Electronic Communications of the EASST
Volume 66 (2013)

Proceedings of the
Automated Verification of Critical Systems
(AVoCS 2013)

Building Traceable Event-B Models from Requirements

Eman Alkhammash, Asieh Salehi Fathabadi, Michael Butler, and Corina Cîrstea

16 pages

Guest Editors: Steve Schneider, Helen Treharne
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Building Traceable Event-B Models from Requirements

Eman Alkhammash[1], Asieh Salehi Fathabadi[2], Michael Butler[3], and Corina Cîrstea [4]

[1 2 3 4] eha1g09,asf08r, mjb, cc2@ecs.soton.ac.uk
ESS, University of Southampton

Abstract: Constructing traceable Event-B models from requirements is crucial in the system development process. It enables the validation of the model against the requirements and allows to identify different refinement levels, which is a key to successful formal modelling with a refinement-based method. Our objective is to present an approach based on the use of semi-formal structures to bridge the gap between requirements and Event-B models and retain traceability to requirements in Event-B models. The presented approach makes use of the UML-B and Atomicity Decomposition (AD) approaches. UML-B provides UML graphical notation that enables the development of an Event-B formal model, while the AD approach provides a graphical notation to illustrate the refinement structures and assists in the organisation of refinement levels. The AD approach also combines several constructor patterns to manage control flows in Event-B. The intent of this paper is to harness the benefits of the UML-B and AD approaches to facilitate constructing Event-B models from requirements and provide traceability between requirements and Event-B models.

Keywords: Event-B, Traceability, UML-B, Atomicity Decomposition

1 Introduction

We present an approach for incrementally constructing a formal model from informal requirements with the aim to retain traceability to requirements in models. The approach helps to identify the modelling elements from requirements, assists the construction of a formal model, and facilitates layering the requirements and mapping the informal requirements to traceable formal models. Traceability supports the process of validation of the model against the requirement document and allows missing requirements to be easily accommodated in the model.

Our approach is based on the Event-B formal method [Abr10]. Event-B is a refinement-based formal method with good tool support for developing various kinds of systems. Event-B covers a range of approaches to support a formal modelling.

In our approach, we make use of UML-B [SB06] and Atomicity Decomposition (AD) [But09a, FBR12] approaches. UML-B provides a graphical modelling environment (UML notation) which enables the development of an Event-B formal model. The AD approach provides a graphical notation to structure refinement and describes the ordering between events. The visual view of the system provided by the UML-B and AD assists in the development of the refinement strategy before the actual work on modelling is performed. The combined AD diagrams, which show the overall refinement structure of the system, can be modified until an acceptable refinement structure is reached. In addition, the AD approach provides several constructor patterns that can be used to manage the flow of events and define event ordering. Moreover, Event-B models corresponding to AD diagrams and UML-B diagrams can be generated automatically by the AD and UML-B tool.

The presented approach comprises of three stages, which are shown in Figure 1. The first

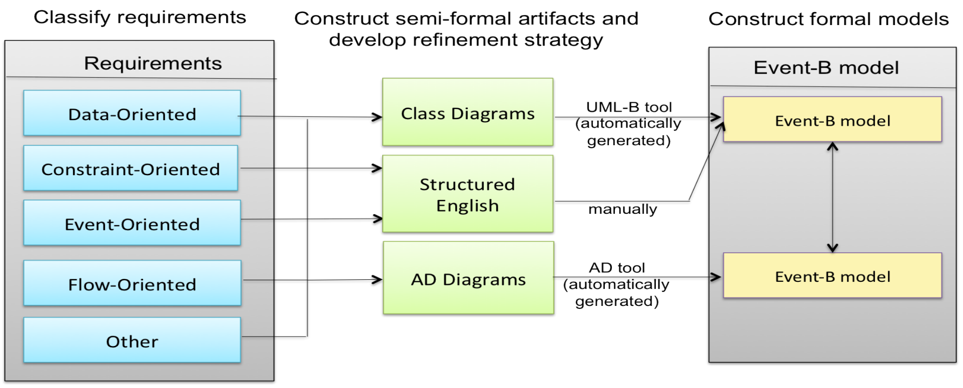


Figure 1: Steps for constructing a traceable formal models

step in our approach is requirement classification. Requirements are classified based on Event-B components. The classification consists of five classes: data-oriented, constraint-oriented, event-oriented, flow-oriented, and others. Data-oriented requirements represent attributes and relationships between attributes, constraint-oriented requirements represent conditions that must remain true in the system, event-oriented requirements represent the activities of the system and its components, flow-oriented requirements represent relationships between events, and “others” represent other requirements that do not fit into the previous classes.

