On Component-Based Reuse for Event-B

Andrew Edmunds[1(✉)], Colin Snook[2], and Marina Walden[1]

[1] Åbo Akademi University, Abo, Finland
aedmunds@abo.fi
[2] University of Southampton, Southampton, UK

Abstract. Efficient reuse is a goal of many software engineering strategies and is useful in the safety-critical domain where formal development is required. Event-B can be used to develop safety-critical systems, but could be improved by a component-based reuse strategy. In this paper, we outline a component-based reuse methodology for Event-B. It provides a means for bottom-up scalability, and can also be used with the existing top-down approach. We describe the process of creating library components, their composition, and specification of new properties (involving the composed elements). We introduce Event-B component interfaces and propose to use a diagrammatic representation of component instances (based on iUML-B) which can be used to describe the relationships between the composed elements. We also discuss the specification of communication flow across component boundaries and describe the additional proof obligations that are required.

1 Introduction

Formal methods can play a useful role in the development of safety-critical systems. Having flexibility in the formal approaches will make them more useful in the development process. Event-B [3] is a formal method, with tool support [11], which has been used in industry. We are seeking to improve the re-use of Event-B artefacts, with the aim of increasing agility. The creation of a library of components and a way to assemble them would facilitate this. Our proposal is based on shared-event composition [23], since we believe that it provides an intuitive abstraction for the encapsulation that is often seen in object-oriented software components.

In its current form, the existing composition approach, and tools, give little guidance as to how machines and their elements should be combined. Components based on shared-event composition provide a useful encapsulation abstraction. The shared-event approach models the interactions between machines using event synchronization, we can view this as an abstraction of method calling in object-oriented components [6]. Since we are focussed on the potential for reuse, we need a way for developers to interpret the intended use of a component. Typically, this is achieved through the use of interfaces, in conventional software engineering practice. In our approach, we introduce *interface events* to make events 'available for use' by potential users of a machine. When considering the

© Springer International Publishing Switzerland 2016
M. Butler et al. (Eds.): ABZ 2016, LNCS 9675, pp. 151–166, 2016.
DOI: 10.1007/978-3-319-33600-8_9

design and reuse of components, we consider how a developer understands what a component does. The state updates are described by events, in the normal way, but to understand the flow of information across the interface boundary we need to introduce additional annotations to represent parameter directions.

Decomposition is a technique for simplifying complex developments or introducing structural partitions. A single machine is split into multiple sub-units, and the equivalence is maintained using a composition technique [22–24]. In this paper, we introduce Event-B components, interfaces, and composite components which builds on the existing composition techniques. To visualize developments, and assist with their specification, iUML-B [26] provides a graphical interface, with state-machines and class diagrams [21,25]. We propose an extension to iUML-B class diagrams to assist with the use of components. We introduce a composed machine diagram showing which machines and components to include in a composition; and we introduce a new component instance diagram to specify how machines and component instances are related. In addition to this, it may be desirable to specify properties involving the elements of newly composed components. We describe how we could extend the existing composition approach, by adding guards to a composed machine, to ensure that these properties are satisfiable.

In Sect. 2, we provide an overview of Event-B, and Sect. 3 describes Event-B composition. Section 4 introduces ideas for component composition and interfaces. Section 5 discusses use of *composition invariant*s, and Sect. 6 introduces proof obligations showing that communication between assembled components is feasible. Section 7 shows an example of a Component. Section 8 discusses related work, and concluding remarks appear in Sect. 9. The work presented here was done as part of the ADVICeS[1] project [28].

2 Event-B

Event-B is a specification language and methodology [1,3] with tool support provided by the Rodin tool [11]. Event-B has received interest from industry, for the development of railway, automotive, and other safety-critical systems [20]. In Event-B, the system, and its properties, are specified using set-theory and predicate logic. It uses proof and refinement [19] to show that the properties hold as the development proceeds. Refinement iterations add detail to the development. Event-B tools are designed to reduce the amount of interactive proof required during specification and refinement steps [8]. Proof obligations 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

[1] The ADVICeS project is funded by Academy of Finland, grant No. 266373.

predicates I to describe the 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 parameter names p; a guarding predicate G; and actions A. State updates (described in the action) can take place only when the guard is true. Guards and actions can refer to the parameters, sets, constants and variables of the machine and seen contexts. For events to occur, the environment non-deterministically chooses an event from the set of enabled events. 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, these can be broken down into more tractable sub-units using decomposition [24].

iUML-B [26] is a graphical modelling approach, for Event-B, for specifying state-machines, and class diagrams [21,25]. The diagrams are linked to a parent machine and contribute to its content using automatic translation tools. State-machine diagrams impose an ordering on the machine's events, and the behaviour can be illustrated using a diagram animator. Class diagrams are used to define data entities and their relationships. We propose to extend class diagrams to expose component interfaces. An example of the extension is shown in Fig. 1, and described in more detail in Sect. 4.

