Andrew Edmunds | Colin Snook | Marina Walden

Towards Component-based Reuse for Event-B

Turku Centre *for* Computer Science

TUCS Technical Report
No 1140, August 2015

Towards Component-based Reuse for Event-B

Andrew Edmunds
Åbo Akademi University,
Department of Computer Science
Joukahaisenkatu 3-5A, 20520 Turku, Finland
aedmunds@abo.fi

Colin Snook
University of Southampton,
Department of Computer Science
Highfield, Southampton, UK
cfs@ecs.soton.ac.uk

Marina Walden
Åbo Akademi University,
Department of Computer Science
Joukahaisenkatu 3-5A, 20520 Turku, Finland
mwalden@abo.fi

TUCS Technical Report
No 1140, August 2015

Abstract

A component-based system for Event-B would improve re-useability of Event-B machines, and also introduce the capability for bottom-up scalability. Tool support, and a theory underpinning composition, already exists. However, this is based on model decomposition, a top-down scalability solution. In that approach, the abstract model is refined by a composed-machine construct, that *includes* the newly decomposed models. Decomposition has been used as a means to simplify complex models, or, for structural partitioning. The process of creating library components, their composition, and specification of new properties (of the composed elements) is relatively unexplored. In this paper we examine the issues surrounding component-based composition. We introduce the notion of Event-B component interfaces; we discuss communication flow across component boundaries, and what additional proofs obligations may be required; and we describe the tool support for supporting the new approach.

1 Preliminaries

1.1 Motivation

As part of the ADVICeS project [18] we are seeking to improve re-use of the Event-B artefacts, with the aim of increasing the agility of Event-B. The creation of a library of components would assist with this. Existing component-based technology is very domain specific [4], and we believe we can produce a more general approach. We will describe how we can extend iUML-B class diagrams [17], add a new component instance diagram, and build on the existing composition techniques, introduced in [13, 14, 15].

In the remainder of Sect. 1 we provide a brief overview of Event-B. In Sect. 2 we describe some of the theory behind the composition approach, and synchronizing parameters. In Sect. 3 we introduce our ideas for Event-B Component Composition, and interfaces. Sect. 4 discusses an extension to the use of the *composition invariant*. Sect. 5 introduces feasibility proof obligations. These are required to show that communication between assembled components is feasible. Sect. 6 gives an example of an Event-B Component, and its use in a development. Concluding remarks appear in Sect. 7.

1.2 Event-B

Event-B is a language and methodology [2] with tool support provided by the Rodin tool [3]. A useful resource is the Rodin User's Handbook [1]. Event-B has received interest from industry, for the development of railway, automotive, and other safety-critical systems [12]. In Event-B, the system and its properties are specified using set-theory and predicate logic; it uses proof and refinement [11] to show that the properties hold as the development proceeds. Refinement is used to iteratively add detail to the development. Event-B tools are designed to reduce the amount of interactive proof required during specification, and refinement steps [6]. Proof obligations (P.O.s) in the form of sequents, are automatically generated by the Rodin tool. The automatic prover can discharge many of the P.O.s, and the remainder can be tackled using the interactive prover. The basic Event-B elements are *contexts*, *machines* and, *composed-machines*. Contexts define the static parts of the system using sets, constants and axioms, which we denote by: s, c, and a. Machines describe the dynamic parts of a system using variables and events: v, and e, and use invariant predicates I to describe properties that should hold. We specify an event in the following way,

$$e \triangleq \textbf{ ANY } p \textbf{ WHERE } G(p, s, c, v) \textbf{ THEN } A(p, s, c, v) \textbf{ END}$$

where e has parameters p; a guarding predicate G; and actions A. For the state updates (described in the action) to take place, the guard must be true. Guards and actions can refer to the parameters, sets, constants and variables of the machine, and seen contexts. For events to occur, we say that the environment non-deterministically chooses an event, from the set of enabled events. When this happens we say that an event "fires'.' For clarity, in the remainder of the paper, we omit sets, and constants from the description where possible; the discussion largely focusses on parameters and machine variables. As development proceeds, the models can become very detailed. So, complex systems can be broken down into more tractable sub-units, using decomposition [15].

iUML-B [17] is a graphical modelling approach, for Event-B, for specifying state-machines, and class diagrams [9, 16]. Diagrams are are embedded in the parent machine, and contribute to its content using automatic translation. State-machine diagrams can be used to impose an order on the machine's events, and can be animated. Class diagrams are used to define data entities, and their relationships. We propose an extension to class diagrams to expose component interfaces. An example is shown in Fig. 1, which is introduced in Sect. 3

