DOI 10.1007/s00165-014-0311-1
BCS © 2014
Formal Aspects of Computing (2015) 27: 499–523

Formal Aspects of Computing

Language and tool support for event refinement structures in Event-B

Asieh Salehi Fathabadi, Michael Butler and Abdolbaghi Rezazadeh

Electronics and Software Systems Group, School of Electronics and Computer Science, University of Southampton, Southampton SO17 1BJ, UK

Abstract. Event-B is a formal method for modelling and verifying the consistency of chains of model refinements. The event refinement structure (ERS) approach augments Event-B with a graphical notation which is capable of explicit representation of control flows and refinement relationships. In previous work, the ERS approach has been evaluated manually in the development of two large case studies, a multimedia protocol and a spacecraft sub-system. The evaluation results helped us to extend the ERS constructors, to develop a systematic definition of ERS, and to develop a tool supporting ERS. We propose the ERS language which systematically defines the semantics of the ERS graphical notation including the constructors. The ERS tool supports automatic construction of the Event-B models in terms of control flows and refinement relationships. In this paper we outline the systematic definition of ERS including the presentation of constructors, the tool that supports it and evaluate the contribution that ERS and its tool make. Also we present how the systematic definition of ERS and the corresponding tool can ensure a consistent encoding of the ERS diagrams in the Event-B models.

Keywords: Event refinement structure, Atomicity decomposition, Event-B, Formal method, Control flow, Refinement

1. Introduction

The Event-B formal method [Abr10] is an evolution of classical B [Abr96]. Event-B is proven to be applicable in a wide range of domains, including distributed algorithms, railway systems and electronic circuits. In the Event-B modelling language states of a system are defined by variables and state changes of a system are defined by guarded actions, called events. The main specification construct is a machine that is comprised of variables and events. Event-B supports refinement [Abr05] in which an abstract model is elaborated towards an implementation in a step-wise manner. During refinement steps a model can be modified and enriched.

One weakness of Event-B is that control flow between events is typically modelled implicitly. Since the Event-B language is a state-based language, ordering between several events can only be modelled in event guards which include conditions on state variables. Because Event-B is also used to model systems with rich control flow properties, it has been observed that explicit control flow specification is beneficial [But00, IIi09].

Correspondence and offprint requests to: A. Salehi Fathabadi, E-mail: asf08r@ecs.soton.ac.uk

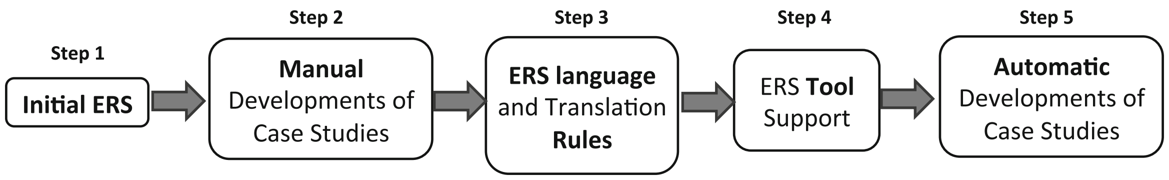

Step 1
Initial ERS
Step 2
Manual Developments of Case Studies
Step 3
ERS language and Translation Rules
Step 4
ERS Tool Support
Step 5
Automatic Developments of Case Studies

Fig. 1. Research road map

New events may be introduced in Event-B refinement and these are often used to decompose the atomicity of an abstract event into a series of steps. A second weakness of Event-B is that there is no explicit link between such new events that represent a step in the decomposition of atomicity and the abstract event to which they contribute. Although the refinement process in Event-B provides a flexible approach to modelling, it is unable to explicitly show the relationships between abstract events and new events introduced during a refinement step.

To address these weaknesses, the event refinement structure (ERS) [But09, SaB10, SRB11, SBR12] addresses the explicit control flow modelling and explicit refinement relationships representation.[1] It provides a graphical notation to structure the refinement process and to illustrate the explicit ordering between events of a model. The ERS graphical notation contains tree structured diagrams based on Jackson Structure Diagrams (JSD) [Jac83]. Semantics are given to an ERS diagram by defining a corresponding Event-B model from it.

The steps carried in the research are presented in Fig. 1. ERS was first introduced by Butler [But09] (step 1). It has been observed that methodological support for ERS, which is outlined in this paper, was weak. So we decided to evaluate and enhance the existing ERS from [But09]. For this reason we manually applied ERS to two sizeable case studies, a multimedia protocol [ZaC09] and a spacecraft system [ESA08] (step 2). The first case study, the multimedia protocol [ZaC09], contains requirements to establish, modify and close a media channel between two endpoints for transferring multimedia data. The second case study is based on a spacecraft system called BepiColombo [ESA08]. Developments of both these case studies involving manual translation of ERS diagrams to Event-B, have been published in [SaB10] and [SRB11] respectively. Insights gained from these case studies enabled us to define a formal description of the ERS language and formal translation rules from ERS diagrams to the Event-B language (step 3). Based on the ERS language and translation rule descriptions, we have developed the ERS tool support, as a plug-in for the Event-B tool-set (step 4). Our ERS tool provides an environment to construct the ERS diagrams and automatically translate them to Event-B models. Finally we re-developed the case study models using the provided ERS tool support (step 5).

The contribution of this paper is to present the full description of the ERS language and translation rules from ERS diagrams to the Event-B language, covering step of Fig. 1. We also outline the development of the ERS tool and the technologies that were used in this tool development (step 4). In order to present these outputs, we are using the automatically constructed models of the case studies (step 5). One of our objectives in this paper is to present the benefits of using ERS in formal modelling development; moreover the evaluation results assess how application of translation rules makes the automatic models of case studies more consistent and systematic, compared with the previous manual ones.

Earlier steps of this research have already been published. An early version of ERS was first introduced by Butler in [But09]; the manual applications of the initial ERS to the case studies have been published in [SaB10] and [SRB11]. Finally in [SBR12], a part of the ERS language, some of the translation rules and the tool support have been published. This paper is an extension of the later publication [SBR12]. In [SBR12] we only presents a part of the ERS language dealing with constructors (three out of seven) and corresponding translation rules that have been applied to the the case studies. In this paper, the full description of the ERS language, including the additional four constructors and corresponding translation rules, are presented. The presentation of translation rules is more detailed and precise, compared to the brief overview in [SBR12]. Also more evaluation results, including the proof obligation statistics, are presented and the related work is improved.

The paper is structured as follows: Sect. 2 outlines the Event-B formal method, the event refinement structure approach and an overview of the case studies requirements; Sect. 3 and Sect. 4 contain the ERS language description and definitions of translation rules respectively; Sect. 5 presents the tool developed to support ERS; In Sect. 6 we evaluate how the ERS language has helped us to enhance the Event-B formal development; finally Sect. 7 presents related work and conclusion.

[1] In [But09] ERS is referred to as event refinement diagram, and in [SaB10, SRB11, SBR12] ERS is referred to as the Atomicity Decomposition approach.

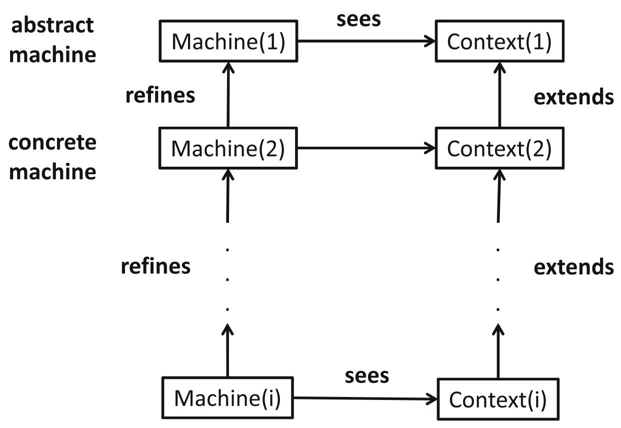


Fig. 2. Machine and context relationships in Event-B

2. Background and related work

2.1. Event-B

The Event-B formal method [Abr10, MAV05] has evolved from classical B [Abr96] and action systems [BaK88]. Event-B is used in modelling and verifying the consistency of chains of model refinements. The modelling language is based on set theory and first order logic.

A model in Event-B can consist of several *Contexts* and *Machines*. Contexts contain the static part (types and constants) of a model while machines contain the dynamic part (variables and events). Contexts provide axiomatic properties of an Event-B model, whereas machines provide behavioural properties of an Event-B model. A machine consists of variables, invariants, events. Invariants constrain variables, and are supposed to hold whenever variables are changed by an event. Each event is composed of a name, a set of guards $G(t, v)$ and some actions $S(t, v)$, where t are parameters of the event and v is state of the system which is defined by variables. All events are atomic and can be executed only when their guards hold. When the guards of several events hold at the same time, then only one of those events is chosen nondeterministically to be executed.

A context can be "extended" by other contexts and "referenced" by machines. A machine can be "refined" by other machines and can reference contexts. The relationships between various contexts and machines are illustrated in Fig. 2.

Building a model in Event-B usually starts with an abstract specification, and continues in successive refinement levels. The abstract model provides a simple view of the system, focusing on the main purposes of the system. Details are added gradually to the abstract model during refinement levels. In Event-B, refinement is used to introduce new functionality or add details of current functionality. One of the important features of Event-B refinement is the ability to introduce new events in a refinement level that have no corresponding abstract event. A new event refines an implicit *skip* event. A *skip* event is an empty event which does not modify any variable. From a given machine, *Machine1*, a new machine, *Machine2*, can be built as a refinement of *Machine1*. In this case, *Machine1* is called an abstraction of *Machine2*, and *Machine2* is said to be a refinement of *Machine1*. Event-B defines proof obligations to verify that events preserve invariants and that refinements are consistent. Also in guard strengthening proof obligations it should be proved that for refining events, the refining guards are stronger than abstract guards.

Event-B is supported by an Eclipse-based tool called Rodin [ABH06] that provides a modelling and proving environment. Rodin generates proof obligations for models and provides a range of automated and interactive provers [ABH06] as well as a model checker [Pro]. Rodin is an open platform, and is an extensible and adaptable modelling tool. We have taken advantage of the extensibility feature of Rodin to develop tool support for the ERS language.

```
Machine M0  sees C0

variables AbstractEvent
invariants
 @inv1 AbstractEvent ⊆ PAR_SET

event INITIALISATION then
    @act1 AbstractEvent ≔ ∅
end

event AbstractEvent any par  where
    @grd1 par ∉ AbstractEvent
   then
    @act1 AbstractEvent ≔ AbstractEvent ∪ {par}
end
```

```
machine M1 refines M0  sees C0

variables Event1 Event2
invariants
 @inv1 Event1 ⊆ PAR_SET
 @inv2 Event2 ⊆ Event1
 @inv3 Event2 = AbstractEvent

event INITIALISATION then
    @act1 Event1 ≔ ∅
    @act2 Event2 ≔ ∅
end

event Event1 any par  where
    @grd1 par ∉ Event1
   then
    @act1 Event1 ≔ Event1 ∪ {par}
end

event Event2 refines AbstractEvent
  any par where
    @grd1 par ∈ Event1
    @grd2 par ∉ Event2
  then
    @act1 Event2 ≔ Event2 ∪ {par}
end
```

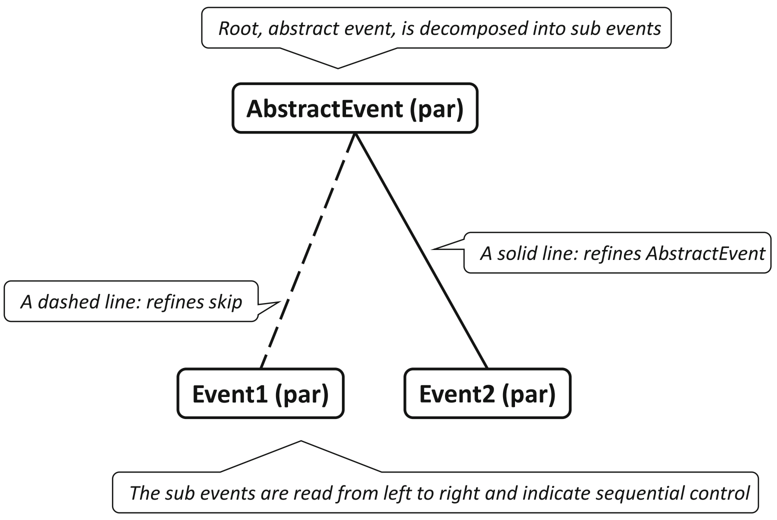

Root, abstract event, is decomposed into sub events
AbstractEvent (par)
A solid line: refines AbstractEvent
A dashed line: refines skip
Event1 (par)
Event2 (par)
The sub events are read from left to right and indicate sequential control

Fig. 3. Event refinement structure diagram

2.2. Event refinement structure

Although refinement in Event-B provides a flexible approach to modelling, it has the weakness that we cannot explicitly represent the relationships between abstract events and new events which are introduced in a refinement level. Another weakness of Event-B modelling is that we are not able to model the sequencing between events explicitly. The control flows are implicitly modelled in guards and actions of the events. Event refinement structure (ERS) approach is proposed to address these limitations. The idea is to augment Event-B refinement with a graphical notation that is capable of representing the explicit relationships between abstract and concrete events as well as explicit representation of the control flows. Figure 3 illustrates these two features of the ERS graphical notation.

