Modularising and Promoting Interoperability for Event-B Specifications using Institution Theory

Marie Farrell, Rosemary Monahan, and James F. Power

Maynooth University, Maynooth, Co. Kildare, Ireland
mfarrell@cs.nuim.ie

Abstract. We motivate the need for modularisation constructs in the Event-B formal specification language. We utilise the specification building operators of the theory of institutions to provide these modularisation constructs and we present an example of a traffic-light simulation to illustrate our approach.

Keywords: Event-B; institutions; refinement; formal methods; modular specification

1 Introduction and Motivation

Modern software development focuses on model-driven engineering: the construction, maintenance and integration of software models, ranging from high-level design documents (often expressed through diagrams) down to program code. For example, a model-based approach to developing software might start with the construction of a design model, such as a UML class diagram, developing class functionality via state machines, and then forward engineering this to program code.

In the field of formal software development we can prove the correctness of a particular piece of software by reasoning logically about the system. Just as for non-formal development, it can be beneficial to model the aspects of a system using a variety of specialised formalisms to ensure different aspects of its correctness. In formal software engineering we can map between these levels of abstraction in a verifiable way through a process known as refinement, which can take place within a single modelling language, or between languages at different levels of abstraction [10]. The ideal scenario has been described as a "theory supermarket", in which a developer can shop for suitable theories with confidence that they will work together [5].

This paper is centered around an illustrative example of a specification in Event-B, inspired by one in the *Rodin User's Handbook* [8]. In Section 2 we provide an overview of the Event-B formalism, identify its limitations and discuss related work. We have identified the theory of institutions as a means of enhancing the Event-B formalism and we define an institution for Event-B in Section 3. The definition of $\mathcal{EVT}$, our institution for Event-B, enables us to utilise the specification-building operators provided by institutions and to re-cast our

example in modular form. We address refinement in Section 4 since this is of central importance in Event-B, and show how this too can be modularised using institutional specification building operations. We summarise our contributions and outline future directions in Section 5.

2 Background: specification and refinement in Event-B

Many tools and formalisms have been developed to facilitate the process of software specification, refinement and verification. Event-B is an industrial-strength language for system-level modelling and verification that combines an event-based logic with basic set theory [1]. A key feature of Event-B is its support for formal refinement, which allows a developer to write an abstract specification of a system and then to gradually add complexity in a provable correct way [11]. The *Rodin Platform*, an integrated development environment for Event-B, ensures the safety of system specifications and refinement steps by generating appropriate proof-obligations, and then discharging these via support for various theorem provers [8]. Event-B has been used extensively in a number industrial projects, such as the Paris Métro Line 14, and is a relatively mature language.

2.1 Modelling the Traffic Lights

We have provided an illustrative example of an Event-B model of a traffic lights system that is inspired by one in the *Rodin User's Handbook* [8]. Figure 1 presents an Event-B machine for a traffic lights system with one light signalling cars and one light signalling pedestrians. In general, machine specifications model the dynamic behaviour of a system and can contain variable declarations (lines 2-3), invariants (lines 4-7) and event specifications (lines 8-33). The goal of the specification is to ensure that it is never the case that both cars and pedestrians receive the "go" signal at the same time (represented by boolean flags on line 3).

Figure 1 specifies five different events (including a starting event called `Initialisation` (lines 9-13)). An event is composed of a guard (predicate) and an action which is represented as a before-after predicate relating the new values of the variables to the old. Events can happen in any order once their guards evaluate to true and the theorem provers check that each invariant is not violated by any event. For example, the `set_peds_go` event as specified on lines 14-19, has one guard expressed as a boolean expression (line 16), and one action, expressed as an assignment statement (line 18). In general an event can contain many guards and actions, though a variable can only be assigned to once (and assignments occur in parallel) [8].