3 Composition of Decomposed Machines

Previous work [23] 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, and events can be combined. The multiple, decomposed sub-units and 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 distinct sub-units (machines). These combined-events are said to *synchronize* (i.e., both of 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 [10], however (unlike CSP) matching event names are not required in the shared-event approach. This is due to one of the features of the composed-machine specification, which allows a developer to select which events to synchronize.

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

Events e_a and e_b may have a set of parameters p in common, with parameters matched by name. Parameter sets are annotated with "!" and "?" to describe

output sets and input sets respectively. The annotations are not part of the parameter name, but simply inform us about the direction of data flow into, and out of, events. The annotation might alternatively be written using the Ada parameter mode style '$p:$ ***in***' for input, and '$p:$ ***out***' for output. To account for multiple machines we use a machine name subscript; the set of output parameter names of an event in machine a is written $p!_a$. This is paired with a set of input parameter names in machine b written $p?_b$. Using syntactic sugar, we can write $e_a(p!_a)$ for $e_a \triangleq$ **ANY** $p!_a \dots$ **END**. Events can have sets of uniquely named, non-shared parameters x_a and x_b, which consist of 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.

The decomposed sub-units, together with the composed-machine construct, form a refinement of the abstract machine. The composed-machine and sub-units can be merged into a single, unifying machine without changing the composition's semantics. This can result in duplication of the events guards, and some simplification may be necessary. The set of communicating parameters of an event $(p!_a \parallel p?_b) \cup (p!_b \parallel p?_a)$ reduces to p when combined. This can be seen 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 a machine variable w in an action, using the assignment $w := q$.

Parameter names are not duplicated when merging input-output pairs. For each input parameter $q?$ that is paired with its output parameter $q!$, after merging we have only a single parameter q, so, $q = q! \parallel q?$.

4 Composition with Components

An important feature of a 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 can synchronize with some other machine. We mark the events on the class diagram with an annotation; interface events have the letter i next to the event name, see Fig. 1. The interface event may involve communication across the component boundary. This will involve parameter passing, so the interface event needs to reveal information about the names and ranges of the *communicating* parameters. Combined-events that communicate via parameters are required to do so through parameters that have the same name. Events that are not

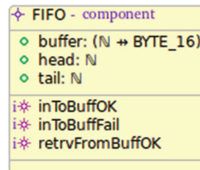

Fig. 1. The FIFO buffer component

marked with the interface annotation may not synchronize: they are 'hidden' from other components. However, they may be non-deterministically selected by the environment, as usual.

4.1 Using Components in a Development

The composition diagram, shown in Fig. 2, is used to import components into a development. It is a new graphical representation of the existing composed machine, but, additionally, it makes use of pre-existing components, which is a new concept. The composed-machine Cm *include*s library machine components L and machines under construction M. In addition, the machines M and L may be associated with an existing refinement chain, or be used to specify a new one. In the diagram, combined-events are represented by dashed lines between the machines. The diagram would be similar to an iUML-B diagram, in that diagrammatic elements are added to the canvas, and the underlying Event-B can be generated, or existing elements linked to it. One shortcoming of the composition diagram is that it gives no information about the number of *instances* of each component. A user should be able to select a component and drop an *instance* onto a canvas. The diagram would be linked to a composed machine, in the style of iUML-B [26]. In this diagram, the number of instances and their relationships with other components and machines can be specified. A component instance diagram, showing this, is depicted in Fig. 4. However, the concepts are best explained using an example, which we defer until Sect. 7.2.

