Applying Atomicity and Model Decomposition to a Space Craft System in Event-B[⋆]

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

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

Abstract. Event-B is a formal method for modeling and verifying consistency of systems. In formal methods such as Event-B, refinement is the process of enriching or modifying an abstract model in a step-wise manner in order to manage the development of complex and large systems. To further alleviate the complexity of developing large systems, Event-B refinement can be augmented with two techniques, namely atomicity decomposition and model decomposition. Our main objective in this paper is to investigate and evaluate the application of these techniques when used in a refinement based development. These techniques have been applied to the formal development of a space craft system. The outcomes of this experimental work are presented as assessment results. The experience and assessment can form the basis for some guidelines in applying these techniques in future cases.

1 Introduction

Event-B [2] is a formal method that evolved from the B-Method [8] and Action Systems [10]. Simplicity of notation and structure is one of the primary reasons for choosing Event-B to develop formal models of our case study. Event-B also is proven to be applicable in different domains including distributed systems [2]. Moreover Event-B supports refinement and uses mathematical proofs to verify consistency of models. Furthermore there is good tool support for modeling and proving.

Exploration of the planet Mercury is the main goal of the BepiColombo mission [13], which consist of two orbiters. One of the orbiters is the Mercury Planetary Orbiter (MPO) which performs global remote sensing and radio science investigations. An important part of this orbiter consist of a core and four devices: Solar Intensity X-ray Spectrometer (SIXS-X and SIXS-P) and Mercury Imaging X-ray Spectrometer (MIXS-T and MIXS-C). The whole system is controlled by mission-critical software. 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 the rest of this paper we refer to the core and the devices including control software

⋆ 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.

as the probe system. Our aim is to present a part of the probe system related to the management of TC and TM communications.

Modeling a large and complex system such as this probe system, can result in large and complex models with difficult proofs [3]. However Event-B provides some techniques to address this problem. One such technique is refinement that allows us to add details during a sequence of models instead of building a model in a flat manner. The Event-B refinement rules are very general and they do not explicitly represent relationships between abstract events and new events, introduced during refinement. Refinement can be augmented with another technique called atomicity decomposition [4] that provides a structuring mechanism for refinement in Event-B. Atomicity decomposition provides definitions and a diagrammatic notation to explicitly represent relationships between refinement levels. Using atomicity decomposition we can also illustrate the explicit sequencing between events of a model that is not always explicit in Event-B model. Model decomposition [6], [7] is another technique to divide a large model into smaller and more easily manageable sub-models.

Figure 1 presents the development architecture of Event-B model of the probe system. In the abstraction, *M0* , the main goal of the system is modeled. The details of the system are added through three refinement levels, *M1*, *M2* and *M3*. Then the last model, *M3*, is decomposed to two sub-models, called *Core* and *Device*. The intention in decomposing *M3* is to decrease the complexity of the produced Event-B sub-models. Also this model decomposition reflects the structure of target architecture by separating the core from the devices. Finally the core sub-model is refined further in two levels of refinement, *M4* and *M5*. During the refinement process both before and after model decomposition, the atomicity decomposition technique is employed to explicitly represent the event sequencing and relationships between abstract and refined events.

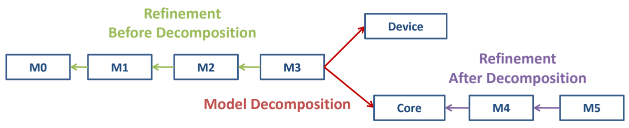


Fig. 1. Development Architecture of Event-B Model

The contribution of this paper is to assess the Event-B atomicity and model decomposition techniques in the development of a complex and large distributed system. In the development process of this case study we will explore how the atomicity decomposition technique will help us to structure refinement steps. After some refinement levels we will see how model decomposition can help us to manage the large model by cutting it into two smaller sub-models. Using the probe system as a carrier, our intention is to identify challenges and provide some solutions by using atomicity and model decomposition techniques. These solutions are presented as assessment results which can lead towards a guideline in using the atomicity decomposition and model decomposition techniques.

This paper is organized into 6 sections. Section 2 outlines the background of this work. Here we overview Event-B method and related techniques, namely atomicity decomposition and model decomposition. In Section 3 we present the Event-B model of the probe system including abstraction and refinement levels. Assessment results are outlined in Section 4 and finally we outline related work in Section 5 and conclude this paper in Section 6.

2 Background

2.1 Event-B and Refinement

The Event-B formal method [2] models the states and events of a system. Variables present the states. Events transform the system from a state to another state by changing the value of variables. The modeling notation is based on set theory and logic. Event-B uses mathematical proof to ensure consistency of a model.

