Contents

Chapter 1

Atomicity Decomposition Part 1 - Overview and Background

1.1 Introduction

The atomicity decomposition approach was first introduced by Butler in [1]. In this chapter we present the atomicity decomposition approach from [1], in Section 1.2. A major contribution of atomicity decomposition approach is structuring refinement in Event-B. To highlight this contribution, Section 1.3 outlines the role of atomicity decomposition diagrams in structuring refinement in Event-B. It is followed by two examples of the atomicity decomposition application from [1], in Section 1.4.

1.2 Overview of Atomicity Decomposition Diagram in Event-B

Although the refinement approach in Event-B provides a flexible approach to modelling, it does not have the ability to show the relationship between one abstract event and the corresponding concrete events. The atomicity decomposition approach is intended to make the relationships between abstract and concrete events clearer and easier to manage than simply using the standard Event-B refinement technique. In this approach course-grained atomicity can be refined to more fine-grained atomicity.

The tree structure notation of the atomicity decomposition approach is first introduced by Butler in [1]. The diagrammatic notation is based on JSD structure diagrams by Jackson [2]. In [1] the atomicity decomposition diagram is presented in two examples containing a parallel execution of an event. Before introducing the parallel notation, we generate a simple view of the atomicity decomposition diagram in order to explain the basic features. It is shown in Figure 1.1. The features explained here are from [1].

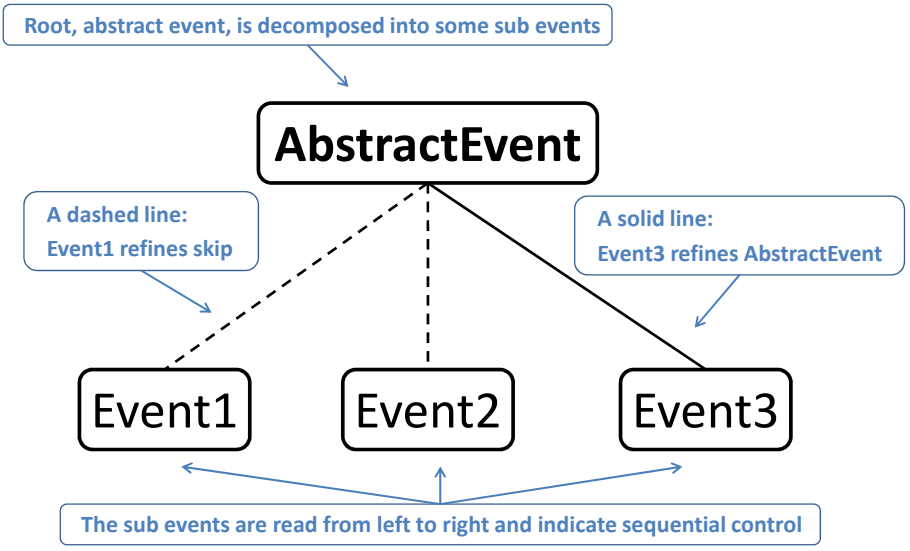


Figure 1.1: Atomicity Decomposition Diagram

The abstract atomic event, *AbstractEvent*, appears in the root node. The diagram shows how the root is decomposed into some sub-events in the refinement model. The number of sub-events can be one or more. In this case we consider three sub-events to explain the features of the diagram. An important feature of diagram, in common with JSD structure diagrams, is that the sub-events are read from left to right and indicate sequential control from left to right. This means that our diagram indicates that the abstract event is realised in the refinement by firstly executing *Event1*, then executing *Event2* and then executing *Event3*.

Sub-events are treated in two ways, one refines abstract event and the others are viewed as hidden events in the abstract model which refine *skip* in the refinement model. So another important feature is types of lines, solid line and dashed line. The sub-events corresponding to dashed lines, *Event1*, *Event2*, are new events which refine *skip* in the abstract model. The sub-event with a solid line, *Event3*, is a refining event which must be proven to refine the abstract event, *AbstractEvent*. A new event introduced in the refinement model which refines *skip*, can be viewed as a hidden event in the abstract model. This kind of event is not visible to the environment of a system in the abstract model, and therefore they are outside the control of the environment [1].

In this case, *Event1* should execute before *Event2*. Also *Event2* should execute before *Event3*. This is done by some control variables in the refinement model. We will see more about control variables later in this chapter.

With the aim of making the point more clear, the possible execution traces of the model, called event trace [1], are presented here.

The execution trace of the abstract model contains a single event and is represented as $< AbstractEvent >$. The execution trace of the refinement model events, *Event1*, *Event2* and *Event3*, is $< Event1, Event2, Event3 >$.

1.3 Event-B Refinement and Atomicity Decomposition Diagrams

One of the important motivations of the atomicity decomposition approach is that it explicitly shows the event ordering and the relationship between an abstract event and the corresponding concrete events, whereas the Event-B text is not able to explicitly show these properties. This can be seen by comparing Figure 1.2 and Figure 1.3.

Assume Event *E21* should execute before event *E22*. And event *E22* should execute before event *E23*. Considering Figure 1.2, the ordering between these events is *implicit*. Whereas the atomicity decomposition diagram in Figure 1.3, *explicitly* shows the event ordering by a sequence execution of events from left to right.

```
events

 event E21
  where
    @grd1 VarE21 = FALSE
  then
    @act1 VarE21 ≔ TRUE
 end

 event E22
  where
    @grd1 VarE21 = TRUE
    @grd2 VarE22 = FALSE
  then
    @act1 VarE22 ≔ TRUE
 end

 event E23 refines E1
  where
    @grd1 VarE22 = TRUE
    @grd2 VarE23 = FALSE
  then
    @act1 VarE23 ≔ TRUE
 end

end
```

Figure 1.2: Event-B Model of Atomicity Decomposition Diagram in Figure 1.3

Considering Figure 1.2, the ordering is implicitly specified by some control variables in the Event-B model. *VarE21*, *VarE22* and *VarE23* are boolean control variables which are initialised to *FALSE*. First event *E21* executes and enables *VarE21* variable. Event *E22* is guarded by *VarE21* variable, *grd1*. Therefore event *E22* can execute only after

event *E21* executes. Also event *E23* is guarded by *VarE22*, *grd1*. So event *E23* can execute only after event *E22* executes.

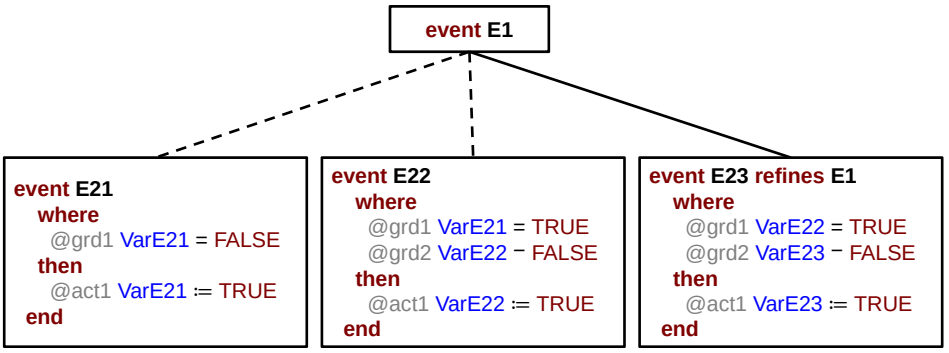


Figure 1.3: Atomicity Decomposition Diagram of Event-B Model in Figure 1.2

Moreover the diagram explicitly illustrates our intention that the effect achieved by event *E1* at the abstract model is realized at the refinement model by execution of event *E21* followed by event *E22* followed by event *E23*, Figure 1.3. Whereas in the standard Event-B model, Figure 1.2, events *E21* and *E22* are refinements of *skip* and there is no explicit connection to abstract event *E1*. Technically, event *E23* is the only event that refines event *E1* but the diagram indicates that we break the atomicity of abstract event *E1* into three sub-events *E21*, *E2* and *E23*.

1.4 Examples of Application

With the aim of making the application of atomicity decomposition diagrams more clear, two examples from [1] are presented here.

Assume the abstract machine contains a single event *Out*, that simply outputs *N* exactly for one time. Considering Figure 1.4, there is only one boolean control variable in the machine, called *Out*, which initialised to false. *Out* event can execute only when it has not executed before, *grd1*. In execution it disabled itself, *act1*. The output value is represented in the parameter *v*, *grd2*.

The output is produced in an atomic event in the abstract machine. We wish to refine the abstract machine by a machine modelling a concurrent accumulation of the output value before outputting it. The refinement structure is presented in an atomicity decomposition diagram in Figure 1.5. The diagram shows that we break the atomicity of abstract *Out* event, to three sub-events. This means that the abstract *Out* event is realised in the refinement by firstly executing the initialisation, then executing the *Increase* event in parallel and then executing *Out* event. The parallel execution here is illustrated with a circle containing "all" and name of a parameter. We call it all-replicator, since it replicates the corresponding sub-events with a new parameter, *p*, and

```
machine M0
variables Out
invariants
  @inv1 Out ∈ BOOL

events
  event INITIALISATION
    then
      @act1 Out ≔ FALSE
  end

  event Out
    any v
    where
      @grd1 Out = FALSE
      @grd2 v = N
    then
      @act1 Out ≔ TRUE
  end

End
```

Figure 1.4: Abstract Model of an Outputting System

Increase event needs to executes for all instances of parameter p before *Out* event execution. Figure 1.5 is slightly different to what Butler presented in [1]. Butler illustrates the parallel execution with a circle containing "par(p)". Since we have improved the atomicity decomposition notations, which will be presented in Chapter 2, we found it more understandable if the diagram presented here is compatible with the improvement of notations in Chapter 2.

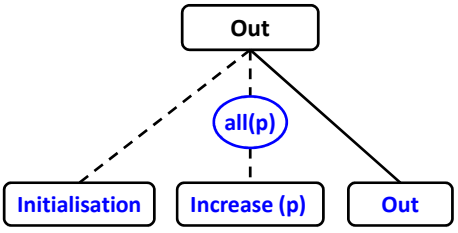


Figure 1.5: Atomicity Decomposition Diagram of an Outputting System

The Event-B model of the refinement machine is presented in Figure 1.6. Each parallel execution of *Increase* event, increments the variable x exactly once. When all N sub-events have incremented x, the value of x is output with execution of *Out* event.

Consider the case where we have two subprocesses, $PROC = \{p1, p2\}$, and $N = 2$. The event traces of the refinement model are as below:

$< Initialisation,\ Increase(p1),\ Increase(p2),\ Out(2) >$
$< Initialisation,\ Increase(p2),\ Increase(p1),\ Out(2) >$

The two possible interleaving of *Increase(p1)* and *Increase(p2)*, represented by two events traces, model their concurrent execution.