In addition to machine specifications, *contexts* in Event-B can be used to model the static properties of a system (constants, axioms and carrier sets). Figure 2 provides a context giving a specification for the data-type $COLOURS$ and uses the axiom on line 7 to explicitly restrict the set to only contain the constants *red*, *green* and *orange*.

```
MACHINE mac1
VARIABLES
  cars_go, peds_go
INVARIANTS
   inv1 : cars_go ∈ BOOL
   inv2 : peds_go ∈ BOOL
   inv3 : ¬ (peds_go = true ∧ cars_go = true)
EVENTS
  Initialisation
    begin
       act1 : cars_go := false
       act2 : peds_go := false
    end
  Event set_peds_go ≙
    when
       grd1 : cars_go = false
    then
       act1 : peds_go := true
    end
  Event set_peds_stop ≙
    begin
       act1 : peds_go := false
    end
  Event set_cars_go ≙
    when
       grd1 : peds_go = false
    then
       act1 : cars_go := true
    end
  Event set_cars_stop ≙
    begin
       act1 : cars_go := false
    end
END
```

Fig. 1: Event-B machine specification for a traffic system, with cars and pedestrians controlled by boolean flags.

```
CONTEXT ctx1
SETS
  COLOURS
CONSTANTS
  red, green, orange
AXIOMS
   axm1 : partition(COLOURS, {red}, {green}, {orange})
END
```

Fig. 2: Event-B context specification for the colours of a set of traffic lights.

```
MACHINE mac2
refines mac1
SEES ctx1
VARIABLES
  cars_colour, peds_colour, buttonpushed
INVARIANTS
   inv1 : peds_colour ∈ {red, green}
   inv2 : (peds_go = TRUE) ⇔ (peds_colour = green)
   inv3 : cars_colour ∈ {red, green}
   inv4 : (cars_go = TRUE)⇔ (cars_colour = green)
   inv5 : buttonpushed ∈ BOOL
EVENTS
  Initialisation
    begin
       act1 : cars_colour := red
       act2 : peds_colour := red
    end
  Event set_peds_green ≙
  refines set_peds_go
    when
       grd1 : cars_colour = red
       grd2 : buttonpushed = true
    then
       act1 : peds_colour := green
       act2 : buttonpushed := false
    end
  Event set_peds_red ≙
  refines set_peds_stop
    begin
       act1 : peds_colour := red
    end
  Event set_cars_green ≙
  refines set_cars_go
    when
       grd1 : peds_colour = red
    then
       act1 : cars_colour := green
    end
  Event set_cars_red ≙
  refines set_cars_stop
    begin
       act1 : cars_colour := red
    end
  Event press_button ≙
    begin
       act1 : buttonpushed := true
    end
END
```

Fig. 3: A refined Event-B machine specification for a traffic system, with cars and pedestrians controlled by a button-activated pedestrian light.

Figure 3 shows an Event-B machine specification, `mac2`, which refines `mac1` from Figure 1. `mac2` refines `mac1` by first introducing the new context on line 3 and then by replacing the truth values used in the abstract machine with new values from the carrier set $COLOURS$. During refinement, the user typically supplies a *gluing invariant* relating properties of the abstract machine to their counterparts in the concrete machine [8]. The gluing invariants shown in lines 8 and 10 of Figure 3 define a one-to-one mapping between the concrete variables introduced in mac2 and the abstract variables of mac1. As specified in lines 7 and 9, the new variables ($peds\_colour$ and $cars\_colour$) can be either red or $green$, thus the gluing invariants map $true$ to $green$ and $false$ to red.

Event-B permits the addition of new variables and events - *buttonpushed* on line 5 and `press_button` on lines 44-47. Also, the existing events from `mac1` are renamed to reflect refinement; for example, on lines 18-19 the event `set_peds_green` is declared to refine `set_peds_go`. This event has also been altered via the addition of a guard (line 22) and an action (line 25) which incorporate the functionality of a button-controlled pedestrian light.

2.2 Limitations of Event-B

Although a very mature formalism, we believe there are two main areas where the Event-B language needs further improvement:

Modularity: The given example highlights features of the Event-B language, but notice how, in Figure 1 the same specification has to be provided twice. The events `set_peds_go` and `set_peds_stop` are equivalent, modulo renaming of variables, to `set_cars_go` and `set_cars_stop`. Ideally, writing and proving the specification for these events should only happen once. Therefore, we can identify one weakness of Event-B as its lack of well-developed modularisation constructs and it is not easy to combine specifications in Event-B with those written in other formalisms [7].