Assume machine *M0*, on the left hand side of Fig. 3, is an abstract Event-B machine which contains the *Initialisation* event and the abstract specification of *AbstractEvent*. Machine *M1* refines machine *M0*. The machine *M1* encodes its control flow (ordering between *Event1* and *Event2*) via guards on the events. This control flow is made explicit in the ERS diagram presented in the right hand side. This diagram explicitly illustrates that the effect achieved by *AbstractEvent* in the abstract machine *M0*, is realized in the refining machine *M1*, by occurrence of *Event1* followed by *Event2*. The sequential ordering of the leaf events is from left to right (this is based on JSD diagrams [Jac83]). The solid line indicates that *Event2* refines *AbstractEvent* while the dashed line indicates that *Event1* is a new event which refines *skip*. In the Event-B model of machine *M1* on the left hand side, *Event1* does not have any explicit connection with *AbstractEvent*, but the diagram indicates that we break the atomicity of *AbstractEvent* into two sub-events in the refinement. One and only one of the children in an ERS diagram is connected to the root event with a solid line. Other leaves have to be connected with dashed lines. This restriction is a result of restrictions in the Event-B model. Since there can be only one occurrence of the abstract event in the refinement level, there is only one refining child (child with the solid line).

The parameter *par* in the diagram indicates that we are modelling multiple instances of *AbstractEvent* and its sub-events. Events associated with different values of *par* may be interleaved thus modelling interleaved execution of multiple processes. The execution effect of an event with parameter *par*, is to add the value of *par* to a control variable with the same name as the event, i.e., $par \in Event1$ means that *Event1* has occurred with value *par*. The use of a set means that the same event can occur multiple times with different values for *par*. The guard of an event with value *par* specifies that the event has not already occurred for value *par* but has occurred for the previous event, e.g., the guard of *Event2* says that *Event1* has occurred and *Event2* has not occurred for value *par*.

2.3. Overview of case studies

This section outlines an overview of our case studies, a multimedia protocol [ZaC09] and a subsystem of a spacecraft system based on BepiColombo [ESA08]. The case studies help to illustrate the translation rules in Sect. 4 and are used in the evaluation of the approach in Sect. 6.

2.3.1. Multimedia protocol

This case study specifies a protocol for establishing, modifying and closing a media channel. A media channel is established between two endpoints for transferring multimedia data. There are three phases in the protocol: establish, modify and close. In the modification phase some properties of the established channel can be modified, such as the codec used for data encoding.

It is worth comparing our approach to the multimedia protocol with the approach taken by Zave and Cheung [ZaC09]. Zave and Chueng present Promela models of the behaviour of each end of the protocol and use the Spin model checker to verify that these models satisfy certain safety and liveness properties. In our approach with Event-B, we start with a more global view of the protocol and then use ERS to arrive at models that have a level of details similar to the Promela models. Development of this case study, involving the application of ERS diagrams in the Event-B modelling, has been published in [SaB10].

2.3.2. Spacecraft system

Exploration of the planet Mercury is the main goal of the BepiColombo mission [ESA08]. One of the BepiColombo subsystems that handles communications between Earth and the satellite is taken for modelling. This subsystem consists of a core and four devices. The core and the control software are responsible for controlling the power of devices and their operation states and to handle TeleCommand (TC) and TeleMessage (TM) communications. In our work, we treat a part of the BepiColombo system related to the management of TC and TM communications. The core software (CSW) plays a management role over the devices. CSW is responsible for communication with Earth on one hand and with the devices on the other hand. Here is the summary of the system requirements:

- A TeleCommand (*TC*) is received by the core from Earth.
- The CSW checks the syntax of the received *TC*.
- Further semantic checking has to be carried out on the syntactically validated *TC*. If the *TC* contains a message for one of the devices, it has to be sent to the device for semantic checking, otherwise the semantic checking is carried out in the core.
- For each valid *TC* a control TeleMessage (*TM*) is generated and sent to Earth.

The development of the Event-B model of this case study includes an abstract level followed by three levels of refinement. Then the third refinement is decomposed to two sub-models, devices and core, and it is followed by two more refinement levels on the core sub-model. In all of the refinement steps, we applied ERS and also we evaluate the application of ERS and model decomposition together. Development of this case study has been published in [SRB11].

3. Event refinement structure language

In order to aid understanding of the ERS language, we provide an abstract example in Fig. 4, and a refinement example in Fig. 5. Considering the example in Fig. 4, in the ERS diagram of the most abstract level of an Event-B model, the name of a process in the system appears in an oval as the root node, and the names of the most abstract events appear in the leaves in order from left to right. All lines have to be dashed lines, since all of leaves are the

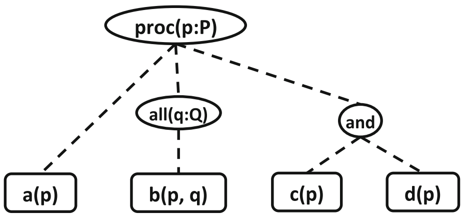


Fig. 4. An ERS diagram example, abstract level

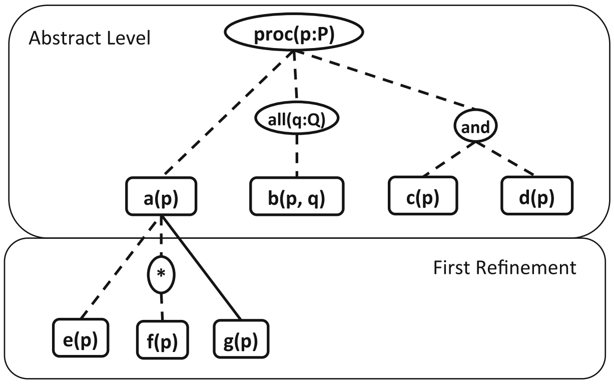


Fig. 5. An ERS diagram example, refinement level

most abstract events and do not refine the root node. In Fig. 4, for each $p \in P$, $proc(p)$ executes $a(p)$ followed by $b(p, q)$ for all $q \in Q$, followed by $c(p)$ and $d(p)$. Event a is further decomposed to three sub-events in the first refinement level, as shown in Fig. 5.

To describe the ERS language syntax, we use Augmented Backus–Naur Form (ABNF) [CrO08]. ABNF is a metalanguage based on Backus–Naur Form (BNF). BNF is a notation for context-free grammars, often used to describe the syntax of languages. It is applied wherever exact descriptions of language syntaxes are needed. The differences between standard BNF and ABNF involve naming rules, repetition, alternatives, order-independence, and value ranges. An ABNF specification is a set of derivation rules, written as

$rule = definition$

The following ABNF operators are used in describing ERS language:

- Terminal values:
  Terminal values are placed between two apostrophes (“Terminal”).
- Alternative: (Rule1 / Rule2)
  A rule may be defined by a list of alternative rules separated by a solidus (“/”).
- Variable repetition: (n*m element)
  To indicate repetition of an element the form (*n*m element*) is used. The optional *n* gives the minimum number of elements to be included with the default of 0. The optional *m* gives the maximum number of elements to be included with the default of infinity. We use *element for zero or more elements, 1*element for one or more elements and 2*element for two or more elements.

The ERS language syntax is presented in Fig. 6. The detailed definition of the ERS constructors are gradually outlined in the next sections. A flow, in Fig. 6, refers to a single root of an ERS diagram. To describe the type of a line (solid/dashed), we use a boolean property, called “ref”. When a sub-event refines the abstract event (solid line) , “ref” is one; otherwise “ref” is zero. Considering Fig. 6, the ABNF of ERS language may be described informally as follows:

- A flow consists of a name, zero or more parameters, and one or more children. Each child of a flow has a “*ref*” property.
- A parameter consists of a name and the type (index set) of the parameter.

```
flow          =  "flow" ("name", *par) ( 1*child (ref) )
par           =  "name" : "type"

child         =  "leaf" ("name") / constructor / 1* flow
cons-child    =  "leaf" ("name") / 1* flow

constructor   =  "loop" ( cons-child )
              /  ("and" / "or" / "xor")  ( 2* cons-child )
              /  ("all" / "some" / "one") (par) ( cons-child )
```

Fig. 6. Syntax of the ERS language

- A child is either a "*leaf*" with a name or a constructor or one or more flow(s), when it is further decomposed in the next refinement level.
- Constructors are divided to three groups:
  - Loop constructor: including "*loop*" with one constructor child (cons-child).
  - Logical constructors: including "*and*", "*or*" and "*xor*", with two or more constructor children (cons-child)
  - Replicator constructors: including "*all*", "*some*" and "*one*" with a parameter, followed by one constructor child (cons-child)
- A cons-child is either a "*leaf*" with a name or one or more flow(s), when it is further decomposed in the next refinement level.

  Following the ERS language syntax definition in Fig. 6, the textual syntax of the example in Fig. 4 is as bellow:

  $flow(proc, p : P)($
  $leaf(a)(0),$
  $all(q : Q)(leaf(b))(0),$
  $and(leaf(c), leaf(d))(0)$
  $)$

  In the textual syntax of the example in Fig. 5, as shown below, *leaf(a)* is replaced by a flow:

  $flow(proc, p : P, 1)($
  flow(a, p:P)(leaf(e)(0), loop(leaf(f))(0), leaf(g)(1))(0),
  $all(q : Q)(leaf(b))(0),$
  $and(leaf(c), leaf(d))(0)$
  $)$

Figure 3 briefly represented how a simple ERS diagram is translated to an Event-B model. The next section outlines the translation rules for all of the ERS constructors presented in this section.

4. Translation rules

Formal semantics are given to an ERS diagram by transforming it into an Event-B model, based on a collection of translation rules. In this section, we discuss these translation rules. The main syntactic elements of an Event-B machine are variables, invariants, guards and actions. The encoding of ERS diagrams in Event-B uses a collection of Event-B syntactic patterns such as typing invariants, sequencing invariants, partitioning invariants, disabling guards, sequencing guards and leaf actions. Our translation scheme defines a separate rule for each of these syntactic patterns. Figure 7 outlines the full list of translation rules. Each translation rule defines a transformation from an ERS source element to an Event-B destination element. Note that for each ERS element usually there is more than one applicable translation rule. We explain the role of each translation rule using snippets taken from the case studies. We first explain the rules related to sequencing of events. After outlining the sequencing rules, we discuss the rules for a solid leaf, the *loop*, logical constructors and replicator constructors.

TR1:	leaf	⟶	leaf variable	**TR10:**	loop	⟶	*loop* guard
TR2:	first leaf	⟶	typing invariant	**TR11:**	loop	⟶	next event guard
TR3:	non-first leaf	⟶	sequencing invariant	**TR12:**	loop	⟶	resetting event
TR4:	dashed leaf	⟶	non-refining event	**TR13:**	solid xor	⟶	gluing invariant
TR5:	leaf	⟶	disabling guard	**TR14:**	dashed xor	⟶	partition invariant
TR6:	non-first leaf	⟶	sequencing guard	**TR15:**	xor	⟶	*xor* guard
TR7:	leaf	⟶	*leaf* action	**TR16:**	replicator leaf	⟶	typing invariant
TR8:	solid leaf	⟶	gluing invariant	**TR17:**	one	⟶	cardinality invariant
TR9:	solid leaf	⟶	refining event	**TR18:**	one	⟶	*one* guard
				TR19:	solid one	⟶	gluing invariant

Fig. 7. Translation rules

4.1. Sequencing rules

As discussed in Sect. 2.2, one major feature of ERS diagrams is the explicit representation of sequencing between events. To illustrate this concept, we have taken a part of the most abstract level diagrams of the BepiColombo system, presented in the upper level of Fig. 8. In the most abstract diagram, the name of the system appears in an oval as the root node, and the names of the most abstract events appear in the leaves ordered from left to right. This diagram specifies an event ordering whereby a *TC* is received by the core, *ReceiveTC* event, and then it is validated by *TC_Validation_Ok* event.

The arrows in Fig. 8 indicate the application of translation rules. For example, the TR1 arrow from the *ReceiveTC* leaf in the diagram to the *ReceiveTC* variable in the Event-B model shows that the application of the TR1 rule to each source leaf produces a variable, with the same name as the leaf, in the Event-B model. The constructed variables are of type set and are used to control the sequencing of the leaf events. We refer to the values in a leaf variable as tokens. So for a parameterised event, the leaf variable added by TR1 is the token set.

