UNIVERSITY OF SOUTHAMPTON

FACULTY OF PHYSICAL AND APPLIED SCIENCES

Electronics and Computer Science

Towards a Systematic Process for Modelling Complex Systems in Event-B

by

Eman Alkhammash

Thesis for the degree of Doctor of Philosophy

May 2014

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF PHYSICAL AND APPLIED SCIENCES
Electronics and Computer Science

Doctor of Philosophy

TOWARDS A SYSTEMATIC PROCESS FOR MODELLING COMPLEX SYSTEMS IN EVENT-B

by Eman Alkhammash

Formal methods are mathematical techniques used for developing large systems. The complexity of growing systems pose an increasing challenge in the task of formal development and requires a significant improvement of formal techniques and tool support.

Event-B is a formal method used for modelling and reasoning about systems. The Rodin platform is an open tool that supports Event-B specification and verification. This research aims to address some challenges in modelling complex systems. The main challenges addressed in this thesis cover three aspects: The first aspect focuses on providing a way to manage the complexity of large systems. The second aspect focuses on bridging the gap between the requirements and the formal models. The third aspect focuses on supporting the reuse of models and their proofs.

To address the first challenge, we have attempted to simplify the task of formal development of large systems using a compositional technique. The compositional technique aims at dividing the system into smaller parts starting from requirements, followed on by a construction of the specification of each part in isolation, and then finally composing these parts together to model the overall behaviour of the system. We classified the requirements into two categories: The first category consists of a different set of requirements, each of which describes a particular component of the system. The second category describes the composition requirements that show how components interact with each other. The first category is used to construct Event-B specification of each component separately from other components. The second category is used to show the interaction of the separated models using the composition technique.

To address the second and the third challenges, we proposed two techniques in this thesis. The first technique supports construction of a formal model from informal requirements with the aim of retaining traceability to requirements in models. This approach makes use of the UML-B and atomicity decomposition (AD) approaches. UML-B provides the 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 second technique supports the reusability of Event-B formal models and their respective proof obligations. This approach adopts generic instantiation and composition approaches to form a new methodology for reusing existing Event-B models into the development process of other models. Generic instantiation technique is used to create an instance of a pattern that consists of refinement chain in a way that preserves proofs while composition is used to enable the integration of several sub-models into a large model. FreeRTOS (real-time operating system) was selected as a case study to identify and address the above mentioned general problems in the formal development of complex systems.

List of Figures

List of Tables

Contents

Declaration of Authorship

I, Eman Alkhammash , declare that the thesis entitled *Towards a Systematic Process for Modelling Complex Systems in Event-B* and the work presented in the thesis are both my own, and have been generated by me as the result of my own original research. I confirm that:

- this work was done wholly or mainly while in candidature for a research degree at this University;
- where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated;
- where I have consulted the published work of others, this is always clearly attributed;
- where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work;
- I have acknowledged all main sources of help;
- where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself;
- parts of this work have been published as: [8]

Signed:.....

Date:.....

.

Acknowledgements

First praise is to ALLAH, the Almighty, on whom ultimately we depend for sustenance and guidance.

I am heartily thankful to my supervisors, Dr Corina Cîrstea and Prof Michael Butler whose encouragement, guidance, support and patience made this thesis come true.

I would also like to thank my thesis reviewer, Dr Thai Son Hoang from ETH Zurich, for his time and efforts while reviewing the thesis and providing with his invaluable comments to improve this work.

This PhD would not have been possible without the support of my beloved parents, and my dear husband, my sisters and brothers, and my friends. To all these people, I cannot thank you enough for your support over the course of my PhD.

Chapter 1

Introduction

Formal methods have been used for constructing the models of complex systems. There are a number of different formal method languages such as B [2], Event-B [4], Z [79], and VDM [22]. The research work focuses on methods that assist in the formal development of the systems and cover issues such as the requirements and reusability. The quality of requirements has a major influence on the construction and validation of formal models, whereas the reuse of the formal models has the potential to reduce the time and effort for developing systems. This chapter introduces the contribution of this thesis and presents relevant background on formal methods. Section 1.1 outlines the thesis motivation and contribution. Section 1.2 is an introduction on formal methods. Sections 1.3, 1.4, 1.5, and 1.6 give an overview of several formal methods including B, Event-B, Z, and VDM. Section 1.7 describes refinement. Section 1.8 shows comparison of B, Event-B, Z, and VDM. Finally, Section 1.9 describes the motivation behind selecting Event-B for modelling FreeRTOS.

1.1 Thesis Motivation and Contribution

One of the activities that addresses the Grand Challenges pilot projects, concerns with the development of approaches supported by tools that cover all the aspects of verified software construction [41, 85].

This thesis attempts to address some challenges in modelling complex systems. The challenges cover three main issues: managing the complexity of large systems, bridging the gap between the requirements and the formal models, and supporting reusability of the models and their respective proof obligations. This thesis presents general techniques for addressing the challenges that have emerged from the FreeRTOS case study.

FreeRTOS [11, 10] is a mini real time kernel for small embedded real time systems. The development is realised with Event-B and the Rodin tool [6, 5, 3, 17]. **Event-B** is a

formal method used for specifying and reasoning about systems. The **Rodin platform** is an open tool that supports the Event-B specification and verification.

Different modelling techniques are investigated throughout the modelling process which are generic instantiation [72], the theory feature [18], UML-B [77], atomicity decomposition approach [15, 33], and composition [73]. The **generic instantiation** technique facilitates the reuse of Event-B models; it provides means of instantiating generic models in a way that ensures the proofs associated to a generic model remain valid in the instantiated one. The **theory feature** is another technique which provides facility to extend the Event-B language and the proving infrastructure. It allows the development of operators, new data types, rewrite and inference rules, and polymorphic theorems. **UML-B** is a graphical modelling environment that allows the development of an Event-B formal model through the use of UML graphical notation. The **atomicity decomposition** approach provides a graphical notation to structure refinement and control flow between the events. Finally, the **shared event composition** approach allows sub-components to interact via synchronisation over shared events.