The second step consists of three stages. Firstly, we use semi-formal artifacts described using UML-B, AD diagrams and structured English to represent requirements. The UML-B is used to represent data-oriented requirements. The AD is used to represent flow-oriented requirements. The structured English is a way of breaking down constraint and event-oriented requirements into shorter sub-requirements and mapping each sub-requirements to the proper class (constraint or event-oriented). The semi-formal artifacts serve as an intermediate representation between requirements and the Event-B formalism. Representing requirements using semi-formal artifacts is reasonably simple, and at the same time the movement from the semi-formal artifacts to the Event-B is straightforward. Secondly, we merge the fragmented structured English of a single event together to facilitate tracing the event components. Thirdly, we combine AD diagrams and use these diagrams to assist the process of developing the refinement strategy.

The third step is to use the UML-B tool and the AD tool to generate Event-B models and also write manually the corresponding Event-B from the structured English representation.

This paper is structured as follows: Section 2 gives an overview of the Event-B formal method, UML-B, and AD approach. The description of the presented approach is introduced in Section 3. Section 4 introduces some related works in requirement traceability. Conclusions are drawn in Section 5 and Future work is presented in Section 6.

2 Preliminaries

2.1 Event-B

Event-B is a formal method developed by Jean-Raymond Abrial, which uses set theory and first order logic to provide a formal notation for the creation of models of discrete systems and the undertaking of several refinement steps. An abstract Event-B specification can be refined by adding more detail and bringing it closer to an implementation. A refined model in Event-B is verified through a set of proof obligations expressing that it is a correct refinement of its abstraction. Event-B may be used for parallel, reactive or distributed system development, and has shown success in the development of different complex real-life systems [But09b, BA05]. The Event-B notation contains two constructs: a context and a machine. The context is the static part of a model in which the data of the model (sets, constants and axioms) is defined. The dynamic and functional behaviour of a model is represented in the machine part, which includes variables to describe the states of the system, invariants to constrain variables, and events to trigger the behaviour of the machine.

Rodin [ABHV06, ABH$^+$10] is a platform for modelling and proving in Event-B models. Rodin exhibits many features and can be extended further with supportive plug-ins. Relevant to this work is the Atomicity Decomposition (AD) plug-in, which provides a graphical notation to structure refinement and to manage flows in Event-B models. An overview of the AD approach is given in Section 2.3.

2.2 UML-B

UML-B is a diagrammatic notation based on UML and Event-B. It provides a graphical modelling environment that allows the development of an Event-B formal model through the use of UML graphical notation. There are four types of UML-B diagrams, namely package diagrams, context diagrams, class diagrams and state machine diagrams. Package diagrams represent the structure and the relationships between Event-B contexts and machines. A context diagrams describes the context part of an Event-B model. Class diagram and state machine diagrams describe state and behaviour and are used in Event-B machines. Class diagrams in UML-B may contain attributes (variables), associations (relationships between two classes), events and state machines (transitions between events). State machine diagrams describe the behaviour of instances of classes as transitions linked to events.

UML-B assimilates the notion of refinement. It is possible to introduce a class in a refined machine that refines a class of its abstract machine. A refined class can keep all attributes of its abstract class, corresponding to the case where a refined machine keeps all the variables of an abstract machine. It is also possible that a refined class drops some of the attributes of the abstract class, corresponding to the case of removing variables through performing data refinement. Moreover, a refined class can introduce new attributes in the class diagram, corresponding to the case of introducing new variables in the refinement levels.

A UML-B tool [SB08] has been developed for the Rodin platform which can be used to generate an Event-B model corresponding to a UML-B development.