Event-B Structure: An Event-B model [11] is made of several components of these two types, $Context$ and $Machine$. Contexts contain the static part(types and constants) of a model while Machines contain the dynamic part(variables and events). A context can be "extended" by other contexts and "referenced" by machines. A Machine can be "refined" by other machines and reference contexts.

Refinement in Event-B: In Event-B development, rather than having a single large model, it is encouraged to construct the system in a series of successive layers, starting with an abstract representation of the system. The abstract model provides a simple view of the system, focusing on main purposes of the system. The details of how the purposes are achieved are ignored in the abstract specification. Details are added gradually to the abstract model in stepwise manner. This process called refinement [3]. 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 step. 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 $Machine1$.

Event-B Tool: Rodin [9] is an Eclipse-based tool for formal modeling and proving in Event-B. Rodin is an extensible tool that can be extended to include new features.

2.2 Atomicity Decomposition

Although refinement in Event-B provides a flexible approach to modeling, it has the limitation that we cannot explicitly represent the relationship between new events in a refinement and abstract events. To overcome this issue, the atomicity decomposition approach is proposed in [4]. The idea is to augment Event-B refinement with a graphical notation that is capable of representing the relations between abstract and concrete events explicitly. Using the atomicity decomposition approach has another advantage which is that we can represent event sequencing explicitly. An example of an atomicity decomposition diagram is pre-

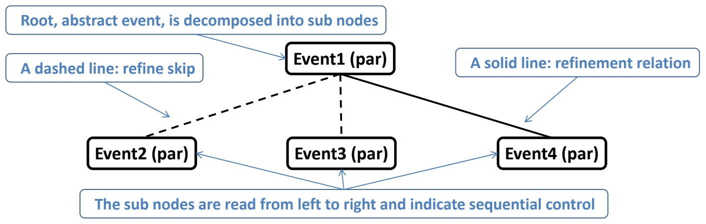
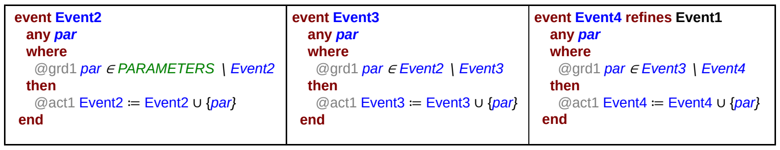

Root, abstract event, is decomposed into sub nodes
A dashed line: refine skip
Event1 (par)
A solid line: refinement relation
Event2 (par)
Event3 (par)
Event4 (par)
The sub nodes are read from left to right and indicate sequential control

Fig. 2. Atomicity Decomposition Diagram

sented in Figure 2. This diagram explicitly illustrates that the effect achieved by $Event1$ at the abstract level is realized at the refined level by occurrence of $Event2$ followed by $Event3$ followed by $Event4$. The execution order of the leaf events is always from left to right (this is based on JSD diagrams of Jackson [5]). We say that $Event1$ is a causal event for $Event2$ since it must occur before $Event2$ and so on. The solid line indicates that $Event4$ refines $Event1$ while the dashed lines indicate that $Event2$ and $Event3$ are new events. In standard Event-B refinement, $Event2$ and $Event3$ do not have any explicit connection with $Event1$. Technically, $Event4$ is the only event that refines $Event1$ but the diagram indicates that we break the atomicity of $Event1$ into three (sub-)events in the refinement.

The parameter par in the diagram indicates that we are modelling multiple instances of $Event1$ and its refining sub-events. Refined sub-events associated with different values of par may be interleaved thus modelling interleaved execution of multiple processes. Further details may be found in [4]. Two more diagrammatic concepts, "XOR case splitting" and "ALL replicator", are used in development of the case study and they will be explained later. Atomicity decomposition has been applied to a distributed file system in [4] and to a multi media protocol in [12]. The Event-B model for the diagram of Figure 2 is pre-



```
event Event2
  any par
  where
   @grd1 par ∈ PARAMETERS ∖ Event2
  then
   @act1 Event2 ≔ Event2 ∪ {par}
 end
```

```
event Event3
  any par
  where
   @grd1 par ∈ Event2 ∖ Event3
  then
   @act1 Event3 ≔ Event3 ∪ {par}
 end
```

```
event Event4 refines Event1
  any par
  where
   @grd1 par ∈ Event3 ∖ Event4
  then
   @act1 Event4 ≔ Event4 ∪ {par}
 end
```

Fig. 3. Event-B Model

sented in Figure 3. The effect of a refined event with parameter par is to add the value of par to a set with the same name as the event, i.e., $par \in Event1$ means that $Event1$ has occured 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 occured for value *par* but has occured for the causal event, e.g., the guard of *Event*3 says that *Event*2 has occurred and *Event*3 has not occurred for value *par*.