Application of TR2 to the first leaf produces an invariant which defines the type of the leaf variable. Application of TR3 to the second leaf produces an invariant which describes the sequencing constraint between two leaf events. The sequencing invariant describes the second leaf variable as a subset of the previous leaf variable, since the second leaf event is allowed to execute only after execution of its previous leaf event.

In the most abstract diagram, since all leaves represent the most abstract events, there is no solid line. For each leaf with a dashed line, TR4 constructs a non-refining event. The parameter of the leaf is transformed to the event parameter. For each leaf, TR5 constructs a disabling guard, which specifies that the leaf event has not executed for a token before. In other words, a token indicates that the event occurred with that value. For each non-first event, like *TC_Validation_Ok* here, another guard is required to make sure that the previous event has been executed for the token before; this translation is carried out via TR6. Finally TR7 defines an action that adds a token to the leaf variable and as a result disables the corresponding leaf event for that token.

The translation rules TR1–TR7 are applicable to leaf nodes and encode sequencing collectively.

4.2. Solid line

In Fig. 9, the abstract atomic *TC_Validation_Ok* event from Fig. 8, is decomposed to three sub-events in a refinement level. Validating a received *TC* is performed in two steps, checking the syntax, in the *TCCheck_Ok* event, and the semantics, in the *TCExecute_Ok* event, of a received *TC*. After syntax and semantic checks, in the third step, *TCExecOk_ReplyCtrlTM*, a control *TM* is produced and sent back to Earth.

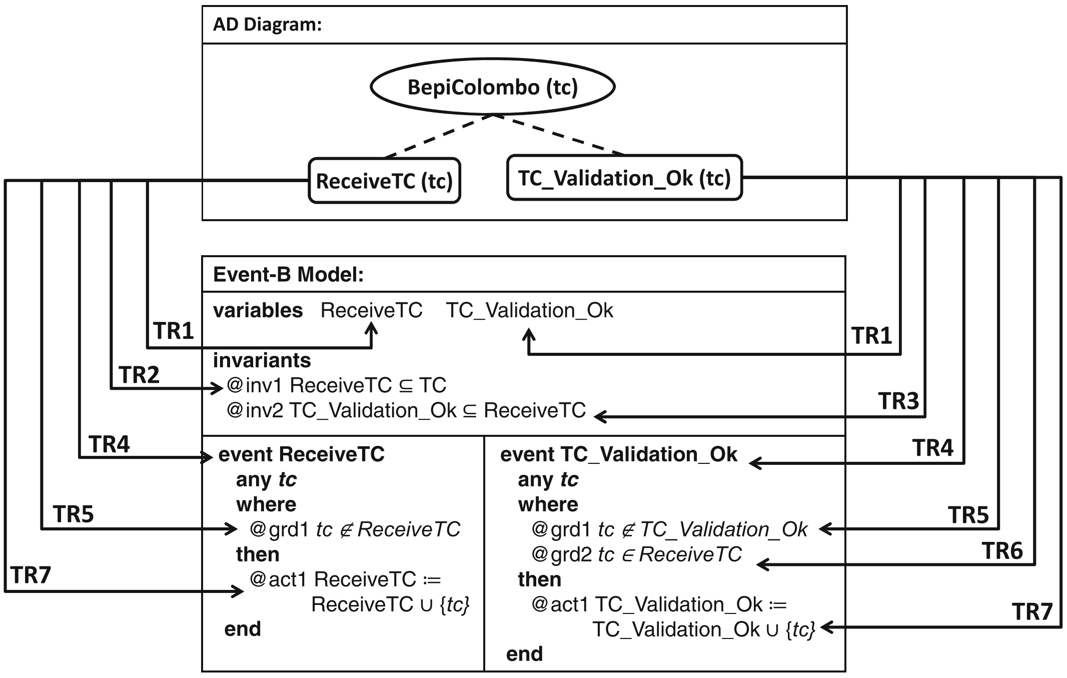


Fig. 8. The most abstract level model, BepiColombo system

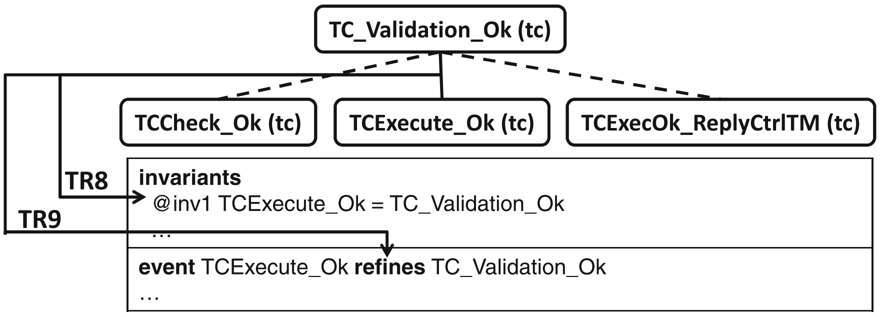

TC_Validation_Ok (tc)
TCCheck_Ok (tc)
TCExecute_Ok (tc)
TCExecOk_ReplyCtrlTM (tc)
invariants
@inv1 TCExecute_Ok = TC_Validation_Ok
…
event TCExecute_Ok refines TC_Validation_Ok
…
TR8
TR9

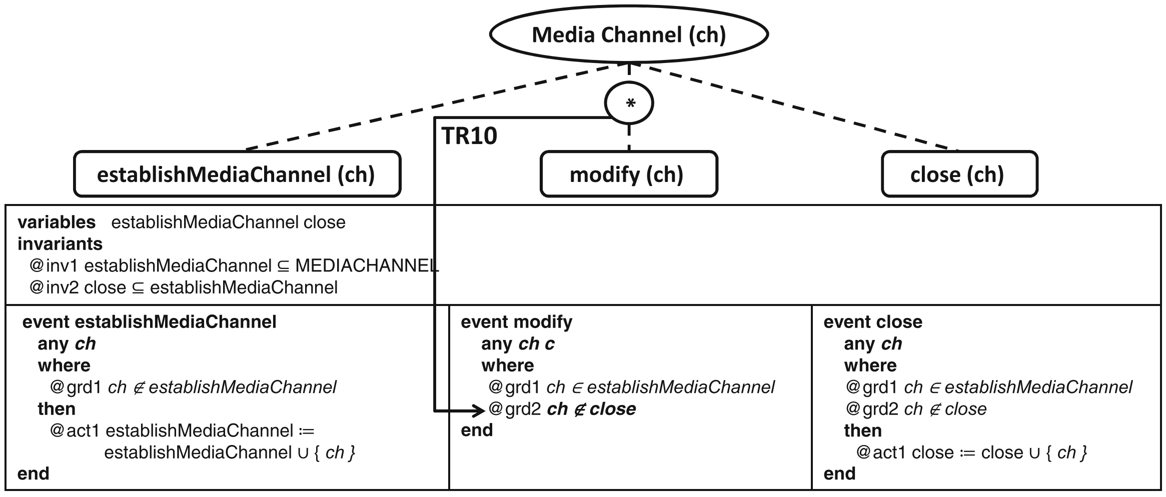
Fig. 9. The ERS diagram of *TC_Validation_Ok*, BepiColombo system

Considering Fig. 9, a solid line in an ERS diagram has two effects. First, it is translated to an invariant which connects the abstract variable to the refinement variable (TR8); this is called a gluing invariant. Second, it is translated to an event which refines the abstract event in the root node (TR9).

4.3. Loop constructor

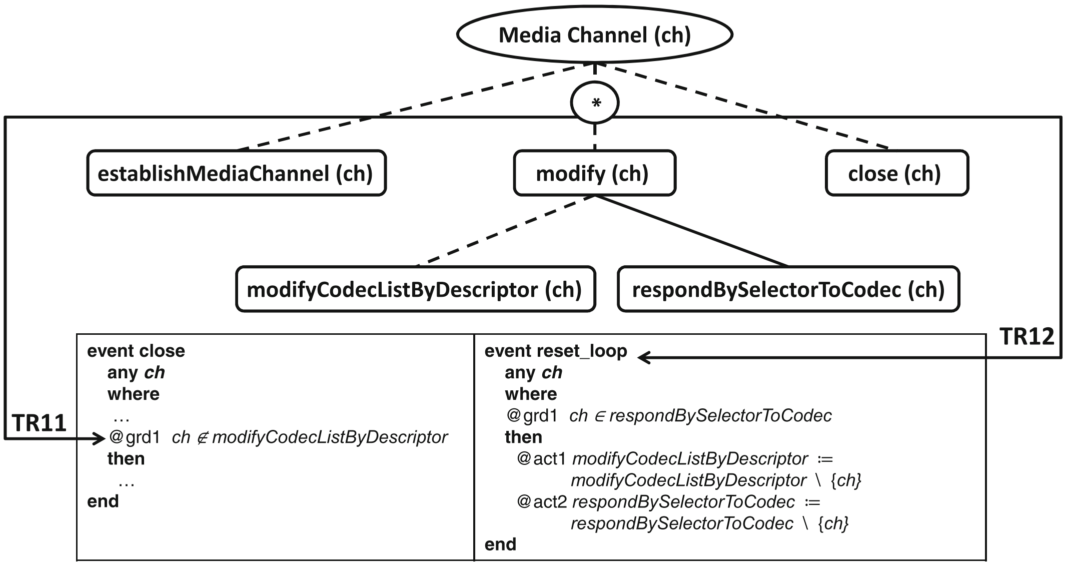
The loop constructor is used to model zero or more executions of a leaf. Figure 10 presents the most abstract ERS diagram of the multimedia protocol which contains the loop constructor as its second child. The diagram states that first a media channel is established, then it can be modified zero or more times and finally it is closed. Considering Fig. 10, there is no variable constructed for the loop leaf (*modify*), since we do not need to record the loop event execution. The event following the loop event (*close*) can execute immediately after execution of the event preceding the loop event (*establishMediaChannel*).

The loop event can execute several times *before* execution of the event that follows it. In Fig. 10, TR10 transforms the loop constructor to a guard in the loop event, *modify*. This guard checks that the event after the loop, *close*, has not already executed for the channel. The other Event-B elements in Fig. 10 are constructed via TR1-TR7 which have been described previously. The *modify* event has no action as we do not need to track its occurrence in order to constrain the event that follows it (*close*), nor does an occurrence of *modify* constrain



```
variables  establishMediaChannel close
invariants
  @inv1 establishMediaChannel ⊆ MEDIACHANNEL
  @inv2 close ⊆ establishMediaChannel

event establishMediaChannel
  any ch
  where
    @grd1 ch ∉ establishMediaChannel
  then
    @act1 establishMediaChannel ≔
          establishMediaChannel ∪ { ch }
end

event modify
  any ch c
  where
    @grd1 ch ∈ establishMediaChannel
    @grd2 ch ∉ close
end

event close
  any ch
  where
    @grd1 ch ∈ establishMediaChannel
    @grd2 ch ∉ close
  then
    @act1 close ≔ close ∪ { ch }
end
```

Fig. 10. The most abstract level, multimedia protocol


Media Channel (ch)
*
establishMediaChannel (ch)
modify (ch)
close (ch)
modifyCodecListByDescriptor (ch)
respondBySelectorToCodec (ch)
TR11
TR12