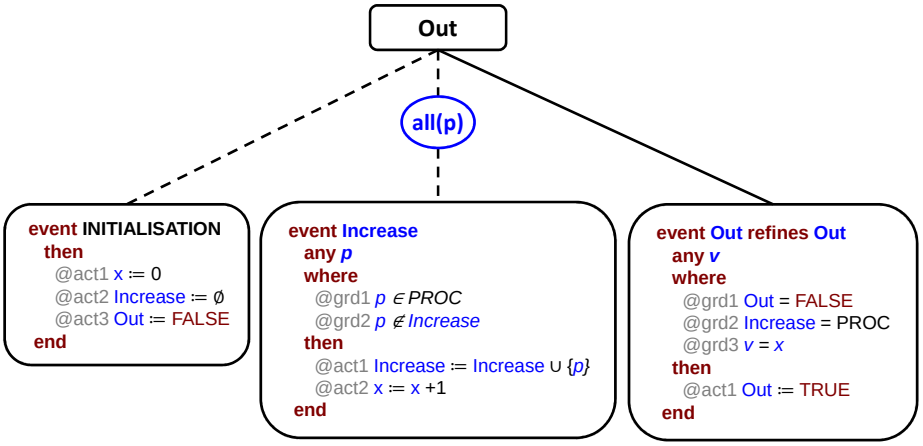


Figure 1.6: Event-B Refinement of an Outputting System

As presented in the first example, *Out* event needs to execute only for one time. Therefore we defined the control variable, *Out*, as a boolean variable, which is disabled in the body of *Out* event after the first execution. Whereas sometimes we wish to model a sequence of events which can execute more than one time for different instances of one or more parameters. Second example presents this case. Later in Chapter 2, first case is called Single Instance (SI) and second case is called Multiple Instance (MI). The type of control variables are different in SI and MI. Considering SI, as seen in first example, control variables are boolean, whereas in the MI case, control variables are sets. Having set type enables multiple instances of an event and event interleaving.

As the second example, consider the atomicity decomposition diagram of a file write system in Figure 1.7. The atomicity of the abstract *Write* event is break to three sub-events in the refinement machine, in order to model the writing of individual pages, *PageWrite* event. The writing of the entire file is no longer atomic. The writing of a file is initiated by *StartWrite* event and ended by *EndWrite* event. Multiple file writes are allowed to be taking place simultaneously in an interleaved fashion. This is indicated by a parameter provided in abstract *Write* event, *f*, and inherited with all sub-events. Also in the refinement model, the pages of an individual file f can be written in parallel hence an all-replicator over *PageWrite* event replicates its parameter with p.

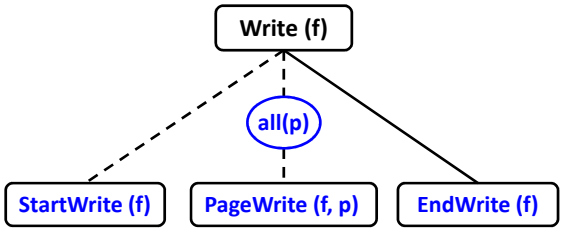


Figure 1.7: Atomicity Decomposition Diagram of File Write

The control variables are sets and the invariants to model event sequencing implied in Figure 1.7 are presented in Figure 1.8. *StartWrite* is a subset of *FILE*, because it is

bounded by parameter *f*, (*inv1*). *PageWrite* is a subset of $FILE \times PAGE$, because it is bounded by parameter *f* and all-replicator parameter *p*, (*inv2*). If a page has been written for a file, then *StartWrite* will already have executed for the file, (*inv3*).

```
invariants
  @inv1 StartWrite ⊆ FILE
  @inv2 PageWrite ⊆ FILE × PAGE
  @inv3 dom(PageWrite) ⊆ StartWrite
```

Figure 1.8: Invariants of File Write Refinement Model

The Event-B model of *StartWrite* and *PageWrite* events are presented in Figure 1.9. The event sequencing is managed with some guards. *PageWrite* is guarded with *StartWrite*, *grd1*, which indicates ordering between *StartWrite* event and each *PageWrite* event.

```
event StartWrite
  any f
  where
    @grd1 f ∈ file
    @grd2 f ∉ StartWrite
  then
    @act1 StartWrite ≔ StartWrite ∪ {f}
end

event PageWrite
  any f p
  where
    @grd1 f ∈ StartWrite
    @grd2 f ↦ p ∉ PageWrite
  then
    @act1 PageWrite ≔ PageWrite ∪ { f ↦ p }
end
```

Figure 1.9: Event-B Model of File Write

The accurate explanation of Event-B model derived from atomicity decomposition diagrams are presented in a pattern based style in Chapter 2. In this section, by using some examples, we tries to make the overall benefits of the atomicity decomposition approach more clear.

1.5 Conclusion

This chapter introduced the atomicity decomposition diagram notation. We have outlined how atomicity decomposition diagrams help to structure refinement in Event-B by showing the relationships between events of different refinement levels and by providing an explicit visual view of the ordering between events. Each node presents one event. The root node contains the name of an abstract event and the child nodes contain the names of concrete sub-events. A refining relationship between an abstract event and a concrete event is indicated with a solid line in the diagram between these two event

nodes, and a non-refining relationship is indicated with a dashed line. The ordering between events is indicated with a sequence from left to right in the diagram.

To make the application of atomicity decomposition diagrams more clear and to highlight the benefits of atomicity decomposition diagrams in structuring refinement, two examples have been outlined. First example covers the case when a single instance (SI) of event executions is need, whereas the second one shows the multiple instance (MI) case.

This chapter presented background material required to understand the atomicity decomposition patterns in Chapter 2.

Chapter 2

Atomicity Decomposition Part 2 - Patterns and Features

2.1 Introduction

The features of the atomicity decomposition approach in [1] are introduced in Chapter 1. Using these features we have developed two case studies. These developments helped us to improve and expand the atomicity decomposition approach by discovering new constructors and features. This chapter presents the constructor patterns and features in Section 2.2 and Section 2.3 respectively. Each pattern outlines the intention and diagrammatic notation of a decomposing constructor and the way that it is encoded in the Event-B model.

2.2 Atomicity Decomposition Diagram Patterns

2.2.1 Introduction

This section presents the atomicity decomposition constructors in a pattern-based style. Each pattern outlines one constructor in one level of refinement.

In the atomicity decomposition approach, we found some common and reusable constructors (as solutions) to some common intentions (as problems). These recurring problem-solution pairings motivated us to use a pattern-based approach to introduce the atomicity decomposition constructors. Moreover organizing the problems and solutions in a pattern-based approach is easy to read, understand and apply.

In total, eight constructor patterns have been delineated. The constructor patterns are divided to four distinct groups:

- Sequence pattern, Section 2.2.2.
- Loop pattern, Section 2.2.3.
- Logical constructor patterns: and-constructor, Section 2.2.4, or-constructor, Section 2.2.5, xor-constructor, Section 2.2.6.
- Replicator patterns: all-replicator, Section 2.2.7, some-replicator, Section 2.2.8, one-replicator, Section 2.2.9.

The logical constructors, including the and-constructor, the or-constructor and the xor-constructor, introduce logical relations between two or more sub-events.

Each replicator constructor, including the all-replicator, the some-replicator and the one-replicator, introduces a new parameter to its related sub-event and replicates the dimension of the related sub-event.

The sequence pattern and the all-replicator pattern have been introduced in [1]. The examples of these two constructors from [1] have been presented in Section 1.4. Here we present them in a way that follows the pattern based style. The other constructs and corresponding Event-B models are derived from developing our case studies.

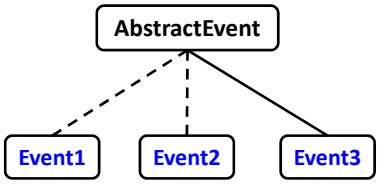
2.2.2 Sequence Pattern

Each pattern is presented in a table. The sequence pattern is presented in Table 2.1. Each pattern table includes the name of the pattern in the first row, followed by a diagrammatic representation of the atomicity decomposition diagram of the pattern for single instance execution (SI) on the left and multiple instances execution (MI) on the right. It is followed by the Event-B model generated from the atomicity decomposition diagrams. The Event-B model contains the invariants and events separately for the SI case and the MI case, labeled as “SI/MI Invariants” and “SI/MI Events”. The Event-B model shown in the table is part of the model which is generated from atomicity decomposition diagrams, user defined Event-B elements like events can be included in the Event-B model but not in any atomicity decomposition diagram. The table interpretation just described, is used for all patterns’ tables.

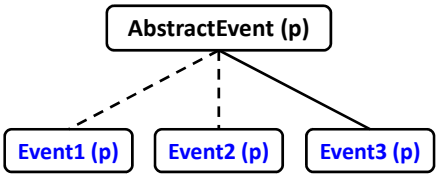
Name: Sequence

Diagrammatic Representation

Single Instance(SI)



Multiple Instance(MI)



Event-B Model

Single Instance(SI) Invariants:

```
invariants
  @inv_Event1_type Event1 ∈ BOOL
  @inv_Event2_seq Event2 = TRUE ⇒ Event1 = TRUE
  @inv_Event3_seq Event3 = TRUE ⇒ Event2 = TRUE
  @inv_Event3_gluing Event3 = AbstractEvent
```

Multiple Instance(MI) Invariants:

```
invariants
  @inv_Event1_type Event1 ⊆ TYPE(p)
  @inv_Event2_seq Event2 ⊆ Event1
  @inv_Event3_seq Event2 ⊆ Event3
  @inv_Event3_gluing Event3 = AbstractEvent
```

Single Instance(SI) Events:

```
event Event1
  where
    @grd Event1 = FALSE
  then
    @act Event1 ≔ TRUE
  end
```

```
event Event2
  where
    @grd_seq Event1 = TRUE
    @grd Event2 = FALSE
  then
    @act Event2 ≔ TRUE
  end
```

```
event Event3 refines AbstractEvent
  where
    @grd_seq Event2 = TRUE
    @grd Event3 = FALSE
  then
    @act Event3 ≔ TRUE
  end
```

Multiple Instance(MI) Events:

```
event Event1
  any p where
    @grd p ∉ Event1
  then
    @act Event1 ≔ Event1 ∪ { p }
  end
```

```
event Event2
  any p where
    @grd_seq p ∈ Event1
    @grd p ∉ Event2
  then
    @act Event2 ≔ Event2 ∪ { p }
  end
```

```
event Event3 refines AbstractEvent
  any p where
    @grd_seq p ∈ Event2
    @grd p ∉ Event3
  then
    @act Event3 ≔ Event3 ∪ { p }
  end
```

Table 2.1: Sequence Pattern

Intention: The atomicity of an abstract event, *AbstractEvent*, is decomposed to sequencing of two or more concrete sub-events. In other words, the behaviour exhibited by an abstract event is realised by the sequential execution of one or more concrete events in the refinement level. Since we are able to describe the features of the sequence pattern by having three sub-events, we minimise the number of sub-events to three, $Event1$, $Event2$ and $Event3$.

Diagrammatic Representation: The name of the abstract event appears in the root node, and sub-events' names appear in leaf nodes in sequence from left to right. A leaf is a node without any child node.

In decomposing the atomicity of an event, two cases are considered. First when a single execution of an event is needed. In this case, there is no control parameter for the event. Moreover control variables are defined with boolean type, since we do not need to record the execution of events for different instances of the parameter(s). This case is called Single Instance (SI). The second case is when multiple instances of an event are needed. It is called Multiple Instances (MI). In this case, there are one or more control parameters for the events. In the diagrammatic representation, control parameter(s) name(s) appear in between parentheses after the event name. In the table, p represents a list of parameters, $p_1, ..., p_n$. We use a set type for control variables. Using sets, enables multiple instances of an event and event interleaving.

Restrictions: One and only one of the leaves in an atomicity decomposition diagram is connected to the root event with a solid line. Other leaves have to connect with dashed lines. This restriction is referred to as the "single solid line" rule in the rest of patterns.

