An Institutional Approach to Modularisation in Event-B

Marie Farrell*, Rosemary Monahan, and James F. Power

Dept. of Computer Science, Maynooth University, Co. Kildare, Ireland

Abstract. This paper presents a formulation of the Event-B formal specification language in terms of the theory of institutions. Our goal is to exploit the specification-building operations of this theory to modularise Event-B specifications. A case study of a traffic-light simulation is presented to illustrate our approach.

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

1 Introduction and Motivation

Event-B is a state-based formalism for system-level modelling and verification, combining set theoretic notation with event-driven modelling. Event-B ensures the safety of a given specification via proof-obligation generation and theorem proving with support for these provided by the *Rodin Platform* [5, 8].

The main pitfalls of Event-B are that it lacks well-developed modularisation constructs and it is not easy to combine specifications in Event-B with those written in other formalisms [7]. Our thesis, presented in this paper, is that the theory of institutions can provide a framework for defining a rich set of modularisation operations and promoting interoperability and heterogeneity for Event-B. Examples of formalisms that have been improved by using institutions in this way are those for UML state machines [9] and CSP [13].

This paper is centered around a case study of a specification in Event-B, inspired by one in the *Rodin Handbook* [8], and we use this to give an overview of Event-B and related work in Section 2. To use the specification-building operators provided by institutions we define an institution for Event-B in Section 3. This allows us to re-cast our case study 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 Event-B

Event-B *machines* are used for modelling the dynamic part of a systems specification [2]. Figure 1 presents an Event-B machine for a traffic lights system

* This project is funded by the Irish Research Council, email: mfarrell@cs.nuim.ie

with one light signalling cars and one signalling pedestrians [8]. 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). In general, machine specifications can contain variable declarations (lines 2-3), invariants (lines 4-7) and event specifications (lines 8-33).

Figure 1 specifies five kinds of event (including a starting event called `Initialisation` (lines 9-13)). Each event specification has a guard part, specifying when it can be activated, and an action part, specifying what happens when the event is activated. 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).

In addition to machine specifications, *contexts* in Event-B can be used 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*.

A central feature of Event-B is its support for refinement, allowing a developer to write an abstract specification of a system and gradually add complexity through a series of refinement steps [3, 11]. Figure 3 shows an Event-B machine specification for `mac2` which refines the machine `mac1` from Figure 1. This machine is refined first by 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 44-46. 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.