2.3 Model Decomposition

The motivation for model decomposition [6], [7] is to decrease the complexity of large models, increase the modularity and reflect the target architecture. After several layers of refinement and as a result of introducing new events, we can end up having to deal with many events and many state variables. The main idea of decomposition is to cut a model into sub-models which can be refined separately and more easily than the initial model. Independent sub-models provides the possibility of team development which seems a very attractive option for the industry.

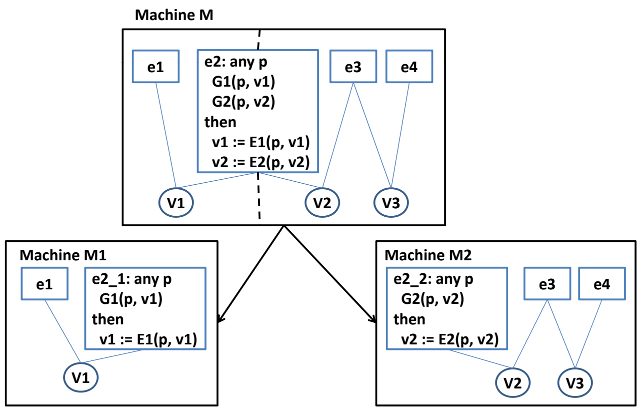


Fig. 4. Model Decomposition, Shared-event Style

In Event-B there are two ways of decomposing a model, shared-variable and shared-event. The shared-event approach is particularly suitable for message-passing in distributed systems, whereas the shared-variable approach is more suitable for concurrent systems. Since the probe system is a distributed system we use the shared-event approach in decomposing its model after three levels of refinement. In the shared-event model decomposition, variables are partitioned among the sub-models, whereas in shared-variable approach, events are partitioned among the sub-models. Shared-event model decomposition is presented graphically in Figure 4. First variables of the initial model M are partitioned among sub-models $M1; ...; Mn$ according to the devised policy. Then events of the initial model, M, are distributed among sub-models $M1; ...; Mn$, according to the variable partitioning. Events that are using variables allocated to different sub-models, called shared events, must be split between these sub-models. For example event $e2$ uses both $v1$ and $v2$ which are going to different sub-models.

Therefore as depicted we have split it to $e2\_1$ and $e2\_2$ corresponding to variable $v1$ and $v2$ respectively. In the next stages the sub-models can be refined independently.

3 Event-B Model of the Probe System

3.1 An overview of System Requirements and Development Process

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.
- For some particular types of *TC*, one or more data *TM*s are generated and sent back to Earth.

As mentioned earlier, we only present the part of the probe system that handles TeleCommands and TeleMessages communications. In Figure 1 of Section 1 we diagrammatically presented the development process of the probe system in Event-B. The development process consists of:

- Machine *M0* models the goal of the probe system. Three main events are receiving a *TC*, validating the received *TC*, and generating one or more *TM(s)* if it is needed.
- In machine *M1* the validation phase is refined and further details of validation process are added.
- In machine *M2* we distinguish between validation checking of *TC*s that should be carried out by the core or the devices.
- In machine *M3* we refine the model to introduce the process of sending related *TC*s to the devices for further validation and processing.
- Machine *M4* and *M5* model producing and sending *TM*s carried out in the core.

3.2 Abstract Specification

In the abstract model, the main goal of the system is modeled. Abstract events are illustrated in Figure 5 with a diagram resembling an atomicity decomposition diagram. Note that the top box is the system name rather than an event name (as the case in an atomicity decomposition diagram). In addition to this we only use solid lines to show the events of the abstract specification. After receiving a TC, three different scenarios are possible. Scenario(a): the received TC is validated and in response to this TC, it is necessary to produce some data. This is achieved by the occurrences of the third event. The response is sent back to Earth in the form of some data TMs by the occurrences of the fourth event. Scenario(b): for some TC's type there is no need to generate data TMs in response. Producing a control TM is later done by refining the $TC\_Validation\_Ok$ event. Scenario(c): it shows the case that the validation of a received TC fails. This is modeled by $TC\_Validation\_Fail$ event.