This restriction can raise two questions:

- First, where is the leaf placed with solid line in the sequence of sub-events in the atomicity decomposition of an abstract event?
- Second, why only one leaf with the solid line can be placed in the atomicity decomposition of an abstract event?

The first question is answered in the next two paragraphs. The short answer for the second question is that 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 event (leaf with the solid line). The second question is clarified at the end of this section using examples of event traces.

In the Event-B model, the EQL (Equality of preserved variable) proof obligation, $(evt/v/EQL)$, ensures that an abstract variable v is preserved in the concrete event *evt*. It means that the EQL proof obligation does not allow an abstract variable to be changed in a new event which refines *skip*. The abstract variable v can be modified

only by a concrete event that refines the abstract event which modifies variable v. Also the SIM (Simulation) proof obligations ensure that each action in a concrete event simulates the corresponding abstract action. It means when a concrete event executes, the corresponding abstract event is not contradicted.

The leaf corresponding to the solid line is encoded to an event which refines the abstract event, appearing as the root node. Considering the limitation which EQL and SIM proof obligations make in the Event-B model, the refining event is the event which simulates the main behaviour of the abstract event by modifying the corresponding abstract variable(s). In our patterns we consider it as the last event, $Event3$.

Event-B Model:

Semantics are given to an atomicity decomposition diagram by generating an Event-B model from it. We now explain how an atomicity decomposition diagram of the sequence pattern is encoded as an Event-B model. The encoded Event-B model for the sequence pattern is presented in Table 2.1.

The middle sub-event in the sequence pattern is replaced by a constructor in the rest of patterns, which are described later. Each constructor can be placed as the first or the last sub-event of the diagram too; the reason that we consider it as the middle sub-event is to show the effect of the previous sub-event (the first sub-event) on the constructor, and the effect of the constructor on the next sub-event (the last sub-event). The sequence pattern is considered as a basic pattern for the rest of atomicity decomposition patterns. Therefore most of the translation rules from the diagram to the Event-B model which are explained in this pattern, are true for the rest of patterns.

For each leaf, a node without any child node, one control variable and one event are generated. The generated event name and variable name are same as the leaf name. All generated new events are labeled as ordinary events. Ordering between leaves is achieved by generating some actions and guards in generated events. The generated event corresponding to the leaf with the solid line refines the abstract event. The leaf with the solid line can have the same name as the abstract event, since it refines the abstract event. In the diagrams of Table 2.1 the rightmost event can have the same name as the abstract event.

Considering the SI case, the boolean control variable's value in the related event, is assigned to TRUE. This assignment enables the next event's guard in sequence. For example, in event $Event1$, variable $Event1$ is assigned to $TRUE$, indicating that event $Event1$ executes. This assignment enables guard $(Event1 = TRUE)$ in event $Event2$. We do not need the sequencing guard in the first event, as there is no event before it in sequence. Another guard is generated for each generated event too. This guard indicates that the current event has not executed before, i.e., $(Event1 = FALSE)$ in event $Event1$.

In the MI case, each event corresponding to a leaf gives rise to a set control variable whose type is based on the type of the parameter(s) of the leaf. In the table, p represents a list of parameters, $p_1, ..., p_n$, of type $TYPE(p_1) \times ... \times TYPE(p_n)$. When an event executes for a specific value of the instance parameter(s), the value is added to the set control variable in the action of that event. This enables the next event's guard in sequence. For example, in event $Event1$, the parameter value is added to the set variable $Event1$. This action enables the next event's guard, $(p \in Event1)$ in event $Event2$. Another guard in each event checks that the event has not executed before, i.e., $(p \notin Event1)$ in event $Event1$.

For each leaf an invariant is generated. The invariants states the sequencing conditions. For example in the SI case, $(Event2 = TRUE \Rightarrow Event1 = TRUE)$ is a condition to show that $Event1$ should executes before $Event2$. In the MI case, the subset invariant $(Event2 \subseteq Event1)$ shows that for instances of variable $Event2$, event $Event1$ has executed before. For the first leaf, we do not need a sequencing invariant. Instead a typing invariant is generated.

A gluing invariant is generated for a leaf with solid line. Leaf $Event3$ connects to the root node with solid line, so the gluing invariant $(Event3 = AbstractEvent)$ is generated.

To make the use of gluing invariant clear, consider a case when machine *M2* refines machine *M1*. Atomicity decomposition diagrams help illustrate the relation between abstract events of *M1* and concrete events of *M2*. Each event E of *M2* corresponding to a leaf with solid line in diagrams, either refines an abstract event A of *M1*, or it is a new event corresponding to a leaf with dashed line refining *skip*. The proof obligations defined for Event-B refinement are based on the following proof rule that makes use of a gluing invariant Inv_Gluing.

- Each *M2.E* refines *M1.A* under Inv_Gluing, if A is defined.
- Each *M2.E* refines *skip* under Inv_Gluing, if E is a new event.

Therefore in order to discharge the refinement proof obligations, some gluing invariants, which define the relationship between abstract variable and concrete variables, are needed.

Event Execution Trace Examples:

Considering the SI case in the sequence pattern, the single event trace of the refinement model is as follow:

< Event1, Event2, Event3 >

Each event trace represents a record of a possible execution trace of the model. It is instructive to relate the event trace of the refinement model with the event trace of the abstract model. The single event trace of the abstract model is

$< AbstractEvent >$

If we remove *Event1* and *Event2* from the trace of the refinement model, we get the trace of the abstract model (considering *Event3* refines *AbstractEvent*):

$< Event1, Event2, Event3 > \setminus \{Event1, Event2\} = < Event3 > = < AbstractEvent >$

Removing events from a trace is the standard way of giving a semantic to hidden events [1, 3] and is used, for example, in CSP. By treating *Event1* and *Event2* as hidden events, traces of the refinement model looks like traces of the abstract model. This illustrates a semantics of refinement of Event-B models. Machine *M1* is a refinement of machine *M0* since any trace of *M1* in which the new events are hidden is also a trace of *M0*. In this point the answer for the second question raised in the *Restriction* part can be made clear. If more than one leaf refines the abstract event in the atomicity decomposition of the abstract event, the refinement semantics in Event-B is violated. Because removing hidden events from the refinement trace does not result in the same abstract trace.

As mentioned in the explanation of the Event-B model, using the set type for control variables, enables multiple instances of an event in an event trace. To make this point clear, we provide some examples of event traces for the MI case here. Considering the MI case in the sequence pattern, assume the case where we have two instances of the parameter, (*p1* and *p2*), two examples of possible event traces are as follows :

$< Event1(p1), Event2(p1), Event3(p1), Event1(p2), Event2(p2), Event3(p2) >$
$< Event1(p1), Event1(p2), Event2(p1), Event2(p2), Event3(p1), Event3(p2) >$

To clarify the sequencing conditions modelled with subset invariants in the MI case, we explain the sequencing invariant, ($Event2 \subseteq Event1$). This invariant holds in the above two event traces. For example in the second trace, after execution of $Event2(p1)$, set variable $Event2 = \{p1\}$ is a subset of set variable $Event1 = \{p1, p2\}$.

2.2.3 Loop Pattern

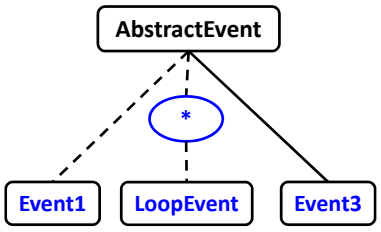
The loop pattern is presented in Table 2.2. The table interpretation is the same as what described in term of the sequence pattern table interpretation in Section 2.2.2.

Intention: In the sequence of sub-events, zero or more execution of an event is needed.

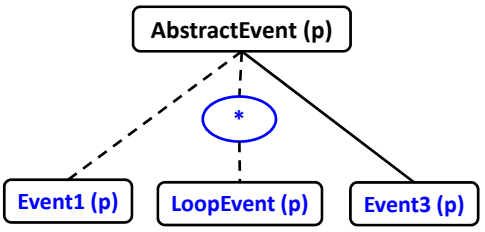
Name: Loop

Diagrammatic Representation

Single Instance(SI)



Multiple Instance(MI)



Event-B Model

Single Instance(SI) Invariants:

```
invariants
  @inv_Event1_type Event1 ∈ BOOL
  @inv_Event3_seq Event3 = TRUE ⇒ Event1 = TRUE
  @inv_Event3_gluing Event3 = AbstractEvent
```

Multiple Instance(MI) Invariants:

```
invariants
@inv_Event1_type Event1 ⊆ TYPE(p)
@inv_Event3_seq Event3 ⊆ Event1
@inv_Event3_gluing Event3  = AbstractEvent
```

Single Instance(SI) Events:

```
event Event1
  where
    @grd Event1 = FALSE
  then
    @act Event1 ≔ TRUE
end
```

```
event LoopEvent
  where
    @grd_seq Event1 = TRUE
    @grd_loop Event3 = FALSE
end
```

```
event Event3 refines AbstractEvent
  where
    @grd_seq Event1 = TRUE
    @grd Event3 = FALSE
  then
    @act Event3 ≔ TRUE
end
```

Multiple Instance(MI) Events:

```
event Event1
  any p where
    @grd  p ∉  Event1
  then
    @act Event1 ≔ Event1 ∪ { p }
end
```

```
event LoopEvent
  any p
  where
    @grd_seq p ∈  Event1
    @grd_loop p ∉  Event3
end
```

```
event Event3 refines AbstractEvent
  any p
  where
    @grd_seq p ∈  Event1
    @grd  p ∉  Event3
  then
    @act Event3 ≔ Event3 ∪ { p }
end
```

Table 2.2: Loop Pattern

Diagrammatic Representation: The loop constructor appears as a circle containing a star. The node connected to the loop, *LoopEvent*, can execute zero or more time after execution of previous sub-event, $Event1$, and before execution of next sub-event, $Event3$, in sequence.

Restrictions: The loop constructor is always connected to the root node with a dashed line. Since the loop event can execute for more than one time, a loop with a solid line does not follow the single solid line rule, which has been explained in the Sequence Pattern (Section 2.2.2). This is clarified at the end of this section using examples of event trace.

Event-B Model:

The encoded Event-B model for the loop pattern is presented in Table 2.2. No control variable is generated for a loop leaf, since we do not need to record the loop event execution. Therefore there is no action for the loop event, *LoopEvent* here.

A guard is generated in the loop event to check that next event has not executed before, i.e., guard ($Event3 = FALSE$) in the SI case and guard ($p \notin Event3$) in the MI case.

The event after the loop event, is guarded by the execution condition of the event before the loop event. Considering the SI case, guard ($Event1 = TRUE$) and considering the MI case guard ($p \in Event1$) in event $Event3$, both check the execution of the event before the loop, $Event1$. This guard allows zero executions of the loop event. Right after execution of event before the loop, with zero execution of the loop event, the event after the loop can execute. That is why we do not need a variable and an action to record the loop execution.

An invariant is generated to show the sequencing between the event before the loop, $Event1$, and the event after the loop, $Event3$. The way that sequencing invariant is described is same as what described in the sequence pattern in Section 2.2.2.