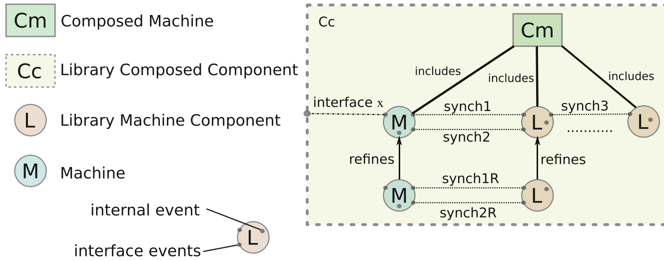

Fig. 2. Using components in a composition diagram

There are two scenarios for instance creation, one is where the library machine links to a machine that initially has no corresponding events. In that case, new events will be added to the machine under construction. The second case is where two existing events are to be synchronized, where a check for compatible parameter names and directions would be done. When no corresponding synchronizing event exists, event stubs can be added. An event stub is a concept taken from programming, where a partial implementation (usually of a method, operation, procedure or function) is generated automatically. This is illustrated in Eq. 2, e_a is an event in the library machine (annotated with **interface**) and e_b is the automatically generated stub. For each output parameter in $p!_a$, we generate an

input parameter in $p?_b$ and vice versa. Order of declaration is not important since parameters are simply matched by name, regardless of the order in which they appear. 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.

$$\begin{aligned}
\textbf{interface } e_a &\triangleq \textbf{ANY } p?_a, p!_a, x_a \textbf{ WHERE } G_a(p_a, x_a, v_a) \\
&\quad\ \textbf{THEN } A_a(p_a, x_a, v_a) \textbf{ END} \\
e_b &\triangleq \textbf{ANY } p?_b, p!_b \textbf{ WHERE } G_b(p_b) \textbf{ END} \\
e_a \parallel e_b &\triangleq \textbf{ANY } p, x_a \textbf{ WHERE } G_a(p, x_a, v_a) \ \wedge \ G_b(p) \\
&\quad\ \textbf{THEN } A_a(p, x_a, v_a) \textbf{ END}
\end{aligned} \tag{2}$$

4.2 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, see Fig. 2. 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. The parameters of the exposed events can be marked with the input/output annotations, ? or !. The composed event of Eq. 2 could be promoted to the composite component interface using the **interface** annotation, as follows,

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

This would make the combined-event available for synchronization with some event outside of the component.

5 The Composition Invariant

5.1 Adding a Guard to Satisfy the Composition Invariant

The existing 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 relating the elements of separate components of a composition. These properties cannot be specified in machine invariants since the elements they refer to reside in separate machines. In the case where a top-down development introduces components in a refinement, and one finds that a particular invariant in the abstraction involves elements that reside in separate components, then, in the refinement, the CI in the composition will reproduce the invariant from the abstraction.

The *composition invariant*, CI(s, c, v), has visibility of *all* of the sets and constants of the included contexts, and variables of the composed-machines s, c,

and v respectively. To identify the sets, constants and variables of the individual machines, in a composition of machines $M_0 \,..\, M_m$, we write $s = s_0 \,..\, s_m$ for sets, $c = c_0 \,..\, c_m$ for constants and $v = v_0 \,..\, v_m$ for variables. The composed-machine invariant CMI is a conjunction of the individual machine invariants $MI_0 \,..\, MI_m$ and CI, where each machine invariant has visibility of its own variables and sets, and also the constants of its seen contexts as follows,

$$CMI(CM, M_0 \,..\, M_m) = CI(s, c, v) \wedge MI_0(s_0, c_0, v_0) \wedge .. \wedge MI_m(s_m, c_m, v_m) \quad (4)$$

To ensure that the composition invariant CI is preserved, we need to add guards G_{CI}, but currently there is no mechanism in the existing tool that does this automatically, so this remains as 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,

$$\begin{aligned} e_a \parallel e_b \triangleq\ & \textbf{ANY}\ p, x_a, x_b\ \textbf{WHERE}\ \boldsymbol{G_{CI}(v)} \wedge\ G_a(p, x_a, v_a)\ \wedge\ G_b(p, x_b, v_b) \\ & \textbf{THEN}\ A_a(p, x_a, v_a)\ \parallel A_b(p, x_b, v_b)\ \textbf{END} \end{aligned} \quad (5)$$