The first step towards achieving the research goal is to choose the appropriate development strategy to manage the complexity of FreeRTOS development. We address complexity by choosing a compositional strategy to build specification of FreeRTOS case study. This involves the decomposition of the FreeRTOS development into components that are easy to manage and then composing these components to show the overall behaviour of the system. Hence, FreeRTOS requirements are analysed into three main set of requirements, each of which corresponds to a particular FreeRTOS components: task, queue, and memory. The division of the requirements based on the system components assist in managing the complexity of the system and allows to focus on modelling a particular component separately without considering its interaction with the other components. This in effect simplifies the development of the system and helps in producing loosely coupled modelling components that can easily be reused in the development of different systems. The interaction between the separated components then became an explicit development task. Hence, another set of requirements called the composition requirements were introduced to show how the different FreeRTOS components are linked together.

Our focus after that was directed to find a way to bridge the gap between the requirements and the Event-B formal models and retain traceability to requirements in Event-B. In order to achieve this purpose, we investigated an approach that consists of three main steps: The first step focuses on categorizing the requirements based on Event-B structure. The second step focuses on using semi-formal artifacts described in UML-B and AD diagrams to represent the requirements. 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. The third step is to use the UML-B

tool and the AD tool to generate the Event-B models. This approach was successfully used to build traceable Event-B model for some features of the queue component.

Moreover, three separate specifications: task, queue, and memory were constructed in Event-B and some general guidelines for modelling the FreeRTOS kernel were drawn from our experiment.

The composition task was carried out based on shared-event composition technique; composition requirements assist in selecting which events to be composed to show the overall behaviour of the system. Since, shared event composition is proved to be monotonic [74], we were able to data refine the models individually.

The basic data structures of FreeRTOS are linked lists. This forces us to think of a systematic approach to reuse the modelling patterns of the linked lists. Therefore, we dedicated some time to study the possibility of building reusable modelling patterns for linked lists that can be used to resume the specification of abstract structure of FreeRTOS models "set" to the concrete data structures of linked lists. As there is little support for modelling the linked lists, we dedicated time to the development of theories that aid the modelling of linked lists using the theory feature. The developed linked list theories have been used to build reusable modelling patterns for linked lists in Event-B. We then focussed on the task of the incorporation of the developed modelling patterns to resume the specification of FreeRTOS models. Therefore, we investigated an approach based on generic instantiation and shared event composition approach to perform the incorporation task. Generic instantiation technique is used to instantiate the modelling pattern. The composition technique is applied to integrate the instantiated pattern into the problem. The approach has been used successfully to refine the abstract "set" of FreeRTOS models to linked list structures. It allows us to incorporate several instances of a modelling pattern into a development with the advantage that all the refinement steps required for the development of the pattern can be incorporated into the development in a single step. This is because generic instantiation allows us to instantiate the pattern several times and the composition technique allows us to incorporate different refinement levels of one or more patterns into the development.

1.1.1 Summary of Contributions

The following points summarise the main contributions of our work:

- Adopted compositional strategy to manage the complexity of systems development. This involves the process of classifying requirements into a set of requirements for each individual component, and a set of requirements for the compositional purposes. The individual components then are modelled separately in Event-B, and then are composed based on the composite requirements using the composition technique.

- Investigated an approach for constructing traceable Event-B models based on UML-B and AD approaches. The investigated approach bridges the gap between requirements and the Event-B models and enables the validation of the models against their corresponding requirements.
- Proposed an approach for reusing the modelling *patterns* in Event-B based on composition and generic instantiation techniques. The approach has been used to carry out the data refinement of the abstract "set" of FreeRTOS specifications to concrete linked lists structures.

We also have other sub-contributions as follows:

- Constructed Event-B models for FreeRTOS using compositional strategy. Three Event-B models have been developed that correspond to the main components of FreeRTOS: task, queue, and memory. The constructed models can contribute to the Verified Software Repository [41, 85].
- Devised general modelling guidelines for RTOS kernels in Event-B. The guidelines introduce Event-B modelling concepts for the basic features of an RTOS such as scheduling, memory allocation, and interrupt handling.
- Developed theories with a set of reusable operators and rules of inference for linked lists using the theory feature. The linked list theories can be used to supply a library of Event-B theories in the future.

1.2 What is a Formal Method?

Formal methods are techniques used in the specification, development and verification of complex systems based on mathematics and formal logics [4].

There are a wide range of formal methods, frequently supported by tools, each of which is suited to a different domain and the emphasize on different aspects of the systems based upon the adoption of different maths and formal logics. Formal specification languages and formal verification are two types of formal methods. Formal specifications are formal description of what systems should do, whereas the formal verification is a mathematical proof of the systems correctness. Formal specifications are normally based on set theory and first order predicate calculus. Event-B, Z and VDM are some dominating formal specification languages. Formal verification, on the other hand, such as model checking [20] are based on temporal logics. Various projects have been successfully developed using formal methods including those pertaining to air traffic control, railway signalling, and smart cards applications [87, 9, 12].

The adoption of the formal method approach for systems development has many advantages; one such advantage is that one is allowed to think more carefully and critically about the system being specified before becoming too much involved in the coding. This leads to a deeper understanding of the system and forces a detailed analysis of the requirements, thus enabling the production of software that meets the requirements. Formal methods provide precise and unambiguous system specifications. Many proofs have to be performed in order to ensure that the final system is "correct by construction". Another important benefit is the ability to detect defects earlier in the software life-cycle, which enables the production of a reliable system. Formal methods can be applied to different aspects of systems and throughout the various phases of the software cycle. It encourages the identification of certain properties and analysis of the system before any code is written, and this is an advantageous feature of formal methods which is not replicated in other software approaches.

This chapter highlights four formal methods: B [2], Event-B [4], Z [79], and VDM [22] with emphasis to Event B as it is the language employed to the model of the FreeRTOS case study.

1.3 B Method

The B-method [2] (classical B) is a state-based method developed by Abrial in the mid 1980s. It is used for the formal development of software systems. The B method uses the set theoretic constructs such as sets, relations and functions to define variables and invariants (constraints of variables). There are a number of safety-critical system applications which have been developed successfully in classical B such as *Paris Metro Line 14* [12].