This 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. Also, the invariants in Figure 1 (lines 6,7,8) are concerned with typing. It is superfluous to check typing as often as invariant preservation. In an ideal world, these fundamental details would be dealt with elsewhere.

```
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 set of pedestrian lights.

2.1 Related Work on modularisation for Event-B

The Event-B formalism lacks modularisation constructs which will improve its scalability for use in industrial projects [7]. One suggested method of providing modularity for Event-B specifications is model decomposition, originally proposed by Abrial and later developed as a plugin for the *Rodin Platform* [3, 17].

Two methods of model decomposition were addressed by Abrial: *shared variable* and *shared event*. *Shared variable* partitions the model into subcomponents based on events sharing the same variables. The *shared event* method partitions the model based on variables participating in the same events. An event that uses variables from different subcomponents must be split into partial versions of the non-decomposed event (restricted to only parameters, guards and actions referring to the relevant variable). To use the plugin the user selects the machine to be decomposed and defines the subcomponents to be generated. They then select the style of decomposition to use and can opt to decompose the contexts in a similar fashion. The tool generates the subcomponents which can undergo further refinement and be recomposed [17]. The moment in development where decomposition takes place is important: decomposing early may yield an overly abstract sub-component that cannot be refined without knowledge of the others; decomposing late may mean that the already concrete model will not benefit from the decomposition. In addition, this approach is quite restrictive in that it is not possible to refer to the same element across subcomponents. Also, it is impossible to select which invariants are allocated to each subcomponent, currently, only those relating to variables of the subcomponent 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 component and are paired with an *interface* describing conditions for incorporating the module into another. Module interfaces list the operations contained in the module. These 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. These imported interface variables may be referred to in invariants/guards and actions but may not be directly updated in an action.

Although similar to the *shared variable* approach proposed by [3] this method is less restrictive, as invariants can be included in the module interface. This provides a mechanism for modularity in Event-B but there are a large number of different components that the user needs to utilise and it is unclear how a model developed using these constructs might be translated into/combined with a different formalism. It does not improve the overall scalability of Event-B as a formalism but rather increases modularity within Event-B only [7].

By providing an institution for the Event-B formalism, we increase the the modularity of Event-B specifications via the use of specification building operators [14]. Furthermore, our approach provides scope for the interoperability of Event-B and other formalisms. The development of $\mathcal{EVT}$, our institution for Event-B, has been closely based on $\mathcal{UML}$, the institution for UML state ma-

chines [9]. Both institutions describe state-based formalisms and, therefore, by keeping $\mathcal{UML}$ in mind during the development of $\mathcal{EVT}$, it should be possible to design meaningful signature morphisms between these formalisms in the future. UML-B (*Rodin* plugin) provides a way of moving from UML models to Event-B code, but this was not developed with institutions in mind [18]. Our HETS implementation of $\mathcal{EVT}$ takes inspiration from that of CSPCASL in HETS which uses $\mathcal{CASL}$ to specify the data parts of CSP processes [12]. This will become clearer when we discuss separating out Event-B specifications in Section 3.

An attempt has already been made to provide an institution and corresponding morphisms for Event-B and UML [4]. However, the definition of Event-B sentences and models are vague, making it difficult to evaluate their semantics in a meaningful way. Also, the models described more closely resemble the set-theoretic foundations of B specifications, whereas we concentrate on event-based models in $\mathcal{EVT}$. While it is always possible to describe formalisms to some extent using institutions, the presentation of a case study in both Event-B and its modular, institutional version is an important element of developing this work.

An institution exists for $\mathcal{CSP}$ in which models are described as traces [13]. The main issue in developing meaningful morphisms between $\mathcal{EVT}$ and $\mathcal{CSP}$ arises from the fact that CSP is a much richer language than Event-B (and thus $\mathcal{EVT}$). This means that although it appears easy to move from $\mathcal{EVT}$ to $\mathcal{CSP}$, the opposite direction will be more difficult. That said, the work of Schneider et al. provides a basis for such a translation by contributing a CSP semantics of Event-B [15].

Other related work includes various plugins that translate between different formalisms and Event-B; however thus far none use institution theory to do so.

3 Institutions

The theory of institutions, originally developed by Goguen and Burstall in a series of papers originating from their work on algebraic specification, was ultimately generalised to include multiple formalisms [6]. The key observation is that once the syntax and semantics of a formal system have been defined in a uniform way, using some basic constructs from category theory, then a set of *specification building operators* can be defined that allow you to write, modularise and build up specifications that can be defined in a formalism-independent manner [14]. Institutions have been defined for many logics and formalisms, most notably algebraic specification and variants of first-order logic, but also programming-related formal languages relevant to Event-B such as UML state machines [9], CSP [13] and Z specifications [10].

Definition(Institution) Formally, an *institution* $\mathcal{INS}$ for some given formalism will consist of definitions for

Vocabulary: a category **Sign** of *signatures*, with signature morphisms $\sigma : \Sigma \rightarrow \Sigma'$ for each signature Σ, $\Sigma' \in |\mathbf{Sign}|$.

Syntax: a functor $\mathbf{Sen} : \mathbf{Sign} \to \mathit{Set}$ giving a set $\mathbf{Sen}(\Sigma)$ of Σ-*sentences* for each signature Σ and a function $\mathbf{Sen}(\sigma) : \mathbf{Sen}(\Sigma) \to \mathbf{Sen}(\Sigma')$ which translates Σ-sentences to Σ'-sentences for each signature morphism σ.

Semantics: a functor $\mathbf{Mod} : \mathbf{Sign}^{op} \to \mathbf{Cat}$ giving a category $\mathbf{Mod}(\Sigma)$ of Σ-*models* for each signature Σ and a functor $\mathbf{Mod}(\sigma) : \mathbf{Mod}(\Sigma') \to \mathbf{Mod}(\Sigma)$ which translates Σ'-models to Σ-models (and Σ'-morphisms to Σ-morphisms) for each signature morphism $\sigma : \Sigma \to \Sigma'$.

Satisfaction: a *satisfaction relation* $\langle \models_{\mathcal{INS},\Sigma} \subseteq |\mathbf{Mod}(\Sigma)| \times \mathbf{Sen}(\Sigma) \rangle$, determining satisfaction of Σ-sentences by Σ-models for each signature Σ, such that for any signature morphism σ the translations $\mathbf{Mod}(\sigma)$ of models and $\mathbf{Sen}(\sigma)$ of sentences preserve the satisfaction relation:

$$M' \models_{\mathcal{INS},\Sigma'} \mathbf{Sen}(\sigma)(\phi) \quad \text{iff} \quad \mathbf{Mod}(\sigma)(M') \models_{\mathcal{INS},\Sigma} \phi$$

for any $\phi \in \mathbf{Sen}(\Sigma)$ and $M' \in |\mathbf{Mod}(\Sigma')|$ [14].

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.

A signature in $\mathcal{EVT}$ is a tuple Σ, where $\Sigma = \langle S, \Omega, \Pi, E, V \rangle$ with $\langle S, \Omega, \Pi \rangle$ a standard first-order signature consisting of a set of sort, operation and predicate names, the latter two indexed appropriately by sort and arity. E is a set of event names which must contain the event name `Initialisation`. V is a set of sort-indexed variable names. Signature morphisms respect arities, sort-indexing and initialisation events.

For any $\Sigma = \langle S, \Omega, \Pi \rangle$ in $\mathcal{FOPEQ}$, a Σ-**sentence** is a set of closed first-order formulae built out of atomic formulae using $(\wedge, \vee, \neg, \Rightarrow, \Longleftrightarrow, \exists, \forall)$. Formulae are algebraic term equalities ($\langle S, \Omega \rangle$-terms) over the predicates, variables and, `true` and `false`.

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(x̄, x̄′)
end
```