2 Composition of Decomposed Machines

Previous work [14] describes the composition of events, arising from the decomposition of one machine into multiple sub-units. We make use of the shared-event

approach for decomposition, where variables are partitioned into different machines. The multiple, decomposed sub-units, together with the composed-machine construct, form a refinement of the abstract machine. The combined-events clause, of the composed-machine, refines an abstract event e. We write $e_a \parallel e_b$ to combine events e_a and e_b, where subscripts a and b also identify the sub-units (machines). These combined-events are said to *synchronize* (i.e., the events are enabled) when the conjunction of the guards are true. The combined actions are composed in parallel. The semantics of synchronizing events is inspired by the CSP semantics of synchronization [7], however (unlike CSP) matching event names are not required in Event-B. This is due to one of the features of the composed-machine specification; a developer is able to select which events synchronize.

$$\begin{aligned} e_a \triangleq\ & \textbf{ANY } p?_a, p!_a, x_a \textbf{ WHERE } G_a(p?_a, p!_a, x_a, v_a) \\ & \textbf{THEN } A_a(p?_a, p!_a, x_a, v_a) \textbf{ END} \\ e_b \triangleq\ & \textbf{ANY } p?_b, p!_b, x_b \textbf{ WHERE } G_b(p?_b, p!, x_b, v_b) \\ & \textbf{THEN } A_b(p?_b, p!_b, x_b, v_b) \textbf{ END} \\ e_a \parallel e_b \triangleq\ & \textbf{ANY } p, x_a, x_b \textbf{ WHERE } G_a(p, x_a, v_a) \ \wedge\ G_b(p, x_b, v_b) \\ & \textbf{THEN } A_a(p, x_a, v_a) \ \parallel A_b(p, x_x, v_b) \textbf{ END} \end{aligned} \tag{1}$$

Events e_a and e_b may have some common parameters p, matched by name. These are annotated with "!" and "?", for output and input resp. This captures the data flow into, and out of, events; so, output parameters of an event in machine a are written $p!_a$, and data flows to input parameters of an event in machine b, written $p?_b$. We can write $e_a(p!_a)$ for $e_a \triangleq ANY\ p!_a$, and so on. Events can have non-shared parameters, x_a and x_b, which are the local variables of the combined event. The guards G_a and G_b, and actions A_a and A_b range over the parameters of the event, and the machine variables, v_a and v_b.

As we mentioned, the sub-units, together with the composed-machine construct, form a refinement of the abstract machine. Sometimes it can be useful to merge the composed-machine, and sub-units, into a single, unifying machine. This is done without changing the composition's semantics, but can result in duplication of the events guards. and we can simplify pairs of matched input-output parameters, when merging. The combination of each $q? \in p?_a$, paired with a $q! \in p!_b$ results in a single parameter q where,

$$q = q! \parallel q? \tag{2}$$

Therefore, the set of all communicating parameters of an event $(p!_a \| p?_b) \cup (p!_b \| p?_a)$ reduce to p when combined, which is the result in the combined-events of Eq. 1. In an event, to pass a machine variable w as an output parameter $q!$, we add a guard $q = w$. To use an input parameter $q?$ we can assign it to machine variable w, in an action, using the assignment $w := q$

3 Composition with Components

The most important feature of a re-usable library component is its interface. It defines how the component reveals itself to the outside world. Since we intend to use the components in shared-event style composition, we need to reveal a set of events, that may synchronize with some other machine. To do this we can mark the event, on the class diagram, with an annotation. We identify interface events with the letter i, shown next to the event name in Fig. 1. The interface event may, or may not, involve communication across the component boundary. But if such communication is required, through parameter passing, the interface event needs to reveal information about the names and types of the *communicating* parameters. Combined-events that communicate through parameters are required to do so through the common parameter names. Events not marked with the interface annotation, may not synchronize; i.e., they are effectively hidden from the outside world. In Event-B the behaviour of hidden events is interpreted as that of CSP, but since we do not have a sequence operator (except at the implementation-level of refinement [5]), it is equivalent to simple interleaving.

3.1 Using Components

We have seen that machines can be composed, and combined-events synchronize. Now we describe how a components may be created, and assembled using these techniques. Fig. 2 shows how components might be used in a composition diagram. The composed-machine Cm can *include* zero or more library machine components L, and machines under construction M. The machines M and L may be refined. In the diagram, combined-events are represented by dashed lines between the machines.

One shortcoming of the composition diagram is that it gives no information about the number *instances* of each components. A user should be able to select a component, and drop an *instance* onto a canvas. A component instance diagram is depicted in Fig. 4. The diagram would be embedded in the composed machine in the style of iUML-B [17]. From this, a composed-machine can be linked to its *includes* machines, and also the number of instances, and their relationships with other components and machines can be specified.