Event Execution Trace Examples:

Considering the SI case diagram in Table 2.2, the event trace of the model in case of zero execution of the loop is:

$<$ *Event1, Event3* $>$

And the event trace of the model in case of two executions of the loop is:

$<$ *Event1, LoopEvent, LoopEvent, Event3* $>$

As mentioned in the restriction, a loop with a solid line is not allowed due to the Event-B restrictions. Assume the loop in the SI case diagram in Table 2.2 is connected to the abstract event with a solid line, and the other two sub-events are connected with dashed

lines. If we remove the hidden sub-events (sub-events with dashed line) from the above event trace, the result is as follow:

$< LoopEvent, LoopEvent >$

Considering what has been explained in the Sequence Pattern in Section 2.2.2 about removing events from a trace, the just mentioned trace is supposed to be same as the abstract event trace, $< AbstractEvent >$, but it is not. Therefore the loop constructor in an atomicity decomposition diagram is always connected to the abstract event with a dashed line.

2.2.4 and-constructor Pattern

The and-constructor pattern is presented in Table 2.3. The table interpretation is same as what was described in terms of the sequence pattern table interpretation in Section 2.2.2.

Intention: The intention is to execute all two or more available sub-events in any order, in the right place in the sequence of other sub-events.

Diagrammatic Representation: The intention stated above is presented in atomicity decomposition diagram with the and-constructor, a circle containing *and*. All nodes connected to the and-constructor execute in any order in the sequence of other sub-events. For simplicity, in this pattern we consider two leaves for the and-constructor.

Restrictions: There are at least two nodes connected to the and-constructor. Following single solid line rule, the and-constructor is always connected to the root node with a dashed line, and all of the corresponding and-constructor events, $AndEvent1$ and $AndEvent2$ here, inherit dashed line from the and-constructor.

Event-B Model:

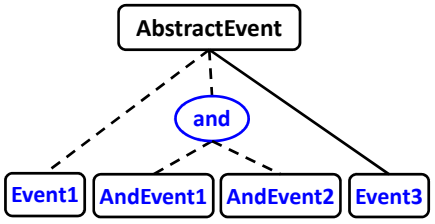
The encoded Event-B model for the and-constructor pattern is presented in Table 2.3. Each and-constructor event can execute only after execution of previous event, $Event1$. This is ensured with a guard, explained in the sequence pattern. The next event after the and-constructor can execute only after execution of all and-constructor events. Therefore a guard is generated in the event after the and-constructor, to ensures that all of the and-constructor events execute before. This guard is a logical conjunction between corresponding control variables generated for the and-constructor leaves. Considering the SI case, guard $(AndEvent1 = TRUE \wedge AndEvent2 = TRUE)$, and in the MI case guard $(p \in AndEvent1 \cap AndEvent2)$, are generated.

Comparing to sequence pattern invariants, the sequencing invariants for the event after the and-constructor is slightly changed in order to show the logical conjunction between control variables of the and-constructor events.

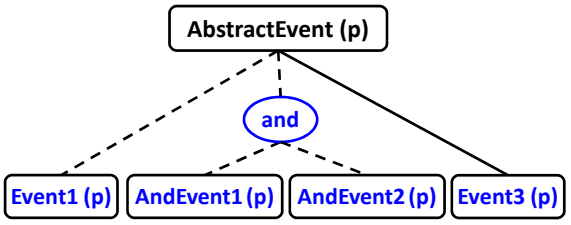
Name: and-constructor

Diagrammatic Representation

Single Instance(SI)



Multiple Instance(MI)



Event-B Model

Single Instance(SI) Invariants:

Invariants
@inv_Event1_type Event1 ∈ BOOL
@inv_AndEvent1_seq AndEvent1 = TRUE ⇒ Event1 = TRUE
@inv_AndEvent2_seq AndEvent2 = TRUE ⇒ Event1 = TRUE
@inv_Event3_seq Event3 = TRUE ⇒ (AndEvent1 = TRUE ∧ AndEvent2 = TRUE)
@inv_Event3_gluing Event3 = AbstractEvent

Multiple Instance(MI) Invariants:

invariants
@inv_Event1_type Event1 ⊆ **TYPE(p)**
@inv_AndEvent1_seq AndEvent1 ⊆ Event1
@inv_AndEvent2_seq AndEvent2 ⊆ Event1
@inv_Event3_seq Event3 ⊆ AndEvent1 ∩ AndEvent2
@inv_Event3_gluing Event3 = AbstractEvent

Single Instance(SI) Events:

event Event1
where
@grd Event1 = FALSE
then
@act Event1 ≔ TRUE
end

event AndEvent1
where
@grd_seq Event1 = TRUE
@grd AndEvent1 = FALSE
then
@act AndEvent1 ≔ TRUE
end

event AndEvent2
where
@grd_seq Event1 = TRUE
@grd AndEvent2 = FALSE
then
@act AndEvent2 ≔ TRUE
end

event Event3 refines AbstractEvent
where
@grd_seq AndEvent1 = TRUE ∧ AndEvent2 = TRUE
@grd Event3 = FALSE
then
@act Event3 ≔ TRUE
end

Multiple Instance(MI) Events:

```
event Event1
  any p where
    @grd  p ∉ Event1
  then
    @act Event1 ≔ Event1 ∪ { p }
end
```

```
event AndEvent1
  any p
  where
    @grd_seq p ∈  Event1
    @grd p ∉ AndEvent1
  then
    @act AndEvent1 ≔ AndEvent1 ∪ { p }
end
```

```
event AndEvent2
  any p
  where
    @grd_seq p ∈  Event1
    @grd p ∉ AndEvent2
  then
    @act AndEvent2 ≔ AndEvent2 ∪ { p }
end
```

```
event Event3 refines AbstractEvent
  any p
  where
    @grd_seq p ∈ (AndEvent1  ∩ AndEvent2)
    @grd p ∉ Event3
  then
    @act Event3 ≔ Event3 ∪ { p }
end
```

Table 2.3: and-constructor Pattern

Event Execution Trace Examples:

Considering the SI case diagram in Table 2.3, the event traces of the model are as follows:

$< Event1, AndEvent1, AndEvent2, Event3 >$
$< Event1, AndEvent2, AndEvent1, Event3 >$

2.2.5 or-constructor Pattern, Multiple Choice

The or-constructor pattern is presented in Table 2.4. The table interpretation is the same as what was described in term of sequence pattern table interpretation in Section 2.2.2.

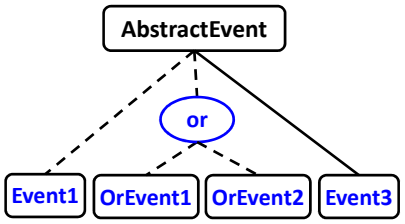
Intention: The intention is to execute one or more sub-events from two or more available sub-events, in any order, in the right place in the sequence of other sub-events.

Diagrammatic Representation: The intention stated above is presented in atomicity decomposition diagram with the or-constructor, a circle containing *or*. One or more nodes connected to the or-constructor execute in any order in the sequence of other sub-events. For simplicity, in this pattern we consider two leaves for the or-constructor.

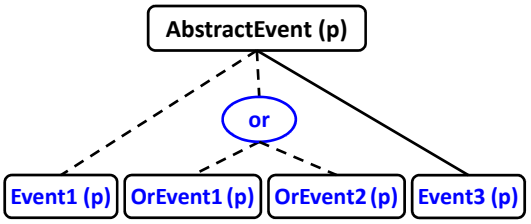
Name: or-constructor

Diagrammatic Representation

Single Instance(SI)



Multiple Instance(MI)



Event-B Model

Single Instance(SI) Invariants:

```
Invariants
  @inv_Event1_type Event1 ∈ BOOL
  @inv_OrEvent1_seq OrEvent1 = TRUE ⇒ Event1 = TRUE
  @inv_OrEvent2_seq OrEvent2 = TRUE ⇒ Event1 = TRUE
  @inv_Event3_seq Event3 = TRUE ⇒ (OrEvent1 = TRUE ∨ OrEvent2 = TRUE)
  @inv_Event3_gluing Event3 = AbstractEvent
```

Multiple Instance(MI) Invariants:

```
invariants
  @inv_Event1_type Event1 ⊆ TYPE(p)
  @inv_OrEvent1_seq OrEvent1 ⊆ Event1
  @inv_OrEvent2_seq OrEvent2 ⊆ Event1
  @inv_Event3_seq Event3 ⊆ OrEvent1 ∪ OrEvent2
  @inv_Event3_gluing Event3 = AbstractEvent
```

Single Instance(SI) Events:

```
event Event1
  where
    @grd Event1 = FALSE
  then
    @act Event1 ≔ TRUE
  end
```

```
event OrEvent1
  where
    @grd_seq Event1 = TRUE
    @grd OrEvent1 = FALSE
  then
    @act OrEvent1 ≔ TRUE
  end
```

```
event OrEvent2
  where
    @grd_seq Event1 = TRUE
    @grd OrEvent2 = FALSE
  then
    @act OrEvent2 ≔ TRUE
  end
```

```
event Event3 refines AbstractEvent
  where
    @grd_seq OrEvent1 = TRUE ∨ OrEvent2 = TRUE
    @grd Event3 = FALSE
  then
    @act Event3 ≔ TRUE
  end
```

Multiple Instance(MI) Events:

```
event Event1
  any p where
    @grd  p ∉ Event1
  then
    @act Event1 ≔ Event1 ∪ { p }
end
```

```
event OrEvent1
  any p
  where
    @grd_seq p ∈  Event1
    @grd p ∉ OrEvent1
  then
    @act OrEvent1 ≔ OrEvent1 ∪ { p }
end
```

```
event OrEvent2
  any p
  where
    @grd_seq p ∈  Event1
    @grd p ∉ OrEvent2
  then
    @act OrEvent2 ≔ OrEvent2 ∪ { p }
end
```

```
event Event3 refines AbstractEvent
  any p
  where
    @grd_seq p ∈  (OrEvent1  ∪ OrEvent2)
    @grd_ p ∉ Event3
  then
    @act Event3 ≔ Event3 ∪ { p }
end
```

Table 2.4: or-constructor Pattern

Restrictions: There are at least two nodes connected to the or-constructor. Following single solid line rule, the or-constructor is always connected to the root node with dashed line, and all of the corresponding or-constructor events, $OrEvent1$ and $OrEvent2$ here, inherit dashed line from the or-constructor.

Event-B Model:

The encoded Event-B model for the or-constructor pattern is presented in Table 2.4. Each or-constructor event can execute only after execution of previous event, $Event1$. This is ensured with a guard, explained in sequence pattern. Next event after the or-constructor in sequence can execute only after execution of at least one of the or-constructor events. Therefore a guard is generated in the event after the or-constructor, to ensures that at least one of the or-constructor events executes before. This guard is a disjunction between the corresponding control variables generated for the or-constructor events. Considering the SI case, guard $(OrEvent1 = TRUE \vee OrEvent2 = TRUE)$, and in the MI case guard $(p \in OrEvent1 \cup OrEvent2)$, are generated.

Comparing to sequence pattern invariants, the sequencing invariants for the event after the or-constructor is changed in order to show the disjunction between control variables of the or-constructor events.

Event Execution Trace Examples:

Considering the SI case diagram in Table 2.4, the event traces of the model are as follows:

$< Event1, OrEvent1, Event3 >$
$< Event1, OrEvent2, Event3 >$
$< Event1, OrEvent1, OrEvent2, Event3 >$
$< Event1, OrEvent2, OrEvent1, Event3 >$

2.2.6 xor-constructor Pattern, Exclusive Choice

The xor-constructor pattern is presented in Table 2.5. The table interpretation is the same as what was described in term of sequence pattern table interpretation in Section 2.2.2.