In the composed-machine, we should demonstrate that the invariants (including the CI) are preserved for all events of the included machines. The invariant preservation proof obligation $INV_{e_a \parallel e_b}$ follows, for each invariant i in I, where local variables x are omitted, and the remainder of the parameters refer to those in events before composition,

$$INV_{ea} : I_a(v_a) \wedge G_a(p_a, v_a) \wedge A_a(p_a, v_a, v'_a) \vdash i_a(v'_a) \quad (6)$$

$$INV_{eb} : I_b(v_b) \wedge G_b(p_b, v_b) \wedge A_b(p_b, v_b, v'_b) \vdash i_b(v'_b) \quad (7)$$

$$\begin{aligned} INV_{e_a \parallel e_b} :\ & CI(v) \wedge I_a(v_a) \wedge I_b(v_b) \\ & \wedge G_a(p_a, v_a) \wedge G_b(p_b, v_b) \wedge \boldsymbol{G_{CI}(v)} \\ & \wedge A_a(p_a, v_a, v'_a) \wedge A_b(p_b, v_b, v'_b) \\ & \vdash i_a(v'_a) \wedge i_b(v'_b) \wedge CI(v') \end{aligned} \quad (8)$$

As seen above, we are required to choose an appropriate guard $\boldsymbol{G_{CI}}$ to show that the invariant holds. It appears in the antecedent of the combined-event's invariant proof obligation.

5.2 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. This is possible for the components that are used in compositions, too, and

allows components/machines to be further refined, by a number of teams, independently. To see how this is possible we comment on the two scopes of visibility in a composition. The top-level scope is defined by the composed machine, which has visibility of all of the sets, constants and variables of the machines that it *includes*, and of the contexts that those machines *see*. The CI resides at the top-level in the composed machine, and can refer to variables of multiple machines. The composed machines can have guards added to combined-events, these also have visibility of the variables of the included machines. An important point here, is that the CI should only describe the properties relating to the composition (i.e. properties that cannot be described in a machine/component in isolation). Otherwise, those properties should reside in the normal machine invariants. Each included machine, and its refinement chain, in a composed machine, forms a lower-level scope of visibility. At the lower level scope, the included machines and refinements can be worked on independently since it contains no information about the composition.

The need to recompose components is a natural consequence of placing constraints on elements residing in different machines of a composition. For each of the included machines and their refinement chains, further refinements can be added independently by adding new variables, strengthening guards, and data refinement. 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.

6 Proof Obligations

6.1 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 the dual role of typing and event-enabling. So, we need to be very clear about the semantics of synchronization, and 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 components are assembled (especially pre-existing components) we require assurance that the data flow across component boundaries is compatible. There should be 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 [18]. 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 the feasibility of synchronization/communication. To do this we introduce pre-condition semantics for communication of data across the interface boundary where we show that, for each parameter, the range A of output parameter values is a subset of the range B of input parameter values, $A \subseteq B$. This is determined by the parameter's range as defined in the event guard.

6.2 Preconditions for Communicating Event Parameters

Design-By-Contract (DBC) [14] is an approach for composing modules using contracts. In DBC, pre-conditions and post-conditions are defined in a specification, pre-conditions should be satisfied by users of the contract and post-conditions should be satisfied by implementers of the contract. It can be seen that contracts define an interface specification for a module, and part of their use deals with ensuring that the communicating parameter values are always within acceptable bounds. In our work, the input and output parameters, and their range (a (non-strict) subset of their type) and direction information, form part of the interface specification. We wish to ensure that, for any input/output pair, the output parameter's value falls within the range of the allowable inputs. To do this, we introduce two functions, to differentiate between the ranges of the inputs $p?$ and the outputs $p!$. Given an event e and input parameter $q?$, function *rangeOfIn* returns the range T of $q?$ as defined in the guard.

$$rangeOfIn(e, q?) = T \tag{9}$$

Also, given an event e and output parameter $q!$, function *rangeOfOut* returns the range T of $q!$.

$$rangeOfOut(e, q!) = T \tag{10}$$

We call the pre-condition style feasibility proof obligation $\boldsymbol{FIS_{preStyle}}$. For the combined-event $e_a \parallel e_b$, we have,

$$\begin{aligned} &FIS_{preStyle}(e_a(p?_a, p!_a), e_b(p?_b, p!_b)) \\ &= \\ &\forall q!, q? \cdot (q! \in p! \wedge q? \in p?) \implies (rangeOfOut(e_a, q!) \subseteq rangeOfIn(e_b, q?) \end{aligned} \tag{11}$$

where $q!$ represents an individual output parameter from the set of output parameters $p!$ of an event, and $q?$ represents an individual input parameter from the set of input parameters $p?$ of an event. To satisfy this proof obligation, for each pair of communicating parameters in an event, the output value must fall within the acceptable range of the input. Consider a concrete example of a combined-event $evt1 \parallel evt2$, where $evt1$ has an output parameter named $prm!$ of range $0 \,..\, 256$ and event $evt2$ has an input parameter $prm?$ of range $\mathbb{N}1$, then,

$$\begin{aligned} &rangeOfOut(evt1, prm!) \subseteq rangeOfIn(evt2, prm?) \\ &= 0 \,..\, 256 \subseteq \mathbb{N}1 \\ &= \bot \end{aligned} \tag{12}$$

In this case, the $\boldsymbol{FIS_{preStyle}}$ proof obligation is not satisfied, since 0 does not belong to ℕ1. If the input range was changed to $prm? \in \mathbb{N}$, it would be satisfied.