Interoperability: Large software systems are often at such a level of complexity that no single formalisation, encoding or abstraction of can aptly represent and reason about the whole system. This results in the system being modelled numerous times, often in separate formalisms, thus requiring proof repetition. For example, when developing software using Event-B, it is at least necessary to transform the final concrete specification into a different language to get an executable implementation.

2.3 Related Work

One suggested method of providing modularity for Event-B specifications is model decomposition, originally proposed by Abrial (shared variable [2]) and Butler (shared event [15]), and later developed as a plugin for the *Rodin Platform* [16]. The *shared variable* approach partitions the model into sub-models based on events sharing the same variables. The *shared event* method partitions

the model based on variables participating in the same events. This approach is quite restrictive in that it is not possible to refer to the same element across sub-models. Also, it is impossible to select which invariants are allocated to each sub-model, currently, only those relating to variables of the sub-model are included.

Another approach is the modularisation plugin for *Rodin* [7], which is based on the *shared variable* method outlined above. Here, *modules* split up an Event-B model and are paired with an *interface* describing conditions for incorporating the module into another. Module interfaces list the operations contained in the module. Modules are similar to machines but they may not specify events. The events of a machine which imports an interface can see the visible constants, sets and axioms, call the imported operations, and the interface variables and invariants are added to the machine. Although similar to the *shared variable* approach proposed by [2] this method is less restrictive, as invariants can be included in the module interface.

Both of these *Rodin* plugins provide some degree of modularisation for Event-B but they do not directly enhance the Event-B formalism itself nor do they provide scope for the interoperability of Event-B with other formalisms and/or logics.

Current approaches to interoperability in Event-B consist of a range of *Rodin* plugins to translate to/from Event-B but these lack a solid logical foundation. Examples include UML-B [17] and EventB2JML [3].

In summary, the existing approaches to addressing modularity and interoperability issues in Event-B tend to be somewhat ad hoc, causing difficulties for interaction, proof sharing and maintainability. The goal of our research is to develop a set of modularisation constructs for Event-B that will be sufficiently generic, so that they are well understood (particularly in formal terms), and so that they can map easily to similar constructs in other formalisms. We also intend to provide scope for the interoperability of Event-B with other formalisms as part of our solution.

The core to ensuring modularisation and interoperability in model-driven engineering is *meta-modelling*: the modelling of modelling languages. Similarly, when dealing with logic-based formalisms that include specification, refinement and proof, the key to ensuring interoperability is a suitable meta-logical framework, that will allow for the specification of specification languages. These ideas have a long history in logic, going back at least to Tarski's work in the 1930s on the definition and classification of consequence relations [18].

3 Institutions - a Meta-Logical Framework

Originating from Tarski's work on metamathematics and consequence, the theory of institutions provides a meta-logical framework in which a set of specification building operators can be defined allowing you to write, modularise and build up specifications in a formalism-independent manner [6, 18]. In order to represent a formalism/logic in this way, the syntax and semantics for it must first

be defined in a uniform way using some basic constructs from category theory [14]. Institutions have been defined for many logics and formalisms, including programming-related formal languages such as UML and CSP [9, 13].

A huge benefit of this approach is that it facilitates the use of specification building operators that provide modularisation constructs to any logic/formalism presented in this way. Examples of formalisms that have been improved by using institutions are those for UML state machines [9] and CSP[13]. Readers familiar with Unifying Theories of Programming may note that the notion of institutions in this way is similar to the notion of a "theory supermarket" in which one can shop for theories with confidence that they will work together [5].

Once a formalism/logic has been described using institutions a range of specification building operators become available [14]. These operators facilitate the modularisation of specifications and describe how specifications can be combined in different formalisms/logics. They facilitate the combination (and, $+$, $\cup$), extension (then), hiding (hide via, reveal) and renaming (with) of specifications.

We have represented Event-B as a logic in the theory of institutions, as such, we gain the use of specification building operators to increase modularity and an embedding in a framework designed to promote and facilitate interoperability with other formalisms. Since a key feature of Event-B is refinement, it is vital that any representation of Event-B maintain the same notion of refinement. The theory of institutions already accounts for this so there is no need to redefine it [14]. Another major benefit of representing Event-B in terms of institutions is that it provides a formal semantics for Event-B that is fully rooted in a mathematical foundation.