AtelierB [21] is a set of tools that provide support for writing B specification including static checkers, automatic and interactive provers, and code generation.

B does not include facility to support user-defined extension of the mathematical language and theory of B. Nevertheless, Event-B (an extension of the B formalism) includes facility of extending the Event-B language and proving infrastructure, and allows users to define new operators and data types.

MACHINE
SETS
CONSTANT
PROPERTIES
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
END

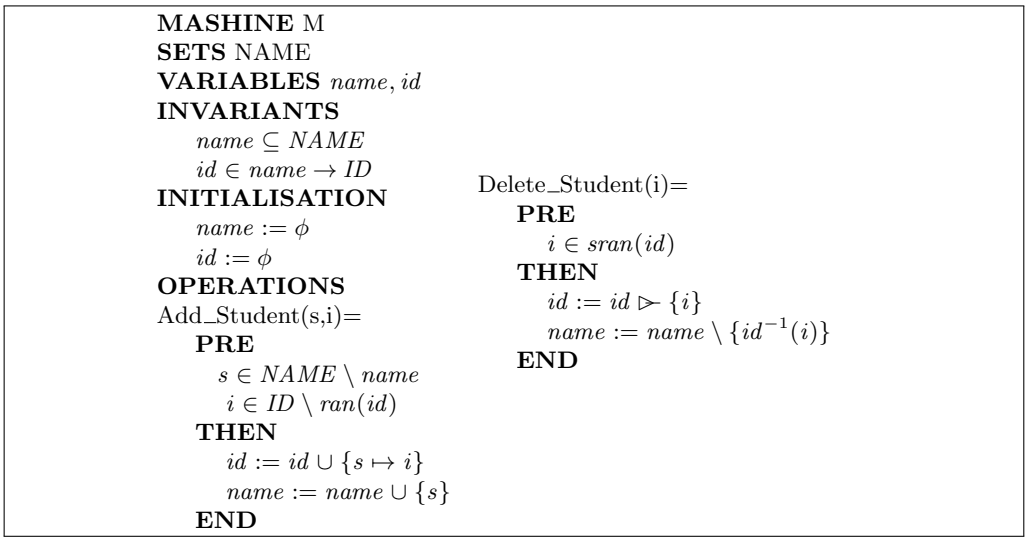


Figure 1.1: An example of B specification.

The structure of B method abstract machine is given above. The **MACHINE** clause specifies the name of the machine, The **SETS** clause specifies all the sets which are used in the machine. The **CONSTANT** clause specifies all the constants which are used in the machine. The **PROPERTIES** clause specifies the constraints on the constants and the sets. The **VARIABLES** clause specifies all the variables which are used in the machine. The **INVARIANT** clause specifies the properties of the variables that must always remain true such as the type of the variables. The **INITIALISATION** clause specifies the initial state of the machine. The **OPERATIONS** clause specifies operations that can cause the variables to change their values.

Figure 1.1 shows a simple example of a B model that describes a student database with two operations *Add_Student* which adds a student to the student data base and *Delete_Student* operation that deletes a student from the student database.

There are two variables in this model: *name* that is defined as a set and *id* that is defined as a function that maps names to their ids. There are two operations in this machine *M*: *Add_Student* operation and *Delete_Student* operation. The **PRE** clause identifies the preconditions which required to hold in order to execute the actions within a **THEN-END** block.

1.4 Event-B

Event-B [4] is a successor language of B developed by Jean-Raymond Abrial. Event-B 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

(a) when $S(c, v)$ then $R(v, c, v')$ end
(b) any t where $S(t, c, v)$ then $R(v, t, c, v')$ end
(c) begin $R(c, v')$ end

Figure 1.2: The outline of an event

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 systems development, and has shown success in the development of different complex real-life systems [16, 9]. The Event-B notation contains two constructs: a context and a machine. The context is the static part in which we can define the data of the model: sets, constants, axioms that are used to specify assumptions about sets and constants, and theorems that are used to describe properties derivable from the axioms. 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, theorems to describe properties that follow from the invariants, and events to trigger the behaviour of the machine. The outline of an event takes one of the three forms described in Figure 1.2. In (a), the body of the event is fired only when the commanded guard is true. $R(v, c, v')$ describes the relationship between the values of the variables before and after the event is executed. The second form (b) takes a local variable, t, that satisfies the guard *S(t, c, v)*, and then executes the body *R(*v, t, c, v'*)*. The third form (c) is free of a guard and can be used for initialisation.

Rodin [6, 5, 3, 17] is a platform used in the development of Event B models. Rodin exhibits many features and can be extended further with supportive plug-ins; for example, ProB [55] is an example of a Rodin plug-in which provides an automatic animation and model checking of Event B models. The theory plug-in [18] allows users to define reusable polymorphic operators, data types, rewrite rules, inference rules and polymorphic theorems. Shared event composition plug-in [73] allows the interaction between the sub-models through the composition of events. The model decomposition plug-in [75] allows to decompose a model into several sub-models. The generic instantiation plug-in [72] allows reusability of existing models by instantiating the model and ensuring that the proofs associated to an existing model remain valid in an instantiated development. The atomicity decomposition plug-in [33] controls the order between the events (flows) and demonstrates the relationship between refinement levels.

We use the student database example here to describe the two events: *Add_Student* and *Delete_Student* event in Event-B.

There are two variables in this machine M: *name* that is defined as a subset of the carrier set *NAME*, the carrier set *NAME* is defined in the context C. The machine M explicitly sees the context C, so it can access the context C components (Here *NAME* set).

CONTEXT C
SET NAME

MACHINE M
SEES C
VARIABLES $name, id$
INVARIANTS
inv1 $name \subseteq NAME$
inv2 $id \in name \rightarrow ID$
Add_Student
Any s
Where
grd1 $s \in NAME \setminus name$
grd2 $i \in ID \setminus ran(id)$
Then
act1 $id := id \cup \{s \mapsto i\}$
act2 $name := name \cup \{s\}$
End

Delete_Student
Any i
Where
grd1 $i \in ran(id)$
Then
act1 $id := \{i\} \rhd id$
act2 $name := name \setminus \{id^{-1}(i)\}$
End

Figure 1.3: An example of Event-B specification.

