Building Traceable Event-B Models from Requirements

Eman Alkhammash\textsuperscript{a}, Michael Butler\textsuperscript{b}, Asieh Salehi Fathabadi\textsuperscript{b}, Corina Cirsiea\textsuperscript{b}

\textsuperscript{a}Department of Computer Science, Taif University, KSA
\textsuperscript{b}Electronics and Computer Science, University of Southampton, UK

Abstract

Bridging the gap between informal requirements and formal specifications is a key challenge in systems engineering. Constructing appropriate abstractions in formal models requires skill and managing the complexity of the relationships between requirements and formal models can be difficult. In this paper we present an approach that aims to address the twin challenges of finding appropriate abstractions and managing traceability between requirements and models. Our approach is 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. In the stepwise refinement approach, design details are gradually introduced into formal models. Stepwise refinement allows each requirement to be introduced at the most appropriate stage in the development. Our approach makes use of the UML-B and Event Refinement Structures (ERS) approaches. UML-B provides UML graphical notation that enables the development of data structures for Event-B models, while the ERS approach provides a graphical notation to illustrate event refinement structures and assists in the organisation of refinement levels. The ERS 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 ERS approaches to facilitate constructing Event-B models from requirements and provide traceability between requirements and Event-B models.

Keywords: Event-B, Traceability, UML-B, Event Refinement Structure (ERS)

1. Introduction

We present an approach for incrementally constructing a formal model from informal requirements with the aim of retaining traceability to requirements in models. The approach helps to identify appropriate 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 identified and captured by the model.

Our approach is based on the Event-B formal method [1]. Event-B is a refinement-based formal method with tool support for developing various kinds of systems. We make use of UML-B [2] and Event Refinement Structure (ERS) [3, 4]. UML-B provides graphical modelling based on UML which supports the development of an Event-B formal model. The visual view of the system provided by UML-B and ERS assists in the development of the refinement strategy before the actual work on modelling is performed. The combined ERS diagrams, which show the overall refinement structure of the system, can be modified until an acceptable refinement structure is reached. In addition, the ERS 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 ERS diagrams and UML-B diagrams can be generated automatically by the ERS plug-in [5] and the UML-B plug-in [6].
Our approach comprises of three stages, which are shown in Figure 1.

![Figure 1: Steps for constructing traceable formal models](image)

The first step in our approach is requirement classification. Requirements are classified based on Event-B modelling structures. 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, ERS diagrams and structured English to represent requirements. UML-B is used to represent data-oriented requirements. ERS 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-requirement to the appropriate requirement 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 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 ERS 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 ERS tool to generate Event-B models and also manually write the corresponding Event-B from the structured English representation.

The work presented in this paper is an extension of [7]. In this paper we apply our approach to a new case study (queue management in an operating system) and add more information regarding the verification of constructed Event-B models using Rodin. In addition we discuss the use of the shared-event composition technique to combine Event-B models generated from UML-B diagrams and Event-B models generated from ERS diagrams. Moreover, some relevant formal developments of operating systems by other researchers are discussed.

This paper is structured as follows: Section 2 gives an overview of Event-B, UML-B, ERS and shared event composition in Event-B. The description of the presented approach is introduced in Section 3. The application of the proposed approach to a queue management case study is introduced in Section 4. Section 5 introduces some related work in requirements traceability. Conclusions and future work are drawn 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 predicate logic to provide a formal notation for the creation of models of discrete systems and the undertaking of refinement steps. Event-B is supported by the Rodin toolset [8, 9], which includes various plugins for features such as theorem-proving, model-checking, model composition and decomposition and translation of diagrammatic representations to textual representation. 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. Event-B models contain two constructs: a context and a machine. The context is the static part of a model in which fixed properties of the model (sets, constants, axioms) are defined. The dynamic 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 specify ways in which the variables can change.

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. Class diagrams in UML-B may contain attributes (variables), associations (relationships between two classes), events and state machines (transitions between states). State machine diagrams describe the behaviour of instances of classes as transitions linked to events.

UML-B supports 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.

2.3. Event Refinement Structure 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 ERS 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 ERS approach has another advantage, namely that it allows event ordering to be represented explicitly. Figure 2 illustrates these two features of the ERS graphical notation.

![Figure 2: Event Refinement Structure diagram](image-url)
Assume machine $M_1$ on the left hand side of Figure 2 refines some machine $M_0$ which contains the abstract specification of $AbstractEvent$. The machine $M_1$ encodes its control flow (ordering between $Event1$ and $Event2$) via guards on the events. This control flow is made explicit in the ERS diagram presented on the right hand side. This diagram explicitly illustrates that the effect achieved by $AbstractEvent$ at the abstract level, machine $M_0$, is realised at the refined level, machine $M_1$, 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 [10]). 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 $M_1$ 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 $par$.

2.4. Shared event composition

Composition [11] is the process of composing several sub-models in a variety of styles. Shared event composition [11] enables sub-models to interact through event synchronisation. Several events can be composed in a single event. The composed event includes the conjunction of guards of sub-model events and combines the actions of sub-model events. The approach has a restriction that prevents the sub-models to include any shared variables. A tool has been developed to support this style of composition in Rodin [11].

![Figure 3: Shared event composition](image)

Figure 3 illustrates this style; suppose we have a model $M_1$ that has events $e1$, $e2$ with variable $v1$. Model $M2$ has events $e3$ and $e4$ with variable $v2$. $e2$ updates $v1$ and $e3$ updates $v2$ and $v1$ and $v2$ are independent variables. Event $e2$ in $M1$ and event $e3$ in $M2$ have the following form:

\[
\begin{align*}
e2 &\triangleq \text{any } p \\
&\text{ where } G1(v1, p) \\
&\text{ then } v1 := E1(v1, p) \\
&\text{ end}
\end{align*}
\]

\[
\begin{align*}
e3 &\triangleq \text{any } p \\
&\text{ where } G2(v2, p) \\
&\text{ then } v2 := E2(v2, p) \\
&\text{ end}
\end{align*}
\]

We can compose $M1$ and $M2$ such that events $e2$ of $M1$ and $e3$ of $M$ are synchronised. The resulting composed model will share the two independent variables: $v1$ and $v2$ and event $e2$ from model $M1$ and $e3$ from model $M2$ are composed by conjoining their guards and combining their actions.
The resulting composed event has the following form:

\[ e_2 \parallel e_3 \triangleq \text{any } p \]

where

\[ G_1(v_1, p) \]
\[ G_2(v_2, p) \]

then

\[ v_1 := E_1(v_1, p) \]
\[ v_2 := E_2(v_2, p) \]

end

3. Steps for Constructing Traceable Event-B Models

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 [12], is given below.