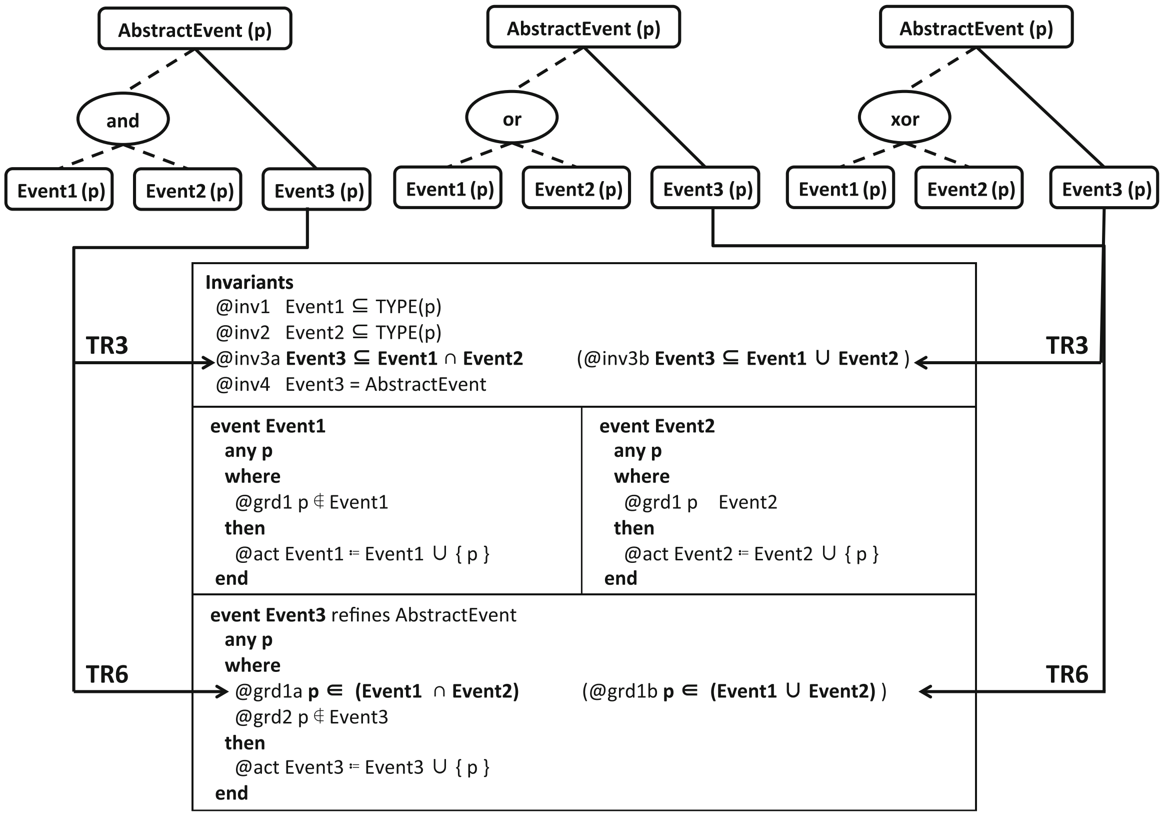
```
event close
  any ch
  where
   …
   @grd1  ch ∉ modifyCodecListByDescriptor
  then
   …
end

event reset_loop
  any ch
  where
  @grd1  ch ∈ respondBySelectorToCodec
  then
   @act1 modifyCodecListByDescriptor ≔
           modifyCodecListByDescriptor \ {ch}
   @act2 respondBySelectorToCodec ≔
           respondBySelectorToCodec \ {ch}
end
```

Fig. 11. Refinement level, multimedia protocol

subsequent occurrences. The *grd1* of *close* event (constructed by TR6) is based on *establishMediaChannel* variable, because *close* event can execute immediately after execution of *establishMediaChannel* event, with zero execution of the loop event.

In the next refinement level, as illustrated in Fig. 11, the abstract atomic *modify* event, is decomposed to two sub-events. First the codec list is modified by sending a descriptor signal from one of the endpoints of the established media channel to another one (*modifyCodecListByDescriptor* event). Then the other endpoint has to respond to the descriptor signal by selecting a codec from the received codec list and sending back a selector signal containing the selected codec (*respondBySelectorToCodec* event).

When a loop leaf, such as *modify*, is further decomposed to some sub-events, two extra translation rules are needed. First it is required that the event after the loop, *close* here, is not allowed to execute in the middle of execution of the loop events, e.g. in Fig. 11, *close* cannot occur in between *modifyCodecListByDescriptor* and *respondBySelectorToCodec*. Considering Fig. 11, TR11 ensures this property by constructing a guard which is added to the next event, *close*. Second we need a reset event. In order to enable more than one execution of the loop events, the loop control variables need to be reset. This is performed as the result of TR12 in Fig. 11. The



```
Invariants
 @inv1   Event1 ⊆ TYPE(p)
 @inv2   Event2 ⊆ TYPE(p)
 @inv3a Event3 ⊆ Event1 ∩ Event2          (@inv3b Event3 ⊆ Event1 ∪ Event2 )
 @inv4   Event3 = AbstractEvent

event Event1                           event Event2
  any p                                  any p
  where                                  where
   @grd1 p ∉ Event1                       @grd1 p    Event2
  then                                   then
   @act Event1 ≔ Event1 ∪ { p }           @act Event2 ≔ Event2 ∪ { p }
 end                                    end

event Event3 refines AbstractEvent
  any p
  where
   @grd1a p ∈ (Event1 ∩ Event2)          (@grd1b p ∈ (Event1 ∪ Event2) )
   @grd2 p ∉ Event3
  then
   @act Event3 ≔ Event3 ∪ { p }
 end
```

Fig. 12. Logical constructors

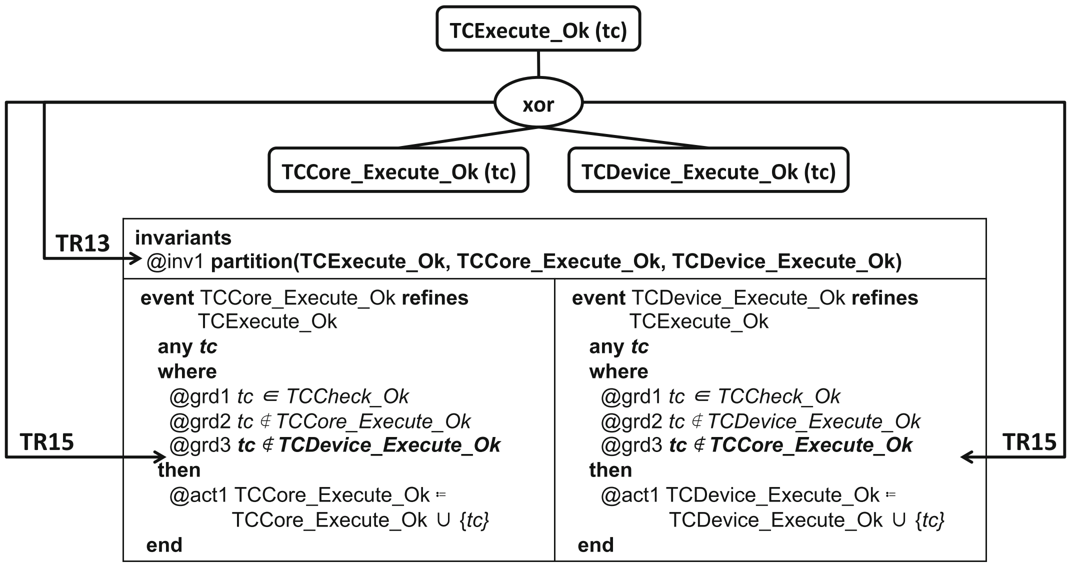
reset_loop event removes a channel token from the control variables of the loop events, which allows them to execute for the same channel again. A reset event is not required in Fig. 10 where the loop leaf is not decomposed further, since there is no control variable constructed for the loop leaf.

4.4. Logical constructor

The ERS logical constructors are presented in Fig. 12. The diagrammatic notation for these is represented in the upper level of the figure using three simple patterns of the ERS diagrams. From left to right, the *and* constructor specifies the requirement to execute all of its sub-events; The *or* constructor specifies the requirement to execute one or more of its sub-events; Finally the *xor* constructor specifies the requirement to execute exactly one of its sub-events.

The encoded Event-B model of the logical constructors is presented in the lower level of Fig. 12. The Event-B models of the logical constructors are similar together and are constructed as the results of the translation rules that have been discussed in the previous sections. There are two differences in the model of the *and* constructor and *or*/*xor* constructors which are highlighted in Fig. 12 and are discussed in this section. First *inv3a* corresponds to the *and* constructor, while *inv3b* corresponds to the *or*/*xor* constructors. Second *grd1a* in *Event3* corresponds to the *and* constructor, while *grd1b* corresponds to the *or*/*xor* constructors.

Considering the Event-B model in Fig. 12, as the result of TR2 (Sect. 4.1), the type of the constructors' sub-events are defined in *inv1* and *inv2*. In the case of the *and* constructor, *inv3a* specifies that *Event3* can execute only after execution of both *and* sub-events. This invariant is constructed as the result of TR3. In the case of the *or*/*xor* constructors, *inv3b* specifies that *Event3* can be executed after execution of one or more of the constructors' sub-events. *inv4*, constructed as the result of TR8 (Sect. 4.2), specifies the connection between the abstract variable and the corresponding concrete variable. The events, their guards and actions are constructed as explained before. Here we highlight the constructed guard (from TR6) in the event after the logical constructors, *Event3*. In the case of the *and* pattern, *grd1a* in *Event3* ensures that *Event3* can execute only for a parameter which all of the *and*



```
invariants
  @inv1 partition(TCExecute_Ok, TCCore_Execute_Ok, TCDevice_Execute_Ok)

event TCCore_Execute_Ok refines
        TCExecute_Ok
  any tc
  where
    @grd1 tc ∈ TCCheck_Ok
    @grd2 tc ∉ TCCore_Execute_Ok
    @grd3 tc ∉ TCDevice_Execute_Ok
  then
    @act1 TCCore_Execute_Ok ≔
            TCCore_Execute_Ok ∪ {tc}
end

event TCDevice_Execute_Ok refines
        TCExecute_Ok
  any tc
  where
    @grd1 tc ∈ TCCheck_Ok
    @grd2 tc ∉ TCDevice_Execute_Ok
    @grd3 tc ∉ TCCore_Execute_Ok
  then
    @act1 TCDevice_Execute_Ok ≔
            TCDevice_Execute_Ok ∪ {tc}
end
```

Fig. 13. The ERS diagram of *TC_Execute_Ok*, BepiColombo system

sub-events has executed before. In the case of the *or* and *xor* patterns, *grd1b* ensures that *Event3* can execute only for a parameter which at least one of the *or*/*xor* sub-events has executed before. In the case of the *xor* pattern an invariant and a guard in each of the *xor* sub-events, are needed to ensures the exclusiveness property of the *xor* sub-events; these invariant and guards are discussed in the next section.

4.5. *xor* constructor

Exclusive choice between two or more events is introduced to the ERS diagrams with a constructor called *xor*. An application of the *xor* constructor in BepiColombo development is presented in Fig. 13. A *TC* either belongs to the core or the devices and not both of them. The figure illustrates a further level of refinement where the atomicity of the semantics checking event, *TCExecute_Ok*, is decomposed to an exclusive choice between two sub-events; *TCCoreExecute_Ok* checks the semantics of a *TC* which belongs to the core and *TCDeviceExecute_Ok* checks the semantics of a *TC* which belongs to the device. Exclusive choice means the system executes *TCCoreExecute_Ok* or *TCDeviceExecute_Ok* but not both.

xor sub-events inherit the type of their line (solid/dashed) from the *xor* constructor. Considering Fig. 13, the *xor* constructor is connected to the root node with a solid line, therefore both *xor* sub-leaves are connected with solid lines and refine the abstract event in the root node.

There are two translation rules for the *xor* constructor in Fig. 13. First the *xor* constructor is transformed to the partitioning invariant (TR13), which ensures exclusivity of execution. The $partition$ operator in Event-B is defined as follows:

$$partition(E, E_1, ..., E_n) \equiv (E = E_1 \cup ... \cup E_n) \wedge (i \neq j \Rightarrow E_i \cap E_j = \emptyset)$$

This states that E is partitioned into n sets, E_1 to E_n.

The constructed partitioning invariant first describes the relationship between the abstract variable and the refinement variables:

$$TCExecute\_Ok = TCCoreExecute\_Ok \cup TCDeviceExecute\_Ok.$$

Second it describes the mutually exclusive property of the *xor* sub-events:

$$TCCoreExecute\_Ok \cap TCDeviceExecute\_Ok = \emptyset.$$

If the *xor* constructor is connected to the root node with a dashed line, there is no relationship between the abstract variable and the refinement variables; Therefore in the case of dashed line there is no corresponding abstract event and the partitioning invariant becomes

$$partition((TCCoreExecute\_Ok \cup TCDeviceExecute\_Ok),\\ TCCoreExecute\_Ok, TCDeviceExecute\_Ok).$$

This is constructed as the result of TR14. In the case of a dashed lined *xor* constructor with just two sub-events, the partitioning invariant can be simplified to specify the mutual exclusive property of the two *xor* sub-events. For example the latest invariant can be simplified as:

$TCCoreExecute\_Ok \cap TCDeviceExecute\_Ok = \emptyset.$

However when the number of *xor* sub-events exceeds two, the partitioning invariant is preferable due to the growing number of mutually exclusive conditions. Therefore, due to the coherence issue, TR14 always construct the partitioning invariant even for a dashed lined *xor* constructor with just two sub-events.

The second translation rule in Fig. 13 (TR15) constructs a guard for each *xor* sub-event. This guard enforces the exclusiveness property of *xor* sub-events. The guard in each *xor* sub-event checks that the other *xor* sub-events have not occurred for the intended value of the *TC*.

