A Systematic Approach to Atomicity Decomposition in Event-B*

Asieh Salehi Fathabadi[1], Michael Butler[2], and Abdolbaghi Rezazadeh[3]

University of Southampton, UK
asf08r[1],mjb[2],ra3[3]@ecs.soton.ac.uk

Abstract. Event-B is a state-based formal method that supports a refinement process in which an abstract model is elaborated towards an implementation in a step-wise manner. One weakness of Event-B is that control flow between events is typically modelled implicitly via variables and event guards. While this fits well with Event-B refinement, it can make models involving sequencing of events more difficult to specify and understand than if control flow was explicitly specified. 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. To address these weaknesses, atomicity decomposition diagrams support the explicit modelling of control flow and refinement relationships for new events. In previous work, the atomicity decomposition approach has been evaluated manually in the development of two large case studies, a multi media protocol and a spacecraft sub-system. The evaluation results helped us to develop a systematic definition of the atomicity decomposition approach, and to develop a tool supporting the approach. In this paper we outline this systematic definition of the approach, the tool that supports it and evaluate the contribution that the tool makes.

1 Introduction

The Event-B formal method [1] is an evolution of classical B [2]. Event-B is proven to be applicable in a wide range of domains, including distributed algorithms, railway systems and electronic circuits. The Event-B modelling language has a simple notation and structure. States of a system are defined by variables and state changes of a system are defined by guarded actions, also called events. The basic specification construct is a machine that comprises of variables and events. Event-B supports refinement [3] 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.

* Acknowledgement: This work is partly supported by the EU research project ICT 214158 DEPLOY (Industrial deployment of system engineering methods providing high dependability and productivity) www.deploy-project.eu.

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 [4], [5].

A second weakness of Event-B is that all refinement relationships between refinement events and the abstract events are not explicit. Refinement in Event-B can consist of introducing new events. Although the refinement process in Event-B provides a flexible approach to modelling, it is not able to explicitly show the relationships between abstract events and new events introduced during a refinement step.

To address these weaknesses, the atomicity decomposition approach [6] addresses the explicit control flow modelling and explicit refinement relationships representation. It provides a graphical notation to structure the refinement process and to illustrate the explicit ordering between events of a model. The atomicity decomposition graphical notation contains tree structured diagrams based on JSD structure diagrams by Jackson [7]. Semantics are given to an atomicity decomposition diagram by generating an Event-B model from it.

In the rest of this paper, "AD" refers to Atomicity Decomposition. The steps carried in our research are presented in Figure 1. AD is first introduced by Butler [6] (step 1). It has been observed that methodological support for AD was weak. So we decided to evaluate and enhance the existing AD approach in [6]. For this reason we manually applied AD to two sizeable case studies, a multi media protocol and a space craft system (step 2). The first case study, the multi media protocol [8], contains requirements to establish, modify and close a media channel between two endpoints for transferring multi media data. Second case study is based on a space craft system called BepiColombo [9]. The manual development processes of these case studies have been published in [10] and [11] respectively. Insights gained from these case studies, enable us to define a formal description of the AD language (ADL) and formal translation rules from AD diagrams to Event-B (step 3). Based on the ADL and translation rule descriptions, we have developed the AD tool support, as a plug-in for the Event-B tool-set, called Rodin (step 4). Our AD tool support, can automatically generate Event-B models from AD diagrams. And finally we re-develop the case study models using the provided AD tool support (step 5).

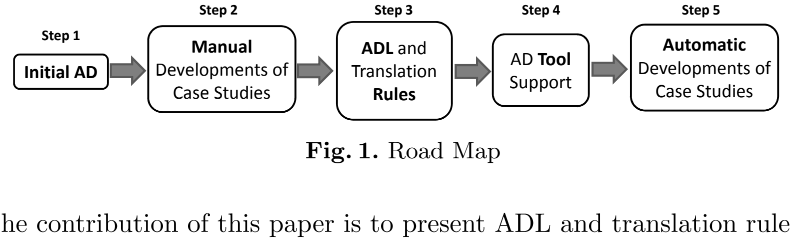

Step 1
Initial AD
Step 2
Manual Developments of Case Studies
Step 3
ADL and Translation Rules
Step 4
AD Tool Support
Step 5
Automatic Developments of Case Studies

Fig. 1. Road Map

The contribution of this paper is to present ADL and translation rules from AD diagrams to the Event-B language, covering steps 3, 4 and 5 of Figure 1. We also outline the development of AD tool and the technologies that used in this tool development. One of our objective in this paper is to assess how application of translation rules, makes the automatic models of case studies more consistent and systematic, comparing with the previous manual ones. Moreover the recent automatic developments of case studies are briefly presented.