2.3 Atomicity Decomposition Approach

Although refinement in Event-B provides a flexible approach to modelling, it has the weakness that it cannot explicitly represent the relationships between abstract events and new events introduced in a refinement level. The Atomicity Decomposition (AD) approach addresses this limitation. The idea is to augment Event-B refinement with a graphical notation that is capable of representing the relationships between abstract and concrete events explicitly. Using the AD approach has another advantage, namely that it allows event ordering to be represented explicitly. Figure 2 illustrates these two features of the AD graphical notation. Assume machine *M1*

```
machine M1 refines M0  sees C0

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

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

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

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

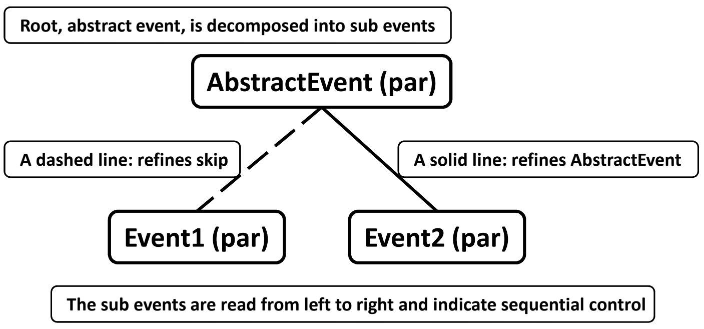

Root, abstract event, is decomposed into sub events
AbstractEvent (par)
A dashed line: refines skip
A solid line: refines AbstractEvent
Event1 (par)
Event2 (par)
The sub events are read from left to right and indicate sequential control

Figure 2: Atomicity decomposition diagram

on the left hand side of Figure 2 refines some machine *M0* which contains the abstract specification of *AbstractEvent*. The machine *M1* encodes its control flow (ordering between *Event1* and *Event2*) via guards on the events. This control flow is made explicit in the AD diagram presented on the right hand side. This diagram explicitly illustrates that the effect achieved by *AbstractEvent* at the abstract level, machine *M0*, is realized at the refined level, machine *M1*, by the occurrence of *Event1* followed by that of *Event2*. The ordering of the leaf events is always from left to right (this is based on JSD diagrams [Jac83]). The solid line indicates that *Event2* refines *AbstractEvent* while the dashed line indicates that *Event1* is a new event which refines *skip*. In the Event-B model of machine *M1* on the left hand side, *Event1* does not have any explicit connection with *AbstractEvent*, but the diagram indicates that we break the atomicity of *AbstractEvent* into two sub-events in the refinement. The parameter *par* in the diagram indicates that we are modelling multiple instances of *AbstractEvent* and its sub-events. Events associated with different values of *par* may be interleaved, thus modelling interleaved execution of multiple processes. The effect of an event with parameter *par* is to add the value of *par* to a set control variable with the same name as the event, i.e., $par \in Event1$ means that *Event1* has occurred with value *par*. The use of a set means that the same event can occur multiple times with different values for *par*. The guard of an event with parameter *par* specifies that the event has not already occurred for value *par* but the previous event has occurred, e.g., the guard of *Event2* says that *Event1* has occurred and *Event2* has not occurred with value *par*.

3 Steps for Constructing Traceable Event-B Models

This paper presents an approach for constructing traceable Event-B models using UML_B and the AD approach. We do not address the question of how requirements are arrived at. Requirements could have been arrived at using use cases [KG12]. In use cases, the system's functionality is described through structured stories in easy-to-understand text form, from which requirements can be derived. Our objectives are to provide a link between requirements and formal models and to facilitate building traceable Event-B formal models from requirements. The following subsections describe the steps proposed to achieve these goals.

3.1 Step 1: Classify Requirements

We classify requirements into the following five classes, based on the structure of Event-B models: data-oriented requirements, constraint-oriented requirements, event-oriented requirements, flow requirements, and other requirements. Each requirement can be placed in at least one category. A detailed description of the requirement classification, with examples of lift controller requirements taken from [Rob10], is given below.

Data-oriented requirements: requirements that describe attributes of nouns and the relationships between nouns. Here are three examples of this requirement class:

REQ1	Each **floor** has one **button** for requesting travelling to another floor
REQ2	The **lift-door** can be **closed** or **opened**
REQ3	The **lift** can be **moving** or **stopped**

The nouns " floor" and "button" in the requirement *REQ1* are identified as data-oriented requirement. The noun "lift-door" and the attributes "closed" and " opened" in the requirement *REQ2* are also identified as data-oriented requirement since they describe states of the door. Similarly, the noun "lift" and the attributes "moving" and "stopped" in the requirement *REQ3* are identified as data-oriented requirement since they describe states of the lift.