7 An Example Illustrating the Required Tool Support

7.1 Specifying a FIFO Buffer Component

We now describe how components might be defined in a version of iUML-B [26] adapted to component (or interface) specification. Figure 1 on Page 4 shows the FIFO class diagram. The *FIFO* class diagram contains the attributes: buffer, head and tail, and three interface events, annotated with the letter i. In the model, but not shown in the diagram, the $FIFO$ instance is represented by the parameter $this_FIFO$. It is automatically generated by the iUML-B tool. We now provide details of $inToBuffOK$ and $retrvFromBuffOK$, two of the events shown in the diagram. The $inToBuffOK$ event models successful receipt of a value and the return of TRUE as an acknowledgement. The $retrvFromBuffOK$ models retrieval of a value (by a consumer) from a buffer. We do not show the $inToBuffFail$ event, it handles the case of failure to receive a value, due to a full buffer, and returns a FALSE acknowledgement.

$inToBuffOK \triangleq$
  ANY $x?,\ ack!,\ this_FIFO$
  WHERE $ack \in BOOL \wedge x \in BYTE_16 \wedge ack = TRUE \wedge$
    $tail(this_FIFO) - head(this_FIFO) < buffSize \wedge \ldots$
  THEN $buffer(this_FIFO) := buffer(this_FIFO) \Leftarrow \{tail(this_FIFO) \mapsto x\} \parallel$
    $tail(this_FIFO) := tail(this_FIFO) + 1 \parallel\ \ldots$
  END

The *inToBuffOK* event shows the input parameter $x?$ of range $BYTE_16$: the value to put in the buffer. It also has an acknowledgement, an output parameter $ack!$ of range BOOL, restricted to ack = 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 an existing value or adds a new value. The value of *tail* is incremented in parallel.

$retrvFromBuffOK \triangleq$
  ANY $y,\ this_FIFO$
  WHERE $y \in BYTE_16 \wedge y = buffer(this_FIFO)(head(this_FIFO)) \wedge\ \ldots$
  THEN $head(this_FIFO) := head(this_FIFO) + 1$
  END

In the $retrvFromBuffOK$ event, we have an output parameter $y!$. The output is modelled in the guard $y = buffer(this_FIFO)(head(this_FIFO))$ where y gets the value of the head of the buffer. The *head* value is incremented in the action.

7.2 Using the FIFO Component

We now introduce a *Producer* class, shown in Fig. 3, that uses two instances of the *FIFO* library component $f1$ and $f2$. Figure 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 instances of the *FIFO*, and another modelling all instances of the *Consumer*. Tool support, for the component instance diagram, can provide stubs for the synchronizing events in the *Producer* and *Consumer* machines, when the connections between an interface event and another machine are defined. The stub event for the *Producer*, called *Producer.inToBuffOK1*, would be provided with the appropriate parameters as follows,

$$Producer.inToBuffOK1 \triangleq$$
$$\textbf{ANY } x!,\ ack?,\ this_Producer,\ this_FIFO$$
$$\textbf{WHERE } ack \in BOOL \wedge x \in BYTE_16 \wedge$$
$$this_Producer \in Producer \wedge this_FIFO = f1(this_Producer)$$
$$\textbf{END}$$

The x and ack parameters modelling the communication,are shown, along with two additional parameters that are introduced by the iUML-B translators, these are used to model the component instances. Namely, the parameters