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

<table>
<thead>
<tr>
<th>Lift</th>
<th>Requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lift1</td>
<td>Each floor has one button for requesting travelling to another floor</td>
</tr>
<tr>
<td>Lift2</td>
<td>The lift-door can be closed or opened</td>
</tr>
<tr>
<td>Lift3</td>
<td>The lift can be moving or stopped</td>
</tr>
</tbody>
</table>

The nouns “floor” and “button” in the requirement Lift1 are identified as data-oriented requirement. The noun “lift-door” and the attributes “closed” and “opened” in the requirement Lift2 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 Lift3 are identified as data-oriented requirement since they describe states of the lift.

**Constraint-oriented requirements**: These are 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 are constraint-oriented requirements:

<table>
<thead>
<tr>
<th>Lift</th>
<th>Requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lift4</td>
<td>The lift door of a moving lift must be closed</td>
</tr>
<tr>
<td>Lift5</td>
<td>The building has 3 floors</td>
</tr>
</tbody>
</table>

Lift4 describes a system property relating the position of the lift door and the lift motion whereas Lift5 describes a property about the maximum floor number.

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

<table>
<thead>
<tr>
<th>Lift</th>
<th>Requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lift6</td>
<td>People on a floor press a button to request a lift</td>
</tr>
</tbody>
</table>

The verbs “press” and “request” denote that Lift6 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**: These are 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:

<table>
<thead>
<tr>
<th>Flow requirements</th>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequencing requirements</td>
<td>LIFT7 The floor door closes before the lift is allowed to move</td>
<td>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.</td>
</tr>
<tr>
<td>selection requirements</td>
<td>LIFT8 If a lift is stopped then the floor door for that lift may be open</td>
<td>In this requirement the lift door can be either opened or left closed when the lift is stopped.</td>
</tr>
<tr>
<td>repetition requirements</td>
<td>LIFT9 There might be more than one external floor request in a particular floor, the lift will respond to them (stop) only once</td>
<td>Here, “more” indicates the iteration of the floor request operation.</td>
</tr>
</tbody>
</table>

Table 1: Description of flow requirements

Flow requirements are not restricted to this classification. Other classes can be identified by analysing more case studies. Moreover, it is sometimes useful to group two or more requirements together that show a particular flow. This is because flow requirements can sometimes be extracted from more than one requirement, and to clarify this, we need to group requirements that show a particular flow together, and separate them from other requirement classes. This point is illustrated in Section 4.

Other requirements: other requirements that do not fit into the previous classes can be considered in this class. This includes requirements that are more difficult 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, ERS, 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:

```
constraint: < constraint requirement > --------> < REQ >
```

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

```
event name
guard: < guard requirement > --------> < REQ >
action: < action requirement > --------> < REQ >
```

In the above notation, the arrow is used for tracing back to the original requirement, and \(REQ\) denotes the requirement identifier of the original requirement.
• Flow requirements are mapped to the appropriate ERS diagram according to Table 2 which summarises the behaviour of the ERS patterns. ERS has constructs that set that support several kinds of flows, therefore 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.

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequence-/and- constructor</td>
<td>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.</td>
</tr>
<tr>
<td>or -constructor</td>
<td>execute one or more events from two or more available events, in any order</td>
</tr>
<tr>
<td>xor- constructor</td>
<td>execute exactly one event from two or more</td>
</tr>
<tr>
<td>loop pattern</td>
<td>execute an event zero or more times</td>
</tr>
<tr>
<td>all-replicator</td>
<td>execute an event for all instances of a defined set</td>
</tr>
<tr>
<td>some-replicator</td>
<td>execute an event for one or more (some) instances of a defined set</td>
</tr>
<tr>
<td>one-replicator</td>
<td>execute an event for exactly one instance of a defined set</td>
</tr>
</tbody>
</table>

Table 2: Description of the ERS patterns

Representing requirements with graphical/structured English notation provides an intermediate level between requirements and models 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 LIFT1 which is introduced in step1 is represented with a class diagram as shown in Figure 4:

![Floor class diagram](image)

Figure 4: The class diagram for LIFT1

The class Floor consists of the attribute FloorButton of type boolean that indicates whether there is a request for the lift to stop at that floor. The multiplicities correspond to the mathematical categorisations of functions: partial, total,..etc. The multiplicity of the attribute FloorButton and instance set for Floor is (0..n). (0..n) means that there is a boolean value for each floor.

The data-oriented requirement LIFT2 is represented using the state machine in Figure 5, which shows two states, “open” and “close”, and two transitions OpenLiftDoor and CloseLiftDoor:

![State machine diagram](image)

Figure 5: The state machine diagram for LIFT2

The data-oriented requirement LIFT3 is represented using the state machine in Figure 6, which shows two states, “stopped” and “moving”, and two transitions LiftStop and LiftMoving.
The constraint-oriented requirement $LIFT4$ is represented as an English constraint as shown in Figure 7:

\[
\text{constraint: The lift door of a moving lift must be closed} \rightarrow LIFT4
\]

Figure 7: The structured English of $LIFT4$

The constraint-oriented requirement $LIFT5$ is represented as an English constraint as shown in Figure 8:

\[
\text{constraint: The building has 3 floors} \rightarrow LIFT5
\]

Figure 8: The structured English of $LIFT5$

The event-oriented requirement $LIFT6$ is represented as structured English as shown in Figure 9:

\[
\begin{align*}
\text{event} & \text{ RequestFloor} \\
\text{guard:} & \text{ a request at floor } f \text{ is made} \rightarrow LIFT6 \\
\text{action:} & \text{ new request is added to the pool of pending requests} \rightarrow LIFT6
\end{align*}
\]

Figure 9: The structured English of requirement $LIFT6$

The flow requirements $LIFT7$, $LIFT8$ and $LIFT9$ can be represented using the ERS diagrams in Figures (10-12). Figure 10 shows that the behaviour of the LiftMove event is realised by executing the CloseLiftDoor event followed by the LiftMove event. The xor-constructor pattern in Figure 11 indicates that the behaviour of the LiftStop event in the root node is realised by executing the LiftStop event followed by either the OpenLiftDoor event or the NotOpenLiftDoor event. Finally, the behaviour of the LiftStop event in Figure 12 is exhibited by executing the RequestFloor event multiple times followed by the LiftStop event. Note that the $LIFT9$ requires one or more occurrences of the RequestFloor event whereas the ERS pattern in Figure 12 allows zero or more occurrences. On Page 12 we show how the model is strengthened to address this.