The paper is structured as follows: Section 2 outlines the Event-B method, atomicity decomposition approach, related works and an overview of case studies requirements; Section 3 contains the ADL description and definitions of translation rules; Section 4 presents the tool developed to support AD; In Section 5 we evaluate how AD tool, has helped us to enhance the development of case studies in a more consistence and systematic way; finally Section 6 concludes.

2 Background and Related Works

2.1 Event-B

The Event-B formal method [1], [12] has evolved from classical B [2] and action systems [13]. Event-B is used in modelling and verifying consistency of models. The modelling language is based on set theory and first order logic.

A model in Event-B consists 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 context can be “extended” by other contexts and “referenced” by machines. A machine can be “refined” by other machines and can reference contexts.

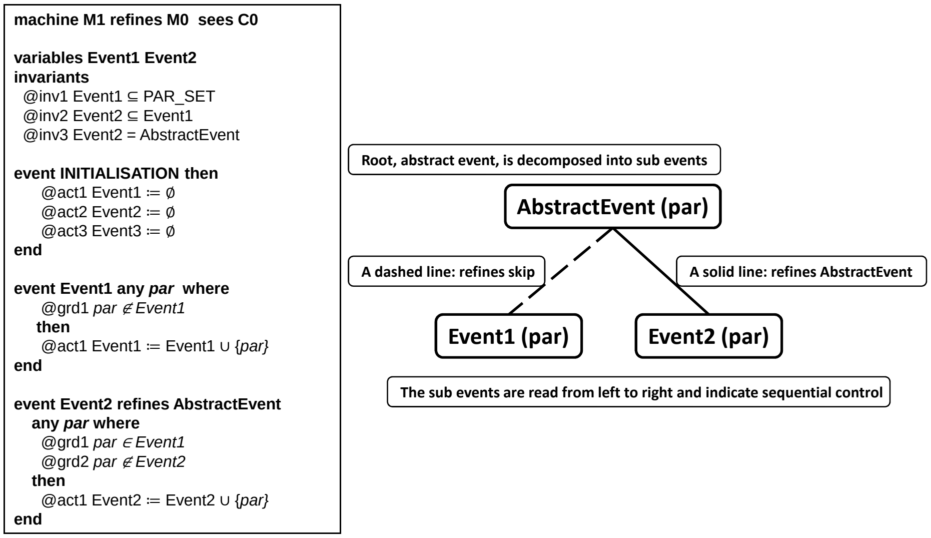
Building a model in Event-B usually starts with an abstract level, and continues in successive refinement levels. The abstract model provides a simple view of the system, focusing on main purposes of the system. Details are added gradually to the abstract model during refinement levels. In Event-B, refinement 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. 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* will said to be a concrete version of *Machine1*.

Rodin [14] is an Eclipse-based tool for formal modelling and proving in Event-B. Rodin has an open platform, and is an extensible and adaptable modelling tool. We have taken the advantage of the extensibility feature of the Rodin to develop a tool support for the AD approach.

2.2 Atomicity Decomposition Approach

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. The AD approach addresses this limitation. The idea is to augment Event-B refinement with a graphical notation that is capable of representing the relationships between abstract and concrete events explicitly. Using the AD approach has another advantage which is that we can represent event ordering explicitly. Figure 2 illustrates these two features of AD graphical notation.