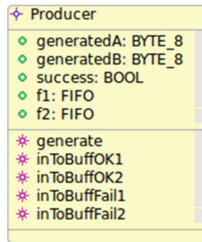

Fig. 3. The producer class

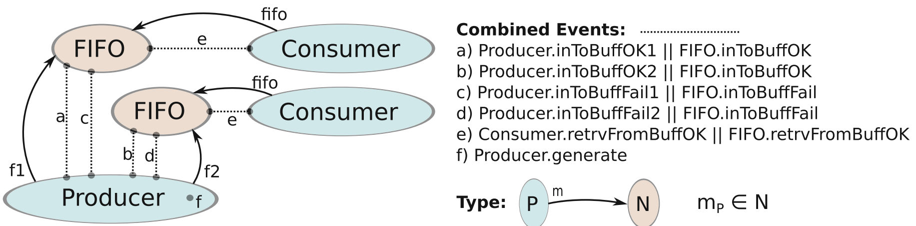

Fig. 4. A component instance diagram

$this_Producer$ and $this_FIFO$. The developer of *Producer* should decide which value to output, and where to assign the input. A possible solution would be to model the output using the variable $value \in BYTE_8$ for output (it is possible that this would be driven by other design concerns) and use $success \in BOOL$ for modelling the acknowledgement. We could then refine the stub with these additions, note the use of the strengthened typing guard,

$Producer.inToBuffOK1 \triangleq$

ANY $x!$, $ack?$, $this_Producer$, $this_FIFO$

WHERE $ack \in BOOL \wedge x \in \boldsymbol{BYTE_8} \wedge \boldsymbol{x = generatedA(this_Producer)} \wedge$
$this_Producer \in Producer \wedge this_FIFO = f1(this_Producer)$

THEN $\boldsymbol{success(this_Producer) := ack}$ **END**

We can see here, that we model the output assignment, of the variable *generatedA* to the parameter x, in the guard, and we model assignment of the return value *ack* to the *success* variable in the action. This is a typical pattern in shared-event synchronization.

Let us 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 synchronize 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 *Producer*, with additional tool support. This relates to the $this_FIFO$ parameter in the *FIFO*, which can be generated by iUML-B tools, from the *FIFO* class diagram. Also, since the shared parameter x has two different ranges in the individual machines, we take the view that the stronger guard should appear in the clause since it makes the weaker guard redundant. The combined event follows,

$Producer.inToBuffOK1 \parallel FIFO.inToBuffOK \triangleq$

ANY x, ack, $this_Producer$, $this_FIFO$

WHERE $ack \in BOOL \wedge x \in BYTE_8 \wedge x = generatedA(this_Producer) \wedge$
$this_Producer \in Producer \wedge \boldsymbol{this_FIFO = f1(this_Producer)} \wedge$
$ack = TRUE \wedge tail(this_FIFO) - head(this_FIFO) < buffSize \wedge \ldots$

THEN $success(this_Producer) := ack \parallel$
$buffer(this_FIFO) := buffer(this_FIFO) \Leftarrow \{tail(this_FIFO) \mapsto x\} \parallel$
$tail(this_FIFO) := tail(this_FIFO) + 1 \parallel \ \ldots$

END

Now we consider the *pre-style* proof obligation of Eq. 11. In our example $Byte_16 = 0 \,..\, 65535$ and $Byte_8 = 0 \,..\, 255$, and we can discharge the proof obligation.

$$\begin{aligned} & rangeOfOut(Producer.inToBuffOK1, x!) \\ & \quad \subseteq rangeOfIn(FIFO.inToBuffOK, x?) \\ & = Byte_8 \subseteq Byte_16 \\ & = 0 \mathinner{..} 255 \subseteq 0 \mathinner{..} 65535 \\ & = \top \end{aligned} \tag{13}$$

7.3 A Composition Invariant

In our example, we may want the FIFO buffer $f1$ to hold odd numbers, and $f2$ to hold even numbers. This is a property of the composition, and should be specified in the composition invariant clause. To do this, we add an invariant stating that values in the producer's $f1$ buffers must have $mod\ 2 = 1$ and those in $f2$ buffers must have $mod\ 2 = 0$. The invariant that constrains $f1$ follows,

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

It states that for each producer p in the domain of the variable $f1$, and for each value v in its buffer, $v \in ran(buffer(f1(p)))$, $v\ mod\ 2 = 1$ 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.

8 Related Work

A concept that is closely related to our approach is that of *Modularisation*. It is an approach for describing components and interfaces in Event-B, by Iliasov et al. [2]. It is based on the shared-variable composition approach. The authors use a pre- and post-condition syntax to specify the component interfaces and behaviour, and they introduce proof obligations to prove refinement. In contrast, shared-event composition provides an appropriate abstraction for the encapsulation that is often seen in object-oriented software components, sharing of variables is usually prohibited here. We also keep the introduction of new syntactic elements to a minimum by extending the existing class diagram techniques. In this way, an implementation of an interface is simply a refinement of that interface. A more detailed discussion of the issues can be found in [7].