\[
\begin{align*}
\text{LiftMove} & \quad \text{CloseLiftDoor} \\
\text{LiftMove} & \quad \text{LiftMove} \\
\text{LiftStop} & \quad \text{OpenLiftDoor} \quad \text{NotOpenLiftDoor} \\
\text{LiftStop} & \quad \text{RequestFloor} \quad \text{LiftStop}
\end{align*}
\]

Figure 10: The ERS diagram for $LIFT7$. Figure 11: The ERS diagram for $LIFT8$. Figure 12: The ERS diagram for $LIFT9$.

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 in the structured English. However, in this small lift case study we do not have requirements that refer to a single event. An example of this merging is given by the following two requirements:

\[
\begin{align*}
\text{TSK1} & \quad \text{Tasks can be created and destroyed} \\
\text{TSK2} & \quad \text{Tasks are assigned priority when created}
\end{align*}
\]

TSK1 and TSK2 are event-oriented requirements that refer to a Task_Create event. The structured English of these requirements are merged in this step.
### 3.2.3. Stage 3: Develop Refinement Strategy

Here we combine the ERS diagrams developed in the first stage in order to organise the refinement levels then we layer UML-B diagrams based on the combined ERS diagram. 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 complexity 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 dependencies to be visualised. 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 ERS diagrams helps to deal with this restriction before modelling. For instance, if the composed ERS includes two events: *addToA* and *RemoveFromA*. Both events must appear in the same refinement level of the composed ERS diagram. It is easy to notice this in the composed ERS diagram and it is important to put appropriate names for such events to make it easy to notice variable dependencies. 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 13 shows the refinement levels for the lift controller case study.

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 ERS diagram in Figure 11 and the tree structure with root *AbstractLiftMove* that corresponds to the ERS diagram in Figure 10. Finally, we use the ERS diagram in Figure 12 to refine the *LiftStop1* event.

After organising refinement strategy then we layer UML-B diagrams based on the combined ERS diagram. This will facilitate integrating Event-B models generated from UML-B and Event-B models generated from ERS in Step3. Figure 14 shows the layered UML-B diagrams.

---

**Figure 13:** The combined ERS diagrams for the lift controller
3.3. Step 3: Construct Formal Models

In stage 3, we organise the refinement levels and structure the hierarchy of the class diagrams according to the combined ERS diagram. In this step, we use the UML-B and ERS 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 model for the class and state machine diagrams are generated using UML-B since UML-B supports the refinement concepts. Each UML-B machine gives rise to both an Event-B context and an Event-B machine. Similarly, each ERS machine gives rise to an Event-B context and an Event-B machine. Thus, we have three contexts generated by the UML-B: $c0$, $c1$ that sees $c0$, and $c2$ that sees $c1$. We also have three machines generated by the UML-B, the abstract machine $m0$, the first machine $m1$ that refines $m0$, and the second machine $m2$ that refines $m1$. Similarly, the ERS gives rise to three contexts: $c0$, $c1$ and $c3$ and three machines: $m0$, $m1$, and $m2$.

This subsection present examples of Event-B models generated by the UML-B and ERS tools, Event-B model corresponds to the structured English, and the process of integrating Event-B models generated by the UML-B and the ERS tools.

The Event-B specification of the semi-formal artefacts represented in Figure 4, Figure 5, and Figure 6, that is generated from the class and state machine diagrams, is given in Figure 15.

![Figure 14: Lift controller layered UML-B diagrams](image)

![Figure 15: Sets, constants and axioms generated from the class and state machine diagrams](image)
sets, constants, axioms, variables, invariants, and events. The Event-B model (Figure 16) of state machines does not only show the states of the system. It also captures the events that specify how the variables can change. Figure 4 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. FloorButton in Figure 4 is translated into a function from Floor to BOOL. The state machine in Figure 5 is translated into Event-B as disjoint sets representation as shown in axiom @distinctstates door STATES. 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 the close state to the open state and CloseLiftDoor is an event that changes the the lift door from the open state to the close state. The state machine in Figure 6 is translated in a manner similar to the state machine in Figure 5.

![Figure 16: Variables, invariants and events generated from the class and state machine diagrams](image)

The structured English that represents constraint-oriented requirements are converted into the appropriate Event-B elements: invariants or axioms. Invariants are predicates that specify constraints about the variables whereas axioms are predicates that specify constraints about the constants.

The structured English of Figure 7 specifies constraints about the position of the lift door and the lift motion which are variables and therefore, it is represented as the following invariant:

\[ \text{lift moving} \Rightarrow \text{door = close} \]

The structured English of Figure 8 specifies assumptions about a constant (i.e. the number of floors). Therefore, it is represented as the following axioms:

\[
\begin{align*}
\text{MaxFloor} &= 3 \\
\text{Floor} &= 0..\text{MaxFloor}
\end{align*}
\]

On the other hand, the structured English that represents event-oriented requirements are converted into events. Events are guarded actions specifying ways in which the variables can change. Sometimes, the variables of the constructed event are already defined in the model generated by the UML-B and the developer only needs to set those variables to appropriate values in the event actions. However, in some cases, the variables of an event might not exist in the model generated by the UML-B tool and therefore such variables need to be defined manually such as the case of the variable requests which is captured by the structured English of Figure 9.
ERS diagrams that represent flow-oriented requirements are converted automatically to variables, invariants, and events.

The events generated from the ERS diagram of Figure 12 are:

```plaintext
variables requests
invariants requests \subseteq Floor

event RequestFloor
where
  grd1 f \in Floor \setminus requests
then
  act1 requests := requests \cup \{f\}
end
```

**Figure 17:** The Event-B specification of the structured English of Figure 9

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 achieved by manually adding 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 further ERS patterns for different requirement types.

So far, we obtained two generated Event-B models: The first one is the model generated by the UML-B tool and the second one is the model generated by the ERS tool. The generated models can be combined using shared-event composition. Shared-event composition merges the variables and the invariants of each of the Event-B models. In each composed machine, events of the model generated by the UML-B are synchronised with events of the model generated by the ERS tool. Figure 19 shows the composed machine cm0 of UML-B.m0: the abstract machine generated by the UML-B tool and ERS.m0: the abstract machine generated by the ERS tool.