Intention: The intention is to execute exactly one event from two or more available sub-events, in the right place in the sequence of other sub-events.

Diagrammatic Representation: The intention stated above is presented in the atomicity decomposition diagram with the xor-constructor, a circle containing *xor*. Exactly one of the nodes connected to the xor-constructor executes in the sequence of other sub-events. The xor-constructor can connect to the root node either with solid line or dashed line. Since only one of the xor-constructor events execute in this pattern, so having solid line for the xor-constructor follows the single solid line rule. It is clarified in examples of event trace at the end of this section. For simplicity, in this pattern we consider two leaves for the xor-constructor.

Restrictions: There are at least two nodes connected to the xor-constructor.

Event-B Model:

The encoded Event-B model for the xor-constructor pattern is presented in Table 2.5. The Event-B model is almost like the or-constructor pattern. In each xor-constructor event, a guard is needed to ensure that other xor-constructor events have not executed. For example, in the SI case, guard $XorEvent2 = FALSE$ is generated in $XorEvent1$, and considering the MI case, guard $p \notin XorEvent2$ is generated in $XorEvent1$.

Also an extra invariant is provided to show that at any time only one of the xor-constructor events has executed or none of them has executed. In the SI case, invariant

$partition(\{XorEvent1, XorEvent2\} \cap \{TRUE\},$
$\{XorEvent1\} \cap \{TRUE\}, \{XorEvent2\} \cap \{TRUE\})$

shows that at any time only one the control boolean variables'value can be $TRUE$. And in the MI case invariant

$partition((XorEvent1 \cup XorEvent2), XorEvent1, XorEvent2)$

shows that the set control variables are disjoints. The *partition* operator in event-B is defined as follows:

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

If the xor-constructor is provided with a solid line, the each xor-constructor sub-event refines the abstract event. Also a gluing invariant is needed. The just stated invariants in the SI case and the MI case respectively are changed to:

$partition(\{AbstractEvent\} \cap \{TRUE\},$
$\{XorEvent1\} \cap \{TRUE\}, \{XorEvent2\} \cap \{TRUE\})$

$partition(AbstractEvent, XorEvent1, XorEvent2)$

These gluing invariant not only describe the exclusive choice property, but also they describe the relation between abstract variable and the xor-constructor control variables. Considering *partition* definition, the gluing invariants in the SI case and the MI case respectively describe:

$\{AbstractEvent\} \cap \{TRUE\} = (\{XorEvent1\} \cap \{TRUE\}) \cup (\{XorEvent2\} \cap \{TRUE\})$

$AbstractEvent = XorEvent1 \cup XorEvent2$

Event Execution Trace Examples:

Considering the SI case diagram in Table 2.5, the event traces of the model are as follows:

$<$ *Event1, XorEvent1, Event3* $>$
$<$ *Event1, XorEvent2, Event3* $>$

As mentioned above, the xor-constructor can be connected to the root node with a solid line. Assume the xor-constructor in the SI case diagram in Table 2.2 is connected to the abstract event with a solid line, and the other two sub-events are connected with dashed lines. If we remove the hidden sub-events (sub-events with dashed line) from the above event traces, the results are as follows:

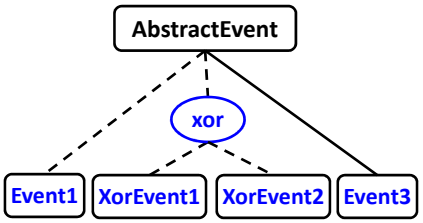
$<$ *XorEvent1* $>$
$<$ *XorEvent2* $>$

Considering what has been explained in the Sequence Pattern in Section 2.2.2 about removing events from a trace, the just mentioned traces are same as the abstract event trace, $<$ *AbstractEvent* $>$, since both xor-constructor events refine the *AbstractEvent*.

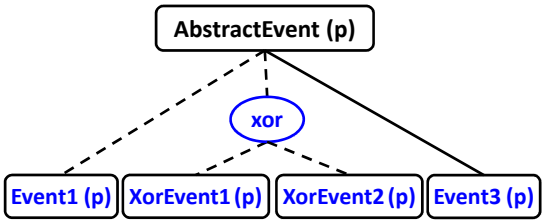
Name: xor-constructor

Diagrammatic Representation

Single Instance(SI)



Multiple Instance(MI)



Event-B Model

Single Instance(SI) Invariants:

```
invariants
  @inv_Event1_type Event1 ∈ BOOL
  @inv_XorEvent1_seq XorEvent1 = TRUE ⇒ Event1 = TRUE
  @inv_XorEvent2_seq XorEvent2 = TRUE ⇒ Event1 = TRUE
  @inv_Event3_seq Event3 = TRUE ⇒ (XorEvent1 = TRUE ∨ XorEvent2 = TRUE)
  @inv_xor partition( {XorEvent1, XorEvent2} ∩ {TRUE} ,
                      {XorEvent1} ∩ {TRUE}, {XorEvent2} ∩ {TRUE})
  @inv_Event3_gluing Event3 = AbstractEvent
```

Multiple Instance(MI) Invariants:

```
invariants
  @inv_Event1_type Event1 ⊆ TYPE(p)
  @inv_XorEvent1_seq XorEvent1 ⊆ Event1
  @inv_XorEvent2_seq XorEvent2 ⊆ Event1
  @inv_Event3_seq Event3 ⊆ XorEvent1 ∪ XorEvent2
  @inv_xor partition((XorEvent1 ∪ XorEvent2), XorEvent1, XorEvent2)
  @inv_Event3_gluing Event3 = AbstractEvent
```

Single Instance(SI) Events:

```
event Event1
  where
    @grd Event1 = FALSE
  then
    @act Event1 ≔ TRUE
  end
```

```
event XorEvent1
  where
    @grd_seq Event1 = TRUE
    @grd XorEvent1 = FALSE
    @grd_xor XorEvent2 = FALSE
  then
    @act XorEvent1 ≔ TRUE
  end
```

```
event XorEvent2
  where
    @grd_seq Event1 = TRUE
    @grd XorEvent2 = FALSE
    @grd_xor XorEvent1 = FALSE
  then
    @act XorEvent2 ≔ TRUE
  end
```

```
event Event3 refines AbstractEvent
  where
    @grd_seq XorEvent1 = TRUE ∨ XorEvent2 = TRUE
    @grd Event3 = FALSE
  then
    @act Event3 ≔ TRUE
  end
```

Multiple Instance(MI) Events:

```
event Event1
  any p where
    @grd  p ∉ Event1
  then
    @act Event1 ≔ Event1 ∪ { p }
 end
```

```
event XorEvent1
  any p
  where
    @grd_seq p ∈  Event1
    @grd p ∉ XorEvent1
    @grd_xor p ∉ XorEvent2
  then
   @act XorEvent1 ≔ XorEvent1 ∪ { p }
 end
```

```
event XorEvent2
  any p
  where
    @grd_seq p ∈  Event1
    @grd p ∉ XorEvent2
    @grd_xor p ∉ XorEvent1
  then
   @act XorEvent2 ≔ XorEvent2 ∪ { p }
 end
```

```
event Event3 refines AbstractEvent
  any p
  where
    @grd_seq p ∈  (XorEvent1  ∪ XorEvent2)
    @grd p ∉ Event3
  then
    @act Event3 ≔ Event3 ∪ { p }
 end
```

Table 2.5: xor-constructor Pattern

2.2.7 all-replicator Pattern

The all-replicator pattern is presented in Table 2.6. The table interpretation is the same as what was described in term of sequence pattern table interpretation in Section 2.2.2.

Intention: The intention is to execute a sub-event for all instances of a new parameter, in the right place in the sequence of other sub-events. The all-replicator is a generalisation of the and-constructor.

Diagrammatic Representation: The all-replicator is presented with a circle containing *all* flowed by name of a new parameter.

Restrictions: Based on the single solid line rule, the all-replicator is always connected to the root event with dashed line, since the all-replicator event can execute for more than one time depending on the number of new introduced *all* parameter instances.

Event-B Model:

The encoded Event-B model for the all-replicator pattern is presented in Table 2.6. The all-replicator parameter, $p2$, is added to the sub-event connected to the all-replicator, $AllEvent$, as a new dimension.

The type of generated control variable for the all-replicator event has got one more dimension compared with other sub-events. Because the all-replicator introduces a new parameter. An invariant is generated to define the type of the all-replicator control variable. For instances considering the SI case, variable $AllEvent$ is a subset of type of new parameter $p2$, $TYPE(p2)$, whereas other control variables are boolean variables. In the MI case $AllEvent$'s variable is defined as a cartesian product of the root parameter's type $TYPE(p1)$ and the all-replicator parameter's type, $TYPE(p2)$.

The event after the all-replicator event in sequence, $Event3$, can execute only after execution of the all-replicator event, $AllEvent$, for all instances of the new parameter, $p2$. A guard is generated in next event, $Event3$, to ensure this property. Guard ($AllEvent = TYPE(p2)$) in event $Event3$ in the SI case, ensures that event $AllEvent$ has executed for all instances of $p2$ before. Also considering the MI case, guard ($AllEvent[\{p1\}] = TYPE(p2)$) plays same role. Relational image r[S] in Event-B is defined as below:

$r[S] = \{y | \exists x.x \in S \land x \mapsto y \in r\}$

Considering relational image definition, guard ($AllEvent[\{p1\}] = TYPE(p2)$) ensures that for $p1$, $AllEvent$ has executed for all instances of $p2$ from set $TYPE(p2)$.

An invariant is generated to model the all-replicator condition: ($p1 \in Event3 \Rightarrow AllEvent[\{p1\}] = TYPE(p2)$) in MI case and ($Event3 = TRUE \Rightarrow AllEvent = TYPE(p2)$ in the SI case.

Event Execution Trace Examples:

Considering the SI case diagram in Table 2.6, assume $p2 \in \{a, b\}$, the the event traces of the model are as follows:

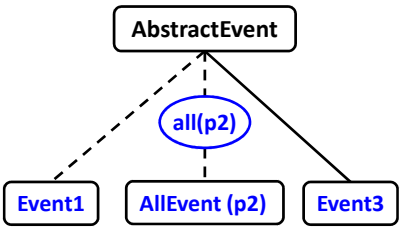
< Event1, AllEvent(a), AllEvent(b), Event3 >
< Event1, AllEvent(b), AllEvent(a), Event3 >

Number of executions of $AllEvent$ is always equal to cardinality of the all-replicator parameter's type set. In this example $AllEvent$ executes for two times, since $card(\{a, b\}) = 2$.