There are two scenarios for instance creation. One is where the library ma-

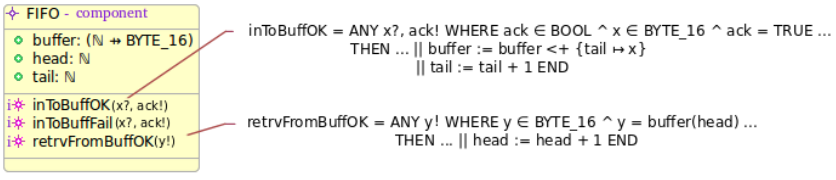

Figure 1: The FIFO Buffer Component

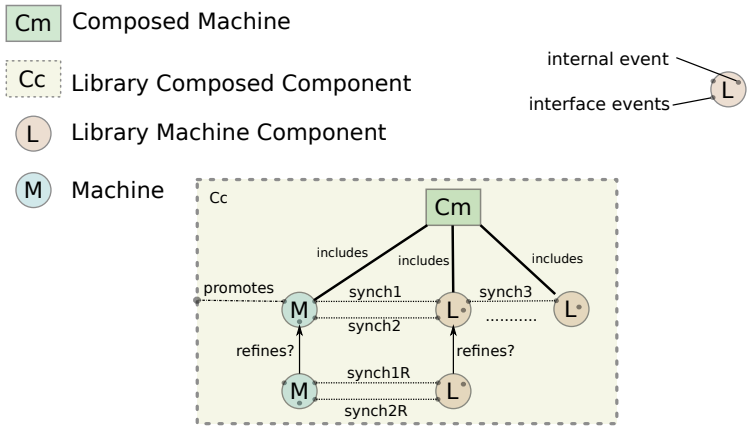

Figure 2: Using Components in a Composition Diagram

chine links to a machine with no corresponding events. That is, its being used for a bottom-up construction. The second is where an existing event must be synchronized with the library event. In this case, a check for compatible parameter names and directions would be done. Feasibility of communication is discussed in Section 5. When no matching event exists, event stubs can be added. This is illustrated in Eq. 3 where e_a is an event in the library machine (annotated with i), and e_b is the generated stub. For each output parameter in $p!_a$, we generate an input parameter in $p?_b$ and vice versa. Typing guards for parameters may be suggested at the time of instantiation, but no other event guards and actions are created automatically. The developer will complete the necessary details, during further development.

$$
\begin{aligned}
i(e_a) \triangleq\ & \textbf{ANY } p?_a, p!_a, x_a \textbf{ WHERE } G_a(p?_a, p!_a, x_a, v_a) \\
& \textbf{THEN } A_a(p?_a, p!_a, x_a, v_a) \textbf{ END} \\
e_b \triangleq\ & \textbf{ANY } p?_b, p!_b \textbf{ WHERE } G_b(p?_b, p!) \textbf{ END} \\
e_a \parallel e_b \triangleq\ & \textbf{ANY } p, x_a \textbf{ WHERE } G_a(p, x_a, v_a) \ \wedge\ G_b(p) \\
& \textbf{THEN } A_a(p, x_a, v_a) \textbf{ END}
\end{aligned}
\tag{3}
$$

3.2 Composite Components

When a composed-machine is defined, it can be added as a library component; the system boundary is then represented by the outer, dashed, box. We need to decide which of the events of the new component are revealed in the interface. We assume that, by default, all events of a composed-machine are hidden, in which case we would need to *promote* some new, or existing, events to the new interface. See Fig. 2. The parameters of the exposed events will need to be marked with the input/output annotations, ? or !. To make it clear to developers which events and parameters, are available for use in a composite component, a separate interface "view" may be useful. This would be similar to the class diagram for simple

components, but it would potentially expose events and parameters of a number of machines. The composed event of Eq. 3 would be promoted to the composite component interface using annotation i, as for a simple components as follows,

$$\begin{aligned} i\ (e_a \parallel e_b) \triangleq\ & \textbf{ANY}\ p!, p?, x_a\ \textbf{WHERE}\ G_a(p, x_a, v_a)\ \wedge\ G_b(p) \\ & \textbf{THEN}\ A_a(p, x_a, v_a)\ \textbf{END} \end{aligned} \tag{4}$$

3.3 Component Development

One of the benefits of the existing decomposition approach is that once decomposition has taken place, the individual machines can be refined independently. For our purposes, we would wish to be able to continue to refine sub-components independently, too. We believe that the addition of a guards to combined-events would reduce this flexibility slightly, since the proof obligations will always need to be discharged for the composition. But this is a natural consequence of placing constraints on composed elements of a model; addition of new variables to a machine, guard strengthening, and data refinement should be able to proceed independently as before. If a composition is complex, it will be possible to add further composed-machines to the refinement chain (which may or may not *include* existing components). Thereby allowing specification of emerging composition properties as development proceeds. We plan to do more investigation into the use of components, and compositions, in team-working. The parallel development of components, and perhaps artefacts within components, is key to making Event-B more agile.