There are two events: *Add_Student* and *Delete_Student*. The guards are the necessary conditions for the event; they must hold in order for the actions to be performed.

1.5 Z Specification Language

Z [79] is a formal specification language based on first order predicate logic and Zermelo-Fraenkel (ZF) set theory. The language was first developed at the Oxford University in the 1980s. Z combines two sub-languages: mathematical language and schema language. States, properties and operations of the system are described by mathematical language, whereas the schema language is used for structuring the specification. A Z specification is constructed as a series of schemas. Schema declarations contain state schemas and operation schemas. State schemas describe states, the relationship between them, as well as the restriction (constraining predicates) on them.

StateSpace
$<$ *variables declaration* $>$

$<$ *invariants* $>$

The structure of state schemas contains two parts: the part above the central line represents the variables declaration part and the variables local to the schema; and the part below the line represents the predicate part of the schema that specifies the invariants or the relationships that remain true in every state of the system. Operation schemas, on the other hand, describe the changes in these states before and after executing the operations. The Schema Calculus offers operators such as conjunction, disjunction, composition, etc that are used to construct a large Z specification from some existing

schemes. Schema can also be extended easily by adding new components, additional invariants and operations.

Z contains a mathematical tool-kit [79] of useful definitions built on top of the basic language. This tool-kit includes a collection of types and operators for sets, products, relations, functions, and sequences. The following schema illustrates the student database example as a Z specification focusing on the addition and deletion of student's information.

StudentDB

$$name : \mathbb{P}\, NAME$$
$$studentId : NAME \pfun ID$$

$$name = \mathrm{dom}\, studentId$$

This schema describes the state space of the system; it contains the variable declarations and invariants. We have two variables: *name* of type *NAME* and *studentId* which is a function that maps a student's name to its id. We have one invariant to indicate that the set *name* is the same as the domain of the function *studentId*, and it should be maintained by every operation on it. This invariant allows the value of the variable *name* to be derived from the value of *studentId*.

AddStudent

$$\Delta StudentDB$$
$$name? : NAME$$
$$id? : ID$$

$$id? \notin ran(studentId)$$

$$studentId' = studentId \cup \{name? \mapsto id?\}$$

$$name' = name \cup \{name?\}$$

DeleteStudent

$$\Delta StudentDB$$
$$id? : ID$$

$$studentId' = studentId \rhd \{id?\}$$

$$name' = name \setminus name(\!| \{id?\} |\!)$$

These two schemas are operation schemas; the first adds new entry where *name?* and *id?* are input parameters. The second, *DeleteStudent* operation, removes the entry associated with a particular id (*id?*).

The Δ convention indicates that states do change; the variables $name'$ and id' are the states after the change, whereas the variables *name* and *id* are the states before the change.

1.6 Vienna Development Method (VDM)

VDM [22, 71] is a popular formal method that was developed in the 1970s by Cliff Jones and Dines Bjrner at IBM labs in Vienna. The method is used to model systems at different levels of abstraction, hence it combines a specification language and an approach to refine the specifications into code. VDM has been successfully used in many projects, such as the development of compilers and in disciplines of programming language semantic such as CHILL, Modula-2, and Ada [48].

There are three dialects of VDM: VDM-SL [36], VDM-RT [81, 82] and VDM++ [35]. VDM-SL is a specification language standardised by the International Standards Institute (ISO) in the 1990s. VDM-RT is a version used to model real time embedded and distributed systems and VDM++ is an object-oriented extension based on VDM-SL that is used for the specication of object-oriented models. VDM is supported by strength tools called VDMTools [1] that were developed by IFAD in the 1990s, and are now owned by CSK Systems.

The VDMTools provide a type checker, theorem prover, and a facility for testing specifications. The tools can be used for producing an executable code of a subset of VDM through the automated code generation feature, or the simulation of model feature.

VDM uses modules for system specifications. The structure of a module is shown as follows:

```
module Module_Name
...
definition types
...
state
....
functions
...
operations
...
end Module_Name
```

Figure 1.4: The structure of VDM-SL module.

```
types
  Name=String;
  String=seq of char;
  ID= nat;
  StudentID= map Name to ID;
functions
AddStudent: Name*ID*StudentID -> StudentID
  AddStudent(name, id, studentID) ==
  studentID munion {name |-> id}
pre id not in set ran studentId;
DeleteStudent: ID*StudentID -> StudentID
  DeleteStudent(id, studentID) ==
  studentID ⩤ { id}
pre id in set ran studentId;
```

Figure 1.5: An example of VDM-SL specifications

The **type** clause is used to specify the various types to be used in the specification. The **state** clause specifies the state definition of the global variables which can be referenced inside the operations. The **operation** clause specifies the behaviour of a system. The **functions** clause defines the functionality of the system used to specify a rule for obtaining a result from zero or more arguments. The difference between operations and functions is that, operations can manipulate both global and local variables but functions can not access global variables or define local variables.

Figure 1.5 shows the simple student database example in VDM-SL that focuses on two operations: *AddStudent*, which adds a new entry to the system, and *DeleteStudent*, which deletes a student's entry from the system.

The specification above has two parts: data definition part and the function definition part. In the data definition part, *Name* is defined as a string, *ID* is defined using a basic data type *nat* which stands for natural numbers, *StudentID* is defined from other types by "mapping type" constructor that consists of two parts domain data type *Name* and range data type *ID*.

The function part consists of two functions, *AddStudent* to add a student to student database. *DeleteStudent* to delete a student from the student database.

The definition of a function in function block of a VDM-SL specification, may include a signature, an explicit function definition, and pre/post conditions. The signature part shows the inputs and outputs data types. For example, the input parameter data types of *AddStudent* function are *Name*, *ID*, and *StudentID* and the output data type is *StudentID*. "munion" is used to add student to *ID* mapping whereas ⩤ is used to delete a student from *ID* mapping.