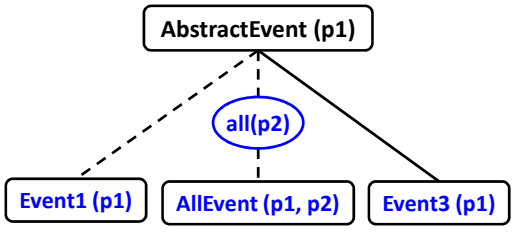
Name: all-replicator

Diagrammatic Representation

Single Instance(SI)



Multiple Instance(MI)



Event-B Model

Single Instance(SI) Invariants:

```
invariants
  @inv_Event1_type Event1 ∈ BOOL
  @inv_AllEvent_type AllEvent ⊆ TYPE(p2)
  @inv_AllEvent_seq AllEvent ≠ ∅ ⇒ Event1 = TRUE
  @inv_Event3_seq Event3 = TRUE ⇒ AllEvent = TYPE(p2)
  @inv_Event3_gluing Event3 = AbstractEvent
```

Multiple Instance(MI) Invariants:

```
Invariants
  @inv_Event1_type Event1 ⊆ TYPE(p1)
  @inv_AllEvent_type AllEvent ⊆ TYPE(p1) × TYPE(p2)
  @inv_AllEvent_seq dom( AllEvent ) ⊆ Event1
  @inv_Event3_seq p1 ∈ Event3 ⇒ AllEvent [ { p1 } ] = TYPE(p2)
  @inv_Event3_gluing Event3 = AbstractEvent
```

Single Instance(SI) Events:

```
event Event1
  where
    @grd Event1 = FALSE
  then
    @act Event1 ≔ TRUE
  end
```

```
event AllEvent
  any p2
  where
    @grd_seq Event1 = TRUE
    @grd p2 ∉ AllEvent
  then
    @act AllEvent ≔ AllEvent ∪ { p2 }
  end
```

```
event Event3 refines AbstractEvent
  where
    @grd_seq AllEvent = TYPE(p2)
    @grd Event3 = FALSE
  then
    @act Event3 ≔ TRUE
  end
```

Multiple Instance(MI) Events:

```
event Event1
  any p where
    @grd p ∉ Event1
  then
    @act Event1 ≔ Event1 ∪ { p }
  end
```

```
event AllEvent
  any p1 p2
  where
    @grd_seq p1 ∈ Event1
    @grd p1 ↦ p2 ∉ AllEvent
  then
    @act AllEvent ≔ AllEvent ∪ { p1 ↦ p2 }
  end
```

```
event Event3 refines AbstractEvent
  any p1
  where
    @grd_seq AllEvent [ { p1 } ] = TYPE(p2)
    @grd p1 ∉ Event3
  then
    @act Event3 ≔ Event3 ∪ { p1 }
  end
```

Table 2.6: all-replicator Pattern

2.2.8 some-replicator Pattern

The some-replicator pattern is presented in Table 2.7. The table interpretation is the same as what was described in term of sequence pattern table interpretation in Section 2.2.2.

Intention: The intention is to execute a sub-event for one or more instances of a new parameter, in the right place in the sequence of other sub-events. The some-replicator is a generalisation of the or-constructor.

Diagrammatic Representation: The some-replicator is presented with a circle containing *some* followed by name of a new parameter.

Restrictions: Based on the single solid line rule, the some-replicator is always connected to the root event with dashed line, since the some-replicator event can execute for more than one time.

Event-B Model:

The encoded Event-B model for the some-replicator pattern is presented in Table 2.7. The some-replicator parameter, $p2$, is added to the sub-event connected to the some-replicator, $SomeEvent$, as a new dimension.

The type of generated control variable for the some-replicator event is defined with an invariant as described in the all-replicator pattern.

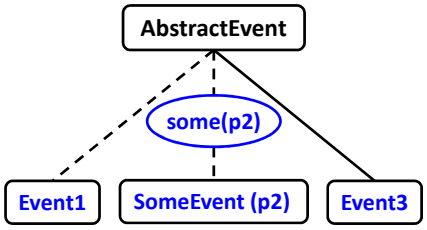
The event after the some-replicator event in the sequence, $Event3$, can execute only after execution of the some-replicator event, $SomeEvent$, at least for one of the instances of the new parameter, $p2$. The sequencing guard ($SomeEvent \neq \varnothing$) in event $Event3$ in the SI case, ensures that event $SomeEvent$ has executed for one or more instances of $p2$ before. Also considering the MI case, guard ($p1 \in dom(SomeEvent)$) ensures that $card(SomeEvent[\{p1\}]) \geq 1$. It means for $p1$, event $Event3$ executes for at least one instance of $p2$.

The sequencing invariant generated for $Event3$, ($Event3 \subseteq dom(SomeEvent)$), also shows one or more execution of $SomeEvent$ before execution of $Event3$.

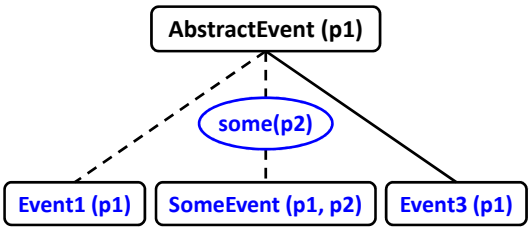
Name: some-replicator

Diagrammatic Representation

Single Instance(SI)



Multiple Instance(MI)



Event-B Model

Single Instance(SI) Invariants:

```
invariants
  @inv_Event1_type Event1 ∈ BOOL
  @inv_SomeEvent_type SomeEvent ⊆ TYPE(p2)
  @inv_SomeEvent_seq SomeEvent ≠ ∅ ⇒ Event1 = TRUE
  @inv_Event3_seq Event3 = TRUE ⇒ SomeEvent ≠ ∅
  @inv_Event3_gluing Event3 = AbstractEvent
```

Multiple Instance(MI) Invariants:

```
Invariants
  @inv_Event1_type Event1 ⊆ TYPE(p1)
  @inv_SomeEvent_type SomeEvent ⊆ TYPE(p1) × TYPE(p2)
  @inv_SomeEvent_seq dom( SomeEvent ) ⊆ Event1
  @inv_Event3_seq Event3 ⊆ dom( SomeEvent )
  @inv_Event3_gluing Event3 = AbstractEvent
```

Single Instance(SI) Events:

```
event Event1
  where
    @grd Event1 = FALSE
  then
    @act Event1 ≔ TRUE
 end
```

```
event SomeEvent
  any p2
  where
    @grd_seq Event1 = TRUE
    @grd p2 ∉ SomeEvent
  then
    @act SomeEvent ≔ SomeEvent ∪ { p2 }
 end
```

```
event Event3 refines AbstractEvent
  where
   @grd_seq SomeEvent ≠ ∅
   @grd Event3 = FALSE
then
    @act Event3 ≔ TRUE
 end
```

Multiple Instance(MI) Events:

```
event Event1
  any p where
    @grd  p ∉ Event1
  then
    @act_Event1 Event1 ≔ Event1 ∪ { p }
 end
```

```
event SomeEvent
  any p1 p2
  where
    @grd_seq p1 ∈  Event1
    @grd p1 ↦ p2 ∉ SomeEvent
  then
    @act SomeEvent ≔ SomeEvent ∪ { p1 ↦ p2 }
 end
```

```
event Event3  refines AbstractEvent
  any p1
  where
    @grd_seq p1 ∈  dom( SomeEvent )
    @grd  p1 ∉  Event3
  then
    @act Event3 ≔ Event3 ∪ { p1 }
 end
```

Table 2.7: some-replicator Pattern

Event Execution Trace Examples:

Considering the SI case diagram in Table 2.7, assume $p2 \in \{a, b\}$, the event traces of the model are as follows:

$<$ *Event1, AllEvent(a), AllEvent(b), Event3* $>$
$<$ *Event1, AllEvent(b), AllEvent(a), Event3* $>$
$<$ *Event1, AllEvent(a), Event3* $>$
$<$ *Event1, AllEvent(b), Event3* $>$

The number of the some-replicator event execution is always less than or equal to the cardinality of the some-replicator parameter's type set. In above event traces, *SomeEvent* executes for one or two times, since $card(\{a, b\}) = 2$.

2.2.9 one-replicator Pattern

The one-replicator pattern is presented in Table 2.8. The table interpretation is the same as what was described in term of sequence pattern table interpretation in Section 2.2.2.

Intention: The intention is to execute a sub-event for exactly one instance of a new parameter, in the right place in the sequence of other sub-events. The one-replicator is a generalisation of the xor-constructor.

Diagrammatic Representation: The one-replicator is presented with a circle containing *one* flowed by name of a new parameter. Following the single solid line rule, the one-replicator can be connected to the root event with either dashed line of solid line, since the one-replicator event can execute for only one instance.

Event-B Model:

The encoded Event-B model for the one-replicator pattern is presented in Table 2.8. The one-replicator parameter, $p2$, is added to the sub-event connected to the one-replicator, *OneEvent*, as a new dimension.

Type of generated control variable for the one-replicator event is defined with an invariant as described in the all-replicator pattern.

The event after the one-replicator event in the sequence, *Event*3, can execute only after execution of the one-replicator event, *OneEvent*, for exactly one of the instances of the new parameter, $p2$. The sequencing guard in event *Event*3 is same as the one in the some-replicator pattern. In order to restrict the number of the one-replicator event executions, we provide a guard in the one-replicator event. Considering the SI case, the guard $(OneEvent = \varnothing)$ in event *OneEvent* ensures that event *OneEvent* can execute

only for one time. And in the MI case, guard ($p1 \notin dom(OneEvent)$) ensures that for $p1$, event $OneEvent$ can execute only for one instance of $p2$.

An invariant is generated to show that the one-replicator event can execute only for one time (for each instance of event parameter in the MI case). In the SI case, $(card(OneEvent) \leqslant 1)$, and the MI case, invariant $(\forall p.card(OneEvent[\{p\}]) \leqslant 1)$.

A gluing invariant is generated for the one-replicator with the solid line. The gluing invariant in the SI case and the MI case respectively are as follows:

$OneEvent \neq \varnothing \Leftrightarrow AbstractEvent = TRUE$

$dom(OneEvent) = AbstractEvent$

Event Execution Trace Examples:

Considering the SI case diagram in Table 2.8, assume $p2 \in \{a, b\}$, the event traces of the model are as follows:

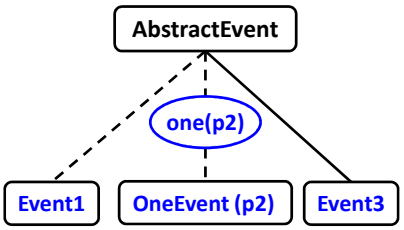
< Event1, OneEvent(a), Event3 >
< Event1, OneEvent(b), Event3 >

The one-replicator event can execute exactly for one instance of the new parameter.