4 Extending the use of the Composition Invariant

The composed-machine CM is made up of the included machines $M_0 .. M_m$, a list of combined-events, and a composition invariant CI. Any of the machines $M_0 .. M_m$ may be library machines. The CI can be used to specify properties of the composition; i.e., properties that cannot be specified in the machine invariant.

$$CI(s, c, v) \tag{5}$$

In Eq. 5 the *composition invariant* has visibility of *all* of the variables of the composed-machines, and their included sets and constants; i.e, s, c, and v resp. So, to distinguish variables, sets and constants, of individual machines in a composition of machines, $M_0 .. M_m$, we write $v = v_0 .. v_m$, sets $s = s_0 .. s_m$, and constants $c = c_0 .. c_m$, resp. The composed-machine invariant CMI, is the conjunction of the individual machine invariants $MI_0 .. MI_m$ and CI. Where each machine invariant only has visibility of its own variables; and sets, and constants of its seen contexts.

$$CMI(CM, M_0 .. M_m,) = CI(s, c, v) \wedge MI_0(s_0, c_0, v_0) \wedge .. \wedge MI_m(s_m, c_m, v_m) \tag{6}$$

To ensure the composition invariant CI of equation 5 is satisfiable, we need to add guards G_{CI}, but currently there is no mechanism in the tool, so this is future work. We would like G_{CI} to range over the whole of v, c and s. That is, the guard requires component-wide visibility of variables; and of the sets, and constants of the seen contexts, of the included machines. The intuitive place to do this is in the composed-machine; where we propose to add an additional guard clause to the combined event clause. We extend the combined event of Eq. 1 with G_{CI}, as follows,

$$\begin{aligned} e_a \parallel eb \triangleq \textbf{ANY } p, x_a, x_b \textbf{ WHERE } \boldsymbol{G_{CI}}(v) \wedge\ G_a(p, x_a, v_a)\ \wedge\ G_b(p, x_b, v_b) \\ \textbf{THEN } A_a(p, x_a, v_a)\ \parallel A_b(p, x_x, v_b) \textbf{ END} \end{aligned} \tag{7}$$

In the composed-machine, we should demonstrate that the invariants (including the CI) are preserved for all events of the included machines. So, for each of the composed events in the composed-machine, we need to show that the invariant holds. Based on the work of [14], we find we need to add the additional guard $\boldsymbol{G_{CI}}$ to the antecedent of the combined event's invariant P.O. Strengthening the guard in this way does not affect invariant preservation as we shall see. We now show invariant preservation, with the additional guard in $INV_{e_j \| e_k}$,

$$INV_{ej} : I_j(v_j) \wedge G_j(p_j, v_j) \wedge A_j(p_j, v_j, v'_j) \vdash i_j(v'_j) \tag{8}$$

$$INV_{ek} : I_k(v_k) \wedge G_k(p_k, v_k) \wedge A_k(p_k, v_k, v'_k) \vdash i_k(v'_k) \tag{9}$$

$$\begin{aligned} INV_{e_j \| e_k} :\ & CI(v) \wedge I_j(v_j) \wedge I_k(v_k) \\ & \wedge G_j(p_j, v_j) \wedge G_k(p_k, v_k) \wedge \boldsymbol{G_{CI}}(v) \\ & \wedge A_j(p_j, v_j, v'_j) \wedge A_k(p_k, v_k, v'_k) \\ & \vdash i_j(v'_j) \wedge i_k(v'_k) \wedge CI(v') \end{aligned} \tag{10}$$

Proof: Assume the hypotheses of $INV_{e_j \| e_k}$,

$$\begin{aligned} & CI(v) \wedge \boldsymbol{G_{CI}}(v) \\ & I_j(v_j) \wedge G_j(p_j, v_j) \wedge A_j(p_j, v_j, v'_j) \\ & I_k(v_k) \wedge G_k(p_k, v_k) \wedge A_k(p_k, v_k, v'_k) \end{aligned} \tag{11}$$

Show:

$$i_j(v'_j) \wedge i_k(v'_k) \wedge CI(v') \tag{12}$$

The proof, as in [14], proceeds, as follows,

$$\begin{aligned} & i_j(v'_j) \wedge i_k(v'_k) \wedge CI(v') \\ & \Longleftarrow \\ & i_j(v'_j) \qquad (GoalEq.\ 8) \\ & \wedge\, i_k(v'_k) \qquad (GoalEq.\ 9) \\ & \wedge\, CI(v') \end{aligned} \tag{13}$$

5 Proof Obligations

5.1 Feasibility of Inputs and Outputs

The use of components, and their interfaces, can be described using a contract, with pre and post-conditions. However, pre-condition semantics are missing in Event-B. So, how do we expect component users to understand what the interface provides? Since we propose using typed/directed event parameters, we can use this information. The parameter's typing guards define the input and output state-spaces. In Event-B guards play a dual role, of typing, and event-enabling. So, we need to be very clear about the semantics of synchronization, about when we expect synchronization and communication to take place. In the existing Event-B approach there is no requirement (in the form of proof obligations) to show that an event is ever enabled. However, we believe that, when assembling components, we do need some reassurance that communication is feasible. That is, there is at least some common set of input and output states that will allow the events to synchronize. A similar concept was explored, in work on feature composition [10]. Our solution is related to the idea of feasibility in Event-B; feasibility proof obligations for non-deterministic assignment, for instance, ensure that there is some initial value in the pre-state that allows a transition to a given post-state. We believe that, in our approach, we should provide some proof of feasibility of synchronization/communication.