3.1 Defining $\mathcal{EVT}$, an institution for Event-B

$\mathcal{EVT}$, our formalisation of Event-B in terms of institutions is based on splitting an Event-B specification into two parts:

- A data part, which can be defined using some standard institution such as that for algebra or first-order logic. We have chosen $\mathcal{FOPEQ}$, the institution for first order predicate logic with equality [14], since it most closely matches the kind of data specification needed.
- An event part, which defines a set of events in terms of formula constraining their before- and after- states. Our specification here is closely based on $\mathcal{UML}$, an institution for UML state machines [9].

While we do not have space to present the details fully formally here, they are not more complex than those normally used for first-order logic, with appropriate assignments for the free variables named in the event specification variables.

In order to build an institution for Event-B, which we call $\mathcal{EVT}$, it is necessary to specify and verify a series of definitions (using category theory) for its syntax and semantics. Once these language elements have been specified, the next step is to verify that the resulting metalogical structure is actually a valid institution. This is ensured by proving the *satisfaction condition* which states, in formal terms, the basic maxim of institutions, that "truth is invariant under change of

notation". We can only outline this process here, but full details of the institution $\mathcal{EVT}$ and the associated proofs are available from our website.[1]

I. Signatures. A signature over $\mathcal{EVT}$ describes the vocabulary that we are allowed to use when writing Event-B specifications, and consists of names for sorts, operations, predicates, events and variables. We assume that each operation, predicate and variable name is appropriately indexed by its sort and arity. Signature *morphisms* provide a mechanism for moving between vocabularies while repspecting arities, sort-indexing and initialisation events. By construction, these morphisms can be extended in a uniform way to models and sentences.

II. Models. A *data state* consists of a set of values for each of the variables in the signature corresponding to the declared sort of the variable. A possible execution of a machine is then represented by a *trace*, which is just a sequence of data states, with each step in this sequence being labelled by an event name. Finally, a *model* of an $\mathcal{EVT}$ signature is a set of such traces, specified as a relation over the states whose constituent tuples are labelled by event names.

III. Sentences. A sentence over $\mathcal{EVT}$ is then an element of an Event-B specification written using the names from the signature. In the *Rodin Platform* Event-B sentences are presented (with suitable syntactic sugaring) as:

$I(\overline{x})$
Event e $\widehat{=}$
  when
    guard-name : $G(\overline{x})$
  then
    act-name : $A(\overline{x}, \overline{x}')$
  end

where $I(\overline{x})$ and $G(\overline{x})$ are predicates representing the invariant(s) and guard(s) respectively over the set of variables $\overline{x}$. In Event-B, actions are interpreted as before-after predicates i.e. the statement $x := x+1$ is interpreted as the predicate $x' = x + 1$. Therefore, a predicate of the form $A(\overline{x}, \overline{x}')$ represents the action(s) over the sets of variables $\overline{x}$ and $\overline{x}'$. Here $\overline{x}'$ is the same set of variables as $\overline{x}$ but with all of the names primed.

Based on this we can define the syntax of $\mathcal{EVT}$ in terms of two types of sentence.

- The first kind of sentence is an *invariant definition*, which is simply a predicate $\phi(\overline{x})$ over the variables in the signature.
- The second kind of sentence represents an *event definition* and consists of a pairing $e \mathrel{\widehat{=}} \psi(\overline{x}, \overline{x}')$ where e is an event name and $\psi(\overline{x}, \overline{x}')$ is a $\mathcal{FOPEQ}$ predicate corresponding to $G(\overline{x}) \wedge A(\overline{x}, \overline{x}')$ in the above Event-B sentence.

IV. Satisfaction. The satisfaction of $\mathcal{EVT}$-sentences by $\mathcal{EVT}$-models is split into satisfaction for each kind of sentence.

