Andrew Edmunds | Marina Walden

Modelling "Operation-Calls" in Event-B with Shared-Event Composition

Turku Centre *for* Computer Science

TUCS Technical Report
No 1144, October 2015

Modelling "Operation-Calls" in Event-B with Shared-Event Composition

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

TUCS Technical Report
No 1144, October 2015

Abstract

Flexible and efficient development approaches are becoming important for developing high-integrity systems. Formal methods, such as Event-B, may be used as part of the engineering process. Event-B is used for the specification, and development, of high-integrity systems, and is underpinned by mathematical rigour. In the ADVICeS project we are working to improve reuse, with the aim of increasing agility. The existing top-down approach would benefit from bottom-up reusability/scalability solutions. Previous work on Event-B components, was based on the shared-event composition approach. We extended this to introduce reusable components, interfaces, and a reuse mechanism. However, the communication between components is modelled at a relatively high level of abstraction. A more concrete specification would include interfaces with 'callable' *interface events*, modelling operations, and have additional syntax to allow modelling of their invocation. In this paper we describe how such *interface events*, and the call syntax, may be introduced to our approach.

1 Introduction

As part of the ADVICeS project [22] we are working to improve reuse in Event-B [1], with the aim of increasing agility. In recent work we introduced new ideas for Event-B components, and interfaces [8]. This is facilitated by the extension of the existing iUML-B class-diagram tool [21], and by extending the composition work of [17, 18, 19]. This specification is undertaken at a relatively high level of abstraction, using the notion of event synchronisation, inspired by the CSP semantics of synchronisation [11]. It relies on a new diagrammatic representation to *include* machines, and components, in a development; and also to instantiate the components, and describe the (communication) relationships between them. However, a more concrete description of the interaction could use 'callable' *interface events*, which are similar in principle to the callable operations of [12]. The specification of these operations are at a suitable level of abstraction to easily relate them to the normal programming concept of operations, and calls.

In this paper we introduce *interface events* which model programming operations, and *interface event* calls, which model programming operation calls. We also describe how certain *interface event* calls can be nested, in the terms of expressions. In Sect. 2 we provide a brief introduction to Event-B, and in Sect. 3 an overview of the recent Event-B components work. In Sect. 4 we introduce, using a simple example, the notion of procedure-style *interface events*. In Sect. 5 we expand on this to describe nested event calls, using function-style *interface events*, that can be used as terms in expressions. In Sect. 6 we compare our approach with the existing modularisation approach, and in Sect. 7 we conclude.

2 Event-B

Event-B is a language and methodology [1] with a supporting tool, Rodin [2]. The approach has received interest from industry, for the development of railway, automotive, and other safety-critical systems [15]. Using Event-B, the system and its properties are specified using set-theory and predicate logic; it uses proof and refinement [14] to show that the invariants hold as the development proceeds. Refinement is used to incrementally add detail to the specification. The tool support is designed to reduce the amount of interactive proof required during specification, and refinement steps [10]. 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 building blocks 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 v and events 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. As development proceeds, the models can become very detailed. Therefore, complex systems can be broken down into more tractable sub-units, using shared-event [18, 19], or shared-variable [3] decomposition. We make use of shared-event composition style [17], where variables are distributed between (and encapsulated by) machines. With this style, communication between components is modelled by event synchronisation inspired by CSP [11]. Synchronizing events are combined in the *combined events* clause of a composed machine.

iUML-B [21] is a graphical modelling approach, influenced by UML [13], for specifying state-machines, and class diagrams [16, 20]. Diagrams are embedded in a parent machine, and contribute to its content, using automatic translation from the diagrammatic representations, to Event-B. State-machine diagrams are used to impose an ordering on the occurrence of a machine's events; they can be animated to improve understanding of a model's behaviour. Class diagrams are used to define data entities, and their relationships.