The composed machine cm0 includes machines UML-B.m0 and ERS.m0. Some invariants of cm0 are generated in order to specify the type of variables. @UML-B.lift.type is type invariant of lift variable in UML-B.m0 machine whereas @ERS.LiftMove.type and @ERS.LiftStop.type are type invariants of the control variables LiftMove and LiftStop in the ERS.m0 machine. The invariant @ERS.LiftStop_Seq is an invariant in ERS.m0 machine that maintains sequence ordering between AbstractLiftStop event and AbstractLiftMove event. Events of cm0 are identified as the parallel composition (interaction) of UML-B.m0 events: LiftStop and LiftMove and ERS.m0 events: AbstractLiftStop and AbstractLiftMove.

Figure 18: The Event-B specification generated for the ERS diagram (loop pattern) of Figure 12
3.4. Verification of Event-B models

The UML-B tool and the ERS tool generate Event-B models corresponding to UML-B and ERS developments, and some of the Event-B models that correspond to events and invariants are manually constructed. After the construction of Event-B models, the Rodin tool is then used for verification purposes. Rodin includes automatic tools to generate proof obligations associated with the generated Event-B models to ensure that the Event-B model is constructed correctly in a consistent manner [9]. Rodin also includes provers [9] that attempt to automatically discharge these obligations. Rodin generates proof obligations to verify well-definedness and invariant preservation as well as correctness of refinement steps. Each component in Event-B (variable, event, etc) has well-defined semantics. For example, \( \text{door} \in \text{door STATES} \), where \( \text{door STATES} \) is defined as: \( \text{partition}(\text{door STATES}, \text{open}, \text{close}) \), allows us to infer that door has only one of the two constant values \text{open} or \text{close}. Invariant preservation proof obligations ensures that events preserve invariants on the variables. The invariant will be true before the event is executed and must remain true when the event terminates. For example, the invariant:

\[
lift = \text{moving} \implies \text{door} = \text{close}
\]

must be true before the \text{CloseLiftDoor} event is executed and must remain true when \text{CloseLiftDoor} event completes. This is guaranteed by the guards \( \text{CloseLiftDoor} = \text{False} \) (guard generated by the ERS diagram of Figure 11), \( \text{door} = \text{Open} \), the invariant \( \text{door} \in \text{door STATES} \), and the action \( \text{door} := \text{close} \). For a refinement step to be valid, every possible execution of a refined machine must correspond to execution steps of its abstract machine. Gluing invariants are used to verify that the concrete machine is a correct refinement. For example, gluing invariants generated by the ERS tool give rise to proof obligations for abstract events and corresponding concrete events.

All the proof obligations for our models were generated and proved using the Rodin tool provers. The total number of proof obligations for the lift controller case study is 50 and all of them are discharged automatically.

The next section present the results of applying the proposed approach to a queue management case study.

4. The Application of the Proposed Approach for Constructing Queue Management Model

This section shows the application of the proposed approach to construct an Event-B model of queue management from the requirements given in Table 3. The requirements presented in Table 3 have been collected from the FreeRTOS book and the FreeRTOS’s source code [13, 14]. FreeRTOS [13, 14] is a mini real time kernel used for small embedded real time systems. In FreeRTOS, an application program consists of independent tasks. Queues are mechanisms used to serve task-to-task communication [14]. The collections of waiting tasks are used to store tasks that are waiting for messages. The requirements describe several functions for queues such as creation of queues, sending messages on queues, receiving messages on queues, waiting for messages and an abstract description of locks which are used to prevent any task from modifying the collections of waiting tasks. In the following steps we apply the proposed approach to construct a queue management model from the requirements.
### 4.1. Step1: Classify Requirements

In order to classify the queue management requirements, we first classify the requirements based on data-oriented class, event-oriented class, and constraint-oriented class. After that, we classify the requirements based on flow-oriented requirements.

Table 3 categorises the queue management requirements based on data-oriented, event-oriented, and constraint-oriented classes.

<table>
<thead>
<tr>
<th>Label</th>
<th>Requirements</th>
<th>Classification</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>TSK1</strong></td>
<td>Tasks can be created and deleted.</td>
<td>Event-oriented and Data-oriented</td>
</tr>
<tr>
<td><strong>QUE1</strong></td>
<td>Queues can be created and deleted.</td>
<td>Event-oriented and Data-oriented</td>
</tr>
<tr>
<td><strong>TSK2</strong></td>
<td>Only one task is running at a time.</td>
<td>Constraint-oriented and Data-oriented</td>
</tr>
<tr>
<td><strong>TSK3</strong></td>
<td>Tasks are assigned priority when created.</td>
<td>Event-oriented and Data-oriented</td>
</tr>
<tr>
<td><strong>QUE2</strong></td>
<td>A queue contains a limited number of items.</td>
<td>Constraint-oriented and Data-oriented</td>
</tr>
<tr>
<td><strong>QUE3</strong></td>
<td>A task can only send an item to a queue when there is enough room in the queue. Similarly, a task can only receive an item from a queue when the queue is not empty.</td>
<td>Event-oriented and Data-oriented</td>
</tr>
<tr>
<td><strong>QUE4</strong></td>
<td>The length of the queue is identified when the queue is created.</td>
<td>Data-oriented and Event-oriented</td>
</tr>
<tr>
<td><strong>QUE5</strong></td>
<td>Each queue has two collections of waiting tasks: tasks waiting to send and tasks waiting to receive.</td>
<td>Data-oriented</td>
</tr>
<tr>
<td><strong>QUE6</strong></td>
<td>A task that fails to send an item to a queue because the queue is full is placed into the collection of tasks waiting to send. Similarly, a task that fails to receive an item from a queue because the queue is empty are placed into the collection of tasks waiting to receive.</td>
<td>Event-oriented and Data-oriented</td>
</tr>
<tr>
<td><strong>QUE7</strong></td>
<td>Every task is mapped at most to one collections of waiting tasks.</td>
<td>Constraint-oriented and Data-oriented</td>
</tr>
<tr>
<td><strong>QUE8</strong></td>
<td>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 (if any) will be removed from the collection of tasks waiting to receive.</td>
<td>Event-oriented and Data-oriented</td>
</tr>
<tr>
<td><strong>QUE9</strong></td>
<td>When a queue becomes available (there is room in the queue), then the highest priority task waiting to send item to that queue will be removed from the collection of tasks waiting to send.</td>
<td>Event-oriented and Data-oriented</td>
</tr>
<tr>
<td><strong>QUE10</strong></td>
<td>A queue should be locked before adding a send-failed task to the collection of tasks waiting to send. Similarly, a queue should be locked before adding the receive-failed task to the collection of tasks waiting to receive.</td>
<td>Event-oriented and Data-oriented</td>
</tr>
<tr>
<td><strong>QUE11</strong></td>
<td>Tasks that are blocked from receiving items from a queue will be unblocked when the required queue becomes non-empty. Similarly, tasks that are blocked from sending items to a queue will be unblocked when the required queue becomes non-full.</td>
<td>Event-oriented and Data-oriented</td>
</tr>
<tr>
<td><strong>QUE12</strong></td>
<td>Full queues that are locked are unlocked when all tasks waiting on that queue are unblocked.</td>
<td>Event-oriented and Data-oriented</td>
</tr>
</tbody>
</table>