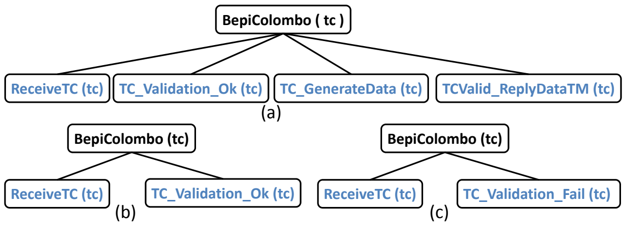
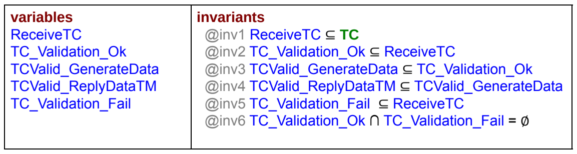


Fig. 5. Abstract Events, Machine $M0$

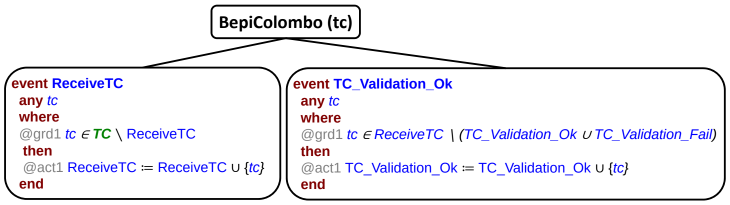
The sequencing between events is specified by following the rules explained in Section2.2. In abstract machine, *M0*, there are five sets used as control variables. Using sets allows multiple instance of a TC to be processed concurrently in an interleaved fashion. Figure 6 shows variables and invariants of *M0*. For each event there is a variable with the same name as the event, and if one event appears after another one in the sequence, its variable is a subset of the variable associated with the former. For example, as described before $TC\_Validation\_Ok$ event can occur only after occurrence of $ReceiveTC$ event, so invariant *inv2* describes $TC\_Validation\_Ok$ variable as a subset of $ReceiveTC$ variable.



variables	invariants
ReceiveTC	@inv1 ReceiveTC ⊆ TC
TC_Validation_Ok	@inv2 TC_Validation_Ok ⊆ ReceiveTC
TCValid_GenerateData	@inv3 TCValid_GenerateData ⊆ TC_Validation_Ok
TCValid_ReplyDataTM	@inv4 TCValid_ReplyDataTM ⊆ TCValid_GenerateData
TC_Validation_Fail	@inv5 TC_Validation_Fail ⊆ ReceiveTC
	@inv6 TC_Validation_Ok ∩ TC_Validation_Fail = ∅

Fig. 6. Variables and Invariants of the Abstract Machine $M0$

To enforce the exact ordering of Figure 5.(a), when a TC is received we add it to the variable $ReceiveTC$ of the $ReceiveTC$ event. This event is represented in Figure 7. The guard of $TC\_Validation\_Ok$ event means that only after this stage, it is possible for the $TC\_Validation\_Ok$ event to occur and to add this TC to the list of validated TCs.



```
event ReceiveTC
  any tc
  where
  @grd1 tc ∈ TC \ ReceiveTC
   then
   @act1 ReceiveTC ≔ ReceiveTC ∪ {tc}
  end
```

```
event TC_Validation_Ok
  any tc
  where
  @grd1 tc ∈ ReceiveTC \ (TC_Validation_Ok ∪ TC_Validation_Fail)
  then
  @act1 TC_Validation_Ok ≔ TC_Validation_Ok ∪ {tc}
  end
```

Fig. 7. Event-B Model of Sequencing between Events of the Abstract Machine $M0$

3.3 First Level of Refinement: Introducing Validation Steps

In the abstract model, the validation process is carried out in a single stage. The outcome can be either ok or fail which is modeled by *TC_Validation_Ok* and *TC_Validation_Fail* events. However validating a received *TC* is not an atomic action, accomplished in a single stage. It is done in two steps, checking the syntax and semantic of a received *TC*. After syntax and semantic checks, in the third step a control *TM* is produced and sent. These details are modeled in the first level of refinement, named machine *M1*. It can be seen in Figure 8 that *TC_Validation_Ok* and *TC_Validation_Fail* are decomposed to sub-events which show further details of the validation process. Checking the syntax of a received *TC* is modeled by *TCCheck_Ok* and *TCCheck_Fail* events. The semantic checking is modeled by *TCExecute_Ok* and *TCExecute_Fail* events. *TCExecOk_ReplyCtrlTM*, *TCExecFail_ReplyCtrlTM* and *TCCheckFail_ReplyCtrlTM* are events for generating control *TM*s. Again the Event-B model can be produced following the rules explained in Section2.2.