where $I(\overline{x})$ and $G(\overline{x})$ represent 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. $x := x + 1$ is interpreted as $x' = x + 1$. Therefore, $A(\overline{x}, \overline{x}')$ 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. We use an appropriate bijective renaming function ι to do this.

Based on this we can define sentences over $\mathcal{EVT}$. Taking $\Sigma = \langle S, \Omega, \Pi, E, V \rangle$, we define two types of **sentences** over $\mathcal{EVT}$:

- This represents the *invariant(s)* in an Event-B specification. It consists of pair $\langle inv, \phi \rangle$ where inv is some keyword for invariant sentences and ϕ is a $\mathcal{FOPEQ}$ sentence with a tuple of free variables $\overline{x} \subseteq V$.
- This represents the *event* in an Event-B specification. It consists of a pair $\langle e, \phi \rangle$ where $e \in E$ and ϕ is a $\mathcal{FOPEQ}$ predicate sentence over $\overline{x}$ and $\overline{x}'$, the free variables in V. ϕ is a sentence in $\mathcal{FOPEQ}$. Pairing ϕ with an event name e provides a definition for *event* e.

Invariants are represented as a special kind of sentence for simplicity. A sentence defines a single event or an invariant (which is a $\mathcal{FOPEQ}$ sentence). As invariants in Event-B are defined outside events we separate their definition from $\langle e, \phi \rangle$. Including them as part of $\langle e, \phi \rangle$ would localise them to each event.