Table 3: Data-oriented, Event-oriented, and Constraint-oriented Requirements classification.
Table 4 categorises the queue management requirements based on flow-oriented requirements

<table>
<thead>
<tr>
<th>Label</th>
<th>Requirements</th>
<th>Classification</th>
</tr>
</thead>
<tbody>
<tr>
<td>QUE3</td>
<td>A task can only send an item to a queue when there is enough room in the queue. Similarly, a task can only receive an item from a queue when the queue is not empty.</td>
<td>Flow1</td>
</tr>
<tr>
<td>QUE6</td>
<td>A task that fails to send an item to a queue because the queue is full is placed into the collection of tasks waiting to send. Similarly, a task that fails to receive an item from a queue because the queue is empty is placed into the collection of tasks waiting to receive.</td>
<td>Flow2</td>
</tr>
<tr>
<td>QUE3</td>
<td>A task can only send an item to a queue when there is enough room in the queue. Similarly, a task can only receive an item from a queue when the queue is not empty.</td>
<td>Flow3</td>
</tr>
<tr>
<td>QUE8</td>
<td>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 (if any) will be removed from the collection of tasks waiting to receive.</td>
<td></td>
</tr>
<tr>
<td>QUE10</td>
<td>A queue should be locked before adding a send-failed task to the collection of tasks waiting to send. Similarly, a queue should be locked before adding the receive-failed task to the collection of tasks waiting to receive.</td>
<td></td>
</tr>
<tr>
<td>QUE11</td>
<td>Tasks that are blocked from receiving items from a queue will be unblocked when the required queue becomes non-empty. Similarly, tasks that are blocked from sending items to a queue will be unblocked when the required queue becomes non-full.</td>
<td></td>
</tr>
<tr>
<td>QUE12</td>
<td>Full queues that are locked are unlocked when all tasks waiting on that queue are unblocked.</td>
<td></td>
</tr>
</tbody>
</table>

Table 4: Flow Oriented Requirements Classification.

In Table 4, we notice a connection between the requirements. Essentially they describe different cases. In Flow1, a task can either send an item to a queue successfully if the queue is available or fails to send that item, thus placing it into the tasks waiting to be sent. A similar scenario arise when a task attempts to receive an item from a queue. In Flow2, we notice a connection between the requirement QUE3 and the requirement QUE8. When an item has been sent out successfully to a queue, the task waiting for that item will be unblocked (removed from task-waiting). Finally, Flow3 shows a connection between the lock and unlock events. The queue will be locked when a task has failed to send or receive an item from a queue. The blocked tasks for that queue will unblock. Following that, the queue will be unlocked.

4.2. Step2: Construct Semi-Formal Artifacts and Develop Refinement Strategy
4.2.1. Stage1: Use Semi-Formal Artifacts (UML-B, ERS, and Structured English)

Due to space limitations, we present the semi-formal artifacts for some requirements which are TSK1, TSK2, QUE3, QUE6, Flow1, Flow2, and Flow3 as follows.

**Requirement TSK1**

<table>
<thead>
<tr>
<th>TSK1</th>
<th>Events can be created and deleted.</th>
</tr>
</thead>
</table>

The verbs “created” and “deleted” identify that TSK1 requirement is of type event-oriented requirement, therefore, it is represented using structured English. Since TSK1 is the first mention of the noun “Task, it introduces a data-oriented requirement. Therefore, tasks are represented using the class diagram shown in Figure 20.

**event**: CreateTask

**action**: new task is added to the pool of tasks

**event**: DeleteTask

**action**: delete an existing task

**Requirement TSK2**

<table>
<thead>
<tr>
<th>TSK2</th>
<th>Only one task is running at a time.</th>
</tr>
</thead>
</table>

Constraint-oriented and Data-oriented.
**TSK2** requirement is a type of constraint-oriented and data-oriented requirement, therefore, it is represented using structured English. *Running* task identifies an attribute of the task class introduced for **TSK2**, therefore we add the attribute **CurrentTask** to Task class to identify the running task as shown in Figure 20.

![Figure 20: The class diagram for TSK1,2.](image)

**Requirement QUE3**

A task can only send an item to a queue when there is enough room in the queue. Similarly, a task can only receive an item from a queue when the queue is not empty.

<table>
<thead>
<tr>
<th>Event-oriented and Data-oriented</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Event-oriented</strong></td>
</tr>
<tr>
<td>TaskQueueSend</td>
</tr>
<tr>
<td>guard: there is enough room in the queue</td>
</tr>
<tr>
<td>action: Task can send an item to a queue</td>
</tr>
</tbody>
</table>

![Figure 21: The class diagrams for QUE3.](image)

**Requirement QUE6**

A task that fails to send an item to a queue because the queue is full is placed into the collection of tasks waiting to send. Similarly, a task that fails to receive an item from a queue because the queue is empty are placed into the collection of tasks waiting to receive.

**Event-oriented** and **Data-oriented**

<table>
<thead>
<tr>
<th><strong>Event-oriented</strong></th>
<th><strong>Data-oriented</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>PlaceOnTaskWaitingToSend</td>
<td>PlaceOnTaskWaitingToReceive</td>
</tr>
<tr>
<td>guard: the queue q is full</td>
<td>guard: the queue q is empty</td>
</tr>
<tr>
<td>action: stores the task into the collection of tasks waiting to send an item to q</td>
<td>action: stores the task into the collection of tasks waiting to receive an item from q</td>
</tr>
</tbody>
</table>

**Requirement Flow1**
A task can only send an item to a queue when there is enough room in the queue. Similarly, a task can only receive an item from a queue when the queue is not empty.