3 An Overview of Event-B Components

In previous work [8], we described how Event-B components 'reveal' events and parameters (using annotations) which allows them to synchronize with other machines, and components. We propose a *composed machine diagram*, see Fig. 1, based on [19]. Here the composed machine is identified as CM, and is used to describe the relationships between the machine(s) under development M, and the included component machines L (which are assumed to be developed to an extent that they are suitable for reuse). An annotated class-diagram, such as the FIFO component example at the lower left of Fig. 1, can be viewed as an interface specification, that identifies the public part of an *interface machine*. Here, the annotation i marks which events of a component are allowed to synchronise, while other events should not. In this approach, synchronisation with communicating parameters is sufficient to describe the communication between modules, and this is an abstraction of what happens in an operation call. This is exactly the technique used in [7]. On the right-hand side of the figure, we see *synch*1, a combined event. It is an example of a synchronisation between two machines, this aspect of the diagram summarises the communication occurring across component boundaries, disregarding component instances.

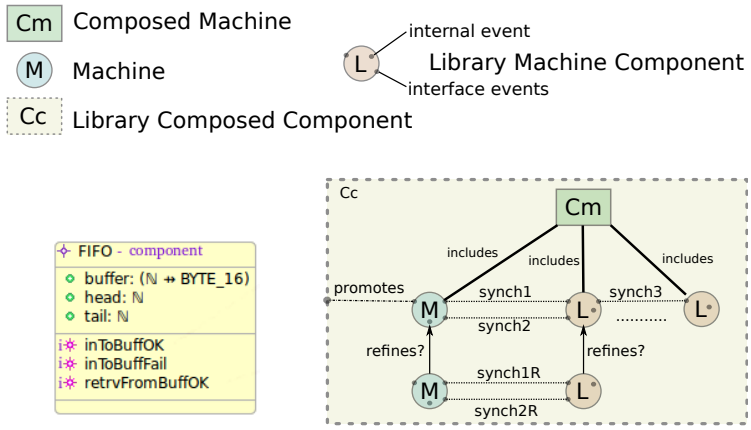

Figure 1: Composition of Library Components [8]

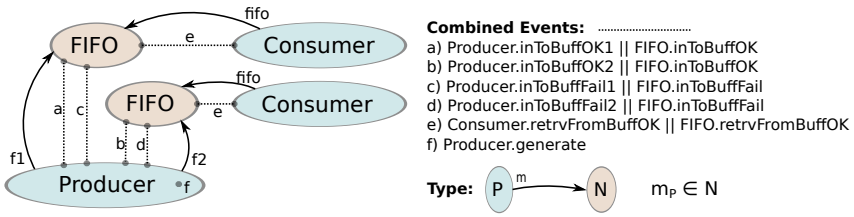

Figure 2: A Component Instance Diagram [8]

In order to describe component instances, and communication across their boundaries more clearly, we introduce a new component instance diagram. A user will connect components to their callers, by linking component instances with connectors, as can be seen in Fig. 2, which is an example from our previous work [8]. Solid, arrowed, lines represent instance containment. Dashed lines represent communication across interfaces. Each dashed line can be associated with two, or more, synchronizing events. Using this information, stubs (a programming term for place-holders) for modelling communication can be generated in the caller, if required. We can add input (output) parameters that match output (input) parameters, in the component, and type them to match. Of course, we can strengthen the guard of any output parameter, since the pre-condition-style PO's will still hold. This approach will also work for communication between two pre-existing components.

In the modularisation approach [12] module interfaces are specified using an entirely different concept. Firstly, an interface module is defined using a pre- and post-condition syntax. Then a machine that 'implements' the interface is defined, which must refine the interface. The approach allows an interface operation to be called, and updates to take place. Refinement rules require the generation of proof obligations to show that an interface is correctly implemented. In our approach, since an *interface machine* is just a machine (with associated interface annotations) any valid refinement of the machine is a valid refinement of the interface.

However, we do produce additional proof obligations, to show that the event's communicating parameters are within certain constraints; any output parameter state-space should be compatible with the corresponding input parameter state space. A more complete comparison of the two approaches is provided in Sect. 6.

4 Procedure-style Interface Events

In order to clarify the use of *interface event* calls, in guards and actions, we refer to the constraints imposed by Ada [4], on function calls and procedure calls. Ada distinguishes between the two, based on the fact that functions can be used in expressions since they must be side effect free, and return a single value which is substituted for the call on its return. Side effects are updates that persist after the call returns, but updates to a single return parameter are permissible. Procedures allow side effects, and return multiple values, and therefore cannot be used in expressions.