One option was to index the events by their invariants, where global invariants would be indexed by all events in the signature, giving one uniform sentence type. However, if we were to take this approach we would encounter problems with amalgamation in $\mathcal{EVT}$, since such invariants projected into a larger signature would no longer be global for that signature. Thus, to ensure that invariants remain global for a signature, we have opted to syntactically differentiate such invariants.

Given $\Sigma = \langle S, \Omega, \Pi, E, V \rangle$, $\mathbf{Mod}(\Sigma)$ provides a category of models where a **model** over Σ is composed of a $\mathcal{FOPEQ}$-model paired with a relation R. For each event/variable name pair (e, v), R contains a $(e, v$-indexed) relation over the carrier set of the corresponding variable's sort. For example, given event name inc with integer variable x and boolean variable y where inc increments x by 1 and sets y to `false`, a model is:

$$\Big\langle A, R = \big\{ \{..., \langle 1, 2 \rangle, \langle 2, 3 \rangle, ...\}_{\text{inc,x}}, \{\langle \texttt{false}, \texttt{false} \rangle, \langle \texttt{true}, \texttt{false} \rangle\}_{\text{inc,y}} \big\} \Big\rangle$$

where A is a standard first-order model over $\langle S, \Omega, \Pi \rangle$ providing semantics for the sorts, operations and predicates in the usual way for first-order logic.

Intuitively, a model over Σ maps each pair (e, v), consisting of an event and variable name, to a relation over the sort carrier set of v, of which each tuple is a pair with the first element corresponding to a before value and the second an after value for v when used in event e. For simplicity we refer to the first element of the tuple as $b_{e,v}$ and the second as $a_{e,v}$, and curry these to get the corresponding variable-to-value mappings b_e and a_e.

Satisfaction for $\mathcal{EVT}$: The satisfaction relation for sentences over $\mathcal{EVT}$ is broken into two parts: satisfaction of invariant sentences and satisfaction of event sentences.

First, consider invariant sentences: Given a Σ-model $\langle A, R\rangle$ and some invariant sentence $\langle inv, \phi\rangle$ over $\mathcal{EVT}$, we define $\langle A, R\rangle \models_{\mathcal{EVT}} \langle inv, \phi\rangle$ if and only if for each event e we have

$$A \models_{\mathcal{FOPEQ}} \phi[\overline{x}/b_e] \wedge \phi[\overline{x}/a_e]$$

where $\phi[\overline{x}/a_e]$ (resp. $\phi[\overline{x}/b_e]$) denotes the evaluation of ϕ using the variable-to-value mapping given by a_e (resp. b_e).

Next, consider event sentences: Given a Σ-model $\langle A, R\rangle$ and some event sentence $\langle e, \phi\rangle$ over $\mathcal{EVT}$ we define $\langle A, R\rangle \models_{\mathcal{EVT}} \langle e, \phi\rangle$ if and only if

$$A \models_{\mathcal{FOPEQ}} \phi[\overline{x}/b_e][\iota^{-1}(\overline{x}')/a_e]$$

where ι^{-1} un-primes the variable names, and thus ϕ is evaluated using the variable-to-value mapping given by a_e and then b_e as above.

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

In our approach an Event-B specification, such as that for `mac1` in Figure 1, is represented as a presentation over $\mathcal{EVT}$. A presentation over a signature is a set of sentences, which are satisfied in a model if the conjunction of the sentences is satisfied in the usual way. Note that this implicitly merges different specifications for the same event name (models must satisfy all the formulae). This incorporates the standard semantics of the *extends* operator for events in Event-B where the extending event implicitly has all the parameters, guards and actions of the extended event but can have additional parameters, guards and actions [3].