A task that fails to send an item to a queue because the queue is full is placed into the collection of tasks waiting to send. Similarly, a task that fails to receive an item from a queue because the queue is empty are placed into the collection of tasks waiting to receive.

FreeRTOS combines several functions and uses different structures such as branches and loops to manage the order of execution of these functions. Requirements QUE3 and QUE6 are an example of branching structure. A task can successfully send/receive an item to/from a queue if the queue is ready, otherwise the task is placed into the collection of waiting tasks. QUE3 and QUE6 are identified as conditional branching and therefore are represented using the “xor” ERS pattern. In this paper we focus on representing flows that describe sending-task process as shown in Figure 22. The sending-task process has equivalent analogues as for the receiving-task process.

**Requirement Flow2**

<table>
<thead>
<tr>
<th>QUE3</th>
<th>A task can only send an item to a queue when there is enough room in the queue. Similarly, a task can only receive an item from a queue when the queue is not empty.</th>
</tr>
</thead>
<tbody>
<tr>
<td>QUE8</td>
<td>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 (if any) will be removed from the collection of tasks waiting to receive.</td>
</tr>
</tbody>
</table>

QUE3 and QUE8 requirements describe a sequence structure. The action of sending/receiving an item successfully to/from a queue, is followed by the action of removing the highest priority task waiting for that item.

QUE3 and QUE8 requirements are identified as flow requirements and therefore are represented using an ERS sequence structure. The “sequence” pattern is used to represent the sequential structure as shown in Figure 23. We also make use of the “xor” pattern to allow a task to be removed from the collection of tasks waiting to receive only if such a task exists. This is because it is possible that the collection of tasks waiting to receive is empty when a task successfully sends an item to a queue.
**Requirement Flow3**

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>QUE0</td>
<td>A queue should be locked before adding a send-failed task to the collection of tasks waiting to send. Similarly, a queue should be locked before adding the receive-failed task to the collection of tasks waiting to receive.</td>
</tr>
<tr>
<td>QUE11</td>
<td>Tasks that are blocked from receiving items from a queue will be unblocked when the required queue becomes non-empty. Similarly, tasks that are blocked from sending items to a queue will be unblocked when the required queue becomes non-full.</td>
</tr>
<tr>
<td>QUE12</td>
<td>Full queues that are locked are unlocked when all tasks waiting on that queue are unblocked.</td>
</tr>
</tbody>
</table>

QUE0,11 and QUE12 requirements describe sequential ordering between the events: LockQueue, PlaceOnTaskWaitingToSend, RemoveFromTaskWaitingToReceive, and UnLockQueue when a task fails to send an item to queue as shown in Figure 24.

---

**Stage 2: Merging the Structured English Representation of a Single Event**

This step merges the fragmented structured English that refers to a single event together. Grouping the fragmented structured English into a single structure, facilitates the process of translating the structured English representation into single events in stage 3.

The actions of TSK1 and TSK3 requirements are merged as follows:

- **event**: CreateTask
  - **action**: new task is added to the pool of tasks
  - **action**: priority of a new task is set

**4.2.2. Stage 3: Develop Refinement Strategy**

In this step, we combine ERS diagrams and develop the refinement strategy. We aim to structure our development in a way that reflects the architecture of an embedded system. We also would like to keep flows of the ERS diagrams in Stage 1 unaffected as they reflect the execution order of FreeRTOS functions. The architecture of an embedded system consists of two main software layers which are the application layer (the uppermost layer) which defines the function and purpose of the embedded system and the RTOS layer which defines in detail how functions of the application layer are achieved. RTOS hides from application software, the hardware details of the processor upon which the application software will run. Therefore, in the abstract level we model the abstract send event that appears in the application level of FreeRTOS. In the first and the second refinement levels we add more details specific to the RTOS level including adding tasks that failed to send item to queue to the collection of tasks waiting to send and queues locking mechanisms. The abstract model includes the TaskQueueSend event and the FailedTaskQueueSend event (an abstract event of PlaceOnTaskWaitingToSend event). The introduction of the FailedTaskQueueSend event in the most abstract level reduces the complexity and allows us to defer the introduction of PlaceOnTaskWaitingToSend event, RemoveFromTaskWaitingToSend event, PlaceOnTaskWaitingToReceive event and RemoveFromTaskWaitingToReceive to the first refinement level. This is because these events are dependent on each other and therefore need to be defined in one modelling level. The atomicity of the FailedTaskQueueSend event is broken down into the first refinement level as PlaceOnTaskWaitingToSend event, RemoveFromTaskWaitingToSend event and RemoveFromTaskWaitingToReceive event. The most abstract level in the Figure 25 demonstrates task-send process. The tree with the root TaskSend corresponds to the ERS diagram shown in Figure 22. The behaviour of the TaskSend process is realised by executing either the TaskQueueSend event when a task successfully sends an item to a queue or by executing the FailedTaskQueueSend event when a task fails to send an item to a queue.
Figure 25: The combined ERS diagram for task-send.
In the first refinement level, the atomicity of TaskQueueSend which corresponds to the ERS diagram shown in Figure 23 is broken down into three events. The abstract TaskQueueSend event is realised in the refinement by firstly executing the refinement TaskQueueSend event, then executing either NoTaskInTaskWaitingToReceive event or RemoveFromTaskWaitingToReceive1 event. Similarly, the abstract FailedTaskQueueSend event, which corresponds partially to the ERS diagram shown in Figure 24, is realised in the refinement by firstly executing the PlaceOnTaskWaitingToSend event, followed by RemoveFromTaskWaitingToReceive2 event for all the tasks placed on the TaskWaitingToReceive. In the second refinement level, the abstract PlaceOnTaskWaitingToSend event is realised by executing the LockQueue event followed by executing the PlaceOnTaskWaitingToSend event. RemoveFromTaskWaitingToReceive2 event, on the other hand, is realised by firstly executing the RemoveFromTaskWaitingToReceive2 event and then the UnlockQueue event.