We have identified three situations where communication would be feasible during synchronization. 1) Events *can* synchronize if they have overlapping input and output state-spaces. For the state-spaces A and B of two communicating variables, it is necessary to have $A \cap B \neq \varnothing$. However, although this simple feasibility requirement is necessary, it is not sufficient in the case of composition as we shall explain. 2) Since overlapping state-space is not sufficient, we must further constrain the values. We might show feasibility as above, and add an additional proof obligation to prevent out-of-bounds inputs from being sent as outputs. More formally for an output state-space A, and input state-space B, The output state-space needs to be constrained to just $A \cap B$. So, outputs in $A \setminus B$ must be prevented. The output state-space would need to be reduced by typing, or by adding an additional guard. This is not an optimal solution, since it requires additional work during composition. 3) A third alternative is to apply pre-condition semantics where the output state-space is a subset of the input state-space, $A \subseteq B$. Since 1) is the naive solution that motivates us to use 3), we examine the first, and last, of the approaches.

5.2 Feasibility: an Overlapping Input-Output State-space

In the first analysis, we assume that for communication to be feasible, parameters should potentially have some common input and output values. For each param-

eter $q \in p$ marked as input "?" or output "!", a typing guard describes a set of possible inputs or outputs such that $typeof(e, q) = T$, for an event e; where T is a non-empty set. T may describe a range of values, if it is an ordered set. In a composed event, we can show that output and input ranges overlap, to ensure that the communication is feasible. We can simplify this by writing, as a guarded action, a **single event e_j** with a set of input and output parameters $p?_j$, $p!_j$, machine variables v_j and local variables x_j:

$$e_j(p?_j, p!_j) \triangleq g_j(p?_j, p!_j, x_j, v_j) \rightarrow a_J(p?_j, p!_j, x_j, v_j) \tag{14}$$

where g_j is the guard and a_j is the action. The individual parameters in the set $p?_j \cup p!_j$ are denoted by x_j, and typed by $typeof(x_j) = T_j$. For example, where $x_j \in 0 \,..\, 5$, then $typeof(x_j) = 0 \,..\, 5$.

For **combined-events $e_j \parallel e_k$** , we reduce parameters in the union, see Eq. 2, $p = (p?_j \cup p!_k)p_{kj} \cup (p?_k \cup p!_j)$.

$$\begin{aligned} e_j(p?_j, p!_j) =& g_j(p?_j, p!_j, x_j, v_j) \rightarrow a_j(p?_j, p!_j, x_j, v_j), \\ e_k(p?_k, p!_k) =& g_k(p?_k, p!_k, x_k, v_k) \rightarrow a_k(p?_k, p!_k, x_k, v_k) \\ e_j(p?_j, p!_j)||e_k(p?_k, p!_k) =& g_j(p, x_j, v_j) \wedge g_k(p, x_k, v_k) \\ & \rightarrow a_j(p, x_j, v_j) \parallel a_k(, x_k, v_k) \end{aligned} \tag{15}$$

To reason about the input-output state spaces we consider each pair of parameters named q. In a matched pair where $q? \in p?_j$ is an input parameter in e_j and $q! \in p!_k$ is the corresponding output parameter in e_k. For communication to be feasible, the set of output values that $q!$ may take, should overlap with the set of values accepted by the input $q?$. We can find the input-output state space based on a function that we define, called *typeof*. Given an event e_j and a parameter q_j, the function returns the type T, as defined in the guard.

$$typeof(e_j, q) = T \tag{16}$$

The overlap feasibility $\boldsymbol{FIS_{overlap}}$ for some common parameters q, is given by a function taking the two composed events. For all input-outputs q there must be a non-empty, intersecting range of potential values available for communication.

$$\begin{aligned} & FIS_{overlap}(e_j(p?_j, p!_j), e_k(p?_k, p!_k)) \\ & = \\ & forall(q \in p).(typeof(e_j, q) \cap typeof(e_k, q) \neq \varnothing) \end{aligned} \tag{17}$$

For example consider the combined event $e_a \parallel e_b$, where event e_a has a parameter $q \in 0 \,..\, 256$ and event e_b has a parameter $q \in \mathbb{N}1$ then,

$$\begin{aligned} & typeof(e_a, q) \cap typeof(e_b, q) \neq \varnothing \\ & = \\ & 0 \,..\, 256 \cap \mathbb{N}1 \neq \varnothing \\ & = \\ & \top \end{aligned} \tag{18}$$

So communication is feasible, that is, it is possible that some value may be communicated. But, how should we interpret attempts to send values outside of the overlap? The composition semantics of Event-B means that the synchronization cannot be guaranteed to take place, since q is non-deterministically selected from the conjunction of typing guards $0\,..\,256 \cap \mathbb{N}1 = 1\,..\,256$. If the output is zero nothing happens. This behaviour could be an unexpected consequence of composition, which could be dangerous in a critical-system. It would be better to highlight the disparity, or prevent it in the first place, using the next approach.