An interesting aspect is that if a variable is not assigned to in an action then, in theory, they may be assigned any arbitrary value even though they have not been updated. This forces us to impose a kind of frame condition when indicating that no other variables may be changed so that they don't get inadvertently assigned to. This is a problem when combining specifications in the presentation. For example, to combine two events of the same name the sensible thing to do is combine their respective actions. But imposing the frame condition doesn't allow these actions be combined because no other variables are allowed to change and the addition of an action will violate this. Therefore, we assume that the default be that the frame condition is disabled thus facilitating the combination of specifications. This frame condition is somewhat analogous to the free operator that will be discussed later.

Another point is that we have not defined the models down to the detail of sequential execution of events, thereby not requiring that the `Initialisation` event happen first which is the norm in Event-B. We have left this to future work and it will be handled when devising an appropriate entailment system for $\mathcal{EVT}$ in Hets.

At first we considered that the relationship between $\mathcal{FOPEQ}$ and $\mathcal{EVT}$ be that of a duplex institution formed from a restricted version of $\mathcal{EVT}$ ($\mathcal{EVT}_{res}$)

```
spec TwoBools over FOPEQ
  Bool
  then
    ops
      i_go, u_go : Bool
    preds
      ¬ (i_go = true ∧ u_go = true)

spec mac1 over EVT
  (LightAbstract with σ1)
    and (LightAbstract with σ2)
  where
    σ1 = {i_go ↦ cars_go, u_go ↦ peds_go,
          set_go ↦ set_cars_go,
          set_stop ↦ set_cars_stop}
    σ2 = {i_go ↦ peds_go, u_go ↦ cars_go,
          set_go ↦ set_peds_go,
          set_stop ↦ set_peds_stop}

spec LightAbstract over EVT
  TwoBools
  then
    Initialisation
      begin
        act1 : i_go := false
      end
    Event set_go ≙
      when
        grd1: u_go = false
      then
        act1: i_go := true
      end
    Event set_stop ≙
      then
        act1: i_go := false
      end
```

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

and $\mathcal{FOPEQ}$ where $\mathcal{EVT}_{res}$ is the institution $\mathcal{EVT}$ but does not contain any $\mathcal{FOPEQ}$ signature items. This approach would allow us to constrain $\mathcal{EVT}_{res}$ by $\mathcal{FOPEQ}$ and thus facilitate the use of $\mathcal{FOPEQ}$ sentences in an elegant way. However, our intent is to implement $\mathcal{EVT}$ in Hets where duplex institutions are not supported [12]. For this reason we opt for a comorphism embedding the simpler institution $\mathcal{FOPEQ}$ into the more complex institution $\mathcal{EVT}$ [14].

The comorphism $\rho : \mathcal{FOPEQ} \to \mathcal{EVT}$ consists of the following: $\rho^{Sign} : Sig[\mathcal{FOPEQ}] \to Sig[\mathcal{EVT}]$ includes signatures and morphisms in $\mathcal{FOPEQ}$ into the category of $\mathcal{EVT}$ signatures by equipping them with empty sets of event and (sort-indexed) variable names. Each $\mathcal{FOPEQ}$ sentence is mapped to an invariant sentence in $\mathcal{EVT}$. For any $\mathcal{FOPEQ}$ signature Σ, $\rho^{Mod}_{\Sigma} : Mod_{\mathcal{EVT}}(\rho^{Sign}(\Sigma)) \to Mod_{\mathcal{FOPEQ}}$ is a functor mapping $\mathcal{EVT}$ models to $\mathcal{FOPEQ}$ models. It achieves this by forgetting about the relation part of the $\mathcal{EVT}$ model.

In order to ensure the institution $\mathcal{EVT}$ has good modularity properties, pushouts must exist and the institution must have the amalgamation property (all pushouts in **Sign** exist and every pushout diagram in **Sign** admits amalgamation) [14]. These amalgamation properties are also a prerequisite for good parameterisation behaviour (section 4.3) [13].

3.3 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 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 constrained to have different values.

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 25-34: 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.

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 [5, 8]. It is therefore essential that any formalisation of Event-B be capable of capturing these concepts.