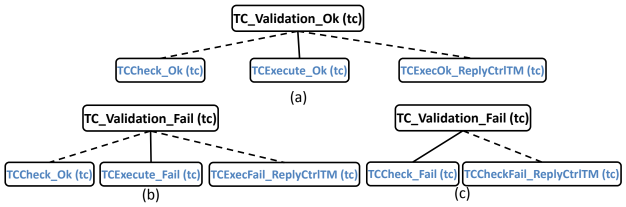


Fig. 8. Atomicity Decomposition of Validation Events, Machine $M1$

For each solid line in atomicity decomposition diagram there is an invariant which shows the relation between the set variable corresponding to abstract event and concrete variable of the refined event. There are three invariants in machine *M1*, shown in Figure 9. For example, *inv9* shows that concrete variable of *TCExecute_Ok* is a subset of abstract variable of *TC_Validation_Ok*, since the *TCExecute_Ok* event refines *TC_Validation_Ok* event.

```
@inv9 TCExecute_Ok ⊆ TC_Validation_Ok
@inv10 TCExecute_Fail ⊆ TC_Validation_Fail
@inv11 TCCheck_Fail ⊆ TC_Validation_Fail
```

Fig. 9. Invariants, Machine $M1$

3.4 Second Level of Refinement: Distinguish between Different Types of TCs

In this stage we are in a position to distinguish between two different types of TCs. There are TCs that should be handled by the core, called *csw* TCs, and TCs that should be sent from the core to the devices, (*mixsc, mixst, sixsp, sixsx*), to be processed. To model this new aspect, we define a new function called PID which maps every TC either to the core or the devices.

So far semantic checking of a received TC is done regardless of considering the type of TC. Now that we have the distinction between the core TCs and the devices TCs. If a received TC belongs to the core, its semantic should be checked in the core, otherwise it should be sent to a one of the devices for validation and processing. It is helpful to emphasis that syntax checking is exclusively carried out in the core.

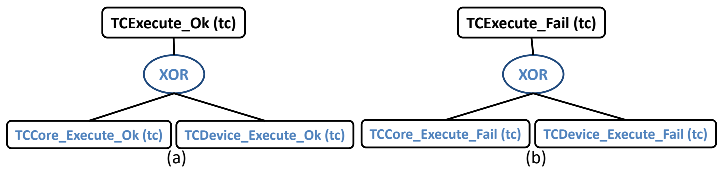


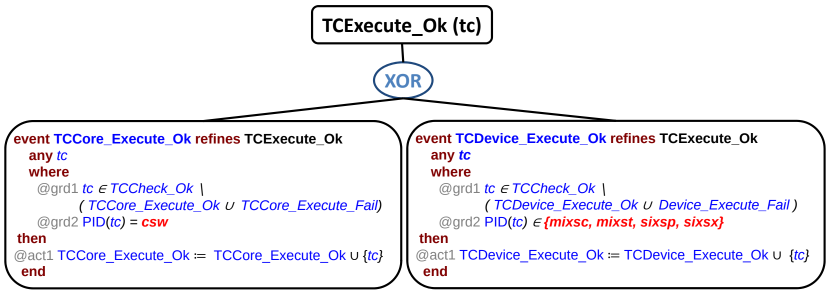
Fig. 10. Case Splitting, Machine $M2$

To model different cases associated with different types of TCs, both *TCExecute_Ok* event and *TCExecute_Fail* event are split into two sub-events. The splitting of these events, illustrated in Figure 10, is carried out using a special construct, called XOR or case splitting. In case splitting, an event is split into some sub-events in a way that only one of them is executed. As it can be seen in Figure 10, XOR, case splitting is graphically represented by a circle containing an "XOR". We draw the attention of the reader to the fact that XOR refers to mutual exclusion of events' execution, but guards of events do not need to be disjoint.

Figure 11 presents the Event-B model of Figure 10.(a). Note that both sub-events refine the abstract event. In the both sub-events we have added a new guard, *grd2*, which check the type of TCs.

3.5 Third Level of Refinement: Refining TCs Processing by the Devices

In the previous level we introduced the distinction between two types of TCs that are processed by the core and the devices respectively. In this level our aim is to refine the case of processing TCs by the devices. As presented in Figure 12, we applied the atomicity decomposition approach to three events of the previous level. By introducing communication between the core and devices, the abstract event, *TCDevice_Execute_ok*, is refined to *SendTC_Core_to_Device*, *CheckTC_in_Device_Ok* and *SendOkTC_Device_to_Core* events. These three events



```
event TCCore_Execute_Ok refines TCExecute_Ok
   any tc
   where
     @grd1 tc ∈ TCCheck_Ok \
            ( TCCore_Execute_Ok ∪ TCCore_Execute_Fail)
     @grd2 PID(tc) = csw
 then
@act1 TCCore_Execute_Ok ≔ TCCore_Execute_Ok ∪ {tc}
 end
```

```
event TCDevice_Execute_Ok refines TCExecute_Ok
   any tc
   where
     @grd1 tc ∈ TCCheck_Ok \
            ( TCDevice_Execute_Ok ∪ Device_Execute_Fail )
     @grd2 PID(tc) ∈ {mixsc, mixst, sixsp, sixsx}
 then
@act1 TCDevice_Execute_Ok ≔ TCDevice_Execute_Ok ∪ {tc}
 end
```

Fig. 11. Event-B Model, Machine $M2$

model the case where a TC is successfully processed by a device and some response is generated for the core. In Figure 12.(b) a very similar approach is followed for the case when processing of a TC fails in a device and the atomicity of the abstract event, *TCDevice_Execute_Fail*, is decomposed to three sub-events based on the atomicity decomposition rules. Note that in Figure 12.(a) and (b) the event with the solid line, which directly refines the abstract event, appears in the middle rather than being the last one. Finally in Figure 12.(c), we show how *TCValid_GenerateData* is refined into two events to represent the case where extra data is produced in response to a TC.