Assume machine *M1* on the left hand side of Figure 2, refines machine *M0* which contains the abstract specification of *AbstractEvent*. 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 AD diagram presented in the right hand side. This diagram explicitly illustrates that the effect achieved by *AbstractEvent* at the abstract level, machine *M0*, is realized at the refined level, machine *M1*, by occurrence of *Event1* followed by *Event2*. The ordering of the leaf events is always from left to right (this is based on JSD diagrams of Jackson [7]). 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.



```
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 ≔ ∅
     @act3 Event3 ≔ ∅
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 dashed line: refines skip
A solid line: refines AbstractEvent
Event1 (par)
Event2 (par)
The sub events are read from left to right and indicate sequential control

Fig. 2. Atomicity Decomposition Diagram

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 effect of an event with parameter *par*, is to add the value of *par* to a set 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 Related Works

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 [15] combining CSP [16] and Z [17]. The combination of CSP and classical B [2] has also been investigated in [4] and [18].

To provide explicit control flow for an Event-B model, a combination of two formal methods is presented in [19] which is based on using CSP alongside Event-B. As presented in Section 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.

UML-B [20] provides a "UML-like" graphical front-end for Event-B. It adds support for class-oriented and state machine modelling. State machines provide us with 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 [21]. This method extends Event-B models with expressions, called flows, defining event ordering. Flows are written in a language resembling those in process algebra.

All techniques outlined in this section, only deal with explicit event sequencing; they do not support the explicit refinement relationship, provided by atomicity decomposition diagrams. The atomicity decomposition approach 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 refinement events. An extra feature of the AD approach is that the graphical front-end of it can provide an overall visualisation of the refinement structure, which is not supported by any of techniques outlined above.

2.4 Overview of Case Studies

This section outlines an overview of our case study systems, a multi media protocol [8] and a space craft system based on BepiColombo [9].

Multi Media Protocol This case study specifies a protocol for establishing, modifying and closing a media channel. A media channel is established for transferring multi-media 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.

It is worth to compare our approach to the multi media protocol with the approach taken by Zave and Cheung [8]. 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 intension of the protocol and then use atomicity decomposition to arrive at models that have similar levels of detail to the Promela models.

Space Craft System Exploration of the planet Mercury is the main goal of the BepiColombo mission. One of the BepiColombo subsystems 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:

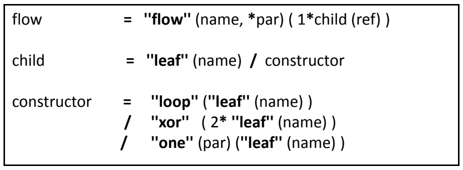
- A TeleCommand (*TC*) is received by the core from Earth.
- The CSW checks the syntax of the received *TC*.
- Further semantic checking has to be carried out on the syntactically validated *TC*. If the *TC* contains a message for one of the devices, it has to be sent to the device for semantic checking,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

3 AD Language and Translation Rules

3.1 Atomicity Decomposition Language

To describe the AD language (ADL) syntax, we adopted Augmented Backus-Naur Form (ABNF) [22]. ABNF is a metalanguage based on Backus-Naur Form (BNF).

An excerpt of the ADL syntax, describing a single AD diagram, is presented in Figure 3. This description is only a subset of the full ADL. It only includes three of AD constructors which are used in our case studies and are explained later in the following sections. There are other AD constructors which are not presented in this paper because of space limitation.



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

child         =  "leaf" (name)  /  constructor

constructor   =   "loop" ("leaf" (name) )
              /   "xor"  ( 2* "leaf" (name) )
              /   "one" (par) ("leaf" (name) )
```

Fig. 3. Syntax of the AD Language (ADL)

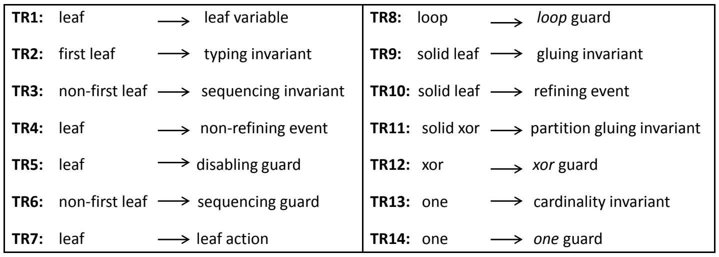
A flow, in Figure 3, refers to a single atomicity decomposition. To describe the type of a line (solid/dashed), we consider a boolean property, called "ref". When a sub-event refines the abstract event (solid line) , "ref" is one; otherwise "ref" is zero. Considering Figure 3, the ABNF of ADL may be described informally as follows:

- A flow consists of a name, zero or more parameters, followed by one or more children. Each child of a flow has a "*ref*" property.
- A child is either a "*leaf*" with a name, or a constructor.
- A constructor is either a "*loop*" with one leaf as its child or a '*xor*" with two or more leaves or an "*one*" with a parameter, followed by one leaf.