Constraint-oriented requirements: requirements that describe properties about the data that should always remain true. They are normally identified by keywords such as *never*, *must not*, *always* etc. The following is a constraint-oriented requirement:

REQ4	The lift door of a moving lift must be closed

REQ4 describes a system property relating the position of the lift door and the lift motion.

Event-oriented requirements: requirements that describe a function or activity of the system or its components. Events are normally identified by the "verbs", such as the following requirement:

REQ5	People on a floor press a button to request a lift

The verb "request" denotes that *REQ5* is of event-oriented type. The part of an event-oriented requirement that describes conditions under which an event can happen is called a guard requirement, whereas the part of an event-oriented requirement that describes how the data defining the state is going to change is called an action requirement.

Flow requirements: requirements that describe the flow of events. We can classify flow requirements generally into three types: sequence requirements which describe sequencing between operations, selection requirements which describe "if-then-else" structure to indicate the selection between two or more operations, and repetition requirements which describe the iteration of a particular operation multiple times. Table 1 provides examples of flow requirements:

Flow requirements	Example	Description
sequencing requirements	REQ6 The floor door closes before the lift is allowed to move	The relationship between the door closing operation and the lift moving operation can be seen as a sequence. After the lift-door closes, the lift is allowed to move.
selection requirements	REQ7 If a lift is stopped then the floor door for that lift may be open	In this requirement the lift door can be either opened or left closed when the lift is stopped.
repetition requirements	REQ8 There might be more than one external floor request in a particular floor, the lift will respond to them (stop) only once	Here, "more" indicates the iteration of the floor request operation.

Table 1: Description of flow requirements

The above classification seems to be more prevalent to many case studies. Nevertheless, flow requirements are not restricted to this classification, and other classes can be identified by analysing more case studies.

Other requirements: other requirements that do not fit into the previous classes can be considered in this class. This includes requirements that are very hard to model in Event-B, such as requirements that represent fairness properties or timing properties.

3.2 Step 2: Construct Semi-formal Artifacts and Develop Refinement Strategy

This step comprises of three stages, described in what follows.

3.2.1 Stage 1: Use semi-formal artifacts (UML-B, AD, and Structured English)

In the first stage, requirements are represented in a semi-formal notation depending on their type:

- **Data-oriented requirements** are represented using UML-B diagrams: nouns or attributes are represented using class diagrams, relationships between nouns are represented using

UML-B associations, and transitions between different attribute values are represented using state machine diagrams.

- **Constraint and Event-oriented** requirements are represented using structured English. The structured English is a way of breaking down constraint and event requirements into smaller requirements and mapping each sub-requirement to the corresponding requirement identifier to facilitate requirement traceability.

  The structured English representation for constraint-oriented requirements has the form:

  $$\boxed{\textbf{constraint}: \; < constraint \;\; requirement > \longrightarrow < REQ >}$$

  The structured English representation for event-oriented requirements has the form:

  $$\boxed{\begin{array}{l} \textbf{event} \;\; name \\ \textbf{guard}: \; < guard \;\; requirement > \longrightarrow < REQ > \\ \textbf{action}: \; < action \;\; requirement > \longrightarrow < REQ > \end{array}}$$

  In the above notation, the arrow is used for tracing back to the original requirement, and *REQ* denotes the requirement identifier.

- **Flow requirements** are mapped to the appropriate AD diagram according to Table 2 which summarises the behaviour of the AD patterns: sequence requirements are mapped to *sequence/and* diagrams, selection requirements are mapped to *or/xor* diagrams, and repetition requirements are mapped to *loop/all/some replicator* diagrams.

Pattern	Description
sequence-/and-constructor	execute events in a sequence. The difference between sequence- and and-constructor is that and-constructor executes all available events in any order, while the sequence constructor executes events in a particular order.
or -constructor	execute one or more events from two or more available events, in any order
xor- constructor	execute exactly one event from two or more