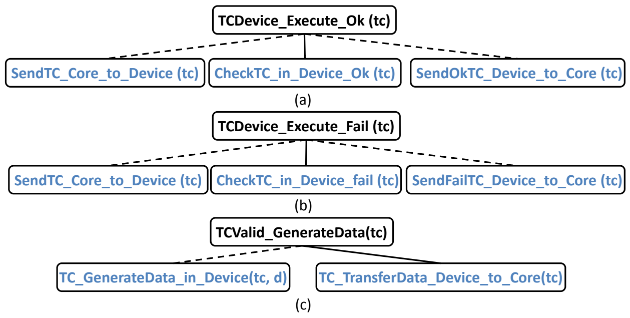

TCDevice_Execute_Ok (tc)
SendTC_Core_to_Device (tc)
CheckTC_in_Device_Ok (tc)
SendOkTC_Device_to_Core (tc)
(a)
TCDevice_Execute_Fail (tc)
SendTC_Core_to_Device (tc)
CheckTC_in_Device_fail (tc)
SendFailTC_Device_to_Core (tc)
(b)
TCValid_GenerateData(tc)
TC_GenerateData_in_Device(tc, d)
TC_TransferData_Device_to_Core(tc)
(c)

Fig. 12. Atomicity Decomposition Diagrams, Machine $M3$

3.6 Decomposing the Probe Model to the Core and Devices Sub-models

So far by applying atomicity decomposition in a few consecutive steps, we have managed to distinguish between events of the core and devices. Also we have reached the stage that we have a big model consisting of several events and many variables. Therefore it is a good time to take the next step and by applying the model decomposition, divide our current Event-B model to two sub-models, namely core and devices. When it comes to model decomposition we can identify three types of events, events that belong to the core or the devices or events that are shared between them. Shared events usually represent communication links and they should be split between the core and devices sub-models.

In Figure 13 shared events are presented using rectangles and variables are presented using ovals. For instance, *SendTC_Core_to_Device* event uses *TCCheck_ok* variable from the core sub-model and *SendTC_Core_to_Device* from the devices sub-model. Therefore it should be slit between these sub-models.

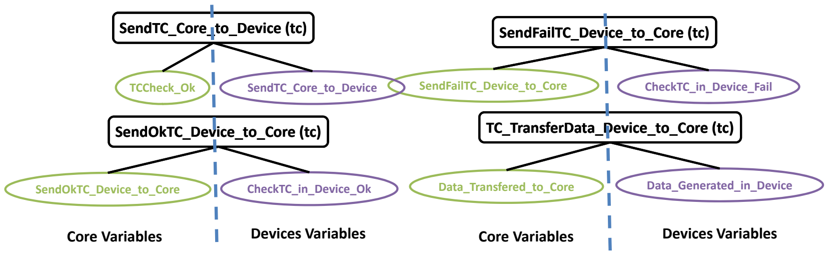


Fig. 13. Shared Events

3.7 Further Refinements of the Core Sub-model

After decomposing our intermediate Event-B model to two sub-models, we have carried out two further refinement of the core sub-model as depicted in Figure 1. These refinements introduce some details about how TMs are produced in response to TCs. We have omitted details of these refinements. Figure 14 presents the *TCValid_ReplyCtrlTM* event and its two consecutive levels of atomicity decomposition. This is modeling the case where a TC has successfully processed and in response some data TMs should be produced and sent back to Earth.

Here using Figure 14 an extra atomicity decomposition concept is explained. In response to a TC, it is possible to produce more than one data TM. To model such a situation we have used a construct [12] called "ALL replicator" applied to $TCValid\_ProcessDataTM$ event. The ALL, parameterized by tm, means that $TCValid\_ProcessDataTM$ occurs for multiple values of tm and the $TCValid\_CompleteDataTM$ can only occur when all the values of tm associated with a tc have occurred. In Event-B we model this by adding a parameter, which is a set containing all possible TMs that should be produced in response to a TC.

Another interesting aspect in Figure 14 is the sequencing order between leaf events. Based on the atomicity decomposition rules, *Produce_DataTM* event should be completed before *TCValid_CompleteDataTM* event. However there is no sequencing enforced between *Send_DataTM* and *TCValid_CompleteDataTM* events. This means that sending TMs to Earth can be carried out before or after occurrence of *TCValid_CompleteDataTM* event. This concept is discussed in more detail in [12].