5.3 Preconditions for Communicating Event Parameters

A better solution is to take a *Design-By-Contract* (DBC) view. In DBC [8], preconditions are defined, and they should be satisfied by users of the interface. In our work the input and output parameters, and their type and direction information, form part of the interface specification. Using this information, we can ensure that matching parameter's output values fall within the range of the allowable inputs.

We need two additional functions to differentiate between input and output parameter types. That is, the types of inputs of $p?$ and outputs of $p!$. Given an event e_j and input parameter $q?$, function *typeOfIn* returns the type T of $q?$, as defined in the guard.

$$typeOfIn(e_j, q?) = T \tag{19}$$

Also, given an event and output parameter $q!$, function *typeOfOut* returns the type T of $q!$.

$$typeOfOut(e_j, q!) = T \tag{20}$$

The pre-condition style feasibility $\boldsymbol{FIS_{preStyle}}$ for two events $e_j \parallel e_k$ is given by,

$$\begin{array}{l} FIS_{preStyle}(e_j(p?_j, p!_j), e_k(p?_k, p!_k)) \\ = \\ forall(q! \in p \wedge q? \in p).(typeOfOut(e_j, q!) \subseteq typeOfIn(e_k, q?) \end{array} \tag{21}$$

As an example, consider event $e_a \parallel e_b$, where e_a has an output parameter $q!$ typed by $q \in 0\,..\,256$, and event e_b has an input parameter $q?$ typed by $q \in \mathbb{N}1$, then,

$$\begin{array}{l} typeOfOut(e_a, q!) \subseteq typeOfIn(e_b, q?) \\ = \\ 0\,..\,256 \subseteq \mathbb{N}1 \\ = \\ \bot \end{array} \tag{22}$$

In this case the pre-condition style P.O., for the same example used in Eq. 18, is not satisfied. This is the desired result, and of course, if the input was of type $\mathbb{N}$, it would be satisfied.

6 An Example

6.1 Specifying a FIFO Buffer Component

We now illustrate how components might be defined in a version of iUML-B [17] adapted to components (or interface) specification. Fig. 1 shows a class diagram with annotations showing some of the event details. The *FIFO* class has buffer, head and tail attributes; and three interface events, marked with i. The buffer holds 16 bit values, which we specify as $BYTE\_16$.

The first event $inToBuffOK$ models successful receipt of a value, and return of a TRUE acknowledgement. The $inToBuffFail$ models failure due to a full buffer, and returns a FALSE acknowledgement. The $retrvFromBuffOK$ models retrieval of a value from a buffer. Two of the events are shown to the right, in the diagram. In the diagram, only the communicating parameters of the event are shown, for clarity. Other parameters may be present, such as local variables; and *instance parameters*, which as the name suggests, (non-deterministically) models instances of a class. We omit instance parameters unless relevant to the discussion. But, we will give a brief explanation of their role. We can see an example of their use in the following fragment, where we are adding a value to a buffer's tail. The $FIFO$ instance is represented by the parameter $this\_FIFO$, and is automatically generated by the iUML-B tool. We have also omitted the *instance parameter* from the description of the events in Fig. 1.

$$buffer(this\_FIFO) := buffer(this\_FIFO) \Leftarrow \{tail(this\_FIFO) \mapsto x\}.$$

The *inToBuffOK* event of Fig. 1 shows the input parameter $x?$ of type $BYTE\_16$; the value to put in the buffer. It also has an acknowledgement, an output parameter $ack!$ of type BOOL, restricted to $ack = \text{TRUE}$; this is returned to the sender on success. The action shows the value x being written to the tail of the buffer, in a statement that overrides ($<+$) any existing value, or adds a new value. The tail is then incremented. In the $retrvFromBuffOK$ event we have an output parameter $y!$ of type $BYTE\_16$, and we see its value being output in the guard $y = buffer(head)$, and the head being incremented in the action.

6.2 Using the FIFO Component

We now introduce a *Producer* class, Fig. 3, that uses two instances of the *FIFO* library component $f1$, and $f2$. We would like to use a component diagram to assist in the creation of the *Producer* since it is related to use of the *FIFO* interface, this feature is future work. Fig. 4 shows how the diagram might look; with two *FIFO* instances connected to a *Producer*, and two *Consumer*s. The combined-events, labelled a .. e, specify synchronizations between the *FIFO* interface, and the Producer/Consumers. Event f does not synchronize with any other event. It should be noted that there is only one machine modelling all instance of the *FIFO*, and another machine modelling all instances of the *Consumer*.