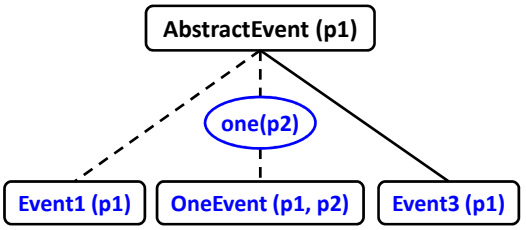
Name: one-replicator

Diagrammatic Representation

Single Instance(SI)



Multiple Instance(MI)



Event-B Model

Single Instance(SI) Invariants:

```
invariants
  @inv_Event1_type Event1 ∈ BOOL
  @inv_OneEvent_type OneEvent ⊆ TYPE(p2)
  @inv_OneEvent_seq OneEvent ≠ ∅ ⇒ Event1 = TRUE
  @inv_Event3_seq Event3 = TRUE ⇒ OneEvent ≠ ∅
  @inv_OneEvent_one card(OneEvent) ≤ 1
  @inv_Event3_gluing Event3 = AbstractEvent
```

Multiple Instance(MI) Invariants:

```
invariants
  @inv_Event1_type Event1 ⊆ TYPE(p1)
  @inv_OneEvent_type OneEvent ⊆ TYPE(p1) × TYPE(p2)
  @inv_OneEvent_seq dom( OneEvent ) ⊆ Event1
  @inv_Event3_seq Event3 ⊆ dom( OneEvent )
  @inv_OneEvent_one ∀p· card( OneEvent [{p}] ) ≤ 1
  @inv_Event3_gluing Event3 = AbstractEvent
```

Single Instance(SI) Events:

```
event Event1
  where
    @grd Event1 = FALSE
  then
    @act Event1 ≔ TRUE
end
```

```
event OneEvent
  any p2
  where
    @grd_seq Event1 = TRUE
    @grd p2 ∉ OneEvent
    @grd_one OneEvent = ∅
  then
    @act  OneEvent ≔ OneEvent ∪ { p2 }
end
```

```
event Event3 refines AbstractEvent
  where
    @grd_seq OneEvent ≠ ∅
    @grd Event3 = FALSE
then
    @act Event3 ≔ TRUE
end
```

Multiple Instance(MI) Events:

```
event Event1
  any p where
    @grd  p ∉ Event1
  then
    @act_Event1 Event1 ≔ Event1 ∪ { p }
end
```

```
event OneEvent
  any p1 p2
  where
    @grd_seq p1 ∈  Event1
    @grd p1 ↦ p2 ∉ SomeEvent
    @grd_one p1 ∉ dom( OneEvent )
  then
    @act OneEvent ≔ OneEvent ∪ { p1 ↦ p2 }
end
```

```
event Event3 refines AbstractEvent
  any p1
  where
    @grd_seq p1 ∈  dom( OneEvent )
    @grd  p1 ∉  Event3
  then
    @act Event3 ≔ Event3 ∪ { p1 }
end
```

Table 2.8: one-replicator Pattern

2.3 Additional Features of the Atomicity Decomposition Approach

2.3.1 The Most Abstract Level

The most abstract level of an Event-B model is illustrated in a diagram that aids understanding, shown in Figure 2.1. The name of a process in the system appears in an oval as the root node, and the names of 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 most abstract events and do not refine the root node. The Event-B model is the same as presented in patterns, Section 2.2. The only difference is that in the most abstract level, there is no refining event (no solid line) and no gluing invariant in the Event-B model.

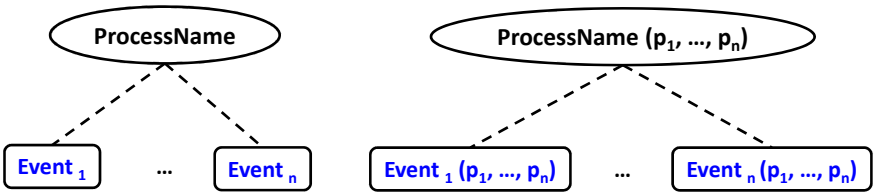


Figure 2.1: The Most Abstract Level Diagrams

2.3.2 Combined Atomicity Decomposition Diagram

In an atomicity decomposition diagram, root node, *AbstractEvent* in described patterns in Section 2.2, is one of the events in $(i)^{th}$ refinement level which decomposed into some sub-events in $(i+1)^{th}$ refinement level. Later each sub-events can be further decomposed to some other sub-events in the next refinement level, $(i+2)^{th}$ refinement level, and so on. The reason in the patterns we called the root node, *AbstractEvent*, is that comparing with sub-events, *AbstractEvent* is placed in an earlier level of refinement which can be considered as an abstract level for the sub-events refinement level.

Starting from the most abstract level diagram, the atomicity decomposition diagrams for different events can be combined in a single diagram. An example is illustrated in Figure 2.2. In this example, there are four abstract events, $Event_1$, $Event_2$, $Event_3$ and $Event_4$, in the most abstract level. In the first refinement level, $Event_2$ is decomposed to $Event_5$ followed by one instance of $Event_6$. Also $Event_4$ is decomposed to three sequential sub-events, $Event_7$, followed by a loop constructor applied to $Event_8$, followed by $Event_9$.

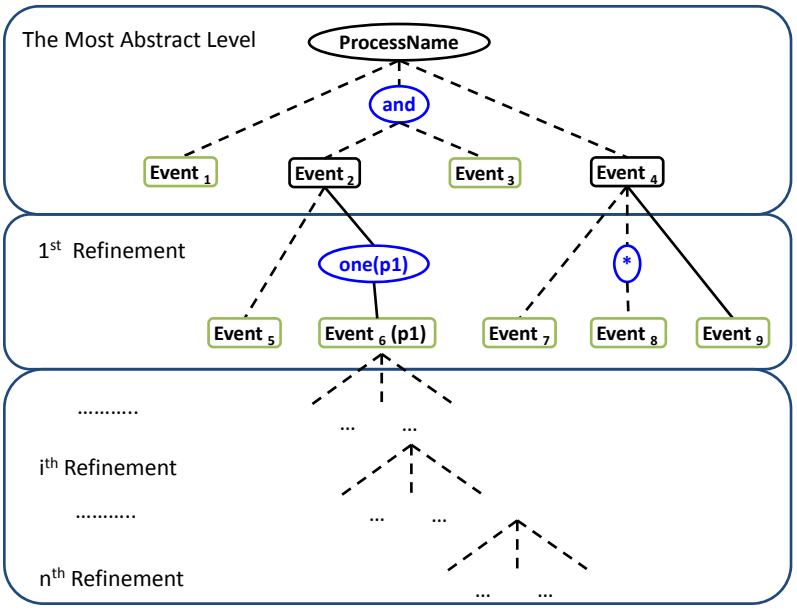


Figure 2.2: Combined Atomicity Decomposition Diagram

The combined atomicity decomposition diagram provides the overall visualization of the refinement structure. In a combined atomicity decomposition diagram, each leaf is encoded as one event in the Event-B model. A leaf is a node without any child. For example, in the first refinement level of Figure 2.2, the leaves are $Event_1$, $Event_5$, $Event_6$, $Event_3$, $Event_7$, $Event_8$, $Event_9$.

The general atomicity decomposition language which describes the structure of the combined atomicity decomposition diagram and translation rules to the Event-B model are presented in Chapter 2.

2.3.3 Several Atomicity Decompositions for a Single Event

A single event can be decomposed to some sub-events in different styles. In other words several atomicity decomposition diagrams can be defined for a same root node. An example is illustrated in Figure 2.3. $Event\_a$ is decomposed in two different diagrams in the next refinement level. The Event-B model follows the rules that presented in patterns, Section 2.2.

2.3.4 Strong Sequencing versus Weak Sequencing

In a combined atomicity decomposition diagram, there are two approaches of sequencing applied to a single root event: Strong Sequencing and Weak Sequencing. Strong/weak sequencing property is applied to each single atomicity decomposition of a root event. If strong sequencing is applied to a root event, then there is a sequencing between all

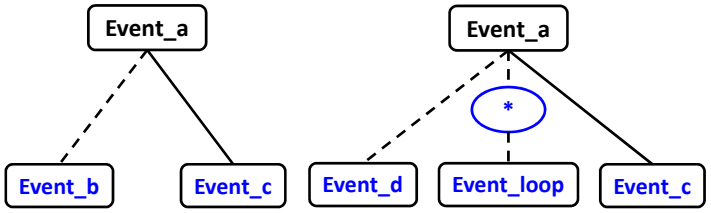


Figure 2.3: Several Atomicity Decomposition for a Single Event, *Event_a*

sub-events of that root and the previous and next sub-events of the earlier refinement level. Whereas in the case of weak sequencing, the sequencing is applied only to the sub-event with solid line of the root and the previous and next sub-events of the earlier refinement level.

To make the point clear, an example of a combined atomicity decomposition diagram is presented in Figure 2.4. *Event_a* is decomposed to four sub-events, *Event_b*, *Event_c*, *Event_d* and *Event_a*, in $(i)^{th}$ refinement level. Then *Event_c* is decomposed to three sub-events, *Event_f*, *Event_c* and *Event_g* in $(i+1)^{th}$ refinement level.

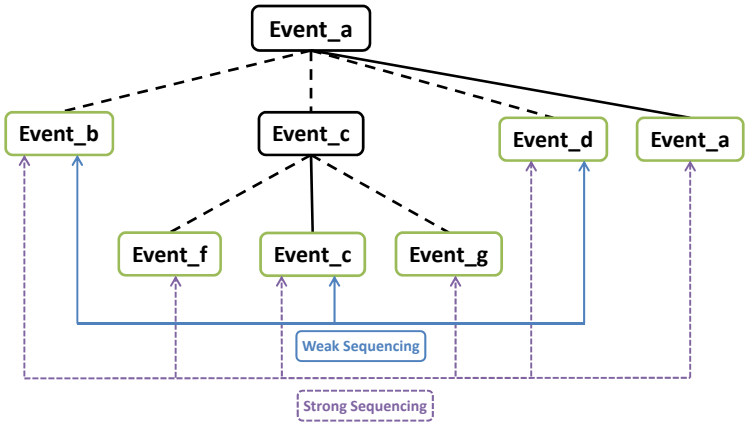


Figure 2.4: Strong Sequencing, Weak Sequencing

Assume atomicity decomposition of *Event_c* root event has strong sequencing, then the only possible event trace is:

$< Event\_b, Event\_f, Event\_c, Event\_g, Event\_d, Event\_a >$

Whereas if atomicity decomposition of *Event_c* has weak sequencing, then on one hand there is an ordering just between the leaf with solid line, *Event_c* and the previous and next leaves in sequence, *Event_b* and *Event_d* respectively. And on the other hand there is no ordering constraints between *Event_b* and *Event_f*, and between *Event_g* and *Event_d*. Therefore, because of weak sequencing, there are more than one possible event trace:

$< Event\_b, Event\_f, Event\_c, Event\_g, Event\_d, Event\_a >$
$< Event\_b, Event\_f, Event\_c, Event\_d, Event\_g, Event\_a >$
$< Event\_f, Event\_b, Event\_c, Event\_g, Event\_d, Event\_a >$
$< Event\_f, Event\_b, Event\_c, Event\_d, Event\_g, Event\_a >$

In all of possible event traces, *Event_c* executes after execution of *Event_b*, before *Event_d*. It is important to mention that in a single atomicity decomposition, there is always an ordering between sub-events of the root event, in both strong and weak sequencing approaches. For example, *Event_f*, *Event_c* and *Event_g* always execute in order.

The weak and strong sequencing is managed with some invariants and guards. The general translation rules to the Event-B model are presented in Chapter 2.

The most abstract atomicity decomposition diagram always has a strong sequencing, since the most abstract diagram is placed in the top level of combined an atomicity decomposition diagram.

2.3.5 Loop Resetting Event

As described in the Loop Pattern in Section 2.2.3, if the loop event is a single event, then we do not consider a variable for the loop event. Considering the example in Figure 2.5, in decomposing the atomicity of *Event_a*, *Event_c* can execute zero or more time before execution of *Event_d*. And the execution of *Event_d* here, does not depend on the loop execution.

In the next refinement level, the loop event is decomposed to some sub-events. So we have to consider some control variables to manage the ordering between the loop events, *Event_e*, *Event_f* and *Event_g*. Also a resetting event is needed to reset the loop control variables to enable more than one execution of the loop. Furthermore an extra guard is needed in *Event_d* to ensure that *Event_d* does not execute in the middle of the execution of the loop events.