4.6. Replicator constructor

There are three replicator constructors, *all*, *some* and *one*, each of which adds a new parameter to its single sub-event. Figure 14 illustrates these constructors using three simple patterns of the ERS diagrams. From left to right, the *all* constructor specifies execution of its sub-event for all instance values of its new parameter, *q*; The *some* constructor specifies execution of its sub-event for some instance values of its new parameter; Finally the *one* constructor specifies execution of its sub-event for exactly one instance value of its new parameter. The *all*, *some* and *one* replicators are generalisations of the *and*, *or* and *xor* logical constructor respectively.

The encoded Event-B model of the replicators is presented in the lower level of Fig. 14. There is a new translation rule (TR16); the rest of the model is constructed as a result of the previously defined translation rules. The replicators add a new parameter, the *q* parameter, to their sub-events, *Event1*. TR16 constructs the type of the replicator leaf variable, *Event1*.

The Event-B models of the replicators are similar together. There are two differences in the model of the *all* replicator and *some/one* replicators which are highlighted in Fig. 14 and are discussed in this section. First *inv2a*, which is constructed as the result of TR3, specifies the sequencing between the *all* sub-event (*Event1*) and the following event (*Event2*). *inv2a* specifies the execution of *Event1* for all instances of the parameter *q* before execution of the *Event2*. This invariant uses relational image, $r[S]$, which in Event-B is defined as follows:

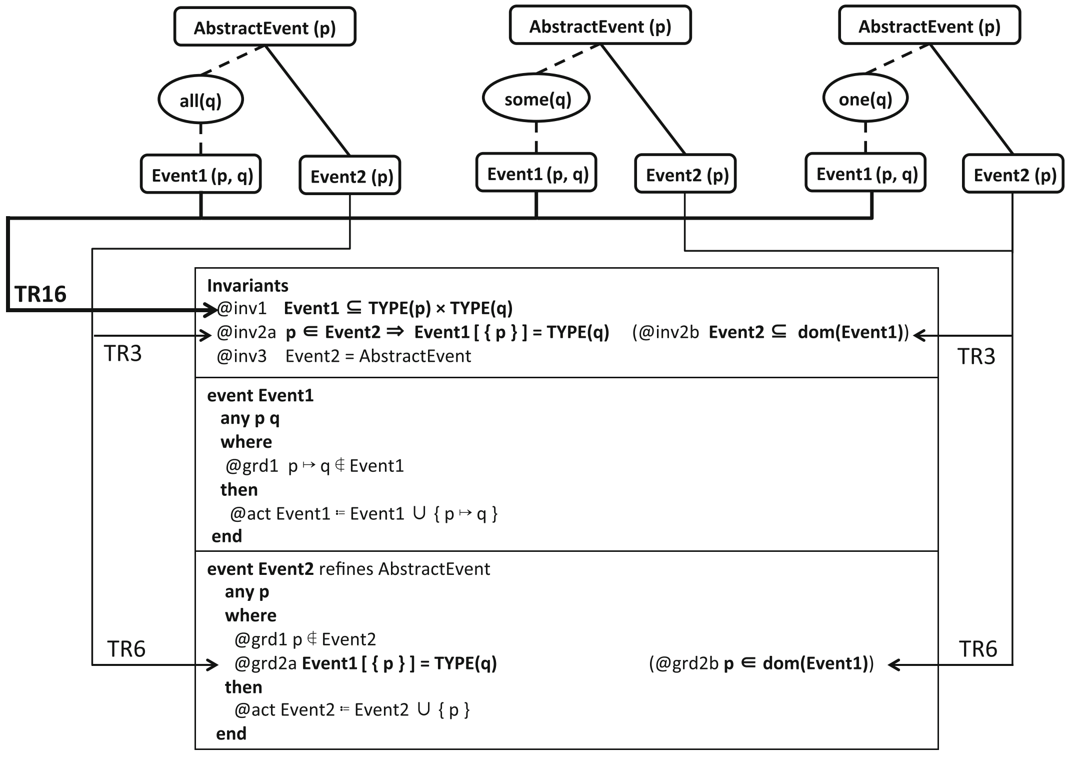
$r[S] = \{y \mid \exists x . x \in S \wedge x \mapsto y \in r\}$

inv2a specifies that *Event2* only can execute for an instance of the parameter *p* for which *Event1* has already executed for all instances of the *all* parameter, *q*. In the case of the *some* and *one* patterns, *inv2b* specifies that *Event2*, which is a subset of *TYPE(p)*, is a subset of the first dimension of the *Event1*. Secondly in a similar way, *grd2a* and *grd2b*, constructed as the result of TR6, enforce the sequencing between sub-events in the case of *all* and *some/one* respectively. In the case of the *one* constructor, an invariant and a guard in the *one* sub-event, are needed to ensures the single execution of the *one* sub-event; these invariant and guards are discussed in the next section.

4.7. *one* constructor

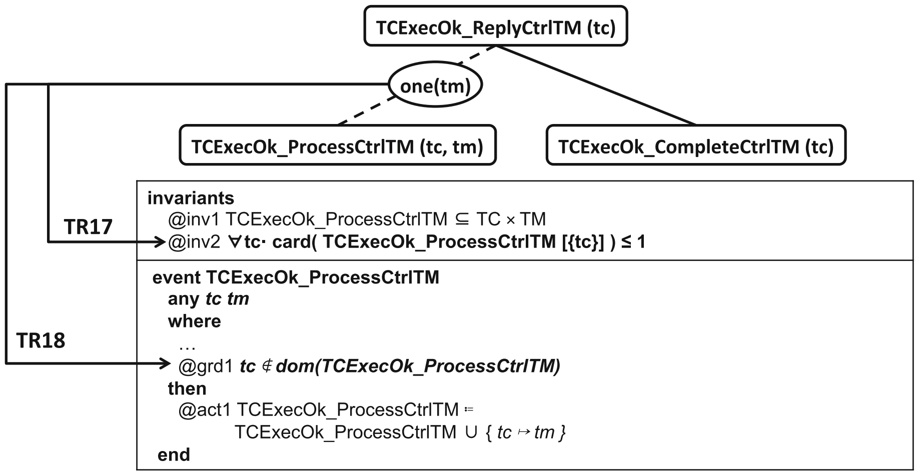
The *one* constructor is a generalisation of the *xor* constructor and specifies execution of an event for exactly one instance value of a new parameter. An application of the *one* constructor in BepiColombo development is presented in Fig. 15. Figure 15 illustrates that the *TCExecOk_ReplyCtrlTM* event is decomposed to produce exactly one *TM*, in the *TCExecOk_ProcessCtrlTM* event, followed by the completion action, *TCExecOk_CompleteCtrlTM*.

As presented in Fig. 15, the *one* constructor adds a new parameter, the *tm* parameter, to its sub-event, *TCExecOk_ProcessCtrlTM*. *inv1* (constructed by TR16) specifies the type of the *one* leaf variable, *TCExecOk_ProcessCtrlTM*. For each validated *tc*, exactly one control *tm* should be processed. To enforce this constraint, the *one* constructor is translated to an invariant and a guard. TR17 constructs an invariant which defines the *one* constructor property, specifying that for each *tc*, the cardinality of the set of processed *tm*s is at most one. TR18 constructs a guard to make sure that the *one* sub-event has not executed for the same value of the intended *tc* before.



```
Invariants
 @inv1   Event1 ⊆ TYPE(p) × TYPE(q)
 @inv2a  p ∈ Event2 ⇒ Event1 [ { p } ] = TYPE(q)    (@inv2b Event2 ⊆ dom(Event1))
 @inv3   Event2 = AbstractEvent

event Event1
  any p q
  where
   @grd1  p ↦ q ∉ Event1
  then
   @act Event1 ≔ Event1 ∪ { p ↦ q }
 end

event Event2 refines AbstractEvent
  any p
  where
   @grd1 p ∉ Event2
   @grd2a Event1 [ { p } ] = TYPE(q)          (@grd2b p ∈ dom(Event1))
  then
   @act Event2 ≔ Event2 ∪ { p }
 end
```

Fig. 14. Replicator constructors


TCExecOk_ReplyCtrlTM (tc)
one(tm)
TCExecOk_ProcessCtrlTM (tc, tm)
TCExecOk_CompleteCtrlTM (tc)
TR17
TR18

```
invariants
  @inv1 TCExecOk_ProcessCtrlTM ⊆ TC × TM
  @inv2 ∀tc· card( TCExecOk_ProcessCtrlTM [{tc}] ) ≤ 1

event TCExecOk_ProcessCtrlTM
  any tc tm
  where
   …
   @grd1 tc ∉ dom(TCExecOk_ProcessCtrlTM)
  then
   @act1 TCExecOk_ProcessCtrlTM ≔
           TCExecOk_ProcessCtrlTM ∪ { tc ↦ tm }
 end
```

Fig. 15. The ERS diagram of *TCExecOk_ReplyCtrlTM*, BepiColombo system

4.8. The formal description of the translation rules

In the previous sections we demonstrated a graphical representation of the application of the translation rules in the case studies. For each translation rule, the source element was presented as a graphical ERS element, and the constructed element was presented as an instantiated Event-B element. The graphical representation of the ERS approach is used by end users and can be specified in a more general manner as the textual ABNF representation of the ERS approach (presented in Sect. 3). This section describes how translation rules are defined precisely in a formal textual manner. Here translation rules are presented based on ABNF representation of the ERS language.

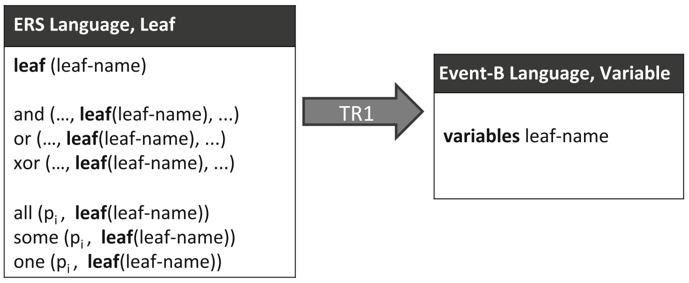

ERS Language, Leaf
leaf (leaf-name)
and (..., leaf(leaf-name), ...)
or (..., leaf(leaf-name), ...)
xor (..., leaf(leaf-name), ...)
all (p_i , leaf(leaf-name))
some (p_i , leaf(leaf-name))
one (p_i , leaf(leaf-name))
TR1
Event-B Language, Variable
variables leaf-name

Fig. 16. TR1 definition

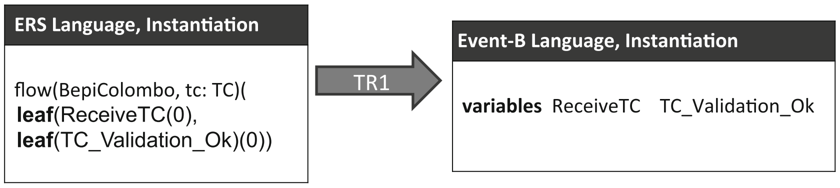

ERS Language, Instantiation
flow(BepiColombo, tc: TC)(
leaf(ReceiveTC(0),
leaf(TC_Validation_Ok)(0))
TR1
Event-B Language, Instantiation
variables ReceiveTC TC_Validation_Ok

Fig. 17. TR1 instantiation for the diagram of the Fig. 8

The source element is an ABNF element of the ERS language, and the constructed element is an Event-B element of the Event-B language.

As summarised in the Fig. 7, the source elements of the translation rules (in the right side of each translation rule arrow) are type of a leaf or the loop constructor or the *xor* constructor or the *one* constructor. These elements are translated into the Event-B elements such as variables, invariants, events, guards and actions. A leaf is transformed to a variable, an invariant, an event, guard(s) and an action in order to manage the sequencing between events and to specify the relationship between the abstract event and the refining sub-event. The loop constructor is transformed to guards and a resetting event to control the loop execution. The xor-constructor is transformed to an invariant and guards to specify the mutual exclusive property of its children. The one-replicator is transformed to an invariant and a guard to limit the number of executions of its child to one.

We explain the textual description of three translation rules, with different target Event-B element, here. The rest of the rules are working in a very similar manner.[2]

First, the formal description of TR1 (introduced in Fig. 8) is presented in Fig. 16. The left-hand box contains the ABNF representation of ERS elements that is transformed to the right-hand box containing the description of the Event-B element. In the case of TR1, the right-hand side contains the representation of seven different types of a leaf in the ERS language: a simple leaf or a leaf of six different ERS constructors. Each of these leaf element is transformed to a variable (with the same name as the leaf) in the Event-B language. The leaf of a *loop* constructor is not appeared in the source element of this rule, since as described in Sect. 4.3, no variable is constructed for a *loop* leaf.

Figure 17 exemplifies the application of TR1 in the BepiColombo system. This figure illustrates the instantiation of the *TR1* in Fig. 16, for the BepiColombo diagram presented in Fig. 8. Based on the ERS language description in Fig. 6, the ABNF representation of the diagram in Fig. 8 is shown in the left-hand box and the result of applying TR1 is shown as the Event-B elements in the right-hand box.