[1] http://www.cs.nuim.ie/∼mfarrell/

```
1  spec TwoBools over FOPEQ
2    Bool
3    then
4      ops
5        i_go, u_go : Bool
6      preds
7        ¬ (i_go = true ∧ u_go = true)

8  spec LightAbstract over EVT
9    TwoBools
10   then
11     Initialisation
12       begin
13         act1 : i_go := false
14       end
15     Event set_go ≙
16       when
17         grd1 : u_go = false
18       then
19         act1 : i_go := true
20       end
21     Event set_stop ≙
22       then
23         act1 : i_go := false
24       end

25 spec mac1 over EVT
26   (LightAbstract with σ1)
27     and (LightAbstract with σ2)
28
29   where
30     σ1 = {i_go ↦ cars_go,
31           u_go ↦ peds_go,
32           set_go ↦ set_cars_go,
33           set_stop ↦ set_cars_stop}
34
35     σ2 = {i_go ↦ peds_go,
36           u_go ↦ cars_go,
37           set_go ↦ set_peds_go,
38           set_stop ↦ set_peds_stop}
```

Fig. 4: A modular institution-based presentation corresponding to the abstract machine `mac1` in Fig 1.

Satisfaction of invariant sentences: If some predicate $\phi(\overline{x})$ is given as an invariant, then an $\mathcal{EVT}$-model m satisfies $\phi(\overline{x})$ if that formula evaluates to true in each data state of the model.

Satisfaction of event sentences: Given an definition of an event e by some predicate $\psi(\overline{x}, \overline{x}')$, then an $\mathcal{EVT}$-model m satisfies this sentence if the predicate $\psi(\overline{x}, \overline{x}')$ evaluates to true for every pair of states in the model labelled by e.

In order to ensure that an institution $\mathcal{EVT}$ has good modularity properties it is necessary to carry out some category theoretic proofs. In particular, pushouts must exist in the category of signatures and the institution must have the amalgamation property [14].

3.2 An Example of specification-building in $\mathcal{EVT}$

Defining $\mathcal{EVT}$, an institution for Event-B, allows us to restructure Event-B specifications using the standard specification building operators for institutions [14]. Thus $\mathcal{EVT}$ provides a means for writing down and splitting up the components of an Event-B system, facilitating increased modularity for Event-B specifications. Figure 4 is a presentation (set of sentences) over the institution $\mathcal{EVT}$ corresponding to the Event-B machine `mac1` defined in Figure 1. The presentation in Figure 4 consists of three specifications:

Lines 1-7: The specification TwoBools, technically in $\mathcal{EVT}$, can be presented as a pure specification over $\mathcal{FOPEQ}$, declaring two boolean variables con-

strained to have different values. The predicate on line 7 here corresponds to the invariant on line 7 of Figure 1.

Lines 8-24: LIGHTABSTRACT is a specification over $\mathcal{EVT}$ for a single traffic light that extends TWOBOOLS (then). It contains the events `set_go` and `set_stop`, with a constraint that a light can only be set to "go" if its opposite light is not.

Lines 25-38: The specification MAC1 combines (and) two versions of LIGHTABSTRACT each with a different signature morphism (σ_1 and σ_2) mapping the specification variables and event names to those in the Event-B machine.

Notice that the specification for each individual light had to be explicitly written down twice in the Event-B machine in Fig 1. In our modular institution-based presentation it is only necessary to have one light specification and simply supply the required variable and event mappings. In this way, $\mathcal{EVT}$ adds much more modularity than is currently present in Event-B, and these constructs are well defined in the theory of institutions providing a formal mathematical foundation for modularisation in Event-B.

4 Refinement in the $\mathcal{EVT}$ institution

Event-B supports three forms of machine refinement: the refinement of event internals (guards and actions) and invariants; the addition of new events; and the decomposition of an event into several events [4, 8]. It is therefore essential that any formalisation of Event-B be capable of capturing these concepts. The theory of institutions provides support for all three types of Event-B refinement as it is, in fact, equipped with a well-defined notion of refinement [14].

4.1 A modular, refined specification

Figure 5 contains a presentation over $\mathcal{EVT}$ corresponding to the main elements of the Event-B specification MAC2 presented in Figures 2 and 3. Here, we present three data specifications over $\mathcal{FOPEQ}$ and three event specifications over $\mathcal{EVT}$.

Lines 1-11: We specify the *Colours* data type with a standard data specification, as can be seen in Figure 2. The specification TWOCOLOURS describes two variables of type *Colours* constrained not both be green at the same time. This corresponds to the gluing invariants on lines 8 and 10 of Figure 3. The specification modularisation constructs used in Figure 5, allow these properties to be handled distinctly and in a manner that facilitates comparison with the TWOBOOLS specification on lines 1-7 of Figure 4.