We choose to apply constraints on function- and procedure-style *interface events*, in the same way that Ada does. We relate Ada function calls to function-style *interface events*, and Ada procedure calls to procedure-style *interface events*. We consider these to be useful abstractions of safe implementations. In addition, we consider that it will be useful to add an annotation to describe an *interface event* as either a function-style *interface event*, or procedure-style *interface event*. This will assist developers to determine its suitability for use in any given situation.

In the remainder of the section, we describe the new syntax for procedural-style *interface events*, and their calls. In Sect. 5 we describe function-style *interface events*.

We begin by showing the public part of an *interface event*. This simply describes the event name, and input and output parameters, in the style of a Java interface [9]. It has no behavioural information associated with it. It takes the following form (with syntactic sugared parameters),

$$\textbf{Procedural Interface Event } evt(ip?, op!) \triangleq \textbf{WHEN } G(ip, op) \textbf{ END} \quad (1)$$

Here, *evt* is the event name, $ip?$ is the set of input parameters, $op!$ is the set of output parameters, and G consists only of typing predicates. The annotations "?" and "!", for input and output sets, are not part of the name. They simply inform us about the direction of data flow, into, and out of, events. Therefore, they do not appear in guard and action clauses. The annotation might alternatively be written using the Ada parameter mode style '$p :$ ***in***' for input, and '$p :$ ***out***' for output.

Below we show a concrete example of Eq. 1, that we will use as a running

example. The syntactic sugared version follows,

Procedural Interface Event $i\_callee(fp1?\ ret!) \triangleq$
WHEN $fp1 \in \mathbb{N} \wedge ret \in \mathbb{N}$
END

This interface has two parameters, $fp1?$ is an input parameter, and $ret!$ an output parameter, both typed as $\mathbb{N}$. The underlying Event-B is below,

$i\_callee \triangleq$
ANY $fp1?\ ret!$
WHERE $fp1 \in \mathbb{N} \wedge ret \in \mathbb{N}$
END

Having defined an interface, we can add a behavioural description, either in the same machine, or in a refinement (depending on our adopted strategy). We specify the return behaviour of the component by adding the guard, in a refinement. We also show the increment of a machine variable *count*, in the action clause, as an illustration of a side effect.

$callee \triangleq$
REFINES i_callee
ANY $fp1?\ ret!$
WHERE $fp1 \in \mathbb{N} \wedge ret \in \mathbb{N} \wedge \boldsymbol{ret \in 10\,..\,(20 + fp1)}$
THEN $\boldsymbol{count := count + 1}$
END

The event has a non-deterministic definition of its return value, the output parameter $ret!$, it should satisfy the guard $ret \in 10\,..\,(20 + fp1)$. Notice that we also retain the weaker typing guard (although it may appear to be redundant) since we may wish to apply separate annotations for typing guards, to assist with later processing, which would be the case if performing code generation [7].

A procedure-style *interface event* call can be used anywhere that an assignment expression can appear (that is, :=). Thus, it can be composed in parallel, but not otherwise nested in an expression. It takes the following form,

$$iEventName(v) \tag{2}$$

where $iEventName$ is the name of an interface event, and v is the list of machine variables that are passed as actual parameters. We show a concrete example, below, where a refinement of the *interface event* i_callee, named *callee*, will be 'invoked' in the *caller* event. Again we use syntactic sugar to hide the underlying Event-B. We can add the call, as an action, in the *caller* event. An iUML-B class

diagram would be a suitable place to do this, since we can generate the underlying Event-B representation when the iUML-B translators run.

$$caller \triangleq \textbf{BEGIN } callee(var1, var2) \textbf{ END} \tag{3}$$

The event *call* refers to actual parameters, $var1$ and $var2$; where $var1$ is a machine variable representing a required input, and $var2$ is a machine variable that holds (is assigned) the return value. Note that this representation of assignment to $var2$ is like the Ada style *out* parameter, which allows multiple return values. The Java style would be $var2 := callee(var1)$. This Java call style, is exactly the style presented in [6], but we moved away from this approach, in favour of a more seamless integration with Event-B.