For the second case, we present *TR9* rule (introduced in Fig. 9) in Fig. 18. This rule is applicable to a sub-event that is connected to its parent with a solid line (represented as $ref = 1$ in the ERS language). Such sub-event is translated to a refining event in the Event-B model. This leaf can be a simple leaf, or a leaf of a refining *xor* constructor, or a refining *one* replicator. Recalling Sect. 2.2, only one occurrence of the abstract event is allowed in the refinement level. Considering definition of the constructors in the previous sections, in the case of *xor* constructor and *one* replicator, only one execution of their sub-events is allowed, but in the case of other constructors, more than one execution is allowed; therefore the other constructors are always non-refining, with dashed lines ($ref = 0$). Therefore *TR9* is applicable to a simple leaf or a leaf of a refining *xor* constructor, or a refining *one* replicator.

[2] The full formal description of the translation rules is presented in the PhD thesis of the first author of this paper: http://eprints.soton.ac.uk/340357/.

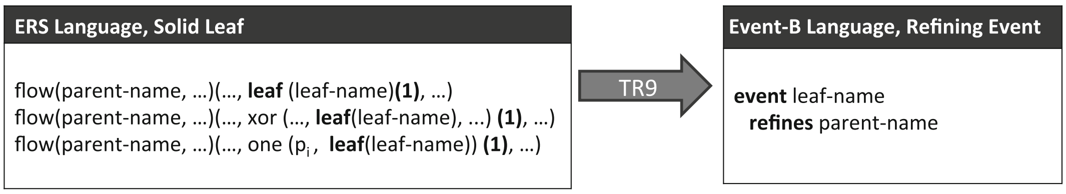

ERS Language, Solid Leaf
flow(parent-name, ...)(..., **leaf** (leaf-name)**(1)**, ...)
flow(parent-name, ...)(..., xor (..., **leaf**(leaf-name), ...) **(1)**, ...)
flow(parent-name, ...)(..., one (p_i, **leaf**(leaf-name)) **(1)**, ...)
TR9
Event-B Language, Refining Event
event leaf-name
refines parent-name

Fig. 18. TR9 definition

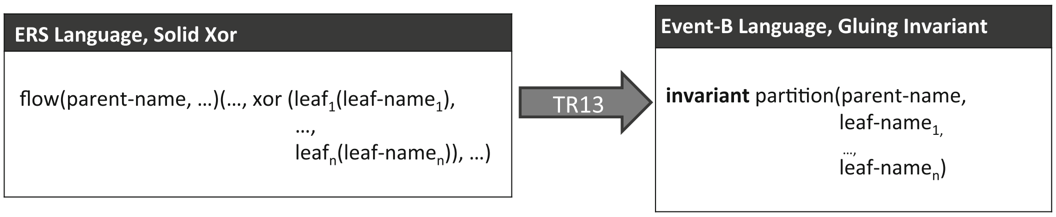

ERS Language, Solid Xor
flow(parent-name, ...)(..., xor ($leaf_1$($leaf\text{-}name_1$), ..., $leaf_n$($leaf\text{-}name_n$)), ...)
TR13
Event-B Language, Gluing Invariant
invariant partition(parent-name, $leaf\text{-}name_1$, ..., $leaf\text{-}name_n$)

Fig. 19. TR13 definition

Finally, Fig. 19 presents *TR13* (introduced in Fig. 13). A solid line *xor* constructor results in construction of an Event-B partitioning invariant, which describes the relationship between the abstract variable and the refinement variables; also it describes the mutually exclusive property of the *xor* sub-events.

5. Tool support

Eclipse [Ecl], is a multi-language Integrated Development Environment (IDE) with an extensible plug-in system. The Rodin platform is an Eclipse-based IDE for Event-B and is further extendable with plug-ins. By taking advantage of the extensibility feature of the Rodin platform, we have developed a plug-tool to support the ERS approach. Since the ERS plug-in addresses automatic construction of the Event-B models in term of control flows and refinement relationships, the ERS plug-in helps developers to build Event-B models in a more consistent and systematic way, compared with manually constructed models. The ERS plug-in allows users to define the ERS diagrams, then the ERS diagram is automatically transformed to an Event-B model.

Concerning the development architecture, we define the ERS language specification in an EMF (Eclipse Modelling Framework) [SBP08] meta-model, called the source meta-model, and then the source meta-model is transformed to the Event-B EMF meta-model as the target meta-model. The transformation is performed using the Epsilon Transformation Language (ETL) [KRP08]. ETL is a rule-based model-to-model transformation language.

ETL rules are direct implementation of the translation rules. The ETL rule for TR1 (from Fig. 16) is as follows:

```
rule Leaf2Variable
    transform l : Source!Leaf
    to v : Target!Variable{
    v.name := l.name;}
```

This rule transforms a leaf from the ERS language meta-model (as the source meta-model) to a variable in the Event-B meta-model (as the target meta-model). In the body of rule the name of the target component (variable) is assigned to the name of the source component (leaf).

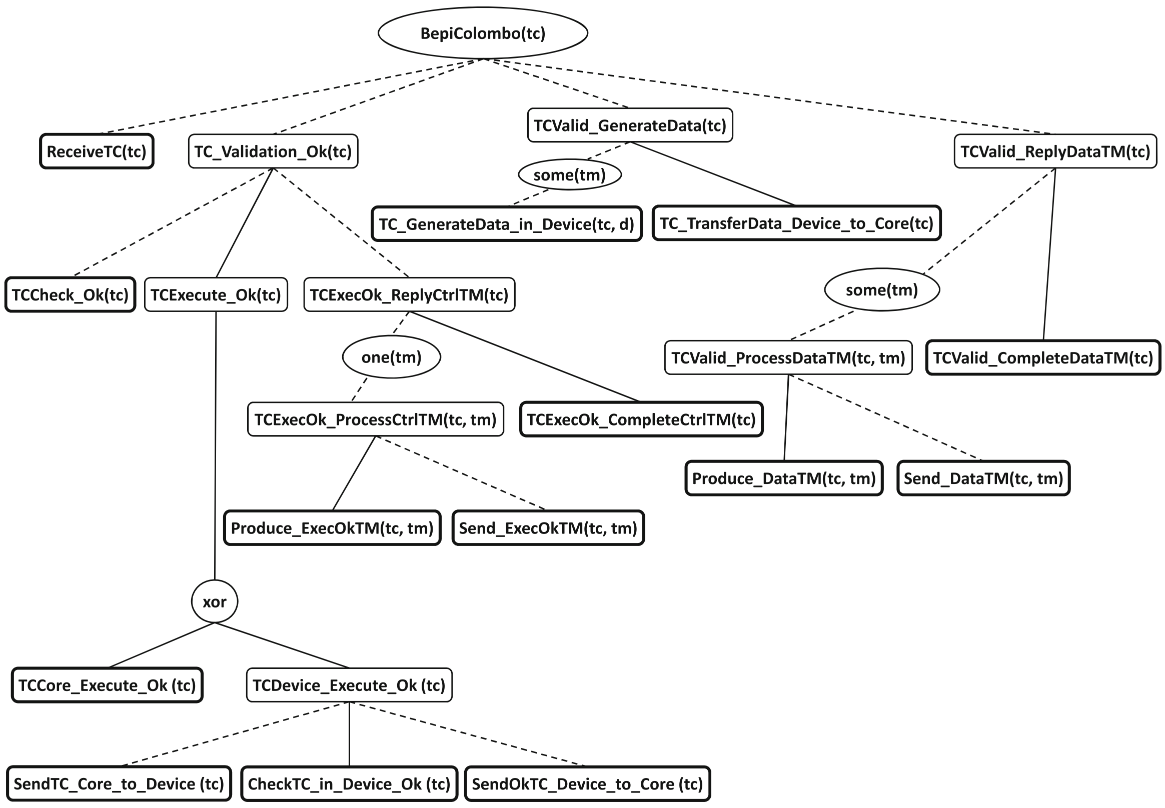


Fig. 20. Overall refinement structure after model decomposition, BepiColombo system

6. Evaluation

The contributions of the ERS approach in specifying the explicit control flows and refinement relationships in the Event-B formal modelling have been outlined in the previous sections. This section discusses other benefits of the systematisation and automatic translation of ERS, in terms of the methodological results and the comparison with previous manual development and recent automatic development of the case studies.

6.1. Overall visualisation of refinement and event tracking

The ERS diagrams provide the overall visualisation of refinement structure. Figure 20 presents a part of the overall refinement structure of the BepiColombo system. Using the overall view of refinement structure gives us the ability to track possible event execution traces by following leaf events from left to right. It provides the visualisation of the behaviour of the entire Event-B model which is more difficult to understand by just reading the Event-B. Event tracking helps us to describe the system requirements which can help us to identify requirements coverage. For instance, in Fig. 20 one of the possible execution traces is shown below. It shows the model covers the case when the validation is ok and the TC belongs to a device.

$< ReceiveTC,$
$TCCheck\_Ok,$
$SendTC\_Core\_to\_Device,\ CheckTC\_in\_Device\_Ok,\ SendOkTC\_Device\_to\_Core,$
$Produce\_ExecOkTM,\ Send\_ExecOkTM,\ TCExecOk\_CompleteCtrlTM,$
$TC\_GenerateData\_in\_Device,\ TC\_TransferData\_Device\_to\_Core,$
$Produce\_DataTM,\ Send\_DataTM,\ TCValid\_CompleteDataTM >$

This trace illustrates that a *tc* is received (the leftmost leaf node), then its syntax is checked (second line), then since the *tc* belongs to a device it will be sent to the device (second child of the *xor*), its semantic is checked in the device and the result is sent back (third line), and so on.

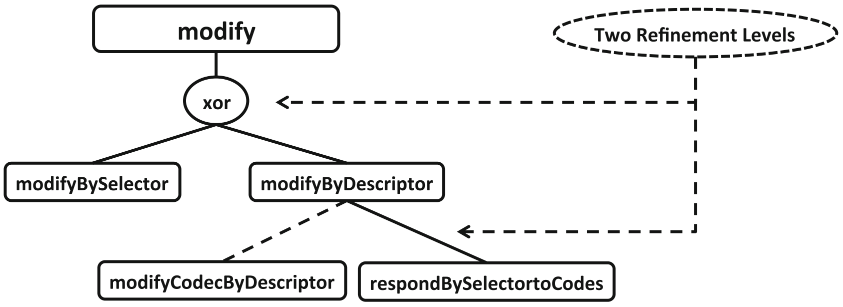


Fig. 21. Decomposing atomicity of *modify* event in two levels of refinement

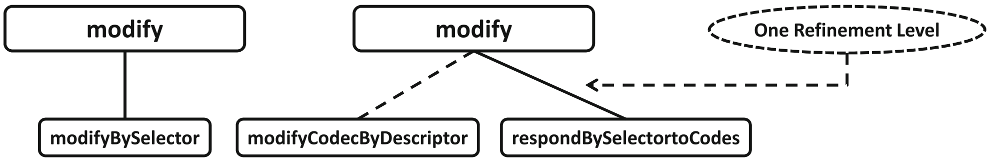


Fig. 22. Decomposing atomicity of *modify* event in one level of refinement

Using the *xor* constructor allows for other event traces. For instance considering *xor* constructor in decomposing the $TCExecute\_Ok$ event into $TCCore\_Execute\_Ok$ and $TCDevice\_Execute\_Ok$ sub-events (bottom left of Fig. 20), another possible event trace, when the TC belongs to the core, is to replace execution of

$< SendTC\_Core\_to\_Device, CheckTC\_in\_Device\_Ok, SendOkTC\_Device\_to\_Core >$

with $TCCore\_Execute\_Ok$.

6.2. Exploring alternatives

The possibility of a diagrammatic view of the developments gives us the chance to consider alternatives in decomposing the atomicity of an event. This decision can be done before making the effort of changing the Event-B model. For instance in the media channel development, we identified two possible ways of refining the $modify$ event, presented separately in Figs. 21 and 22. The atomicity decomposing of the $modify$ event is done in two levels of refinement in Fig. 21 whereas by using the second decomposition in Fig. 22, we can reduce it to one level of refinement. In the second way we separate the case split in two separate decomposition diagrams (simple sequence of events without using constructors), shown in Fig. 22. We chose ERS in Fig. 22 with fewer refinements to reduce the effort of modelling. This case shows how we can explore event refinement alternatives using ERS diagrams before creating the Event-B model. Therefore the ERS approach can help us find good ways of refining events before getting involved with the complex Event-B model.