3.2 Translation Rules

Semantics are given to an AD diagram by generating an Event-B model from it, based on some translation rules. In this section, we discuss these translation rules. Here, due to space limitation, we only present translation rules that are used in our two case studies. [1] The initial AD diagrammatic notation in [6] has been extended with some AD constructors. Three of them, *loop*, *xor* and *one*, used in our case study developments are introduced here.

The main syntactic elements of an Event-B machine are variables, invariants, guards and actions. The encoding of AD 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 4 outlines the full list of translation rules used in this paper. Each translation rule defines a transformation from an AD source element to an Event-B destination element. Note that for each AD element usually there are 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, then the rules for the *loop* constructor, a solid leaf, the *xor* and *one* constructors.



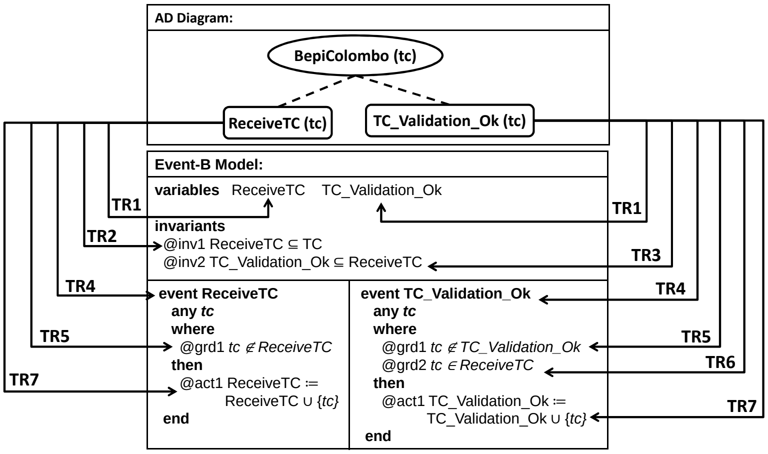
TR1:	leaf	⟶	leaf variable	**TR8:**	loop	⟶	*loop* guard
TR2:	first leaf	⟶	typing invariant	**TR9:**	solid leaf	⟶	gluing invariant
TR3:	non-first leaf	⟶	sequencing invariant	**TR10:**	solid leaf	⟶	refining event
TR4:	leaf	⟶	non-refining event	**TR11:**	solid xor	⟶	partition gluing invariant
TR5:	leaf	⟶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

Fig. 4. Translation Rules

[1] The full set of translation rules is presented in the PhD thesis of the first author of this paper.

Sequencing Rules As discussed in Section 2.2, one major feature of AD diagrams is to explicitly represent sequencing between events. To illustrate this concept, we have taken a part of the most abstract level diagrams of BepiColombo system, presented in higher level of Figure 5. In a 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 in an order from left to right. This diagram illustrates the scenario when a *TC* is received by the core, *ReceiveTC* event, and then it is validated by *TC_Validation_Ok* event.

The arrows in Figure 5 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 TR1 rule to each source leaf, produces a variable in the Event-B model. The generated variables are later used to control the flow of the leaf events.



```
variables  ReceiveTC   TC_Validation_Ok
invariants
 @inv1 ReceiveTC ⊆ TC
 @inv2 TC_Validation_Ok ⊆ ReceiveTC
```

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

```
event TC_Validation_Ok
  any tc
  where
   @grd1 tc ∉ TC_Validation_Ok
   @grd2 tc ∈ ReceiveTC
  then
   @act1 TC_Validation_Ok ≔
            TC_Validation_Ok ∪ {tc}
  end
```

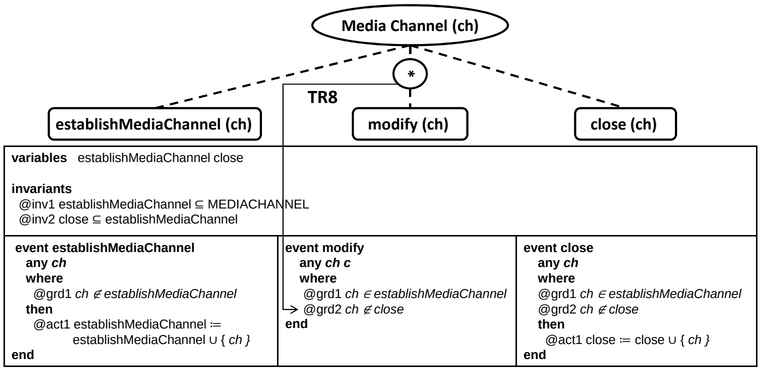
Fig. 5. The Most Abstract Level Model, BepiColombo System