Lines 12-28: A specification for a single light is provided in LIGHTREFINED which uses TWOCOLOURS to describe the colour of the lights. As was the case with LIGHTABSTRACT in Figure 4, the specification makes clear how a single light operates. An added benefit here is that a direct comparison with the abstract specification can be done on a per-light basis.

Lines 29-46: The specifications BOOLBUTTON and BUTTONSPEC account for the part of the MAC2 specification that requires a button. These details were

```
1  spec Colours over FOPEQ
2    then
3      sorts
4      Colours free with red|green|orange

5  spec TwoColours over FOPEQ
6      Colours
7      then
8        ops
9          icol, ucol : Colours
10       preds
11         ¬(icol = green ∧ ucol = green)

12 spec LightRefined over EVT
13   TwoColours
14   then
15     Initialisation
16       begin
17         act1 : icol := red
18       end
19     Event set_green ≙
20       when
21         grd1 : ucol = red
22       then
23         act1 : icol := green
24       end
25     Event set_red ≙
26       then
27         act1 : icol := red
28       end

29 spec BoolButton over FOPEQ
30   Bool
31   then
32     ops
33       button : Bool

34 spec ButtonSpec over EVT
35   BoolButton
36   then
37     Event gobutton ≙
38       when
39         grd1 : button = true
40       then
41         act1 : button := false
42       end
43     Event pushbutton ≙
44       then
45         act1 : button := true
46       end

47 spec mac2 over EVT
48   (LightRefined with σ3)
49   and
50   (LightRefined and
51         (ButtonSpec with σ5)
52         with σ4)
53
54 where
55   σ3 = {i_col ↦ cars_colour,
56         u_col ↦ peds_colour,
57         set_green ↦ set_cars_green,
58         set_red ↦ set_cars_red}
59
60   σ4 = {i_col ↦ peds_colour,
61         u_col ↦ cars_colour,
62         set_green ↦ set_peds_green,
63         set_red ↦ set_peds_red}
64
65   σ5 = {gobutton ↦ press_button}
```

Fig. 5: A modular institution-based presentation corresponding to the refined machine `mac2` specified in Fig 3.

woven through the code in Figure 3 (lines 5, 11, 22, 25, 45) but the specification-building operators allow us to modularise the specification and group these related definitions together, clarifying how the button actually operates.

Lines 47-65: Finally, to tie this all together we must combine a copy of LightRefined with a specification corresponding to the sum (and) of LightRefined and ButtonSpec with appropriate signature morphisms. This second specification combines the event `gobutton` in ButtonSpec with the event `set_green` in LightRefined thus accounting for `set_peds_green` in Figure 3. One small issue involves making sure that the name replacements are done correctly, and in the correct order, hence the bracketing on lines 48-52 is important.

The combination of these specifications involves merging two events with different names: `gobutton` from ButtonSpec with the event `set_green` from LightRefined. To ensure that these differently-named events are combined into an event of the same name we use the signature morphism σ_5 to give `gobutton` the same name as `set_green` before combining them. By ensuring that the events have the same name, and combines both events' guards and actions and the morphism σ_4 names the resulting event `set_peds_green`. The resulting specification will also contain the event `pushbutton`.

Note that the labels given to guards/actions are syntactic sugar to make the specification aesthetically resemble the usual Event-B notation for guards/actions.

5 Conclusion and Future Work

Our specification of $\mathcal{EVT}$ has enabled us to address the limitations in the Event-B language that we have identified in Section 2.2 as follows:

Modularity: By defining $\mathcal{EVT}$ and carrying out the appropriate proofs, we gain access to an array of generic specification building operators [14]. These facilitate the combination (and, $+$, $\cup$), extension (then), hiding (hide via, reveal) and renaming via signature morphism (with) of specifications. Representing Event-B in this way provides us with a mechanism for combining and parameterising specifications. Most importantly, these constructs are formally defined, a crucial issue for a language used in formal modelling.