4.1 Translation of the Call

The call, described above, is syntactic sugar for the following event, which can be generated automatically by translation tools.

$$\begin{array}{l} caller \triangleq \\ \quad \textbf{ANY } fp1!\ ret? \\ \quad \textbf{WHERE } fp1 \in \mathbb{N} \land ret \in \mathbb{N} \\ \quad \land\ fp1 = var1 \land var2 = ret \\ \quad \textbf{END} \end{array}$$

In the generated *caller* event, we introduce input and output parameters that correspond to the input output parameters of the *callee*. Note that we can automatically add parameter direction annotations, the *callee*'s input $fp1?$ is matched to an output in the *caller*, $fp1!$, and the return value $ret?$ is similar. The guards constraining the input, $fp1 = var1$, and output $var2 = ret$ model the relationship between formal and actual parameters. The parameter's typing guards are matched to those of the *callee*.

We propose to make use of the composition semantics of [5], where the call is modelled by a merge of (conjoined) guards and (parallel) actions. The whole caller and callee update is atomic, we discuss this next.

4.2 The (Merged) Combined Event Representation

The complete behaviour of an event call can be described as a merge of the $callee \parallel caller$ events. We call this two-way synchronisation, since it involves two events. We propose to use shared-event composition to model the call, so, in practice it is not necessary for the merge to actually take place; but the semantics of machine composition using the shared-event style are equivalent to a merge. The merge

provides a useful way for us to describe the result of a composition; a combined event, $callee \parallel caller$, embodies the atomic synchronisation in the merged event,

$$
\begin{array}{l}
callee \parallel caller \triangleq \\
\quad \textbf{ANY } fp1\ ret \\
\quad \textbf{WHERE} \\
\quad fp1 \in \mathbb{N} \wedge ret \in \mathbb{N} \\
\quad \wedge\ fp1 = var1 \wedge var2 = ret \\
\quad \wedge\ ret \in 10 \mathinner{..} (20 + fp1) \\
\quad \textbf{THEN } count := count + 1 \\
\quad \textbf{END}
\end{array}
$$

In the combined event, we do not duplicate parameters and guards. Parameters are matched by name, and the direction annotations can be removed, since the event models both the *caller* and *callee*.

5 Function-style Interface Events for use in Expressions

In order to demonstrate how a call is used as a (nested) term, we extend our example. We introduce a new event $callee2$, and add a call in a new event $newCallee$. The $newCallee$ is similar to *callee* in the previous example. Indeed it could refine the same interface, if we were not distinguishing between procedural- and function-style events. However, since we wish to use the call as a term, it is necessarily a function-style event. It has a single return value, $ret2$, and no side effects; as previously asserted in 4. We define $callee2$ as follows,

$$
\begin{array}{l}
\textbf{Functional Interface Event } \ ret2! \leftarrow i\_callee2(fp1?) \triangleq \\
\quad \textbf{WHERE } fp2 \in \mathbb{N} \wedge ret2 \in \mathbb{N} \textbf{ END}
\end{array}
$$

We add a behavioural description, defining the return value $ret2$ in the guard (highlighted),

$$
\begin{array}{l}
callee2 \triangleq \\
\quad \textbf{REFINES } i\_callee2 \\
\quad \textbf{ANY } fp1?\ ret2! \\
\quad \textbf{WHERE } fp2 \in \mathbb{N} \wedge \boldsymbol{ret2 \in \mathbb{N} \wedge ret2 = 2 * fp1} \\
\quad \textbf{END}
\end{array}
$$

Now, to show the use of a call in an expression, we write $ret \in 10 \,..\, callee2(fp1)$, see Fig. 3. To show its use (highlighted), we introduce *newCallee*, shown below,

$$
\begin{array}{l}
newCallee \triangleq \\
\quad \textbf{REFINES } i\_callee \\
\quad \textbf{ANY } fp1?\ ret! \\
\quad \textbf{WHERE } fp1 \in \mathbb{N} \wedge ret \in \mathbb{N} \wedge \boldsymbol{ret \in 10 \,..\, callee2(fp1)} \\
\quad \textbf{END}
\end{array}
$$

During a call, actual parameters replace formal parameters, and we model this in the guard. As previously mentioned, the *interface event* used in an expression is a function with exactly one return parameter. In the expression, we substitute $callee2(fp1)$ with the return parameter $ret2$, to get $\ldots \wedge ret \in 10 \,..\, ret2$. This will be observable in the final merge of Eq. 5.1 described later.