The structure of the class diagrams that reflect the decided refinement strategy is shown in Figure 26. The UML-B class diagrams are layered through the refinement based on the combined ERS diagrams of Figure 25. Therefore, the abstract model specifies the abstract send/receive events. The first refinement specifies the collection of the waiting tasks and their events, whereas, the second refinement level specifies the queue lock mechanism.

4.3. Step3: Construct Formal Models

In stage 3, we obtained three modelling levels in the combined ERS diagram and three modelling levels in the layered class diagrams. In this step we use the UML-B tool and the ERS tool to convert the resulted diagrams into Event-B models and use shared-event composition to enable the integration of Event-B models. We show some parts of the models generated from class diagrams and ERS diagrams, as well as the model constructed from a structured English representation.

The Event-B model corresponds to the class diagrams shown in the first refinement of Figure 26 is:

![Figure 26: Queue management layered class diagrams.](image)
The Event-B model generated by the ERS tool that corresponds to \textit{Flow2} is:

Each event corresponding to a leaf gives rise to a set control variable whose type is based on the type of the parameter $c$ of the leaf, where $c$ is of type \textit{Task} (carrier set). Therefore, three set control variables are generated: \textit{TaskQueueSend} set, \textit{NoTaskWaitingToReceive} set, and \textit{RemoveFromTaskWaitingToReceive1} set. The invariant labelled @inv\_xor invariant ensures that at any time only one of the xor-constructor events can be executed. The @inv\_NR\_seq and @inv\_RR\_seq invariants ensure that \textit{NoTaskWaitingToReceive} event and \textit{RemoveFromTaskWaitingToReceive1} event can only executed after \textit{TaskQueueSend} event.

@grd\_NR in \textit{NoTaskWaitingToReceive} event ensures that \textit{NoTaskWaitingToReceive} event is executed after \textit{TaskQueueSend} event. Similarly, @grd\_RR in \textit{RemoveFromTaskWaitingToReceive1} event ensures that \textit{RemoveFromTaskWaitingToReceive1} event is executed after \textit{TaskQueueSend} event. @grd\_xor in \textit{NoTaskWaitingToReceive} event and @grd\_xor in \textit{RemoveFromTaskWaitingToReceive1} event ensures that exactly one of these events executes in the sequence of \textit{TaskQueueSend} event.

The Event-B model corresponds to the structured English of \textit{PlaceOnTaskWaitingToSend} of \textit{QUE6} is:
The shared event composition tool is used to integrate Event-B models generated from UML-B diagrams and Event-B models generated from ERS diagrams. The Event-B components of the structured English are added manually to the composed machines. More detailed discussion about the integration steps of UML-B, ERS and refinement is given in Step 3 of Section 3.

All the proof obligations for our models were generated and proved using the Rodin provers. The total number of proof obligations for the queue management model is 116 and they are discharged automatically. At the beginning there were violation of some proof obligations due to some missing guards and actions. For instance, in the DeleteQueue event there was only one action corresponding to TSK1 which was: Queue := Queue \ \{ q \}. That action caused violation in the DeleteQueue event at the first refinement during the introduction of QueueItem variable. Rodin indicated that DeleteQueue\QueueItem.type/INV is the proof obligation that must be verified to show that the event DeleteQueue preserves the invariant (INV) with the label QueueItem.type. Thus, the action QueueItem := QueueItem \{ q \} was added to DeleteQueue event. After that amendment, the violated proof obligations of DeleteQueue event was discharged automatically.

5. Related Work

This section presents some of the related work regarding the use of formal methods in operating systems and also some works in the area of requirements structuring and requirements traceability.

5.1. Formal Development of Operating Systems

Craig’s work is one of the fundamental sources in formal modelling of operating systems (OS) [15, 16]. He focuses on the use of formal methods in OS development, and the work is introduced in two books. The first book is dedicated to specifying the common structures in operating system kernels in Z [17] and Object Z [18], with some CCS (Calculus of Communicating Systems) process algebra used to describe the hardware operations. It starts with a simple kernel with few features and progresses on to more complex examples with more features. For example, the first specification introduced in the book is called a simple kernel, and involves features such as task creation and destruction, message queues and semaphore tables. However, it does not contain a clock process or memory management modules, whereas other specifications of swapping kernel contain more advanced features including a storage management mechanism, clock, interrupt service routines, etc.

The second of Craig’s books [16] is devoted to the refinement of two kernels, a small kernel and a micro kernel for cryptographic applications. The work specifies proofs written by hand, with some mistakes and some missing properties resulting due to manual proofs, some of which have been highlighted by Freitas [20].

Freitas [20, 21] has used Craig’s work to explore the mechanisation of the formal specification of several kernels constructed by Craig using Z/Eves theorem prover. This covers the mechanisation of the basic kernel components such as the process table, queue, and round robin scheduler in Z. The work contains an improvement of Craig’s scheduler specification, adapting some parts of Craig’s models and enhancing it by adding new properties. New general lemmas and preconditions are also added to aid the mechanisation of kernel scheduler and priority queue. Mistakes have been corrected in constraints and data types for the sake of making the proofs much easier, for instance, the enqueue operation in Craig’s model preserves priority ordering, but it does not preserve FIFO ordering within elements with equal priority; this has been corrected by Freitas in [20].

Furthermore, Déharbe et al. [22] specify task management, queues, and semaphores in Classical B. The work specifies mutexes and adopts some fairness requirements to the scheduling specification. The formal model built was published in [23].
There is also an earlier effort by Neumann et al [24] to formally specify PSOS (Provably Secure Operating System) using a language called SPECIAL (SPECification and Assertion Language) [25]. This language is based on the modelling approach of Hierarchical Development Methodology (HDM). In this approach, the system is decomposed into a hierarchy of abstract machines; a machine is further decomposed into modules, each module is specified using SPECIAL. Abstract implementation of the operations of each module are performed and then is transformed to efficient executable programmes. The work began in 1973 and the final design was presented in 1980 [24]. PSOS was focusing on the kernel design and it was unclear how much of it has been implemented [26]. Yet, there are other works inspired by the RSOS design such as Kernelized Secure Operating System (KSOS) [27] and the Logical Coprocessing Kernel (LOCK) [28].

The aforementioned examples follow a top-down formal method approach, where the specification is refined stepwise into the final product. On the other hand, there are also some earlier efforts in the area of formal specification and correctness proofs of kernels based on the bottom-up verification approach. The bottom-up approach adopts program verification methods to verify the implementation.