In general for institutions, a refinement from an abstract specification A to some concrete specification C is defined as $|Mod|_C \subseteq |Mod|_A$ when $Sig[A] = Sig[C]$. This definition is credible, as new variable or event names cannot be added if the signatures stay the same. This provides only one option: strengthen the formulae in event definitions, which will result in at most the same number of models. This accounts for the first form of refinement in Event-B.

The second and third forms of refinement in Event-B cause the signatures to change because the set of event names E will get larger when adding or decomposing events. We address this using the specification building operator hide via σ which allows the definition of a signature morphism from the abstract to the concrete specification [14]. In this way we interpret the concrete specification over the same signature as the abstract specification and thus translate the models accordingly. This method of reasoning is supported by Schneider's work on a CSP semantics for Event-B [16].

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.

```
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 (LightRefined and (ButtonSpec with σ5)with σ4)

50 where
51   σ3 = {i_col ↦ cars_colour, u_col ↦ peds_colour,
52         set_green ↦ set_cars_green, set_red ↦ set_cars_red}
53   σ4 = {i_col ↦ peds_colour, u_col ↦ cars_colour,
54         set_green ↦ set_peds_green, set_red ↦ set_peds_red}
55   σ5 = {gobutton ↦ set_peds_green}
```

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

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 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 47-56: 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 48-49 is important.

```
1  constructor κ0 : Sig[Colours]⇒Sig[Bool]
2    sorts
3      Bool = Colours
4    ops
5      true = green
6      false = red

7  constructor κ1 : Sig[TwoColours]⇒Sig[TwoBool]
8    K0
9    ops
10     i_go = icol
11     u_go = ucol

12 constructor κ2 : Sig[mac2]⇒Sig[mac1]
13   ops
14     set_peds_go = set_peds_green
15     set_peds_stop = set_peds_red
16     set_cars_go = set_cars_green
17     set_cars_stop = set_cars_red

18 constructor κ3 : Sig[empty]⇒Sig[mac2]
19   mac2
20
```

Fig. 6: Constructors defining the modularised refinement relationship between the concrete and abstract presentations.

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_go`. 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.

4.2 Specification refinement with constructors

In the case when the signatures are different, which they are in our specifications, then A is refined to C when we have $\kappa(Mod[C]) \subseteq (Mod[A])$ for a suitably defined constructor $\kappa : Sig[C] \Rightarrow Sig[A]$. This restricts the concrete model to only contain elements of the abstract signatures via the appropriate mappings.

Figure 6 splits up the specification of the concrete machine `mac2` into constructors specifying each of the three separate forms of refinements:

Lines 1-6: κ0 defines the refinement of *Bool* into *Colours*, with an appropriate mapping for the values. Note that the lack of a mapping for the colour *orange* at this point implicitly hides it in the specification.

Lines 7-11: κ1 defines the refinement of the two boolean variables into their corresponding variables of type *Colour*. In combination with κ0, this corresponds to lines 8 and 10 of Fig. 3.

Lines 12-17: κ2 defines the refinement relation between the four events: this corresponds to the refines statements on lines 19, 28, 33 and 40 of Fig. 3.

Finally the constructor κ3 ties this together (lines 18-19), adding nothing else to the specification mac2. The specification empty over the initial signature $\Sigma_\varnothing$ has no axioms and one model `empty`. Intuitively, reaching Empty means that there are no more unresolved parts of the specification to be constructed. By including it here, we are signalling that the refinement is complete at this point. The full system refinement can be written as $(\kappa 2 + \kappa 1)(\kappa 3(\text{empty}))$, where $+$ takes the amalgamated union of the constructors κ1 and κ2.

4.3 An alternative construction using parameterisation

Parameterised specifications both take specifications as parameters and return specifications. These parameter and return specifications correspond to (user-defined) specification building operators. Parameterisation provides a more general way of combining specifications than that of the specification building operators providing λ-abstraction for user-defined abbreviations where variables in β-reduction now range over specifications [14].