We are now at the point where we consider the call of the *newCallee interface event*, itself. We write,

$$caller \triangleq \textbf{BEGIN } var2 := newCallee(var1) \textbf{ END}$$

As in the previous examples this is syntactic sugar, and a translation will be performed to generate the following Event-B,

$$
\begin{array}{l}
caller \triangleq \\
\quad \textbf{ANY } fp1!\ ret? \\
\quad \textbf{WHERE } fp1 \in \mathbb{N} \wedge fp1 = var1 \wedge ret \in \mathbb{N} \wedge var2 = ret \\
\quad \textbf{END}
\end{array}
\tag{4}
$$

In the caller, the formal and actual parameter assignments are modelled by the constraints in the guard, $fp1 = var1 \wedge var2 = ret$. As before, input and output parameters are paired; with $fp1!$ and $ret?$ being added to *caller*, to complement the corresponding parameters $fp1?$ and $ret!$ in *newCallee*.

5.1 The (Merged) Combined Event Representation

As in the procedural-style example, when all the callers, and callees are composed, we can represent this as a three-way merge with the following result,

$$
\begin{array}{l}
caller \parallel callee2 \parallel newCallee \triangleq \\
\quad \textbf{ANY } fp1\ ret\ ret2 \\
\quad \textbf{WHERE} \\
\quad fp1 \in \mathbb{N} \qquad -- \ In\ Parameter\ Type \\
\quad \wedge\ fp1 = var1 \qquad -- \ Actual\ parameter\ subst.\ on\ entry\ to\ caller \\
\quad \wedge\ ret \in 10\,..\,ret2 \qquad -- \ Subst.\ call\ for\ return\ parameter \\
\quad \wedge\ ret2 \in \mathbb{N} \qquad -- \ Out\ Parameter\ Type \\
\quad \wedge\ ret2 = 2 * fp1 \qquad -- \ Nested\ return\ value \\
\quad \wedge\ var2 = ret \qquad -- \ Final\ assignment \\
\quad \textbf{END}
\end{array}
$$

Since, we identify communication across interface boundaries by parameter name, and by using input-output direction annotations, we have discovered a new case, in three-way synchronisation, that does not exist in two-way synchronisation. With two-way synchronisation we simply match a pair of parameters "p?" and "p!" and this defines a pair engaging in communication. By introducing three-way synchronisation, as in the last example, we can see that the input parameter $fp1?$, can be used to communicate its value across two component (interface) boundaries, representing two calls. However, the value will always remain the same, since input parameters cannot be assigned to. This avoids us having to redefine the parameter on 'entry', and percolates down through successive calls, as can be seen in Fig. 3. For multi-way synchronisations we can freely pass input parameters across component boundaries by reusing the formal input parameter name. This is not the case for returning output parameters, however, as can be seen in Fig. 3. Returning parameters involve assignment, or substitution in expressions. As the 'calls' return, we model the substitutions, and are required to use the return parameters to record the changes.

6 A Comparison with the Modularisation Approach

We now compare our approach to the existing modularisation approach [12]. The modularisation approach introduces a new interface module construct, which is based on specifying pre- and post-conditions. We do not introduce a 'separate' specification, as such, preferring to extend iUML-B class-diagrams to include the required detail. In our approach a syntactically-sugared version (a pretty-print view) of the interface specification could be shown to the user if required. This could indeed be presented as a pre- and post-condition specification, since these