1.7 Refinement

Refinement [29, 7] refers to the process of transforming the abstract specification to a more concrete specification through a series of refinement steps. Refinement assists in managing the complexity of a system being developed. The core functions of a system can be focused on in an abstract model whereas other aspects of the system can be focused on through different refinement levels. The posit-and-prove and transformational are two main refinement styles [14, 53]. In the first style, a refined model is verified through a set of proof obligations to prove that it is a correct refinement of its abstraction. This is done using theorem provers or model checkers (i.e. $S \sqsubseteq S'$ if and only if S' satisfies all desired properties that S satisfies). The latter style, however, is performed by transforming the abstract model into a concrete one by applying transformation rules in the abstract model to automatically extract a concrete model satisfying its abstraction.

There are also two aspects of refinement to make a model closer to implementation: horizontal refinement and vertical refinement [4]. Horizontal refinement or superposition is also known as a feature augmentation refinement, and centres on enriching the model gradually by integrating the system's details through different refinement steps [4]. In the horizontal refinement stage, the specifier follows the system requirements document and extracts the variables and events, formalising them through several steps of refinement. It is possible to add new variables and guards, or strengthen guards in a refined model. It is also possible to add new actions to an event or even extra events. This kind of refinement is usually finalised when the specifier reaches the point where there is no further requirement element left to be formalised.

Vertical refinement or data refinement is a type of refinement in Event-B which centres on the replacement of the abstract model with more design details in each refinement step down to an implementation [4]. Data refining finite set variables to array structure is a typical example of the vertical refinement [4]. In this kind of refinement, we can eliminate unnecessary variables of the prior models and identify gluing invariant(s) that relate the new concrete variables and the abstract ones [4].

The concrete level of the model must reflect the behaviour of the abstract level to ensure the correctness of the system. This can be achieved through the validation of the refinement level via specific proofs, called the refinement proofs.

The refinement in classical B is one to one; each abstract operation is refined by only one concrete operation and it is not allowed to introduce new operations. In Event-B, several different refined events may refine the same abstract event. It is also possible in Event-B to introduce new events during the stepwise refinement steps. Z employs some rules for the operation and data refinement. The validity of the refinement is ensured by simulations. A simulation is a representation of one state by another. It combines some rules to validate a refinement. There are two ways of simulations: downward simulation

(ensures the correctness of refinement from an abstract state to a concrete one) and upward simulation (ensures the correctness of refinement from a concrete state to an abstract one) [86]. The refinement mechanism in VDM is done through data reification and operation decomposition; data reification maps the abstract data types to more concrete ones, while operation decomposition is performed later and is used to map operations and functions to algorithmic representations [22].

1.8 Comparison of B, Event-B, Z and VDM

B, Event-B, Z, and VDM are model based languages that view a system as a state machine with a set of states and a collection of operations over these states. Event-B, Z and VDM are different but they all describe a system precisely as a mathematical entity and use set theory and first-order predicate logic. This section gives a comparison of Event-B, Z and VDM in terms of syntactic changes, data types, tool support and mathematical extension and refinement.

Syntactic Changes: As mentioned previously, the syntaxes differ between B, Event-B, Z, and VDM. In what follows, we will mention two examples of the most obvious syntax changes between the three languages. The first example is of state changing specification, Z uses prime variables to distinguish the states of the model after change. B and Event-B do not use specific symbols to denote the initial value of the variable before and after change. VDM, however, uses hooked variable for the before state and unhooked variable for after state.

The second example is of inputs and outputs specification, B and Event-B do not have any explicit way of distinguishing inputs from outputs. Input variable names in Z, however end up in "?" whereas output variable names end with "!". In VDM *rd* "read only" is used for states that are not allowed to be changed and *wr* "write" keyword is used for states that can change.

Data types: The basic data types in B, Event B and Z are sets and relations. VDM does not have the notion of relation data type and consequently it does not support relational image which Z and Event-B do support.

VDM and Z include record types and B does not; however, in Event B, it is possible to specify record types following the Evans and Butler approach [32].

VDM has different data types such as character type that are not found in Z, B and Event-B.

Tool Support: Z has some proof tools such as Proofpower-Z [54] and Z/EVES [66] theorem prover. AtelierB is a set of tools for B formalism that provides syntax checker, automatic and interactive provers, and code generation. RODIN is a platform used

for developing Event-B models that provides support for refinement and mathematical proof. It has different supportive plug-ins such as proof obligation generator, automated and interactive provers, theory plug-in that provides capabilities of mathematical extension in Event-B language and the proving infrastructure, and moreover RODIN offers code generation plug-in that aids in translating Event-B into a C or Ada code. VDM-Tools support VDM and include set of tools support software development and provides a bunch of features such as a type checker, document generator, theorem prover, and code generation to C++ or Java.

Mathematical Extension: B lacks the ability to extend the B mathematical language; Event-B, on the other hand, adopts theory plug-in, which is a recent feature providing support for extending Event-B language and proving infrastructure. It allows the development of new data types, set of operators, inference and rewrite rules, and theorems.

Z has an extensible mathematical tool-kit that provides a library of standard mathematical notions of operators and data types useful for building a system specification.

VDM, on the other hand, is a rich language of operators and data constructors such as union, sequences and record types from which user-defined data types can be built. It also has a large expression and statement language such as if-then-else and while-do.

Refinement: With respect to the refinement, operations in B, VDM and Z are "refined" on a one-to-one basis, therefore, one abstract operation is refined by only one concrete operation. In Event-B, however, an abstract event can be refined by one or more concrete events. It is also possible in Event-B to introduce new events (stuttering steps) in the refinement step.

1.9 The Motivation Behind Selecting Event-B For Modelling FreeRTOS

Event-B is a natural candidate for the FreeRTOS case study. We chose Event-B to develop FreeRTOS because Event-B is a stepwise formal method which has a platform supported with various plugins. The stepwise methodology allows the complexity to be managed through several refinement steps. Rodin, the platform for Event-B, is supported by useful plugins. As FreeRTOS uses complex data structures (circular linked lists), we make use of several plug-ins that facilitate building and reusing complex formal models involving complex data structures such as linked lists. We use the theory plug-in to develop reusable data structures including operators, polymorphic theorems, and inference rules. Generic instantiation is used to create an instance of a generic model while preserving the proofs associated with the generic model. A composition technique is used to compose the separated modelling components into a single model. We also

make use of the atomicity decomposition technique to manage flows between events and organise refinement steps. We also combine several techniques in Event-B to propose new approaches that facilitate building traceable formal models and support the reuse of Event-B modelling patterns. Chapter 4 and chapter 7, outline respectively an approach for building traceable Event-B models and an approach for reusing Event-B modelling patterns.