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 illustrate the most abstract events, there is no solid line. For each leaf with a dashed line, TR4 generates a non-refining event. The parameter of the leaf is transformed to the event parameter. For each leaf, TR5 generates a disabling guard, which describes that the leaf event has not executed for the same instance of the parameter before. For each non-first event, like *TC_Validation_Ok* here, another guard is needed to make sure that the previous event has been executed for the parameter value before; this translation is carried out via TR6. Finally TR7 adds an action for each leaf, which disables the corresponding leaf event for a specific parameter value.

Translation rules (TR1-TR7) that are outlined in Figure 4 and applied in this section, are only applicable to leaf nodes and encode sequencing collectively. We discuss the rest of rules in the following sections.

Loop Constructor The loop constructor is used to model zero or more executions of a leaf. Figure 6 presents the most abstract AD diagram of the multi media 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 Figure 6, there is no variable generated for the leaf connected to the loop constructor, since we do not need to record the loop event execution. The event after the loop event can execute after execution of the event before the loop event (in the case of zero executions of the loop event).



```
variables   establishMediaChannel close

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

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

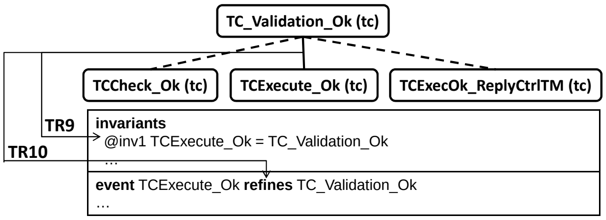
```
event modify
  any ch c
  where
    @grd1 ch ∈ establishMediaChannel
    @grd2 ch ∉ close
end
```

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

Fig. 6. The Most Abstract Level, Multi Media Protocol

The loop event can execute several times *before* execution of next event. TR8 transforms the loop constructor to a guard in the loop event, *modify*. This guard checks that the event after loop, *close*, has not executed before, for the intended channel. The other Event-B elements in Figure 6 are generated via TR1-TR7 which have been described in the previous section (Figure 4).

Solid Line The abstract atomic *TC_Validation_Ok* event in Figure 5, is decomposed to three sub-events in a refinement level. Figure 7 presents the AD diagram of this decomposition. Validating a received *TC* is not atomic. It is done in two steps, checking the syntax, in *TCCheck_Ok* event, and the semantics, in *TCExecute_Ok* event, of a received *TC*. After syntax and semantics checks, in the third step, *TCExecOk_ReplyCtrlTM* event, a control *TM* is produced and sent back to the earth.

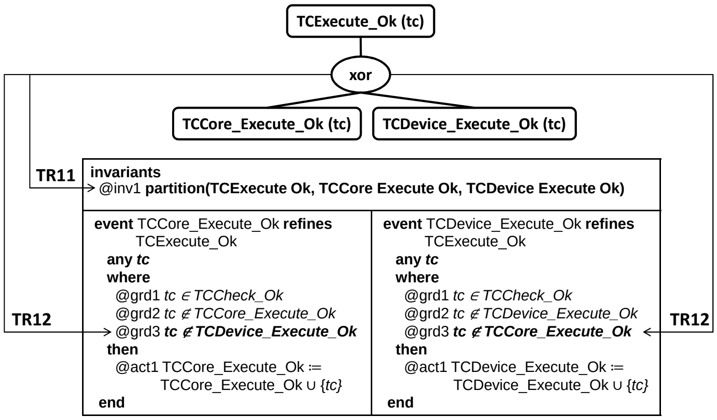