loop pattern	execute an event zero or more times
all-replicator	execute an event for all instances of a defined set
some-replicator	execute an event for one or more (some) instances of a defined set
one-replicator	execute an event for one instance of a defined set

Table 2: Description of the AD patterns

Representing requirements into graphical/structured English notation provides an intermediate level of tracing information and enables the validation of the model against the requirements.

Assuming that requirements are analysed based on the described requirement classification, the following examples illustrate how to represent each requirement class in a graphical or structured English notation.

The data-oriented requirement ***REQ1*** is represented as follows:

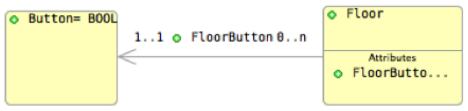


Figure 3: The class diagram for REQ1

The class *Floor* consists of the association *FloorButton* of type *Button* which is defined as a boolean that indicates whether there is a request for the lift to stop at that floor. The multiplicity property for the association *FloorButton* specifies a many-to-one relationship (i.e., total function). That is, there are n+1 floors and a boolean for each floor to indicate whether there is a request for the lift to stop at that floor.

The data-oriented requirement ***REQ2*** is represented using the state machine in Figure 4, which shows two states, “open” and “ close”, and two transitions *OpenLiftDoor* and *CloseLiftDoor*:

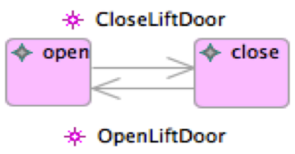


Figure 4: The state machine diagram for REQ2

The data-oriented requirement ***REQ3*** is represented using the state machine in Figure 5, which shows two states, “stopped” and “ moving”, and two transitions *LiftStop* and *LiftMoving*.

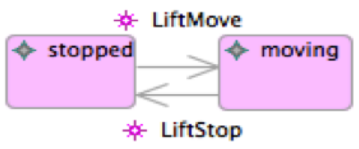


Figure 5: The state machine diagram for REQ3

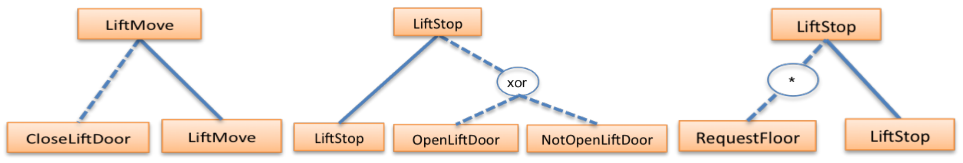
The constraint-oriented requirement ***REQ4*** is represented as follows:

constraint : $\mathit{The\ lift\ door\ of\ a\ moving\ lift\ must\ be\ closed} \longrightarrow \mathit{REQ4}$

The event-oriented requirement ***REQ5*** is represented as follows:

event $\mathit{RequestFloor}$
guard : $\mathit{a\ request\ at\ floor}\ \mathtt{f}\ \mathit{is\ made} \longrightarrow \mathit{REQ5}$
action : $\mathit{new\ request\ is\ added\ to\ the\ pool\ of\ pending\ requests} \longrightarrow \mathit{REQ5}$

The flow requirements ***REQ6***, ***REQ7*** and ***REQ8*** can be represented using the AD diagrams in Figure 6. Figure 6a shows that the behaviour of the $LiftMove$ event is exhibited by executing the *CloseLiftDoor* event followed by the *LiftMove* event. The xor-constructor pattern in Figure 6b indicates that the behaviour of the $LiftStop$ event in the root node is exhibited by executing the *LiftStop* event followed by either the *OpenLiftDoor* event or the *NotOpenLiftDoor* event. The latter event has skip action and is used to skip from applying any change to the lift-door status; this is because the “xor” pattern forces the execution of only one leaf. Finally, the behaviour of the $LiftStop$ event in Figure 6c is exhibited by executing the *RequestFloor* event multiple times followed by the *LiftStop* event.



(a) The ADD for REQ6 (b) The ADD for REQ7 (c) The ADD for REQ8

Figure 6: The AD diagrams for REQ6, REQ7 and REQ8

3.2.2 Stage 2: Merging the structured English of a single Event

It is possible that two or more structured English requirements refer to a single event. If such requirements exist, we merge them here. However, in this small case study we do not have requirements that refer to a single event. An example of this aspect is given by the following two requirements:

TSK1	Tasks can be created and destroyed
TSK2	Tasks are assigned priority when created

TSK1 and *TSK2* are event-oriented requirements that refer to a *Task_Create* event. The structured English of these requirements needs to be merged in this step.

3.2.3 Stage 3: Develop Refinement Strategy

Here we combine the AD diagrams developed in the first stage in order to organise the refinement levels. Flow is one criterion that can be considered in devising the refinement strategy. The nature of the requirements, the nature of the architecture that the refinement is aiming towards and the nature of the data types being refined are other important criteria that might come before the flow criterion since they may influence the flow requirements. The visualisation of the overall structure of the system gives more insight into the development of the refinement strategy before any Event-B modelling is carried out. It allows the developer to illustrate visually the hierarchy of the model based on the important criteria the developer is aiming at, and also helps to control the size of the model and view the number of events in each refinement level. Another advantage of the diagrammatic view of the refinement strategy is that it allows to visualise event dependencies and structure/variable