For example, we could replace lines 8 and 9 of Figure 4 with:

spec LIGHTABSTRACT over $\mathcal{EVT}$ = λX : Spec(Sig[TWOBOOLS]) • X

thus giving a parameterised version of LIGHTABSTRACT that takes as parameter a specification over the signature of TWOBOOLS (which can be seen in Figure 4). It is now possible to define

spec LIGHTREFINED over $\mathcal{EVT}$ = LIGHTABSTRACT(TWOCOLOURS hide via σ) with σ

where σ is the morphism from Sig[TWOBOOLS] to Sig[TWOCOLOURS] such that $\sigma = \{Bool \mapsto Colours, i\_go \mapsto icol, u\_go \mapsto ucol\}$.

Here, TWOCOLOURS hide via σ takes a specification over TWOCOLOURS as its parameter and hiding maps the sorts and operations in TWOCOLOURS to those in TWOBOOLS giving a specification over TWOBOOLS. Applying with σ maps elements of Sig[TWOBOOLS] to elements of Sig[TWOCOLOURS] to produce the refined light specification.

5 Conclusion and Future Work

The Heterogeneous Tool-Set HETS provides a framework for heterogeneous specifications where each formalism is represented as a logic and understood in the theory of institutions. A logic is represented as an entailment system which consists of a category of signatures with corresponding morphisms, a set of sentences with corresponding translation maps and an entailment relation between sets of sentences and sentences for each signature [12]. Our logic for $\mathcal{EVT}$ utilises the already existing institution $\mathcal{CASL}$ [1] to account for the $\mathcal{FOPEQ}$ parts of the $\mathcal{EVT}$ institution thus taking advantage of the interoperability/heterogeneity supplied by HETS. $\mathcal{CASL}$ provides sorts and predicates like those written in Figure 4 lines 4-7. Our HETS implementation of $\mathcal{EVT}$ uses $\mathcal{CASL}$ formulae to represent the components of $\mathcal{EVT}$ sentences that require predicates[1].

Currently we can parse, statically analyse and combine specifications written over $\mathcal{EVT}$. Future work includes developing comorphisms to translate between $\mathcal{EVT}$ and other logics in HETS as well as integrating with the provers currently available in HETS (e.g. Isabelle). Comorphisms between these theorem provers and $\mathcal{EVT}$ will allow us to prove our specifications correct in HETS. We envisage that development should take place here to fully take advantage of the prospects for interoperability. A translation from Event-B to $\mathcal{EVT}$ in the future will enable us to fully utilise both tools.

[1] Supplementary information is available at: http://www.cs.nuim.ie/∼mfarrell/

References

1. Casl reference manual. In P. D. Mosses, editor, *The Complete Documentation of the Common Algebraic Specification Language*, volume 2960 of *LNCS*. 2004.
2. J.-R. Abrial. *The B-book: Assigning Programs to Meanings*. Cambridge University Press, New York, NY, USA, 1996.
3. 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.
4. A. Achouri and L. Jemni Ben Ayed. UML activity diagram to Event-B: A model transformation approach based on the institution theory. In *Information Reuse and Integration*, pages 823–829, Aug. 2014.
5. 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.
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 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. D. Lucanu, Y.-F. Li, and J. S. Dong. Institution morphisms for relating OWL and Z. In *Software Engineering and Knowledge Engineering*, pages 286–291, 2005.
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. S. Schneider, H. Treharne, and H. Wehrheim. A CSP approach to control in Event-B. In *Integrated formal methods*, volume 6396 of *LNCS*, pages 260–274, 2010.
16. S. Schneider, H. Treharne, and H. Wehrheim. The behavioural semantics of Event-B refinement. *Formal Aspects of Computing*, 26:251–280, 2014.
17. R. Silva, C. Pascal, T. S. Hoang, and M. Butler. Decomposition tool for event-b. *Software: Practice and Experience*, 41(2):199–208, 2011.
18. 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.