```
invariants
  @inv1 TCExecute_Ok = TC_Validation_Ok
  …
event TCExecute_Ok refines TC_Validation_Ok
…
```

Fig. 7. The AD Diagram of *TC_Validation_Ok*, BepiColombo System

Considering Figure 7, a solid line in an AD diagram has two effects. First, it is translated to an invariant which connects the abstract variable to the refinement variable (TR9); it is called a gluing invariant. Second, it is translated to an event which refines the abstract event in the root node (TR10).

xor **Constructor** Exclusive choice between two or more events is introduced to the AD diagram with a new constructor called *xor*. An application of the *xor* constructor in BepiColombo development is presented in Figure 8. A *TC* either belongs to the core or the device and not both of them. The figure illustrates a further level of refinement where the atomicity of semantics checking event, *TCExecute_Ok*, is decomposed to an exclusive choice between two sub-events; *TCCoreExecute_Ok* event checks the semantics of a *TC* which belongs to the core and *TCDeviceExecute_Ok* event checks the semantics of a *TC* which belongs to the device.



```
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. 8. The AD Diagram of *TC_Execute_Ok*, BepiColombo System

xor sub-leaves inherit the type of their line (solid/dashed) from the *xor* constructor. Considering Figure 8, 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.

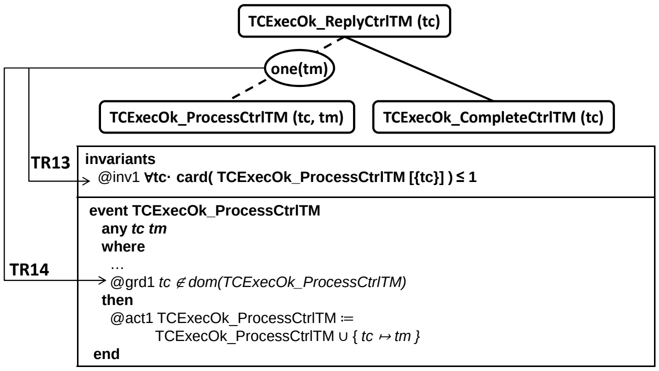
There are two translation rules for the *xor* constructor. First the *xor* constructor is transformed to the partitioning invariant (TR11), which ensures exclusivity of execution. The *partition* operator in Event-B is defined as follows:

$partition(E_0, E_1, ..., E_n) \equiv (E_0 = E_1 \cup ... \cup E_n) \wedge (i \neq j \Rightarrow E_i \cap E_j = \emptyset)$

The generated partitioning invariant first describes the relationship between the abstract variable and the refinement variables:
$TCExecute\_Ok = TCCoreExecute\_Ok \cup TCDeviceExecute\_Ok$.
Second it described the mutually exclusive property of the the *xor* sub-events:
$TCCoreExecute\_Ok \cap TCDeviceExecute\_Ok = \emptyset$.

The second translation rule (TR12) generates a guard for each *xor* sub-event. This guard describes the exclusiveness property of *xor* sub-events. The guard in each *xor* sub-even, checks that the other *xor* sub-events have not occurred for the intended value of the *TC*.

***one* Constructor** The *one* constructor specifies execution of an event for exactly one instance value of a new parameter. An application of *one* constructor in BepiColombo development is presented in Figure 9. Figure 9 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* event.