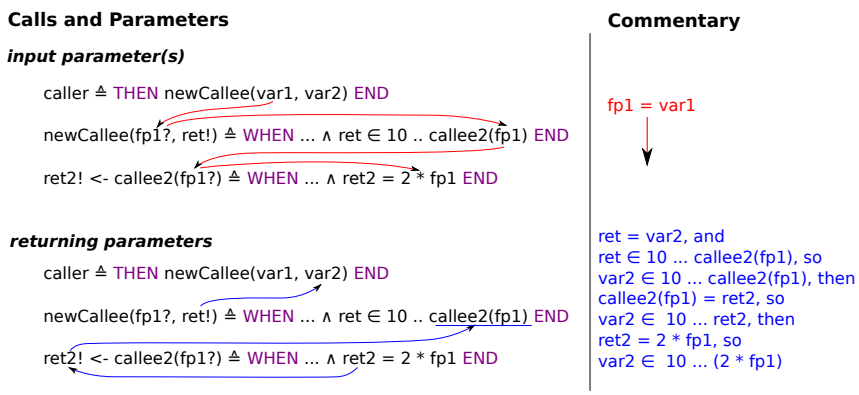

Figure 3: Call Parameters as Guards

details can be derived from our specification. Our components can have update actions, and actions are a form of post-condition; so we consider that they do not require separate specification. We prefer the idea that interfaces (communication across component boundaries) should be kept separate from the behavioural aspects (that is, behaviour of the component). In some sense, modularisation appears to blur the distinction between an interface and the component by having post-conditions. Our approach is driven by the 'programming view' of an interface, one that describes the data flow across the component boundary; such as Ada [4] or Java [9].

The pre-conditions, that are introduced in the modularisation approach, are not explicitly defined in our approach. We take the view that pre-conditions should refer only to input and output parameters (which we understand to be different for modularisation) and we do not need a separate specification for them. This is because parameters are typed in guards, and are associated with direction annotations. Therefore, we can reason about the suitability of input-output ranges, as described in [8]. We can generate pre-style proof obligations, to show the feasibility of communication. We could also derive a pre-condition specification from this information, by interpreting input parameter guards as pre-conditions. We know that the corresponding output parameters would satisfy the pre-conditions. Generally, for any guard G and pre-condition P; when $G = P$, $G \implies P$ is true.

Modularisation is based on shared-variable style decomposition [3]. In this approach the shared variables can be exposed by the interface, to allow modification from outside the component. It could be argued that, in relation to the encapsulation techniques used in programming, this is a departure from the normal practice of component encapsulation through its interface, since direct modification of variables is usually prohibited. Our approach, using shared-event composition, is much more in keeping with the usual interpretation of encapsulation.

In the modularisation approach, the underlying Event-B model is different, in that they model the three steps of assigning, actual parameters to formal parameters, evaluate the action, and assignment to the return parameter, in three

separate events. This requires special ordering guards, and variables, to ensure that no other event takes place while this sequence of events takes place. In our approach, this is modelled in a merger of events, which represents a single atomic step; no additional guards and variables are required. It is also the case that modularisation operations allow side effects. Now, we consider that this is acceptable when the call is not nested in an expression; but it is much less clear that this should be permitted when the calls are nested in expressions. This is usually advised against in programming circles, due to concerns about order of evaluation of terms. Our initial stance is to prohibit side effects in the *interface event* calls that are used in expressions, and acknowledge that it may warrant further investigation. It is, therefore, useful to distinguish between function-style *interface events*, and procedure-style *interface events*, using an additional annotation.

7 Conclusion

The aim of the work described in this paper is to improve reuse of Event-B artefacts, as part of an effort to make Event-B more agile. In this paper we describe an extension of Event-B components and interfaces, that we introduced in [8]. Previous work relies on diagrammatic representations to facilitate instantiation of pre-existing components. Diagrams also describe machine and component inclusion, in a composition; and describes the communications between them. It is based on an abstract description of the interaction between communicating components, and models operation calling, using *interface events*, at a relatively high level of abstraction. Any machine with an interface, we can describe as an *interface machine*.

The extension, presented here, describes how a more concrete specification of communication can be provided. Using a programming-like notation, we introduce 'callable' *interface events*, a low-level description, facilitating the modelling of operations. These are further categorized as procedure-style or function-style *interface events*, where procedure-style *interface events* model operations that allow state updates (similar to Ada procedures). Function-style *interface events* model side effect free operations, for use as terms nested in expressions. We also introduce a new syntax for 'calling' the *interface events*.