6.3. Prevention of wrong event decomposition

Using ERS diagrams can result in earlier detection of wrong refinements in the modelling process. Figure 23 presents one possible way of decomposing the atomicity of validation phase in the development of the BepiColombo system. Figure 23 states that a validation can succeed, *TC_Validation_Ok*, or fail, *TC_Validation_Fail*. A successful validation means successful syntax validation, *TCCheck_Ok* event, followed by a successful semantic validation, *TCExecute_Ok* event. And a failed validation fails either in the syntax check, *TCCheck_Fail* event, or the semantics check, *TCExecute_Fail*. The possible event executions of Fig. 23 are:

$< TCCheck\_OK(tc), TCExecute\_OK(tc) >$
$< TCCheck\_Fail(tc) >$
$< TCExecute\_Fail(tc) >$

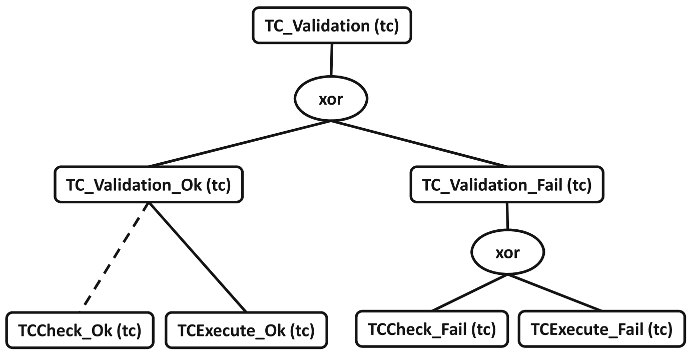


Fig. 23. Wrong ERS

These traces do not cover the following trace where the syntax validation is ok but the semantic check fails:

$< TCCheck\_OK(tc), TCExecute\_Fail(tc) >$

A wrong refinement could be identified without using the ERS diagrams as well, but discovering the mistake needs effort of the Event-B modelling and model checking.Whereas using ERS diagram helped us to prevent a wrong refinement earlier, before doing the effort of Event-B modelling.

6.4. Assessment of the automatic models

Our ERS tool addresses automatic construction of control flow in Event-B modelling. Moreover using the ERS plug-in to create the Event-B model of a system, ensures a consistent encoding of the ERS diagrams in a systematic way. In order to investigate the contributions of the ERS approach, first we developed the Event-B models of the case studies manually. The insights gained from the manual developments helped us to improve the ERS approach and develop the tool supporting the approach. The manually constructed Event-B models are less systematic and less consistent, since at the time of developing them our experience of ERS applications was less mature. The versions of the case studies reported in this paper are referred to as automatically constructed models. We applied the tool to the two case studies and compared the automatic models with the manual models, reported in our earlier work. There are some differences between the automatic models and the manual models, of which some of the more notable ones are described in this section.

6.4.1. Naming convention

In the automatic Event-B models (like Fig. 8), each control variable has the same name as the corresponding event name. Whereas in the manual Event-B models, there was no specific naming convention for variable names. Providing a unique naming protocol makes it easier to understand the model and to track the ordering between events more easily.

6.4.2. Alternative approaches of control flow modelling in Event-B

There are several approaches to modelling control flow in Event-B. In the automatic Event-B translation, we adopted the subset approach to model ordering between sequential events. Consider Fig. 8 where the second control variable is a subset of the first one (*inv2*: $TC\_Validation\_Ok \subseteq ReceiveTC$). The alternative approach is disjoint sets. Using the disjoint sets, the token is removed from one set before it can be moved to the next set. The Event-B model of disjoint sets for the diagram in Fig. 8 is presented in Fig. 24. In this way the parameter *tc* is removed from *ReceiveTC* set variable in the body of *TC_Validation_Ok* event. The set variables *ReceiveTC* and *TC_Validation_Ok* are always disjoint, as specified in *inv1* ($ReceiveTC \cap TC\_Validation\_Ok = \emptyset$).

```
invariants
  @inv1 ReceiveTC ∩ TC_Validation_Ok = ∅
```

```
event ReceiveTC
  any tc
  where
    @grd1 tc ∉ ReceiveTC
  then
    @act1 ReceiveTC ≔ ReceiveTC ∪ {tc}
 end
```

```
event TC_Validation_Ok
  any tc
  where
    @grd1 tc ∈ ReceiveTC
  then
    @act1 ReceiveTC ≔ ReceiveTC / {tc}
    @act2 TC_Validation_Ok ≔ TC_Validation_Ok ∪ {tc}
 end
```

Fig. 24. Disjoint sets in the most abstract level, BepiColombo system

One of the advantages of using the subset relationships in the Event-B models is that the sequencing relationships between the control variables can be specified in the invariants of the model. Considering Fig. 8, *inv2* specifies the ordering relationship between control variables. This ensures that the orderings are upheld in the Event-B model more strongly than if specified only in the event guards. Moreover, having disjoint set variables would not allow us to model some of the constructors in a simple way as subset variables provide. For example, in the case of *and* constructor, a logical *and* between two events, *a* and *b*, means four states as follows:

- none has happened
- *a* happened but not *b*
- *b* happened but not *a*
- *a* and *b* both happened

Using subset sets allows us to model these combinations using two set variables. But disjoint set variables does not allow this by using only two set variables. Using disjoint set variables to model these combinations would requires four state variables explicitly. As a result the Event-B models of the *and* constructor corresponding to the disjoint set approach are larger and more complex comparing to the subset approach models.

6.4.3. Complex guard versus simple guard

Considering the automatic Event-B model in Fig. 8, there is a separate guard for each predicate (*grd1* and *grd2* in the *TC_Validation_Ok* event). These separate guards are constructed as a result of different translation rules (TR5 and TR6 respectively). Whereas in the manual Event-B model, we modelled all of the pre-condition predicates in a single guard. For instance, guards of *TC_Validation_Ok* event in Fig. 8, can be merged as a more complex guard:

$tc \in ReceiveTC \setminus TC\_Validation\_Ok.$

To verify the correctness and consistency of an Event-B model, some proof obligations are generated by Rodin provers. Some of the generated proof obligations are related to the guard verification. Proving such proof obligations generated for the manual Event-B models requires more effort compared to the proof obligations generated for the automatic Event-B models, since the corresponding separated guards are simpler predicates compared to a merged guard.

6.4.4. More refinement levels

In the manual Event-B model, we did not have a one-to-one relation between control variables and events. In the media channel system case study, there are two different ways of opening a channel: open a channel with a codec and open a channel without a codec. Considering the manual events in Fig. 25, both *OpenWithRealCodecs* event and *OpenWithoutCodecs* event change the state of a channel, *ch*, to *open*. Whereas in the automatic Event-B model, as presented in Fig. 26, there is a one-to-one relation between control variables and the events. Each event change the state of a media channel to a unique state with same name as the event.

In the manual model there is a further refinement level in order to introduce a unique state for each event; for instance, concrete variables, *OpenWithCodecs* and *OpenWithoutCodecs*, replace the single abstract variable, *open*.

```
event OpenWithCodes
  any ch
  where
    @grd ch ∉ open
  then
    @act open ≔ open ∪ {ch}
 end
```

```
event OpenWithoutCodes
  any ch
  where
    @grd ch ∉ open
  then
    @act open ≔ open ∪ {ch}
 end
```

Fig. 25. Sharing a state variable in the manual model, media channel system

```
event OpenWithCodes
  any ch
  where
    @grd ch ∉ OpenWithCodes
  then
    @act OpenWithCodes ≔
           OpenWithCodes ∪ {ch}
 end
```

```
event OpenWithoutCodes
  any ch
  where
    @grd ch ∉ OpenWithoutCodec
  then
    @act OpenWithoutCodec ≔
           OpenWithoutCodec ∪ {ch}
 end
```

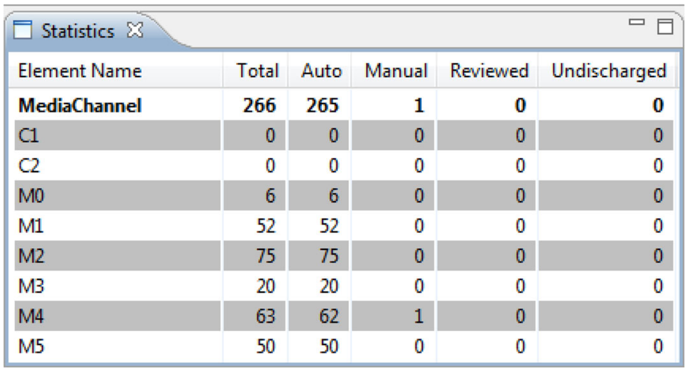
Fig. 26. Unique state variables in the automatic model, media channel system

The further refinement level makes the manual model larger and more complex, compared with the automatic model. Also more effort is need to define the gluing invariants between abstract variables and concrete variables.

6.5. Overview of proof obligations

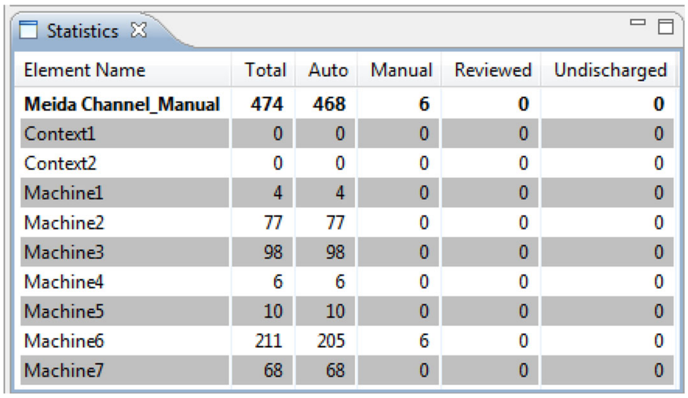
The result of the proof effort in the Rodin platform for the automatic Event-B model of the multimedia system, is outlined in Fig. 27. The *Total* column shows the total number of proof obligations generated for each level. The *Auto* column represents the number of those proof obligations that are proved automatically by the prover and the *Manual* column shows the number of proof obligations which are proved interactively. In Fig. 27, almost all proof obligations are proved automatically.

Figure 28 presents the proof effort for the manual Event-B model. The total number of proofs is more than the total number of proofs in the automatic model, because the extra refinement level in the manual model (*Machine6*), outlined in Sect. 6.4.4, significantly increases the number of proof obligations. A large number of proof obligations are caused by gluing invariants. Gluing invariants which that are needed to define the relations between the abstract non-unique states and concrete unique states, should be proved to be preserved by each action of each event. Also there are six proof obligations in *Machine6* which needed to be proved interactively. The interactive proofs are the gluing invariant preservation proofs. Therefore, recalling Sect. 6.4.4, introducing the unique states in an extra refinement level, not only makes the model large and complex, but also it makes the proof more complex.



Element Name	Total	Auto	Manual	Reviewed	Undischarged
MediaChannel	**266**	**265**	**1**	**0**	**0**
C1	0	0	0	0	0
C2	0	0	0	0	0
M0	6	6	0	0	0
M1	52	52	0	0	0
M2	75	75	0	0	0
M3	20	20	0	0	0
M4	63	62	1	0	0
M5	50	50	0	0	0

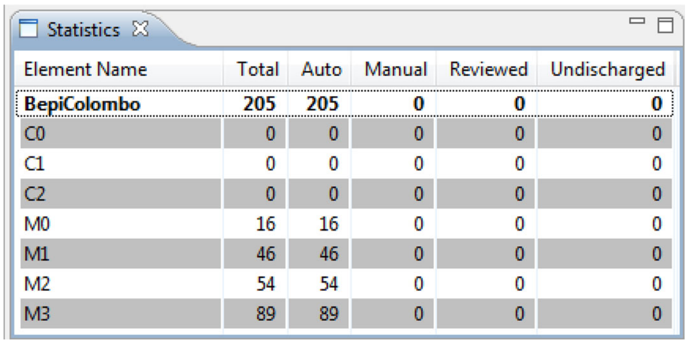
Fig. 27. Proof obligation statistics for the automatic multimedia Event-B model



Statistics

Element Name	Total	Auto	Manual	Reviewed	Undischarged
Meida Channel_Manual	**474**	**468**	**6**	**0**	**0**
Context1	0	0	0	0	0
Context2	0	0	0	0	0
Machine1	4	4	0	0	0
Machine2	77	77	0	0	0
Machine3	98	98	0	0	0
Machine4	6	6	0	0	0
Machine5	10	10	0	0	0
Machine6	211	205	6	0	0
Machine7	68	68	0	0	0

Fig. 28. Proof obligation statistics for the manual multimedia Event-B model

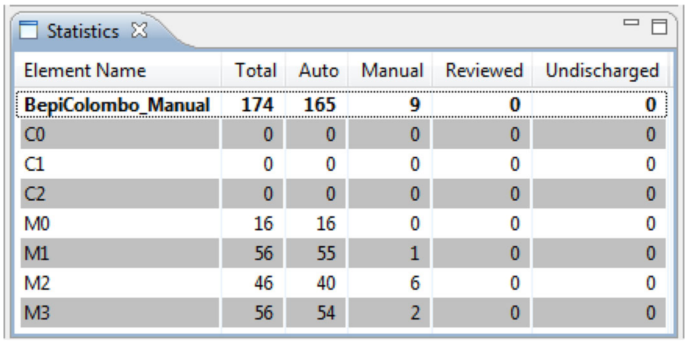


Statistics

Element Name	Total	Auto	Manual	Reviewed	Undischarged
BepiColombo	**205**	**205**	**0**	**0**	**0**
C0	0	0	0	0	0
C1	0	0	0	0	0
C2	0	0	0	0	0
M0	16	16	0	0	0
M1	46	46	0	0	0
M2	54	54	0	0	0
M3	89	89	0	0	0

Fig. 29. Proof obligation statistics for the automatic spacecraft Event-B model

A summary of the proof obligations for the automatic Event-B model of the spacecraft system can be seen in Fig. 29. The overall 205 generated proof obligations discharged automatically. Most of the proof obligations are related to gluing invariants preservation and guard strengthening. Figure 30 presents the summary of the proof obligations for the manual Event-B model for the BepiColombo system. The number of proof obligations in the manual model is slightly less than the automatic ones. As described in Sect. 6.4.3, having separate guards in the automatic model increases the number of proof obligations, though they are individually simpler. However all of the automatic model's proofs are discharged automatically, whereas in the manual model, nine proofs had to be discharged interactively.