Interoperability: Institution comorphisms can be defined enabling us to move between different institutions, thus providing a mechanism by which a specification written over one institution can be represented as a specification over another. Devising meaningful institutions and corresponding morphisms to/from Event-B provides a mechanism for not only ensuring the safety of a particular specification but also, via morphisms, a platform for integration with other formalisms and logics.

Another benefit of developing an institution-based specification for Event-B is that it provides a formal semantics for the language, something that has not been explicitly developed thus far, although developed informally [1].

We have successfully specified an institution for the Event-B formalism and proved the relevant properties that allow for the use of the modularisation constructs. Our current task is that of implementation using the Heterogeneous Tool-Set, Hets, a framework for institution-based heterogeneous specifications [12]. A significant future challenge is the integration of proofs for Event-B, developed using the *Rodin* platform, into the more general Hets environment.

Devising meaningful institutions and corresponding morphisms to/from Event-B provides a mechanism for not only ensuring the safety of a particular specification but also, via morphisms, a potential for integration with other formalisms. Interoperability and heterogeneity are significant goals in the field of software engineering, and we believe that the work presented in this paper provides a basis for the integration of Event-B with other formalisms based on the theory of institutions.

References

1. J.-R. Abrial. *Modeling in Event-B: System and Software Engineering.* Cambridge University Press, New York, NY, USA, 1st edition, 2010.

2. J.-R. Abrial and S. Hallerstede. Refinement, decomposition, and instantiation of discrete models: Application to Event-B. *Fundamenta Informaticae*, 77(1-2):1–28, 2007.
3. N. Cataño, T. Wahls, C. Rueda, V. Rivera, and D. Yu. Translating B machines to JML specifications. In *27th Annual ACM Symposium on Applied Computing*, pages 1271–1277, New York, NY, USA, 2012. ACM.
4. K. Damchoom, M. Butler, and J.-R. Abrial. Modelling and proof of a tree-structured file system in Event-B and Rodin. In *Formal Methods and Software Engineering*, volume 5256 of *LNCS*, pages 25–44. 2008.
5. J. Fitzgerald, P. G. Larsen, and J. Woodcock. Foundations for model-based engineering of systems of systems. In *Complex Systems Design & Management*, pages 1–19. Springer, 2014.
6. J. A. Goguen and R. M. Burstall. Institutions: abstract model theory for specification and programming. *Journal of the ACM*, 39(1):95–146, 1992.
7. 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*, volume 5977 of *LNCS*, pages 174–188. 2010.
8. M. Jastram and P. M. Butler. *Rodin User's Handbook: Covers Rodin V.2.8*. CreateSpace Independent Publishing Platform, USA, 2014.
9. A. Knapp, T. Mossakowski, M. Roggenbach, and M. Glauer. An institution for simple UML state machines. In *Fundamental Approaches to Software Engineering*, volume 9033 of *LNCS*, pages 3–18. 2015.
10. C. Morgan. *Programming from Specifications*. Prentice Hall, U.K., 2nd edition, 1998.
11. C. Morgan, K. Robinson, and P. Gardiner. *On the Refinement Calculus*. Springer, 1988.
12. T. Mossakowski, C. Maeder, and K. Lüttich. The heterogeneous tool set, Hets. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4424 of *LNCS*, pages 519–522, 2007.
13. T. Mossakowski and M. Roggenbach. Structured CSP - a process algebra as an institution. In *Recent Trends in Algebraic Development Techniques*, volume 4409 of *LNCS*, pages 92–110. 2007.
14. D. Sanella and A. Tarlecki. *Foundations of Algebraic Specification and Formal Software Development*. Springer, 2012.
15. R. Silva and M. Butler. Shared Event Composition/Decomposition in Event-B. In *International Symposium on Formal Methods for Components and Objects*, pages 122–141. Springer, 2010.
16. R. Silva, C. Pascal, T. S. Hoang, and M. Butler. Decomposition tool for Event-B. *Software: Practice and Experience*, 41(2):199–208, 2011.
17. C. Snook and M. Butler. UML-B: Formal modeling and design aided by UML. *ACM Trans. on Software Engineering and Methodology*, 15(1):92–122, 2006.
18. A. Tarski. On some Fundamental Concepts of Metamathematics. In *Logic, semantics, metamathematics: papers from 1923 to 1938*, chapter 3. Hackett Publishing, 1983.