TCExecOk_ReplyCtrlTM (tc)
one(tm)
TCExecOk_ProcessCtrlTM (tc, tm)
TCExecOk_CompleteCtrlTM (tc)
TR13
TR14

```
invariants
  @inv1 ∀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. 9. The AD Diagram of *TCExecOk_ReplyCtrlTM*, BepiColombo System

As presented in Figure 9, the *one* constructor adds a new parameter, the *tm* parameter, to its sub-event, *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. TR13 generates an invariant which defines the *one* constructor property describing that for each *tc*, the cardinality of the set of processed *tm*s is at most one. And TR14 generated a guard to make sure that the *one* sub-event has not executed for the same value of intended *tc* before.

There are two more constructors, *all* and *some*, which adds a new parameter to their sub-events. The *all* constructor specifies execution of an event for all instance values of a new parameter. And the *some* constructor specifies execution of an event for some instance values of a new parameter. In this paper, we skip define them in depth.

Using the formal description of ADL, presented in in Figure 3, the translation rules outlined in Figure 4, can be defined formally. For instance, the formal description of TR1 is presented in Figure 10. The left box contains the AD element description that transformed to the right box containing the description of the Event-B element. In the case of TR1, each leaf (not loop leaf) is transformed to a variable (with same name as the leaf) in destination Event-B model.

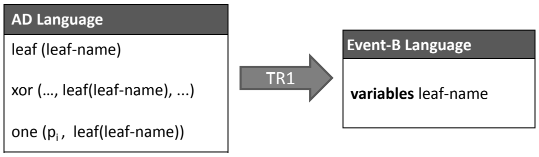


Fig. 10. TR1 Definition

4 Tool Support

Eclipse [23], is a multi-language software development environment comprising an integrated development environment (IDE) and 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 for Event-B, we have developed a plug-in as tool support for the AD approach. The AD plug-in helps developers to build Event-B models more easily, since the AD plug-in addresses automatic generation of the Event-B models in term of control flows and refinement relationships. The AD plug-in allows users to define the AD diagram; then the AD diagram is automatically transformed to an Event-B model.

The development architecture is briefly presented in Figure 11. We define the ADL specification in an EMF (Eclipse Modelling Framework) [24] meta-model, called source meta-model, and then the source meta-model is transformed to the Event-B EMF meta-model as the target meta-model. Currently AD diagrams are build as an EMF model, included in an Event-B machine. However we consider developing a graphical environment for the plug-in as future work. The transformation is done using the Epsilon Transformation Language (ETL) [25]. ETL is a rule-based model-to-model transformation language.

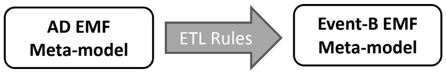


Fig. 11. AD Tool Support Architecture

The ETL rule for TR1 (presented in Section 3.2) is as follow:

```
rule Leaf2Varibale
    transform l : Source!Leaf
    to v : Target!Variable {
    v.name := l.name; }
```

This rule transforms a leaf from the ADL 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).

5 Evaluation

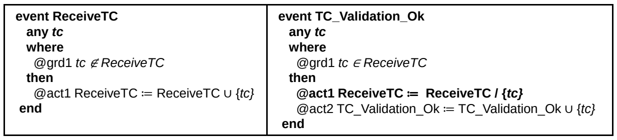
Our AD tool addresses automatic generation of control flow in Event-B modelling. Moreover using the AD plug-in to create the Event-B model of a system, ensures a consistent encoding of the AD diagrams in a systematic way. The manually generated Event-B models are less systematic and less consistent, since on the time of developing them our experience of AD applications were not enough. The versions of the case studies reported in this paper are referred to as automatically generated models. We applied the tool to the two case studies and compared the automatic models with the manual models, reported in our earlier works. There are some differences between the automatic models and the manual models, which some of the more notable ones are described in this Section.

5.1 Naming Protocol

In the automatic Event-B models (like Figure 5), each control variable has the same name as the corresponding event name. Whereas in the manual Event-B models, there was no specific naming protocol for variables name. Providing a unique naming protocol helps to understand the model more easily, and can help to track the ordering between events more easily.

5.2 Alternative Approaches of Control Flow Modelling in Event-B

There are different approaches to model control flow in Event-B. In the automatic Event-B model, we adopted the subset approach to model ordering between sequential events. Considering Figure 5, the second control variable is a subset of the first one (*inv1*). The alternative way is disjoint sets. The Event-B model of disjoint sets for the diagram in Figure 5 is presented in Figure 12. In this way the parameter *tc* is removed from *ReceiveTC* set variable in the body of *TC_Validation_Ok* event.



```
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. 12. 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 subset relationships between the control variables that represent different states of the model can be specified in the invariants of the model. Considering Figure 5, invariant *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.

5.3 A Merged Guard versus Separate Guards

Considering the automatic Event-B model in Figure 5, there is a separate guard for each predicate (*grd1* and *grd2* in the *TC_Validation_Ok* event). These separate guards are generated as a result of different translation rules (TR5 and TR6 respectively). Whereas in the manual Event-B model, we modeled all of the precondition predicates in a single guard. For instance, guards of *TC_Validation_Ok* event in Figure 5, can be merged as a single 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 guards verification. Proving such proof obligations generated for the manual Event-B models needs more effort comparing 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 Conclusion

In the previous publications we have demonstrated how the atomicity decomposition (AD) approach provides a means of introducing explicit flow control into Event-B development process. In this paper, we have presented the formal description of the atomicity decomposition language (ADL) and translation rules from the ADL to the Event-B language. We have developed a tool, supporting the atomicity decomposition methodology; the tool support is developed as a plug-in for the Event-B tool-set, Rodin. A brief description of AD tool development has been illustrated. Using translation rules developed in the AD 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 recent automatic models are more consistent and systematic. Some aspects of this improvement have been outlined.