Statistics

Element Name	Total	Auto	Manual	Reviewed	Undischarged
BepiColombo_Manual	**174**	**165**	**9**	**0**	**0**
C0	0	0	0	0	0
C1	0	0	0	0	0
C2	0	0	0	0	0
M0	16	16	0	0	0
M1	56	55	1	0	0
M2	46	40	6	0	0
M3	56	54	2	0	0

Fig. 30. Proof obligation statistics for the manual spacecraft Event-B model

7. Related work and conclusion

7.1. Related work

The desire to explicitly model control flow is not restricted to Event-B. To address this issue usually a combination of two formal methods are suggested. A good example of such an approach is Circus [WoC02] combining CSP [Hoa85] and Z [DaW96]. The combination of CSP and classical B [Abr96] has also been investigated in [But00] and [ScT04].

To provide explicit control flow for an Event-B model, a combination of two formal methods is presented in [STW10] which is based on using CSP alongside Event-B. As presented in Sect. 2.2, control flow can only be implicitly modelled in state variables and event guards in Event-B. On the other hand CSP is a process-based formalism, which explicitly supports specifying control flow via processes. [STW10] presents an integrated formal method, a combination of Event-B as a state-based formalism and CSP as a control-based formalism, to explicitly model control flow in Event-B.UML-B [SBS08, SBS09] provides a "UML-like" graphical front-end for Event-B. It adds support for class-oriented and state machine modelling. State machines provide a graphical notation to explicitly define event sequencing. Events are represented by transitions on a state machine, and control flow is specified by defining the source and target state of each transition. Another method to explicitly define control flow properties of an Event-B model is suggested in [IIi09] and [IIi10]. This method extends Event-B models with expressions, called flows, defining event ordering. Flows are written in a language resembling those in process algebra.

A comparison between ERS and other techniques outlined above, is provided as follows:

- All the outlined techniques only deal with explicit event sequencing; they do not support the explicit refinement relationship, provided by ERS diagrams. ERS provides a graphical front-end to Event-B along with other features such as supporting explicit event sequencing and expressing refinement relationships between abstract and concrete events. The graphical front-end of ERS can provide an overall visualisation of the refinement structure, which is not supported by any of the techniques outlined above.
- In integrated formal methods, the control flow constructs rely on the constructs in the process-based formalism of the integration. CSP constructs are used to model control flow in integrations of CSP and Z/B/Event-B. CSP constructs include prefix, deterministic choice, nondeterministic choice, parallel, interleaving, hiding and recursion.
  ERS constructs, as presented in Sect. 3, contain the sequence construct, the loop construct, logical constructs, e.g. *and*/*or*/*xor*, and *all*/*some*/*one* constructs as generalisation of the *and*/*or*/*xor* constructs.
  The CSP constructs and the ERS constructs can be compared as follows:
  - The prefix operator in CSP is used to describe the sequence of events and is equivalent to the sequence construct in ERS.
  - The choice operators in CSP are equivalent to the *xor* construct in ERS. We do not distinguish between deterministic and nondeterministic choice in ERS. The *one* construct in ERS is generalisation of the *xor* construct; CSP also supports a generalisation of its choice operators similar to our *one* construct.
  - The parallel operator is CSP is equivalent to the *all* construct in ERS. In ERS, the *all* construct is generalisation of the *and* construct; the *and* construct is also supported by parallel operator in CSP.
  - The interleaving operator is supported in CSP. In ERS, different diagrams can be interleaved based on the Event-B interleaving.
  - CSP includes an event hiding operator. In the Event-B refinement, a new event introduced in a refining machine, may be considered as a hidden event in the abstract machine. In ERS, we decomposed the atomicity of an abstract event to new concrete events and a refining concrete event. The new events connected with dashed lines to the abstract event, are considered as hidden events in the abstract machine.
  - CSP supports recursion (which makes it possible to model loops). ERS supports loops but not recursion.

  - There is no equivalences for the *or* construct and the *some* construct (as generalisation of *or*) of ERS, in CSP. Recalling the *or* construct in Fig. 12, in (*Event1 or Event2*), one or both may occur which is different from choice and different from interleaving.

  The flow language, presented in [IIi09] and [IIi10], is based on process algebra. The flow language constructs contain sequential composition, parallel composition, choice and loop.
  Control flow in Event-B can be modelled in state machine supported by UML-B [SBS08, SBS09]. Sequencing, choice and loop can be encoded in state machines, state machines do not have explicit constructs for these. State machines have explicit constructs for parallel regions. The *or* construct and the *some* construct (as generalisation of *or*) of ERS, are not supported in UML-B state machine.

- A Classical B operation can be called by other operations. It is the responsibility of the caller to ensure that the called operation pre-conditions hold. In contract in Event-B, an event contain guards and the enabled events are continually executed in a nondeterministic manner.
  In the integration of CSP and classical B presented in [ScT04], classical B operations are called with CSP description. CSP description allows us to make sure that pre-conditions of called operations hold. In the integration of CSP and Event-B presented in [STW10], the authors do not need to deal with pre-conditions, as Event-B events contain guards rather than preconditions.
- In the integration of CSP and Event-B technique presented in [STW10], a new tool framework for reasoning about combined specifications would be required. In contrast in ERS and UML-B state machines the graphical representation is directly transformable to the Event-B formalism. This in turn means that verification effort can be carried out in the existing Event-B tool-set, Rodin, which is already familiar to the Event-B users. Also in the combined CSP with classical B approach presented in [But00], CSP specifications are converted into standard B specifications.
- As [STW10] suggests, in combining formal method descriptions we may not be able to express all invariants as state predicates; because the control flow requirements are separated in a process-based description. While in ERS, control flow requirements are translated into Event-B; and Event-B invariants have access to all state variables in one place, the Event-B model.

7.2. Conclusion

In the previous publications [SaB10, SRB11] we have demonstrated how the event refinement structure (ERS) approach provides a means of introducing explicit flow control and explicit refinement relationships into Event-B development process. In this paper, we have presented the formal description of the ERS language and translation rules from the ERS language to the Event-B language. We have developed a tool, supporting ERS; the tool support was developed as a plug-in for the Event-B tool-set, Rodin. A brief description of ERS tool development has been illustrated. Using translation rules developed in the ERS tool, has helped us to develop the models of the previous case studies in an automatic way. Compared to the previous manual models of the case studies, the automatic models are more consistent and systematic. Some aspects of this improvement have been outlined.

The current ERS tool does not provide a graphical environment of ERS diagrams. Instead an ERS diagram is represented as an EMF model that is manipulated using an EMF structure editor. We consider developing a graphical environment of ERS diagrams as future work. Also future work is needed in order to improve the ERS language and translation rules. For this reason, further applications of ERS using the ERS tool are being undertaken.

Acknowledgements

This work is funded by the FP7 ADVANCE Project, http://www.advance-ict.eu.

References

[ABH06] Abrial JR, Butler M, Hallerstede S, Son Hoang T, Mehta F, Voisin L (2010) Rodin: an open toolset for modelling and reasoning in Event-B. STTT 12:447–466
[Abr96] Abrial JR (1996) The B-book: assigning programs to meanings. Cambridge University Press, Cambridge

[Abr05] Abrial JR (2005) Refinement, decomposition and instantiation of discrete models. In: Proceedings of the 12th international workshop on abstract state machines, pp 17–40
[Abr10] Abrial JR (2010) Modeling in Event-B: system and software engineering. Cambridge University Press, Cambridge
[BaK88] Back RJ, Kurki-Suonio R (1988) Distributed cooperation with action systems. ACM Trans Program Lang Syst, vol 10, pp 513–554
[But00] Butler M (2000) csp2B: a practical approach to combining CSP and B. Form Asp Comput 12:182–196. ISSN 0934-5043
[But09] Butler J (2009) Decomposition structures for Event-B. In: IFM2009. LNCS, vol 5423. Springer, Berlin
[CrO08] Crocker D, Overell P (2008) Augmented BNF for syntax specifications: ABNF. STD 68, RFC 5234
[Ecl] Eclipse [Online] (2013). http://www.eclipse.org
[ESA08] ESA Media Center, Space Science (2008) Factsheet: Bepicolombo. http://www.esa.int/esaSC
[Hoa85] Hoare CAR (1985) Communicating sequential processes. Prentice Hall, Englewood Cliffs. ISBN 0-13-153289-8
[IIi09] Iliasov A (2009) On Event-B and control flow. Technical Report, School of Computing Science, Newcastle University, http://deploy-eprints.ecs.soton.ac.uk/144
[IIi10] Iliasov A (2010) Tutorial on the flow plugin for Event-B. Workshop on B Dissemination [WOBD] Satellite event of SBMF, Natal, Brazil
[Jac83] Jackson MA (1983) System development. Prentice-Hall, Englewood Cliffs
[KRP08] Kolovos D, Rose L, Paige R (2008) The epsilon book. http://www.eclipse.org/gmt/epsilon/doc/book
[MAV05] Metayer C, Abrial JR, Voisin L (2005) Event-B language. RODIN Project Deliverable 3.2. http://rodin.cs.ncl.ac.uk/deliverables/D7.pdf
[Pro] ProB Animator and Model Checker [Online] (2013). http://wiki.event-b.org/index.php/ProB
[SaB10] Salehi Fathabadi A, Butler M (2010) Applying Event-B atomicity decomposition to a multi media protocol. In: FMCO formal methods for components and objects, pp 89–104
[SBP08] Steinberg D, Budinsky F, Paternostro M, Merks E (2008) EMF: eclipse modeling framework, 2nd edn. Part of the eclipse series. Addison-Wesley Professional, Reading
[SBR12] Salehi Fathabadi A, Butler M, Rezazadeh R (2012) An approach to atomicity decomposition in the Event-B formal method. In: SEFM software engineering and formal methods, pp 78–93
[SBS08] Said MY, Butler M, Snook C (2008) UML-B and Event-B: an integration of languages and tools. In: The IASTED international conference on software engineering, pp 336–341
[SBS09] Said MY, Butler M, Snook C (2009) Language and tool support for class and state machine refinement in UML-B. In: FM2009-16th international symposium on formal methods, pp 579–595
[ScT04] Schneider S, Treharne H (2004) Verifying controlled components. In: Proc IFM, Springer, Berlin, pp 87–107
[SRB11] Salehi Fathabadi A, Rezazadeh A, Butler M (2011) Applying atomicity and model decomposition to a space craft system in Event-B. In: NASA formal methods, pp 328–342
[STW10] Schneider S, Treharne H, Wehrheim H (2010) A CSP approach to control in Event-B. In: Proceedings of the 8th international conference on integrated formal methods, pp 260–274
[DaW96] Woodcock J, Davies J (1996) Using Z: specification, refinement and proof. Prentice hall international series in computer science. ISBN 0-13-948472-8
[WoC02] Woodcock J, Cavalcanti A (2002) The semantics of circus. In: ZB 2002: Formal specification and development in Z and B, 2nd international conference of B and Z users, pp 184–203
[ZaC09] Zave P, Cheung E (2009) Compositional control of IP media. IEEE Trans Softw Eng 35(1):46–66

Received 22 August 2013
Revised 27 May 2014
Accepted 8 July 2014 by George Eleftherakis and Jim Woodcock
Published online 12 September 2014