Loop resetting can be done in three ways. Each of them for the example in Figure 2.5, is illustrated with a state diagram and its Event-B model in Figure 2.6, Figure 2.7 and Figure 2.8.

First, as illustrated in Figure 2.6, the reset event is considered as a separate event, called *Reset* here. The ordering between loop events are managed with some control variables, *Event_e*, *Event_f* and *Event_g*. The first event in the Loop, *Event_e* checks that the next event after the loop has not execute before, $(Event\_d = FALSE)$. A guard in *Event_d* ensures that it can not execute in the middle of the loop, $(Event\_e = FALSE)$.

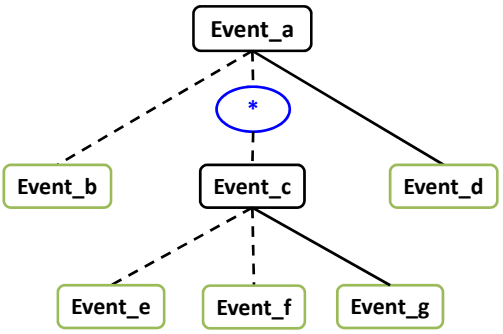
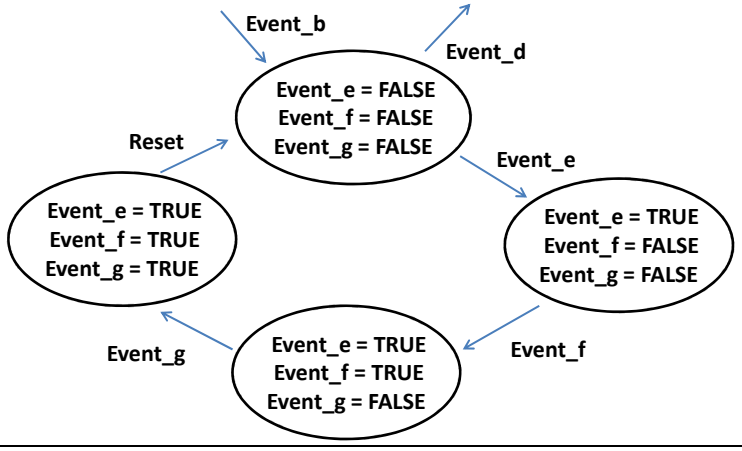


Figure 2.5: Loop Resetting Example



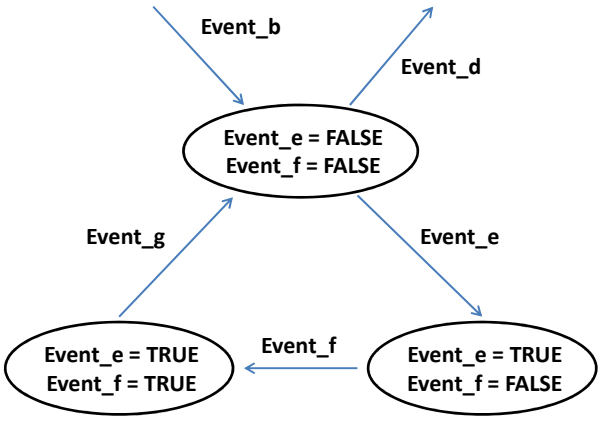

```
event Event_e where Event_b = TRUE ∧ Event_d = FALSE ∧ Event_e = FALSE then Event_e := TRUE end
event Event_f  where Event_e = TRUE ∧ Event_f = FALSE  then  Event_f := TRUE  end
event Event_g where Event_f = TRUE ∧  Event_g = FALSE  then  Event_g := TRUE end

event Reset where  Event_g = TRUE  then  Event_e := FALSE,
                                         Event_f := FALSE,
                                         Event_g := FALSE  end

event Event_d where Event_b = TRUE ∧ Event_e = FALSE ∧ Event_d = FALSE then Event_d := TRUE end
```

Figure 2.6: Loop Resetting as a Separate Event

Second, as illustrated in Figure 2.7, the resetting is merged in the last event of the loop, *Event_g*. In this case we do not need a control variable for the last event, since the last event resets the loop.

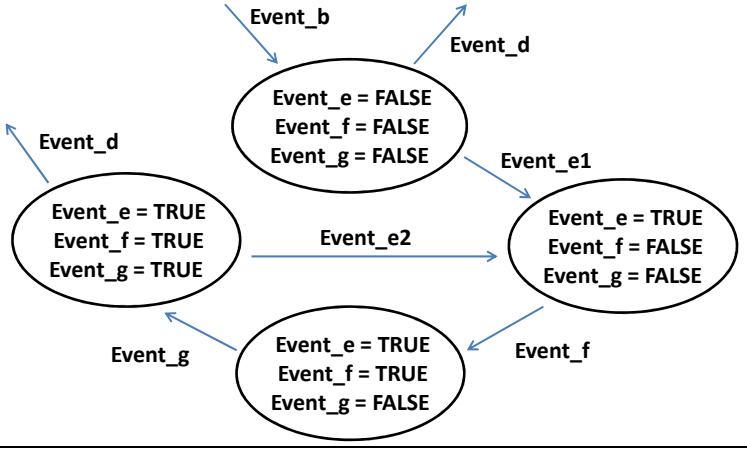
Last, as illustrated in Figure 2.8, the resetting is merged in the first event of the loop, *Event_e*. In this case we have to consider a separate event for the first event of the loop, *Event_e1*. The resetting is done in *Event_e2*. In this case *Event_d*'s guard is complex, since we need to consider two cases. First zero execution of the loop, ($Event\_e = FALSE$) and second, one or more execution(s) of the loop, ($Event\_g = TRUE$).

We adopted the separate resetting event for the loop in Figure 2.6. Considering the example in Figure 2.9, assume the case when the first sub-event in decomposing the




```
event Event_e where Event_b = TRUE ∧ Event_d = FALSE ∧ Event_e = FALSE then Event_e := TRUE end

event Event_f  where  Event_e = TRUE ∧ Event_f = FALSE  then  Event_f := TRUE  end

event Event_g where  Event_f = TRUE  then  Event_e := FALSE,
                                           Event_f := FALSE   end

event Event_d where Event_b = TRUE ∧ Event_e = FALSE ∧ Event_d = FALSE then Event_d := TRUE end
```

Figure 2.7: Loop Resetting in the Last Event




```
event Event_e1 where Event_b =TRUE ∧ Event_d=FALSE ∧ Event_e =FALSE  then Event_e:= TRUE end
event Event_e2 where Event_g = TRUE ∧ Event_d= FALSE  then Event_f:= FALSE,
                                                           Event_g:= FALSE  end

event Event_f  where  Event_e = TRUE ∧ Event_f = FALSE  then   Event_f := TRUE  end
event Event_g where  Event_f = TRUE ∧ Event_g = FALSE  then   Event_g := TRUE end

event Event_d where Event_b = TRUE ∧ ( Event_e = FALSE   ∨  Event_g = TRUE ) ∧ Event_d = FALSE
then Event_d := TRUE end
```

Figure 2.8: Loop Resetting in the First Event

loop event, *Event_c*, is either the and-constructor or the or-constructor or the xor-constructor. Then the resetting approach presented in Figure 2.8, needs to be applied to all of the constructor children, *Event_e* and *Event_f* here. Also in the resetting approach presented in Figure 2.7, if the last sub-event is either the and-constructor or the or-constructor or the xor-constructor, then the resetting needs to be applied to all of the constructor children, and this can make the Event-B model large and complex comparing to the approach when we provide the separate resetting event.

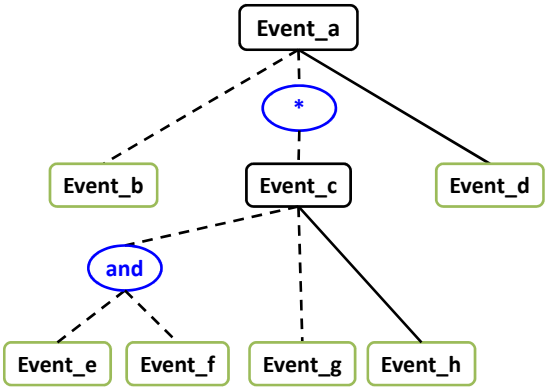


Figure 2.9: Loop Resetting Example

Using a separate event to reset loop, the Event-B model of the example presented in Figure 2.5, is presented in Figure 2.10, in the MI case (having one parameter).

event Event_e where $p_1 \in$ Event_b $\wedge$ $p_1 \notin$ Event_d $\wedge$ $p_1 \notin$ Event_e **then** Event_e $\coloneqq$ Event_e $\cup \{p_1\}$
event Event_f where $p_1 \in$ Event_e $\wedge$ $p_1 \notin$ Event_f **then** Event_f $\coloneqq$ Event_f $\cup \{p_1\}$ **end**
event Event_g where $p_1 \in$ Event_f $\wedge$ $p_1 \notin$ Event_g **then** Event_g $\coloneqq$ Event_g $\cup \{p_1\}$ **end**

event Reset where $p_1 \in$ Event_g **then** Event_e $\coloneqq$ Event_e $/ \{p_1\}$,
Event_f $\coloneqq$ Event_f $/ \{p_1\}$,
Event_g $\coloneqq$ Event_g $/ \{p_1\}$ **end**

event Event_d where $p_1 \in$ Event_b $\wedge$ $p_1 \notin$ Event_e $\wedge$ $p_1 \notin$ Event_d **then** Event_d $\coloneqq$ Event_d $\cup \{p_1\}$

Figure 2.10: Loop Resetting with Parameter

2.4 Conclusion

Several atomicity decomposition constructors, which were discovered during case study developments, have been presented in this chapter. A pattern-based style was used to present the atomicity decomposition constructors. Each pattern is defined to satisfy a particular intention in decomposing the atomicity of an abstract event, and contains one constructor in a single level of refinement. Each pattern is encoded in terms of Event-B using some variables, invariants, events, guards and actions. The diagrammatic notation

of a constructor and corresponding encoded Event-B model are presented both for single instance (SI) execution of an event and multiple instance (MI) execution.

In total eight constructors were presented as follows:

- The intention to model a sequential execution of two or more events is represented by the Sequence pattern.
- The Loop pattern represents zero or more execution of an event.
- The logical constructor patterns (and-constructor, or-constructor and xor-constructor) model a logical execution between two or more events.
- The replicator patterns, all-replicator, some-replicator and one-replicator, are generalisations of the logical constructor patterns, and-constructor, or-constructor and xor-constructor, respectively.

Each pattern contains three children in decomposition of an abstract event in one refinement level. In all patterns, except the sequence pattern, the middle sub-event is a loop or a logical constructor or a replicator.

References

[1] Butler, M. J. Decomposition Structures for Event-B. In *Integrated Formal Methods iFM2009* (2009). URL http://deploy-eprints.ecs.soton.ac.uk/51/.

[2] Jones, C. B. *Systematic software development using VDM (2nd ed.)* (Prentice-Hall, Inc., 1990). URL http://portal.acm.org/citation.cfm?id=94062.

[3] Butler, M. J. Incremental Design of Distributed Systems with Event-B. In *Marktoberdorf Summer School 2008 Lecture Notes* (IoS, 2008). URL http://deploy-eprints.ecs.soton.ac.uk/49/.