Eiffel [14] is another modular approach, based on Design-by-Contract; this too, makes use of pre- and post-condition specifications. We prefer not to use pre- and post-conditions, and present a more integrated method that does not diverge so greatly from the existing iUML-B approach [26]. The CODA component model, of Butler et al. [5], describes how components can be represented on a UML-B style diagram [27]. The underlying model is used to simulate communication between components which are joined using ports and connectors. This makes use of the ProB model checker [13] and uses an oracle to compare various simulation runs. In CODA, the focus is not on reuse. Rather, it is a way

of modelling message queuing over time, and it embodies the communication style found in VHDL [17] which makes it very domain specific.

Hallerstede and Hoang describe interface refinement in [9]. This makes use of the shared-variable composition approach, where external variables, and a corresponding external invariant are specified in the interface. We believe that by using the shared-event approach, we avoid having to consider the effects of sharing variables. By using interface events, as the means for interacting with components, this simplifies reasoning and proof: the encapsulation of traditional software components is closely represented by the shared-event abstraction. Banach extends the interface refinement concepts in [4], by using a CONNECTS construct to make use of interface events: continuing with the shared-variable style.

In other work on components, Kessel and Atkinson discuss reuse of software components [12] focussing on partial matches for suitability in situations where a component's intended use differs from its ultimate use. For Event-B, in the development of high-integrity systems, it will be very important to fully understand the behaviour of a component and underspecification must be judiciously applied to accommodate unforeseen variability. Other notions include location aware components, such as the distributed computation notion of components in CommUnity, which is presented by Oliveira and Wermelinger in [15]; and another concept is the component approach used in the formal modelling of agent interactions with Event-B, from [16]. Both of the latter use quite different notions of components; our components' main purpose is reuse.

9 Conclusions

In the domain of software engineering, the concept of a *component* has many different meanings. Our use of the term component is comparable with its use in the object-oriented software world [29], where a component is an element that is intended for reuse and the flow of data across the component boundary is described by its interface. In the work presented here, we propose an extension to the existing composition approach by introducing Event-B components. The existing composition approach was primarily designed to work as a top-down decomposition method. 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.

Using our diagrammatic extension we can describe a collection of communicating components. We introduce *interface events* as a concept to describe which events can be synchronized with other events. Non-interface events cannot be synchronized, but they can be non-deterministically chosen by the environment, as usual. We add input, and output specifiers, "?" and "!" to annotate the event parameters, in order to clarify the flow of information across the interface boundary. We introduce some new features to a class diagram, to create an interface class, which is annotated to show the interface events. In all other ways, class diagrams are unchanged. We introduce a new composition diagram to describe

machines that are included in the composition. It is a diagrammatic representation of the composed machine construct, and is used to aid visualization of the composed machines and library components. We can describe which events synchronize and show which events are promoted to the interface of a composite machine, but we do not provide information about specific instances. To do this, we introduce a new component instance diagram describing the composition of components as class instances showing the links that describe their synchronizations. We model multiple components using the existing concept of instance parameters where there may be several instances of a particular component in the composition. All instances of a particular component are modelled in a single machine, and there may be multiple components.

Properties involving a number of components may be described in the composition invariant (CI) of the composed machine. These properties extend beyond component boundaries and should be used to describe properties that cannot be described in a single component. The guards related to the CI should go in the combined event. The feasibility of communication across the interface boundaries, for composed events, can be checked by generating additional proof obligations which ensure that, for each parameter, the output values fall completely within the range of values accepted by the corresponding input parameter. This style of feasibility proof will be particularly useful when composing pre-existing components since it is necessary to ensure that the data flow across component boundaries is compatible.

As future work, we plan to do more investigation into the use of components and compositions for team-working, and to provide the additional diagrammatic tool support. In addition, new translators will be required for generating Event-B from the diagrams. Additional tool changes are also required, to add guards to the composed machine's combined event, in order to satisfy the composition invariant.