Much of the approach is comparable to the callable operations of the modularisation approach [12]; although (as we have shown in Sect. 6) the mechanism is quite different. In the modularisation approach, the underlying Event-B model is different, in that they model the three steps of assigning, actual parameters to formal parameters, evaluate the action, and assignment to the return parameter, in three separate events. This requires special ordering guards, and variables, to ensure that no other event takes place while this sequence of events takes place. In our approach, this is modelled in a merger of events, which represents a single atomic step; no additional guards and variables are required. It is also the case

that modularisation operations allow side effects. Now, we consider that this is acceptable when the call is not nested in an expression; but it is much less clear that this should be permitted when the calls are nested in expressions. This is usually advised against in programming circles, due to concerns about order of evaluation of terms. Our initial stance is to prohibit side effects in calls that are used in expressions, and acknowledge that it may be necessary to distinguish between function-style *interface events* that are side effect free, and procedure-style *interface events*, which allow side effects. Procedure-style calls were also introduced by Walden in [23], where they discuss a distributed load balancing algorithm, in the B-Action systems paradigm.

We restrict pre-conditions to typing guards, this is not so for modularisation (as far as we can tell). We consider that the analogue between *interface event* calls, and its sequential programming equivalent, only extends so far. In Event-B it is expected that all calls (and their nested calls) evaluate atomically. This may be difficult to reason about at the implementation level; where the behavioural aspects of components (blocking behaviour in particular) is developed in isolation, and then composed. Due consideration should be given to handling false guards, for instance. If updates are implemented sequentially, then roll-back would be required in the event of a guard being false. It would be very easy to add a few guards to some events, and then have a complex implementation arising from it. We prefer to keep things simple initially, then through experimentation, we can see if our approach is sufficient.

In future work we would like to provide tool support for our approach; and then perform some experiments, evaluating the usability, and applicability for solving various problems in a industrial setting. We are also interested in deriving a behavioural description from a component specification, and combining it with the interface specification, for the purposes of improving location, and retrieval, of the most relevant components, from component libraries.

8 Acknowledgements

This work was undertaken during the ADVICeS project, funded by Academy of Finland, grant No. 266373.

References

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

[2] 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.

[3] J.R. Abrial and S. Hallerstede. Refinement, Decomposition, and Instantiation of Discrete Models: Application to Event-B. *Fundam. Inform.*, 77(1-2):1–28, 2007.

[4] John Barnes. Ada 2012 Rationale. 2013.

[5] M. Butler. Decomposition Structures for Event-B. In *Integrated Formal Methods iFM2009, Springer, LNCS 5423*, volume LNCS. Springer, February 2009.

[6] A. Edmunds. *Providing Concurrent Implementations for Event-B Developments*. PhD thesis, University of Southampton, March 2010.

[7] A. Edmunds, A. Rezazadeh, and M. Butler. Formal Modelling for Ada Implementations: Tasking Event-B. In *Ada-Europe 2012: 17th International Conference on Reliable Software Technologies*, June 2012.

[8] A. Edmunds, C. Snook, and M. Walden. Towards Component-Based Reuse for Event-B. Technical Report 1140, 2015.

[9] J. Gosling, B. Joy, G. Steele, G. Bracha, and A. Buckley. *The Java Language Specification*. Java (Addison-Wesley). Addison Wesley, 2014.

[10] 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.

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

[12] A. Iliasov, E. Troubitsyna, L. Laibinis, A. Romanovsky, K. Varpaaniemi, D. Ilic, and T. Latvala. Supporting reuse in Event B development: Modularisation approach. In *Abstract State Machines, Alloy, B and Z*. Springer, 2010.

[13] Object Management Group (OMG). UML 2.0 Superstructure specification. Available at http://www.omg.org/technology/uml/index.htm.

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

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

[16] M.Y. Said, M. Butler, 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.

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

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

[19] R. Silva and M. Butler. Shared Event Composition/Decomposition in Event-B. In *FMCO Formal Methods for Components and Objects*, November 2010.

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

[21] 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.

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

[23] M. Waldén. Distributed Load Balancing. In Emil Sekerinski and Kaisa Sere, editors, *Program Development by Refinement*, Formal Approaches to Computing and Information Technology FACIT, pages 255–300. Springer London, 1999.

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

ISSN 1239-1891