An example of this approach is a work by Walker et al (1980) [29] on the formalisation of the UCLA Unix security kernel. The work is developed at the University of California at Los Angeles for the DEC PDP-11/45 computer. The kernel was implemented in Pascal due to its suitability for low-level system implementation and the clear formal semantics [30, 31]. Four levels of specification for the security proof of the kernel were conducted. The specifications were ranging from Pascal code at the bottom to the top-level security properties. After that, the verification based on the first-order predicate calculus was applied that involves the proof of consistency of different levels of abstraction with each other. Yet, the verification was not completed for all components of the kernel.

Finally, there was an effort by Klein et al [32, 33] on the formal verification of the seL4 kernel starting with the abstract specification in higher-order logic, and finishing with its C implementation. The design approach is based on using the functional programming language Haskell [34] that provides an intermediate level that satisfies bottom-up and top-down approaches by providing a programming language for kernel developer and at the same time providing an artefact that can be automatically translated into the theorem prover. A formal model and C implementation are generated from seL4 prototype designed in Haskell. The verification in Isabelle/HOL [35] shows that the implementation conforms with the abstract specification.

5.2. Requirements Structuring and Requirements Traceability

SOFL (Structured Object-Oriented Formal Language) [36] 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 to offer a visual and formal specification of a system. The graphical and textual formal notation serve as a good communication mechanism between a user and a developer. While our approach is supported by the UML-B tool and the ERS tool, SOFL is supported by a prototype of a tool for writing specifications of SOFL approach [37]. In addition, our approach provides requirements classification and offers guidance on which kind of semi-formal structure is used to model each requirement class. Similarly, the semi-formalization process in SOFL method is carried out based on specific guidelines which are mentioned in [38].

Behaviour trees [39] are formal, graphical modelling language developed by Dromey to represent natural requirements [39]. Behaviour trees are of two forms: Requirement behaviour trees and Integrated behaviour trees. Requirement behaviour trees are used to graphically capture all functional behaviour in each individual natural language requirement. Integrated behaviour trees are used to compose all the individual requirement behaviour trees where every individual requirement is expressed as a behaviour tree and has a precondition associated with it. The integrated behaviour trees check that all preconditions are satisfied so defects can be discovered and corrected. The similarity between the behaviour trees and the use of ERS in our approach is that both approaches allow requirements to be expressed in detail and at an abstract level. The behaviour trees allow behaviour to be easily partitioned and separated out and the ERS allows the system to be visualised at different abstract levels. Behaviour trees and ERS diagrams are composable, however, behaviour trees can expose different behavioural defects such as aliases, inconsistencies, redundancies associated with the requirements information. Moreover, ERS can be translated into Event-B formal models and there is some work on linking a subset of behaviour trees to CSP [40].

Jastram et al [41] present an 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 artefacts (the restriction on states and transitions). The artefacts 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 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 [41] 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.

Yeganefard and Butler [42] describe 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 [42], the approach used in [42] 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 textual requirement document is difficult. We believe that the visualisation of Event-B components using ERS diagrams gives a clear overview of the whole system and helps decide which feature to model in each refinement level. It should be possible to combine our approach with that of [42] to obtain more effective guidelines for developing traceable Event-B models for control systems.

KAOS and \( i^* \) are goal-oriented requirement engineering methods that specify the high level goals of the system. A goal is a statement of a system whose satisfaction is determined by the cooperation of the agents of the system such as humans, devices, etc. Goals drive requirement details which leads to domain-specific requirements that could be implemented. Goals may be organised in an AND-refinement hierarchy [43]. The higher-level goals are strategic and coarse-grained whereas lower-level goals are technical and fine-grained. In our work we used ERS to structure the behaviour of the models. ERS patterns are used to manage the execution between events whereas KAOS is used to structure requirement goals. Some work, however, have been done to generate high-level Event-B system from KAOS requirements and there is a tool support that links KAOS/Objectiver tool and the Event-B/Rodin tools [44].

6. Conclusions and Future work

We have described an approach which facilitates constructing Event-B models via semi-formal requirements structures and provides clear traceability between requirements and the Event-B model. The approach is based on the use of the UML-B and ERS approaches. UML-B provides UML-like graphical modelling that allows the development of an Event-B model, whereas the ERS 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 ERS approach at the requirement level assists a developer in the process of deciding which features to be modeled in each refinement step. Moreover, part of the Event-B model is generated automatically by the ERS tool, which reduces the burden of the manual work especially in the development of complex systems. The combined ERS diagrams provide an overall visualisation of the refinement structure and demonstrate the relationships between events even before any model is written.

From the application of our approach to the queue management case study, four conclusions were drawn: Firstly, we found that most of the requirements were classifiable according to the classification scheme. Several requirements can be classified as event/constraint and data oriented requirements such as TSK3, QUE4, and QUE7. It is possible to define clearly data-oriented requirement and separate them from event/constraint-oriented requirements. We can consider that data-oriented requirement only describe attributes of the system, constraint-oriented requirements describe properties about the system and event-oriented requirements describe the activities of the system. Then, we ignore all the nouns and attributes mentioned in the constraint/event-oriented requirements. Therefore, TSK3 can be restructured as follows:
In the above formulation, **TSK3-1** is a data-oriented requirement whereas **TSK3-2** is an event-oriented requirement. In **TSK3-2**, we only focus on the action of assigning a task priority when created, and thus consider **TSK3-2** as an event-oriented requirement. We regard priority as an attribute of a task in **TSK3-1** and classify **TSK3-1** as a data-oriented requirement. Secondly, we found that the flow requirements can sometimes be extracted from more than one requirement as shown in Table 4. Thirdly, ERS 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 ERS pattern together with an additional manual flag can be used to represent this particular case. Finally, it is possible that a particular event becomes a leaf in different ERS diagrams. In some cases however, it is necessary to change the name of the repeated leaf to avoid an invalid combination of ERS 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 ERS 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 considered to evaluate the composition of ERS diagrams and to explore more useful patterns for managing the flows.

The application of the proposed approach to several case studies is the primary goal of future work. In this paper we have described one kind of constraint-oriented requirements, namely requirements on the system being developed, such as requirement **LIFT4**. We also need to investigate another type of constraint-oriented requirements, which describe assumptions on the environment, such as the following requirement:

**LIFT10** 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 ERS 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 issue of scalability. Finally, further investigation of several ERS patterns is necessary to support a larger class of flow requirements.

References