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

References

1. Jean-Raymond Abrial: Modeling in Event-B: System and Software Engineering. Cambridge University Press, (2010)

2. Jean-Raymond Abrial: The B-book: Assigning Programs to Meanings. Cambridge University Press, (1996)
3. Jean-Raymond Abrial: Refinement, Decomposition and Instantiation of Discrete Models. In Abstract State Machines, pages 17-40, (2005)
4. Michael Butler: csp2B: A Practical Approach to Combining CSP and B. Formal Aspects of Computing, vol. 12, pp. 182-196, ISSN 0934-5043, (2000)
5. Alexei Iliasov: On Event-B and Control Flow. Technical Report, School of Computing Science, Newcastle University, (2009)
6. Michael J. Butler: Decomposition Structures for Event-B. In IFM2009, volume LNCS 5423. Springer, (2009)
7. M.A Jackson: System Development. Prentice-Hall, Englewood Cliffs (1983)
8. Pamela Zave and Eric Cheung: Compositional Control of IP Media. IEEE Trans. Software Eng., 35(1):46–66, (2009)
9. ESA Media Center, Space Science. Factsheet: Bepicolombo. http://www.esa.int/esaSC (2008)
10. Asieh Salehi Fathabadi and Michael J. Butler: Applying Event-B Atomicity Decomposition to a Multi Media Protocol. In FMCO Formal MEthods for Components and Objects, pages 89-104, (2010)
11. Asieh Salehi Fathabadi, Abdolbaghi Rezazadeh and Michael J. Butler: Applying Atomicity and Model Decomposition to a Space Craft System in Event-B. In NASA Formal Methods, pages 328-342, (2011)
12. C. Metayer, J-R Abrial and L. Voisin: Event-B language. RODIN Project Deliverable 3.2. http://rodin.cs.ncl.ac.uk/deliverables/D7.pdf, (2005)
13. Ralph-Johan Back and Reino Kurki-Suonio: Distributed Cooperation with Action Systems. ACM Trans. Program. Lang. Syst., pages 513-554, (1988)
14. Jean-Raymond Abrial, Michael J. Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta and Laurent Voisin: Rodin: An Open Toolset for Modelling and Reasoning in Event-B. STTT, volume 12, pages 447-466, (2010)
15. Jim Woodcock and Ana Cavalcanti: The Semantics of Circus. ZB2002, pages 184-203 (2002)
16. C. A. R. Hoare: Communicating Sequential Processes. Prentice Hall. ISBN 0-13-153289-8, (1985)
17. Jim Davies and Jim Woodcock Using Z: Specification, Refinement and Proof. Prentice Hall International Series in Computer Science. ISBN 0-13-948472-8, (1996)
18. S. Schneider and H. Treharne: Verifying Controlled Components. In In Proc IFM, Springer, pp. 87-107, (2004)
19. Steve Schneider, Helen Treharne and Heike Wehrheim: A CSP Approach to Control in Event-B. IFM, pages 260-274, (2010)
20. M. Y. Said, M. Butler and C. Snook: Language and Tool Support for Class and State Machine Refinement in UML-B. In: FM2009 - 16th International Symposium on Formal Methods, Eindhoven. pp. 579-595, 2-6th November (2009)
21. Alexei Iliasov: Tutorial on the Flow plugin for Event-B. Workshop on B Dissemination [WOBD] Satellite event of SBMF, Natal, Brazil, (2010)
22. Crocker, D. and Overell, P.: Augmented BNF for Syntax Specifications: ABNF. STD 68, RFC 5234, (2008)
23. Eclipse [Online] http://www.eclipse.org
24. Dave Steinberg, Frank Budinsky, Marcelo Paternostro and Ed Merks: EMF: Eclipse Modeling Framework Published by Addison-Wesley Professional Second Edition Part of the Eclipse Series series, (2008)
25. Dimitrios Kolovos, Louis Rose and Richard Paige: The Epsilon Book http://www.eclipse.org/gmt/epsilon/doc/book, (2008)