References

1. The Rodin User's Handbook. http://handbook.event-b.org/
2. Iliasov, A., Troubitsyna, E., Laibinis, L., Romanovsky, A., Varpaaniemi, K., Ilic, D., Latvala, T.: Supporting reuse in Event B development: modularisation approach. In: Frappier, M., Glässer, U., Khurshid, S., Laleau, R., Reeves, S. (eds.) ABZ 2010. LNCS, vol. 5977, pp. 174–188. Springer, Heidelberg (2010)
3. Abrial, J.R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press, Cambridge (2010)
4. Banach, R.: The landing gear case study in hybrid Event-B. In: Boniol, F., Wiels, V., Ait Ameur, Y., Schewe, K.-D. (eds.) ABZ 2014. CCIS, vol. 433, pp. 126–141. Springer, Heidelberg (2014)
5. Butler, M., Colley, J., Edmunds, A., Snook, C., Evans, N., Grant, N., Marshall, H.: Modelling and refinement in CODA. In: Refine, pp. 36–51 (2013)
6. Edmunds, A., Butler, M.: Tasking Event-B: an extension to Event-B for generating concurrent code. In: PLACES 2011, February 2011
7. Edmunds, A., Walden, M.: Modelling "operation-calls" in Event-B with shared-event composition. Technical report 1144 (2015)

8. Hallerstede, S.: Justifications for the Event-B modelling notation. In: Julliand, J., Kouchnarenko, O. (eds.) B 2007. LNCS, vol. 4355, pp. 49–63. Springer, Heidelberg (2006)
9. Hallerstede, S., Hoang, T.S.: Refinement by interface instantiation. In: Derrick, J., Fitzgerald, J., Gnesi, S., Khurshid, S., Leuschel, M., Reeves, S., Riccobene, E. (eds.) ABZ 2012. LNCS, vol. 7316, pp. 223–237. Springer, Heidelberg (2012)
10. Hoare, C.A.R.: Communicating Sequential Processes. Prentice Hall, Upper Saddle River (1985)
11. Abrial, J.R., et al.: Rodin: an open toolset for modelling and reasoning in Event-B. Softw. Tools Technol. Transf. **12**(6), 447–466 (2010)
12. Kessel, M., Atkinson, C.: Ranking software components for pragmatic reuse. In: 2015 IEEE/ACM 6th International Workshop on Emerging Trends in Software Metrics (WETSoM), pp. 63–66. IEEE (2015)
13. Leuschel, M., Butler, M.: ProB: a model checker for B. In: Proceedings of Formal Methods Europe 2003 (2003)
14. Meyer, B.: Design by contract: the Eiffel method. In: TOOLS, vol. 26, p. 446. IEEE Computer Society (1998)
15. Oliveira, C., Wermelinger, M.: The community workbench. In: Proceedings of the 26th International Conference on Software Engineering, pp. 709–710. IEEE Computer Society (2004)
16. Pereverzeva, I.: Formal development of resilient distributed systems. Ph.D. thesis, Åbo Akademi University (2015)
17. Perry, D.L.: VHDL, 2nd edn. McGraw-Hill, New York (1994)
18. Poppleton, M.R.: The composition of Event-B models. In: Börger, E., Butler, M., Bowen, J.P., Boca, P. (eds.) ABZ 2008. LNCS, vol. 5238, pp. 209–222. Springer, Heidelberg (2008)
19. Back, R., Wright, J.: Refinement Calculus: A Systematic Introduction. Springer Science & Business Media, New York (2012)
20. Romanovsky, A., Thomas, M.: Industrial Deployment of System Engineering Methods. Springer, Heidelberg (2013)
21. Said, M.Y., Butler, M., Snook, C.: Language and tool support for class and state machine refinement in UML-B. In: Cavalcanti, A., Dams, D.R. (eds.) FM 2009. LNCS, vol. 5850, pp. 579–595. Springer, Heidelberg (2009)
22. Silva, R.: Towards the composition of specifications in Event-B. In: B 2011, June 2011
23. Silva, R.: Supporting development of Event-B models. Ph.D. thesis, University of Southampton, May 2012
24. Silva, R., Butler, M.: Shared event composition/decomposition in Event-B. In: FMCO Formal Methods for Components and Objects, November 2010
25. Snook, C.: Event-B Statemachines (2011). http://wiki.event-b.org/index.php/Event-B_Statemachines
26. Snook, C.: iUML-B Statemachines. In: Proceedings of the 5th Rodin User and Developer Workshopp (2014)
27. Snook, C., Butler, M.: UML-B: formal modelling and design aided by UML. ACM Trans. Softw. Eng. Methodol. **15**, 92–122 (2006)
28. The ADVICeS Team: The ADVICeS Project. https://research.it.abo.fi/ADVICeS/
29. Wikipedia: Component-Based Software Engineering - Software Component. https://en.wikipedia.org/wiki/Component-based_software_engineering