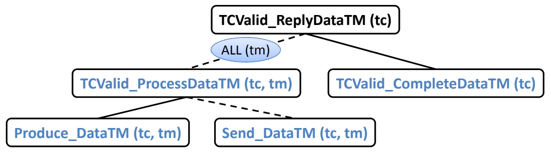


Fig. 14. "*ALL*" Construct, the Core Sub-Model

4 Assessment

In this section we discuss how the atomicity and model decomposition techniques helped us in enhancing the development process of the probe system. We also explain notable effects of these techniques in term of methodological contribution that can form a basis for a set of future guidelines. As a part of our formal modeling, we have developed a substantial set of Event-B models including three levels of refinement before model decomposition and two levels of refinement after it. In total the Rodin tool produced 174 proofs, 158 of them discharged automatically. The remaining proofs are discharged interactively. Atomicity decomposition diagrams enabled us to explore and explain our formal development without getting into technical details of the underlying Event-B models. We consider this as an advantage of the atomicity decomposition technique. The next important advantage of this technique is that we can explicitly represent refinement relations between events of different levels. Another merit of atomicity decomposition technique is the capability of representing sequencing between events of the same model. Further aspects are discussed in the following sections.

4.1 Providing Insight for Appropriate Event Decomposition

During the development process, atomicity decomposition diagrams helped us to spot some flaws in our decomposition approach. For example if the adapted approach did not cover all desired scenarios, we managed to discover this from the diagrams before attempting to produce any Event-B code.

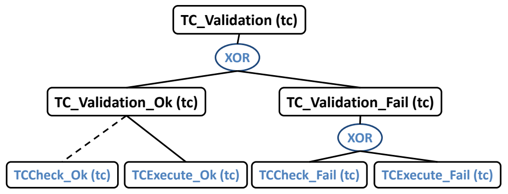


Fig. 15. An Example of Wrong Atomicity Decomposition

To clarify this further, in Figure 15 we present one possible way of decomposing the atomicity of TC validation process. Applying two successive levels of atomicity decomposition to the abstract event $TC\_Validation$ results in four subevents. The diagram shows that the possible scenarios are: <TCCheck_OK(tc)

and TCExecute_OK(tc)> or <TCCheck_Fail(tc)> or <TCExecute_Fail(tc)>. Clearly this approach does not cover the case where *TCCheck_Ok* and *TCExecute_Fail* events can happen together as described in Section 3.3. This helped us to go back to the abstract level and followed an appropriate of atomicity decomposition which was presented in Section 3.3.

4.2 Assessing the Influence of Atomicity Decomposition and Model Decomposition over each other

In this case study we have used both atomicity decomposition and model decomposition together. One interesting aspect is to investigate whether by analyzing atomicity decomposition diagrams a decision can be made on a proper point that model decomposition can be applied. Atomicity decomposition diagrams provide an overall visualization of the refinement process. By grouping relevant events together, it is easier to decide about the point at which we can apply model decomposition.

Usually when we develop a system, we have a target architecture in mind. Therefore the outcome of the model decomposition should give us the desired sub-models. To be able to decompose an Event-B model, all events should either belong to one of sub-models or otherwise they should model communication links between its sub-models. In this regard model decomposition can provides us with some hint to which events, atomicity decomposition should be applied as a preparation stage for model decomposition.

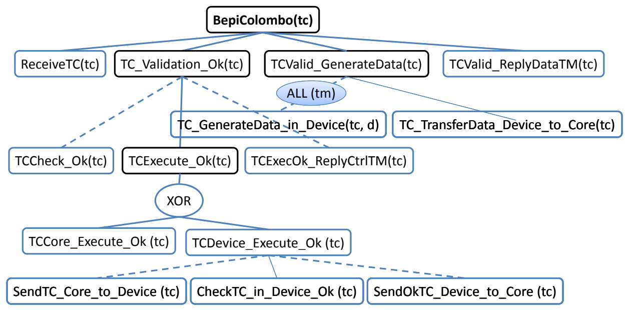


Fig. 16. Overall Refinement Structure before Model Decomposition

To clarify this aspect we use a part of development process presented in Figure 16. As a preparation for model decomposition, we have applied atomicity decomposition to events such as *TCExecute_Ok* to distinguish between functionality of the core and devices. Note that leaf events satisfy the pre-mentioned condition that either should belong to one of the sub-models or represent communication links.