Chapter 2

Techniques in Event-B

This chapter describes some of the techniques in Event-B used in this thesis. The techniques cover: proof obligations, shared event (de)composition, generic instantiation, the theory definition, the atomicity decomposition and UML-B.

2.1 Introduction

This chapter presents relevant technical background of some techniques in Event-B which are adopted in the research. During the development of FreeRTOS, we adopted a number of techniques in Event-B such as: (de)composition, generic instantiation, the theory feature, the atomicity decomposition and UML-B. Applying the decomposition technique results in obtaining simpler models and enables the developer to focus on certain aspects of an Event-B model. For the purpose of developing reusable modelling component "patterns", generic instantiation technique provides us with a mechanism for instantiating Event-B models in a way that ensures the validity of the instantiated model proofs. On the other hand, the theory technique provides us with the facility of defining a set of reusable operators, polymorphic theorems, rewrite and inference rules. The atomicity decomposition approach provides a graphical notation to illustrate the refinement structures and assists in the organisation of the refinement levels. The UML-B provides UML graphical notation that enables the development of an Event-B formal model. Finally, the shared event composition allows sub-components to interact via synchronisation over shared events to show the overall behaviour of the FreeRTOS. Chapter 4 and Chapter 7 present new approaches for bridging the gap between the requirements and the formal models and support reusability in Event-B using some existing Event-B techniques.

Section 2.2 to section 2.7 describe the proof obligations, the composition and decomposition techniques, the generic instantiation technique, the theory feature, the atomicity decomposition, and the UML-B.

2.2 Proof Obligations

A proof obligation is a mathematical formula that requires to be proven, in order to ensure the correctness of an Event-B model. Proofs in Event-B are constructed using a number of sequents. A sequent is of the form ($H \vdash G$), where H is is a hypothesis that contains a finite set of predicates and G is the goal. The proofs of the sequent are carried out using inference rules that prove that the goal is a consequence of the hypotheses. The POs can be classified into consistency and refinement POs. In order to check the consistency of an Event-B machine, three main proof obligations should be proven true for each event: the well-definedness PO (WD), the event feasibility PO (FIS) and the invariant preservation PO (IP). The well-definedness PO ensures that invariants, guards, events, axioms, and variants (used to prove that a convergent event do not take over the execution) are well defined, for instance, in order to prove the well-defindeness of the cardinality of a set s: *card(s)*, the set s must be a finite set. The event feasibility states that it should be possible to execute an event from any state when both the machine invariant and the event guard hold. The invariant preservation states that the invariant should always be maintained. On the other hand, there are several proof obligations generated for refinement. Here we mention three important POs for refinement: the guard strengthening PO (GRD), the action simulation PO (SIM) and the variant decreasing PO (VAR). The guard strengthening PO ensures that the concrete guard in the refined event is stronger than the abstract one. The action simulation PO ensures that the concrete event action simulates the abstract event actions. Finally, the variant decreasing PO ensures that the new convergent event reduces the variants to ensure that the convergent event do not take over the execution. The variants are defined as an integer number or a finite set and has to become smaller each time the convergent event is executed to prove that the convergent events do not execute forever.

The POs are generated by the proof generator and can be discharged either automatically or interactively. The automatic theorem prover discharges many proofs automatically, however, it is possible that some POs are not discharged automatically, in which case they can be discharged by the user using the interactive prover.

Assuring consistency of requirements is important. The properties of the system driven by requirements are encoded as invariants in Event-B model. The invariants must not contradict each other. Proving that an invariant is always true for a given Event-B model through initialisation and event consistency POs ensures consistency (i.e. Absence of contradiction from the invariants). However, adding more invariants to an Event-B model may add inconsistency, but, the standard POs will detect that and it can be fixed.

2.3 Composition and Decomposition

Composition [73] is the process of composing several sub-models in a variety of styles. Composition has the potential of model reusability where it provides a way of constructing specifications as a combination of other different specifications. We focus on shared event composition style [73]. In this style, sub-models interact through synchronised events. 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 [73].

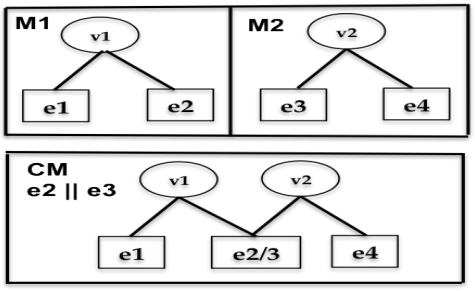


Figure 2.1: Shared event composition style.

Figure 2.1 illustrates this style; suppose we have a model *M1* that has events *e1*, *e2* with variable *v1*. Model *M2* has events *e3* and *e4* with variable *v2*. Events *e2* and *e3* can be synchronised since *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:

$e2 \triangleq$ **any** p
where
$G1(v1, p)$
then
$v1 := E(v1, p)$
end

$e3 \triangleq$ **any** p
where
$G2(v2, p)$
then
$v2 := E(v2, p)$
end

Therefore, *e2* from model *M1* and *e3* from model *M3* are composed. The resulted composed model *cm*, therefore, share the two independent variables: *v1* and *v2*.

The resulted composed event has the following form:

$e2 \parallel e3 \triangleq$ **any** p
where
G1(v1, p)
G2(v2, p)
then
$v1 := E(v1, p)$
$v2 := E(v2, p)$
end

Composition may also be applied in reverse in a top-down way by factorising a model into a composition of smaller models. We refer to this as decomposition. The decomposition approach is used to divide a model into sub-models that can be refined separately than the whole. Large system models involve a large number of features (state variables and machine events) which are extended throughout the refinement steps. As the model becomes larger, some state variables and machine events are considered in the next level. Thus, decomposition [7] is used as a means of dividing the system into small and manageable parts that ease dealing with the whole system.