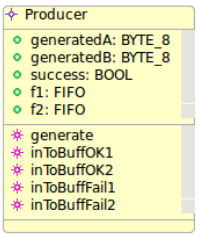

Figure 3: The Producer Class

The tool support for the component diagram could provide (event) stubs for the synchronization, in the *Producer* and *Consumer* machines, when the connections between an interface event and another machine are defined. The stub event for the *Producer.inToBuffOK1* would have the corresponding communicating parameters added. So, recalling that output values are defined in the guard, the stub would be generated as follows,

$$inToBuffOK1 \triangleq \textbf{ ANY } x!, ack? \textbf{ WHERE } ack \in BOOL \wedge x \in BYTE\_8 \textbf{ END}$$

The developer of *Producer* should decide which value to output, and where to the assign the input. A possible solution would be have machine variables $value \in BYTE\_8$ as output, and $success \in BOOL$ for recording the acknowledgement. So we could refine the stub with,

$$\begin{array}{l} inToBuffOK1 \triangleq \\ \quad \textbf{ANY } x!, ack? \textbf{ WHERE } ack \in BOOL \wedge x \in BYTE\_8 \wedge \boldsymbol{x = value} \\ \quad \textbf{THEN } \boldsymbol{success := ack} \textbf{ END} \end{array}$$

Let's now consider the combined event, where we look at the underlying Event-B, showing the *instance parameter*. From Fig. 3 we see that the *Producer* has two *FIFO* instances, $f1$, and $f2$. To perform some synchronization with a library component, the user of the interface requires a separate event for each instance of the component. This is why we have two events in the *Producer* related to the event *FIFO.inToBuffOK*. In the event $Producer.inToBuffOK1$, below, we can see $f1(this\_Producer)$ being used to identify which *FIFO* it is related to. The parameter, $this\_FIFO$, and the guard, could be generated automatically in the

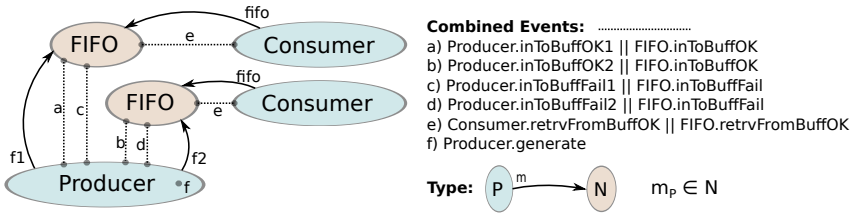

Figure 4: A Component Instance Diagram

Producer by the new tool support. It relates to the parameter, $this\_FIFO$, in the *FIFO* which is generated by iUML-B tools from the *FIFO* class diagram. Also since the shared parameter x has two different types in the individual machines, we take the view that the stronger guard should appear in the clause, since it makes the weaker guard redundant.

$$
\begin{array}{l}
Producer.inToBuffOK1 \parallel FIFO.inToBuffOK \triangleq \\
\quad \textbf{ANY } x!, ack?, this\_Producer, this\_FIFO \\
\quad \textbf{WHERE } ack \in BOOL \wedge x \in BYTE\_8 \wedge x = value(this\_Producer) \wedge \\
\qquad \boldsymbol{this\_FIFO = f1(this\_Producer)} \wedge this\_Producer \in Producer \\
\quad \textbf{THEN } success(this\_Producer) := ack \textbf{ END}
\end{array}
$$

Then we consider our *pre-style* proof obligation of Eq. 21. In our example $Byte\_16 = 0 \mathinner{..} 65535$ and $Byte\_8 = 0 \mathinner{..} 255$, which in this case would be discharged.

$$
\begin{array}{l}
typeOfOut(Producer.inToBuffOK1, x!) \\
\quad \subseteq typeOfIn(FIFO.inToBuffOK, x?) \\
= \\
Byte\_8 \subseteq Byte\_16 \\
= \\
0 \mathinner{..} 255 \subseteq 0 \mathinner{..} 65535 \\
= \\
\top
\end{array}
\tag{23}
$$

6.3 A Composition Invariant

In our example we may want the fifo buffer $f1$ to hold odd numbers, and $f2$ to hold even numbers. This a property of the composition, and should be specified in the composition invariant clause. To specify this, we add an invariant, stating that all of the producer's $f1$ buffers must have $mod2$ of 1, and $f2$ buffers must be $mod2$ of 0; as in the following,

$$\forall p \cdot p \in dom(f1) \implies (\forall v \cdot v \in ran(buffer(f1(p))) \implies v \bmod 2 = 1)$$

It states that for each producer p in the domain of the variable $f1$, and for each value in its buffer, $v \in ran(buffer(f1(p)))$, $v \bmod 2 = 1$ must hold, there is a similar guard stating that $f2$'s values must be even. It would not be possible to specify this in the *Producer* machine since it does not have visibility of FIFO's $buffer$ variable. But this causes an additional problem, since we require the buffers' initial values to satisfy two different invariants, depending on where they are used. We need to make the built-in initialisation non-deterministic enough to accommodate this; and add an additional initialisation event, and a $flag$ to record the occurrence of the event.

We then use the flag in the invariants above, and in any affected combined events. We also add, to the combined events description (in the combined machine) guards preserving the remainder of the composition invariant. This is subject to a future tool enhancement.