dependencies. For example, in Event-B, events that update a particular variable should be introduced in the same modelling level. This is a restriction imposed by Event-B and is often only discovered during the modelling activity. The visual view of events given by the AD diagrams helps to deal with this restriction before modelling. Moreover, using this view, the developer can first introduce the basic properties of the system, and then introduce more complex properties that depend on the basic ones in the refinement levels. For instance, the developer of a real-time operating system (OS) can introduce basic properties of the processes used by the application developer in the abstract model, and complex properties that are used by the real-time OS to handle the processes in the refinement levels.

Figure 7 shows the refinement levels for the lift controller case study.

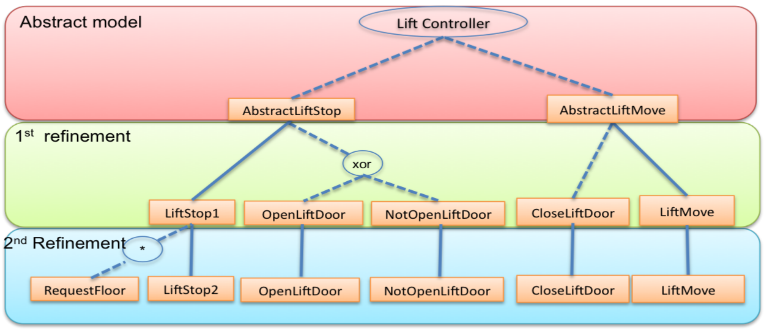


Figure 7: The combined AD diagrams for the lift controller

In the abstract level, we decided to model two abstract events: *AbstractLiftStop* and *AbstractLiftMove*. We use a sequence pattern to indicate the sequencing between the abstract events. In the first refinement, we decided to combine the tree structure with root $AbstractLiftStop$ that corresponds to the AD diagram in Figure 6b and the tree structure with root *AbstractLiftMove* that corresponds to the AD diagram in Figure 6a. Finally, we use the AD diagram in Figure 6c to refine the *LiftStop1* event. We note that, because of a restriction in the AD tool, event names in the combined AD diagrams are changed slightly from their names in the individual AD diagrams. The AD tool automatically generates flags and gluing invariants according to event names. The gluing invariants show the relationship between the abstract flags and concrete flags. Therefore, the flag names for the abstract events must be different from the flag names for the refined events.

3.3 Step 3: Construct Formal Models

In this step, we use the UML-B and AD tools to convert the diagrams of step 2 to Event-B notation. We also manually convert the structured English representation of step 2 into Event-B.

The Event-B specification of the requirements *REQ1*, *REQ2*, and *REQ3*, generated from the class and state machine diagrams, is given below. The sets, constants and axioms generated from these diagrams are as follows:

SETS
$Floor\_SET, door\_STATES, lift\_STATES$
CONSTANTS
$open$
$close$
$moving$
$stopped$
AXIOMS
$open.type\ \ open \in door\_STATES$
$close.type\ \ close \in door\_STATES$
$distinctStates\_door\_STATES\ \ partition(door\_STATES, \{open\}, \{close\})$
$distinctStates\_lift\_STATES\ \ partition(lift\_STATES, \{moving\}, \{stopped\})$
End

Figure 8: Sets, constants and axioms generated from the class and state machine diagrams

Figure 3 contains a class represented by the variable *Floor*. This variable is defined as a subset of $Floor\_SET$, which represents the set of all possible instances of *Floor*. The set of instances of the *Button* class is defined as the boolean type. The UML-B associations are translated into variables whose type is a function from the class set to the attribute type. Hence, *FloorButton* in Figure 3 is translated into a function from *Floor* to *Button*. The multiplicity of an association determines the type of the function: partial, total, injective etc. Here $(0..n \rightarrow 1..1)$ is translated into a total function that maps *Floor* to *Button*. The state machine in Figure 4 is translated into Event-B as shown in axiom $distinctstates\_door\_STATES$ of Figure 8. The states *open* and *close* are translated into constants of type $door\_STATES$. Each transition is translated into an event whose guard specifies the source state and whose actions specify its target state. Hence, *OpenLiftDoor* is an event that changes the lift door from *close* state to *open* state and *CloseLiftDoor* is an event that changes the lift door from *open* state to *close* state. The state machine in Figure 5 is translated in a similar manner to the state machine in Figure 4.

variables
$Floor\ \ Button\ \ FloorButton\ \ door\ \ lift$
invariants
$Floor \in \mathbb{P}(Floor\_SET)$
$Button = BOOL$
$FloorButton \in Floor \rightarrow Button$
$door \in Floor \rightarrow door\_STATES$
$lift \in lift\_STATES$

event *CloseLiftDoor*
any *self*
where
$self \in Floor$
$door(self) = open$
then
$door(self) := close$
end

event *OpenLiftDoor*
any *self*
where
$self \in Floor$
$door(self) = close$
then
$door(self) := open$
end

event *LiftMove*
where
$lift = stopped$
then
$lift := moving$
end

event *LiftStop*
where
$lift = moving$
then
$lift := stopped$
end

Figure 9: Variables, invariants and events generated from the class and state machine diagrams

The Event-B specification written manually for the requirement *REQ4* is:

$$\boxed{\forall f.f \in dom(door) \wedge lift = moving \implies door(f) = close}$$

Also, the structured English for the requirement *REQ5* can be formalised manually as follows:

```
event  $RequestFloor$
$any\ \ f$
where
  $grd1\ \ f \in Floor \setminus request$
then
  $act1\ \ request := request \cup \{f\}$
```

Figure 10: The Event-B specification of the structured English for the requirement *REQ5*

The Event-B specification generated from the AD diagram for the requirement *REQ8* is:

```
event  $RequestFloor$
where
  $LiftStop2 = FALSE$
end
```

```
event  $LiftStop2$  refines  $LiftStop1$
where
  $LiftStop2 = FALSE$
then
  $LiftStop2 = TRUE$
end
```

Figure 11: The Event-B specification generated from the AD diagram (loop pattern)

According to the loop pattern rule, the *RequestFloor* event can be executed zero or more times before the execution of the *LiftStop* event. Thus, the *RequestFloor* event does not have a variable and an action to record the loop execution. It only has one guard $LiftStop2 = FALSE$ that allows zero executions of the loop event. We need to make a slight change to this pattern to allow the *RequestFloor* event to be executed at least one time before the execution of the *LiftStop* event. This can be achived by adding manually a boolean flag *RequestFloor* together with the action *RequestFloor:=TRUE* in the *RequestFloor* event instead of the guard *LiftStop2 = FALSE* in the *RequestFloor* event. Also we add the guard *RequestFloor=TRUE* to the *LiftStop* event to check the execution of the *RequestFloor* event. That way, *RequestFloor* must be executed at least one time before the *LiftStop* event. This modification can be considered as a new repetition pattern that allows the execution of an event one or more times before the execution of other events. Clearly, there is a need to investigate different AD patterns for different requirement types.

The following gluing invariant is generated by the AD tool for the leaf with solid line in the loop AD diagram in Figure 6c:

$$\boxed{LiftStop2 = LiftStop1}$$

The gluing invariant defines the relationship between the abstract flag *LiftStop1* and the concrete flag *LiftStop2* and is used to discharge the refinement proof obligation.

Requirements *REQ6* and *REQ7* are dealt with in a similar way. The generated Event-B models from AD diagrams and UML-B diagrams can be combined using shared-event composition [SB11]. This concludes the application of the general approach described in this section.

We note that the presented approach has been applied also to a larger case study of 32 requirements. The case study is an Event-B specification for queue management in FreeRTOS. FreeRTOS [Bar10] is an open source, mini kernel developed by Richard Barry to serve real-time application requests. Data transfer is established by means of queues. Queues are mechanisms used to serve communication between a task-to-task or a task-to-Interrupt Service Routine (ISR) [Bar10]. The requirement categorisation for the queue management case study was useful: 9 requirements were classified as data-oriented requirements, 3 as constraint-oriented requirements, 20 as event-oriented requirements and 7 as flow requirements. Four conclusions were drawn from the application of our approach to the queue management case study. Firstly, we found that flow requirements can sometimes be extracted from more than one requirement. For example, the sequencing between the event responsible for sending an item successfully to a queue in the requirement *QUE1* and the event responsible for removing the highest-priority task from the collection of tasks waiting to receive in the requirement *QUE2*.

QUE1	A task can only send items to a queue when there is enough room in the queue
QUE2	When a queue becomes available (there is an item in the queue to be received) then the highest-priority task waiting for item to arrive on that queue will be removed from the collection of tasks waiting to receive

Secondly, AD patterns do not cover all possible flows, we sometimes need to modify them to represent the exact flow we are looking for, or even explore some new patterns. For example, one might need to represent “one or more” executions of an event. This is currently not supported by the existing patterns, however, the loop AD pattern together with an additional manual flag can be used to represent this particular case. Thirdly, it is possible that a particular event becomes a leaf in different AD diagrams. In some cases however, it is necessary to change the name of the recurrent leaf to avoid an invalid combination of AD flags. Assume that an event x is a leaf in a sequence diagram and also a leaf in an “xor” diagram. If this leaf has the same name in both trees, then the AD tool will generate “xor” flags and sequence flags for the event x. Mixing flags together in a single event can result in mis-behaviour of the intended flows. Overall, further investigations should be carried out to evaluate the presented approach and to explore more useful patterns for managing flows. Finally, the proposed approach allows different requirements to be represented in the same modelling element (variable, event, etc). For example:

TSK3	Task is an object in FreeRTOS
TSK4	Task can send an item to a queue

Task is a shared variable between the requirement *TSK3* and the requirement *TSK4*. *TSK3* is classified as a data-oriented requirement whereas *TSK4* is classified as an event-oriented requirement.

4 Related Works

This section presents three works in the area of requirements traceability. SOFL (Structured Object-Oriented Formal Language) [Liu04] is an approach that uses graphical and textual formal notation for system construction. It is an integration of Data Flow Diagrams, Petri Nets, and VDM-SL. The graphical and textual formal notation serve as a good communication mechanism between a user and a developer. One of the main differences between our work and [Liu04] is that our work makes use of structured English and graphical notation represented in UML-B and the AD approach to bridge the gap between requirements and Event-B models, whereas the semi-formal artifacts used in the SOFL approach are used to document requirements.

Jastram et al [JHLG10] presented another approach to achieving requirement traceability. They structure the requirements based on WRSPM. WRSPM is a model used for the formalisation of system requirements. It differentiates between phenomena (state space and transitions of the system) and artifacts (the restriction on states and transitions). The artifacts are classified into groups: Domain Knowledge (W), Requirements (R), Specifications (S), Program (P) and Programming Platform (M). Once the requirements are structured using WRSPM, the second step is to use a formal model for system specification. WRSPM elements are mapped to Event-B. This mapping provides a way for traceability between requirements and the Event-B model. They distinguish three types of possible traces: evolution traces, explicit traces, and implicit traces. Evolution traces are explored through the requirement evolution over time. Explicit traces are used to link each non-formal requirement to a formal statement. Implicit traces are discovered via refinement relationships, references to model elements or proof obligations. The main difference between our approach and the [JHLG10] approach is that the latter focuses more on traceability and uses intermediate constructs based on WRSPM to provide traceability between requirements and Event-B models. On the other hand, the intermediate constructs which we use are based on a requirement classification derived from Event-B components. As a result, the process of converting the semi-formal artifacts into an Event-B model is straightforward. Moreover, our approach focuses not just on building Event-B models but also on traceability.