Event-based decomposition is a style of decomposition proposed by Butler [74]. In this style, the machine events are split into sub-models and variables can not be shared between the sub-models. Figure 2.1 can be used to illustrate this style. The modeler can start with the model *cm*, and then decomposes it into two sub-models *M1* and *M2*. Where variables *v1* and *v2* of model *cm* are partitioned between *M1* and *M2* and event *e2/3* is split into two events *e2* and *e3*. Event *e2* updates variable *v1* and therefore is located in *M1* whereas event *e3* updates variable *v2* and is located in *M2*.

A decomposition plug-in [75] has been developed for the Rodin tool which can be used to support event based decomposition.

2.4 Generic Instantiation

Instantiation means the reuse of a pattern (chain of machines and contexts) where variables, events, constants and carrier sets can be renamed in an instance and reused in another development [72]. It ensures that the proofs associated with a generic development (pattern) remain valid in the instantiated development. This requires the creation of a new context (called "*parameterisation* context") containing the carrier sets and constants that will be used by the instance machine (or chain of machines). After the creation of an instance from the pattern, the pattern axioms are converted into theorems in the instantiated machine to ensure the correctness of the instantiation [72].

To illustrate this approach, supppose we have a generic pattern P with machines p_0 to p_n. Suppose that in some points in another development M, we find that its useful to reuse development P with some slight modification to resume development M. This

can be achieved by instantiating the various variables, events of P. The pattern axioms of development P should become theorems in development M in order to show the correctness of the instantiation by satisfying the pattern assumptions. This approach is also supported by a tool [72].

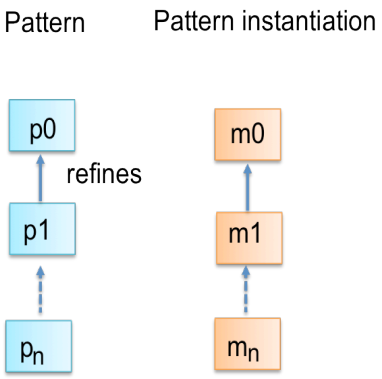


Figure 2.2: Generic instantiation.

Since refinement is monotonic, it is possible to go from m_0 to m_n directly by instantiating p_n. The instantiated machine m_n will be a refinement of m_0. Assuming that the pattern machine p_n is parameterised by some sets s and constants c, the pattern instantiations is parameterised by some sets t and constants d. The instantiation performed by replacing s and c by some expression $E(t)$ and $F(t,d)$. If the pattern has axioms $A(s, c)$, the pattern instantiation has theorems $B(t,d)$. The POs that ensure that the proofs associated to the pattern P remain valid in an instantiated development M are: $B(t, d) \Rightarrow A(E(t), F(t, d))$.

2.5 The Theory Extension

The theory feature [18] is a Rodin extension that allows users to define new data types, operators, rewrite and inference rules, and theorems. With the theory plug-in, users are able to define different data types and operators of their Event-B specifications. The polymorphic nature of the definitions of data types and operators provide the ability to reuse them across different modelling patterns. Here is a simple example of a sequence operator:

operator *seq*
prefix
args S
well-definedness condition $S \subseteq T$
definition $seq(S) \triangleq \{f, n \cdot f \in 1 \mathinner{..} n \rightarrow S \mid f\}$

Here, *seq* is defined as an operator that takes an argument S. S is a subset of T as given by the well-definedness condition. T is a type parameter, thus *seq* is polymorphic on T. The definition of *seq* is given by the final clause where *seq(S)* is the set of total functions from $1..n$ to S.

Polymorphic theorems provide reusable properties on reusable operators; they use to verify that definitions of operators are valid and to ensure that operators capture the intending behaviour of the system. The rewrite rules provide means to rewrite formulae to equivalent forms, whereas the inference rules are a special kind of polymorphic theorems which include hypothesis and conclusions. Rewrite and inference rules can be used to facilitate proofs.

The definition of inference rules in the theory component is presented in Figure 2.3:

inference *name*
[**automatic**] [**interactive**]
vars $x_1, .., x_n$
given $G_1, .., G_n$
infer I

Figure 2.3: The structure of inference rule definitions.

The structure of the inference rule can be read as follows: given conditions $G_1, .., G_n$, one can infer I. The developer decides whether the rule can be applied automatically or interactively with intervention from the user.

Figure 2.4 shows an inference rule for the sequence theory [18]:

inference *FiniteSeq*
vars s, m
given $s \subseteq T$
$m \in seq(s)$
infer $finite(m)$

Figure 2.4: Definition of sequence operator.

2.6 Atomicity Decomposition Approach

The atomicity decomposition (AD) approach [33] is introduced to show the relationships between abstract and concrete events and represents an explicit ordering between events

of a single level of refinement. The AD approach provides a graphical notation that helps to refine structure in Event-B by showing the relationships between events of different refinement levels. The AD diagrams can provide visual view of the ordering between some events as well as the overall structure of several refinement levels.

Figure 2.5 shows an example of an AD diagram.

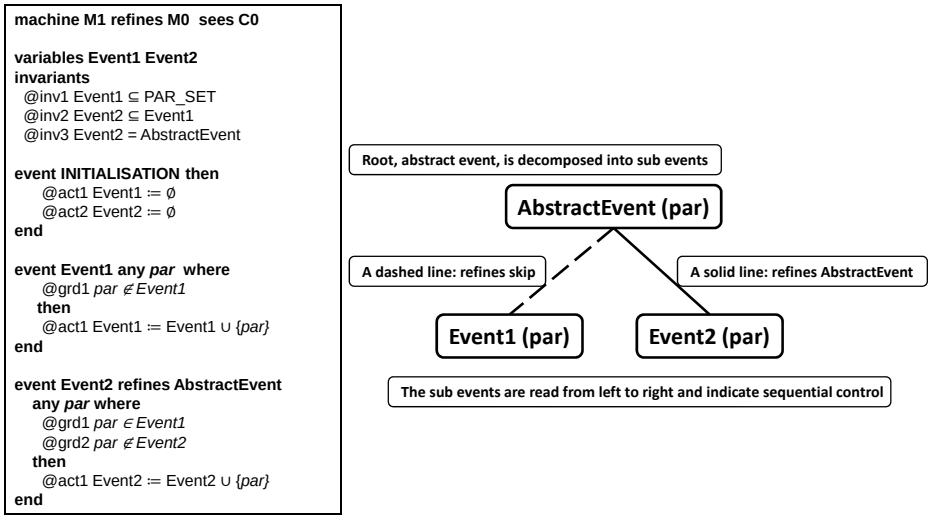


Figure 2.5: Atomicity decomposition diagram [33].