5 Related Work

The desire to explicitly model control flow is not restricted to Event-B. To address this issue usually a combination of two or more formal methods are suggested. A good example of such approach is Circus [14] combining CSP [15] and Z [16]. The combination of CSP and Classical B [8] also has been investigated in [17] and [18] with some differences. To explicitly define event sequencing in Event-B the Flows Approach is suggested in [19]. Another method to provide explicit control flow for an Event-B model is presented in [20] which is again based on using CSP alongside Event-B. These methods only deal with event sequencing; they do not support the explicit refinement of atomicity decomposition diagrams. UML-B [21] provides a "UML-like" graphical front-end for Event-B. It adds support for class-oriented and state machine modeling. State machines provide us with a graphical notation to explicitly define event sequencing.

Atomicity decomposition approach provides a graphical front-end to Event-B along other features such as supporting event sequencing and expressing refinement relations between concrete and abstract events. Also it can be combined effectively with other techniques such as model decomposition.

6 Conclusion

In this paper we demonstrated how atomicity decomposition diagrams provide a systematic means of introducing control structure into the Event-B development process. It also provides a means to express refinement relations between events of different refinement levels, through a set of hierarchal diagrams. In addition it can be merged with model decomposition technique to manage the complexity of large models. We have done an assessments of this approach and some merits of it explained in the previous section. In future work we hope that the outcomes of this stage can contribute toward providing some guidelines for atomicity and model decomposition. During the development of this case study, translation from atomicity decomposition diagrams to Event-B was carried out manually. As a continuation of this work, currently we are working on a tool providing support for producing atomicity decomposition diagrams as well as translating them to Event-B. This tool will be developed as a plug-in for the Rodin toolset.

References

1. Jean-Raymond Abrial: Formal Methods: Theory Becoming Practice. J.UCS 13(5): 619-628, (2007)
2. Jean-Raymond Abrial: Modeling in Event-B: System and Software Engineering. Cambridge University Press, (2010)
3. Jean-Raymond Abrial: Refinement, Decomposition and Instantiation of Discrete Models. In Abstract State Machines, pages 17–40, (2005)
4. Michael Butler: Decomposition Structures for Event-B. In Integrated Formal Methods iFM2009, volume LNCS 5423. Springer, (2009)
5. M.A Jackson: System Development. Prentice-Hall, Englewood Cliffs (1983)

6. Renato Silva, Carine Pascal, T.S. Hoang and Michael Butler: Decomposition Tool for Event-B. ABZ, (2010)
7. Carine Pascal, Renato Silva: Event-B Model Decomposition. DEPLOY Plenary Technical Workshop, (2009)
8. Jean-Raymond Abrial: The B-book: Assigning Programs to Meanings. Cambridge University Press, (1996)
9. Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta and Laurent Voisin: Rodin: An Open Toolset for Modelling and Reasoning in Event-B. Technical Report, DEPLOY Project, http://deploy-eprints.ecs.soton.ac.uk/130/, (2009)
10. Ralph-Johan Back, Reino Kurki-Suonio: Distributed Cooperation with Action Systems. ACM Trans. Program. Lang. Syst., 10(4):513–554, (1988)
11. Stefan Hallerstede: Justifications for the Event-B Modelling Notation. In B, volume LNCS 4355, pages 49–63. Springer, (2007)
12. Asieh Salehi Fathabadi, Michael Butler: Applying Event-B Atomicity Decomposition to a Multi Media Protocol. In FMCO Formal MEthods for Components and Objects, (2010)
13. ESA Media Center, Space Science. Factsheet: Bepicolombo. http://www.esa.int/esaSC/120391_index_0_m.html
14. F. Zeyda, A. Cavalcanti: Mechanised Translation of Control Law Diagrams into Circus. In M. Leuschel and H. Wehrheim, editors, Integrated Formal Methods, volume 5423 of LNCS, pages 151-166, Springer, (2009)
15. C. A. R. Hoare: Communicating Sequential Processes. Prentice Hall. ISBN 0-13-153289-8, (1985)
16. Jim Davies, Jim Woodcock Using Z: Specification, Refinement and Proof. Prentice Hall International Series in Computer Science. ISBN 0-13-948472-8, (1996)
17. Michael Butler: csp2B: A Practical Approach to Combining CSP and B. Formal Aspects of Computing, vol. 12, pp. 182-196, ISSN 0934-5043, (2000)
18. S. Schneider, H. Treharne: Verifying Controlled Components. In In Proc IFM, Springer, pp. 87-107, (2004)
19. Iliasov, Alexei: Tutorial on the Flow plugin for Event-B. In: Workshop on B Dissemination [WOBD] Satellite event of SBMF, Natal, Brazil, November 8th-9th 2010
20. Steve Schneider, Helen Treharne and Heike Wehrheim: A CSP Approach to Control in Event-B. IFM, pages 260-274, (2010)
21. M. Y. Said, M. Butler, 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)