7 Conclusions

We propose an extension to the existing composition approach, and introduce Event-B components. The existing composition approach was primarily designed to work with the top-down decomposition approach. We wish to have bottom-up composition for re-use. That is not to say that we intend to dispose of the top-down approach, rather we should have the flexibility to include, and work with, existing artefacts as and when required.

We begin by describing an Event-B component interface: adding input, and output specifiers, "?" and "!" to annotate the event parameters. Otherwise, communicating parameters are matched by name, as in the existing approach. We propose a modification of, or extension to, class diagrams. We would like to use a diagrammatic approach to describe components, and their interfaces. We will add annotations to identify externally visible events, and hide others. To facilitate assembly of components into compositions, we consider a new component diagram; which would work with a new translator to Event-B to support the approach. It should be possible to create new components from compositions, which would create nested components. In this case, we would also need an interface for the newly constructed component. If, by default, all events of the newly constructed component are hidden, we will need to introduce a new annotation to *promote* existing events; thereby exposing them to users through a *composite component* interface.

In our new approach we use the existing *composition invariant* in a new way. We allow the invariant to refer the state inside the composed-machines. We show how we can add additional guards to discharge the new proof obligations. We will add this to the composed-machine's combined-events clause, which requires a tool enhancement.

Ensuring that communication between composed-machines is feasible, requires additional proof obligations. One requirement is to ensure that typing guards are defined in away that provides overlapping sets of input and output values. Otherwise synchronization will never occur, since the guards will never be true in both combined-events. To ensure that the semantics of synchronization are well-defined, we consider the use of a pre-condition style of synchronization. So, proof obligations will be generated to ensure that the values of an output parameter fall completely in the range of values accepted by its corresponding input parameter.

Other future work would be to investigate how composition affects the refinement of composed components. The existing composition approach allows inde-

pendent refinement of sub-models, for team-work, and so on. This may or may not be simple for components, since the *composition invariant* covers multiple machines.

8 Acknowledgements

This work was carried out within the project ADVICeS, funded by Academy of Finland, grant No. 266373.

References

[1] The Rodin User's Handbook. Available at http:// handbook.event-b.org/.

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

[3] J.R. Abrial, M. Butler, S. Hallerstede, T.S. Hoang, F. Mehta, and L. Voisin. Rodin: An Open Toolset for Modelling and Reasoning in Event-B. *Software Tools for Technology Transfer*, 12(6):447–466, November 2010.

[4] M. Butler, J. Colley, A. Edmunds, C. Snook, N. Evans, N. Grant, and H. Marshall. Modelling and Refinement in CODA. In *Refine*, pages 36–51, 2013.

[5] A. Edmunds and M. Butler. Tasking Event-B: An Extension to Event-B for Generating Concurrent Code. In *PLACES 2011*, February 2011.

[6] S. Hallerstede. Justifications for the Event-B Modelling Notation. In J. Julliand and O. Kouchnarenko, editors, *B*, volume 4355 of *Lecture Notes in Computer Science*, pages 49–63. Springer, 2007.

[7] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[8] B. Meyer. Design by Contract: The Eiffel Method. In *TOOLS (26)*, page 446. IEEE Computer Society, 1998.

[9] M. Butler M.Y. Said and C. Snook. Language and Tool Support for Class and State Machine Refinement in UML-B. In *FM 2009: Formal Methods*, pages 579–595. Springer, 2009.

[10] M. Poppleton. The Composition of Event-B Models. In *Abstract State Machines, B and Z*, pages 209–222. Springer, 2008.

[11] J. Wright R. Back. *Refinement Calculus: a systematic introduction*. Springer Science & Business Media, 2012.

[12] A. Romanovsky and M. Thomas. *Industrial deployment of system engineering methods*. Springer, 2013.

[13] R. Silva. Towards the Composition of Specifications in Event-B. In *B 2011*, June 2011.

[14] R. Silva. *Supporting Development of Event-B Models*. PhD thesis, University of Southampton, May 2012.

[15] R. Silva and M. Butler. Shared Event Composition/Decomposition in Event-B. In *FMCO Formal Methods for Components and Objects*, November 2010. Event Dates: 29 November - 1 December 2010.

[16] C. Snook. Event-B Statemachines. at http://wiki.event-b.org/index.php/Event-B_Statemachines, 2011.

[17] C. Snook and M. Butler. UML-B and Event-B: An Integration of Languages and Tools. In *The IASTED International Conference on Software Engineering - SE2008*, February 2008.

[18] The ADVICeS Team. The ADVICeS Project. available at https://research.it.abo.fi/ADVICeS/.

Turku
Centre *for*
Computer
Science

Joukahaisenkatu 3-5 A, 20520 TURKU, Finland | www.tucs.fi

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

Turku School of Economics
- Institute of Information Systems Sciences

Åbo Akademi University
- Computer Science
- Computer Engineering

ISBN ?
ISSN 1239-1891