Yeganefard and Butler [YB12] described an approach for structuring requirements of control systems to facilitate refinement-based formalisation. The approach has three stages: In the first stage, requirements are categorised into monitored (MNR) requirements, commanded (CMN) requirements and controlled (CNT) requirements. The second step involves layering requirements by modelling one feature in each refinement level; the developer chooses which feature to model in each refinement level. The authors suggest modelling the main role of the system with a minimum set of requirements in the very abstract model. The third step is based on revising the requirement document and the formal model to investigate any inconsistent, ambiguous or missing requirements. Comparing our work with [YB12], the approach used in [YB12] is specific to control systems whereas the approach of this paper is based on Event-B structures. We also think that structuring refinement levels based on a textual requirement document is difficult. We believe that the visualisation of Event-B components using AD diagrams gives a clear overview of the whole system and helps decide which feature to model in each refinement level. It is possible to combine our approach with that of [YB12] to obtain more effective guidelines for developing traceable Event-B models for control systems.

5 Conclusions

We presented an approach which facilitates constructing Event-B models and provides clear traceability between requirements and the Event-B model. The approach is based on the use of the UML-B and AD approaches. UML-B provides UML graphical modelling environment that allows the development of an Event-B model, whereas the AD approach provides a graphical notation to structure refinement and manage flows in an Event-B model.

Applying UML-B at the requirement level facilitates the mapping from data-oriented requirements to Event-B. Event-B models of the UML-B diagrams are generated automatically by the UML-B tool. On the other hand, applying the AD approach at the requirement level assists a developer in the process of deciding which features to be modeled in each refinement step. Moreover, the Event-B model is generated automatically by the AD tool, which reduces the burden of the manual work especially in the development of complex systems. The combined AD diagrams provide an overall visualisation of the refinement structure and demonstrate the relationships between events even before any model is written.

6 Future work

The application of the proposed approach to several case studies is the primary goal of future work. In this paper we describe one kind of constraint-oriented requirements, namely requirements on the system being developed, such as requirement *REQ4*. We also need to investigate another type of constraint-oriented requirements, which describe assumptions on the environment, such as the following requirement:

REQ9	The lift can transition from stopped to moving-up or moving-down, from moving-up or moving-down to stopped, but not from moving-up to moving-down or vice versa

Exploring the scalability of the graphical models is another direction for future work. The visual view of the refinement strategy provides some support for scalability: the ADD diagrams are hierarchical and it is always possible to partition the diagram into sub-hierarchies; UML-B class diagrams can also be layered through refinement. Further work is needed to investigate the scalability issue. Finally, further investigation of several AD patterns is necessary to support a larger class of flow requirements.

Acknowledgements: Butler and Salehi were partly funded by the FP7 ADVANCE Project (287563), http://www.advance-ict.eu/.

Bibliography

[ABH+10] J. Abrial, M. Butler, S. Hallerstede, T. Hoang, F. Mehta, L. Voisin. Rodin: an open toolset for modelling and reasoning in Event-B. *STTT* 12(6):447–466, 2010.

[ABHV06] J. Abrial, M. Butler, S. Hallerstede, L. Voisin. An open extensible tool environment for Event-B. In *ICFEM 2006*. LNCS, pp. 588–605. Springer, 2006.

[Abr10] J. Abrial. *Modeling in Event-B - System and Software Engineering*. Cambridge University Press, 2010.

[BA05] F. Badeau, A. Amelot. Using B as a high level programming language in an industrial project: roissy VAL. In *Proc. 4th International Conference on Formal Specification and Development in Z and B*. ZB'05, pp. 334–354. Springer, 2005.

[Bar10] R. Barry. The FreeRTOS Project. http://www.freertos.org/, 2010.

[But09a] M. Butler. Decomposition Structures for Event-B. In *Proc. 7th International Conference on Integrated Formal Methods*. IFM '09, pp. 20–38. Springer, 2009.

[But09b] M. Butler. Using Event-B Refinement to Verify a Control Strategy. *University of Southampton*, 2009.

[FBR12] A. Fathabadi, J. Butler, A. Rezazadeh. A Systematic Approach to Atomicity Decomposition in Event-B. In *SEFM*. Pp. 78–93. 2012.

[Jac83] M. Jackson. *System Development*. Prentice Hall, Englewood Cliffs, 1983.

[JHLG10] M. Jastram, S. Hallerstede, M. Leuschel, A. G. Russo. An Approach of Requirements Tracing in Formal Refinement. In *VSTTE*. Pp. 97–111. 2010.

[KG12] D. Kulak, E. Guiney. *Use Cases: Requirements in Context*. Pearson Education, 2012.

[Liu04] S. Liu. *Formal Engineering for Industrial Software Development: Using the SOFL Method*. Springer, 2004.

[Rob10] K. Robinson. *System Modelling and Design*. Draft book on Event-B, 2010.

[SB06] C. Snook, M. Butler. UML-B: Formal modeling and design aided by UML. *ACM Trans. Softw. Eng. Methodol*, pp. 92–122, 2006.

[SB08] C. Snook, M. Butler. UML-B: A Plug-in for the Event-B Tool Set. In *Proc. 1st international conference on Abstract State Machines, B and Z*. ABZ '08, pp. 344–344. Springer, 2008.

[SB11] R. Silva, M. Butler. Shared event composition/decomposition in Event-b. In *Proceedings of the 9th international conference on Formal Methods for Components and Objects*. FMCO'10, pp. 122–141. Springer, 2011.

[YB12] S. Yeganefard, M. Butler. Control Systems: Phenomena and Structuring Functional Requirement Documents. In *ICECCS*. Pp. 39–48. 2012.