Assume we have a machine *M1* that refines machine *M0* which contains the abstract specification of *AbstractEvent*. The control flow is shown in the AD diagram presented in Figure 2.5. The leaf events are read from left to right and indicate sequential control from left to right (this is based on JSD diagrams [44]). Thus, the AD diagram shown in Figure 2.5 indicates that *AbstractEvent* at the abstract level *M0* is realised at the refined level by the execution of *Event1* followed by that of *Event2*. The refining relationship between an *AbstractEvent* and a concrete event *Event2*, is indicated with a solid line in the AD diagram, whereas the non-refining relationship is indicated with a dashed line to indicate that *Event1* is a new event which refines *skip*. The corresponding Event-B model of the AD diagram is shown on the left hand side of Figure 2.5. The ordering between *Event1* and *Event2* is represented by guards on the events.

In the diagrammatic representation, control parameter name *par* appears in between parentheses after the event name. In the Event-B model, *par* represents a list of parameters, $par1, ..., par_n$. The parameter *par* appears in the AD diagram to indicate that *AbstractEvent* and its sub-events can execute several times (for different values of the parameter). The set control variables *Event1* and *Event2* have same name as the events and are used to show that an event can occur multiple times with different values for the parameter *par*. The guard $par \in Event1$ of *Event2* means that *Event1* has occurred with value *par* whereas the guard $par \notin Event2$ means that *Event2* has not occurred for value *par*.

At the most abstract level of an Event-B model, the root of the AD diagram appears as an oval and the leaves represent the most abstract events as can be shown in Figure 2.6. They are connected to the root via dashed lines since the most abstract events do not refine the root node.

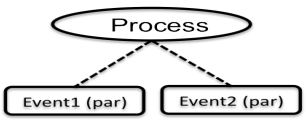


Figure 2.6: Atomicity decomposition diagram for the most abstract level.

There are several AD patterns such as sequence/and patterns, or/xor patterns, loop/all/some patterns, and one pattern. The description of these AD patterns are given in the following table:

Pattern	Description
sequence-/and-constructor	Execute events in a sequence. The difference between 'sequence' and 'and-constructor' is that an 'and-constructor' executes all the available events in any order, whereas the sequence constructor executes the 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 2.1: Description of the AD patterns.

2.7 UML-B

UML-B [77] 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: package diagrams, context diagrams, class diagrams and state machine diagrams. The package diagrams represent the structure and the relationship between Event-B contexts and machines. A context diagram describes the context part of an Event-B model. The class diagrams and state machine diagrams describe the state and the behaviour and are used in Event-B machines. The class diagrams in UML-B may

contain attributes (variables), associations (relationships between two classes), events and state machines (transitions between states). The state machine diagrams describe the behaviour of instances of classes as transitions linked to the events.

UML-B assimilates the notion of refinement. It is possible to introduce a class in a concrete machine that refines a class of its abstract machine. A refined class can keep all the 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 [78] has been developed for the Rodin platform which can be used to generate an Event-B model corresponding to a UML-B development.

Chapter 3

Experiences, Challenges and Contribution

This chapter outlines a number of challenges that we have faced during the course of our research and shows our experience of addressing some of theses challenges with two useful approaches in Event-B: The first approach supports the construction of traceable Event-B models from requirements and the second approach supports the reusability of Event-B models to resume other development.

3.1 Introduction

FreeRTOS is a real-time operating system. It is a complex system which involves complex data structures such as linked lists. The complexity of the system poses an increasing challenge on its development using Event-B. The following listing outlines a number of challenges involved in the development of complex systems that must be considered and overcome. Although, these challenges have emerged from the FreeRTOS case study, they can also be applied to other kinds of systems such as complex real-time and distributed systems.

- Handling large and complex system functionality
- Building Event-B models from the requirements and retaining traceability of requirements in Event-B models
- Deriving requirements/specifications from the source code to assist code maintenance and evolution
- Constructing modelling guidelines to aid the development of building Event-B models

- Supporting reusability to facilitate the construction of Event-B models
- Modelling complex data structures
- Verifying linear temporal properties

The remainder of this chapter is organised as follows: Section 3.2 gives an overview of the RTOS and FreeRTOS. Section 3.3 briefly describes the FreeRTOS models. Section 3.4 outlines our experience, challenges, and contribution of modelling FreeRTOS. Section 3.5 describes briefly the combination of some of the Event-B techniques to investigate an approach for reusing Event-B formal models and an approach for building traceable Event-B models from requirements. Finally, Section 3.6 discusses some related work regarding the use of formal methods in operating systems.

3.2 An Overview of RTOS

RTOS is a class of operating systems that is used for applications which have time constraints (Real Time Application). These kinds of operating systems are characterised by features such as fault tolerant design and fast task scheduling as they always have specific timing requirements, and they are usually small in size. The structure of an RTOS is shown in Figure 3.1; as can be seen, the RTOS forms the intermediate layer that masks the hardware details to the application level.

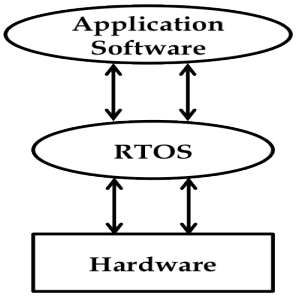


Figure 3.1: The RTOS structure.

The components of most RTOSes are [56]:

Scheduler: Special algorithms that are used to schedule objects. Some of the commonly used scheduling algorithms are: preemptive priority-based scheduling and round-robin scheduling. In preemptive priority-based scheduling, each task must be assigned a priority. At every clock tick, the scheduler runs the highest priority task that is ready to run. In round-robin scheduling, the tasks with equal priority get an equal share of processing time.

Objects: Entities that are used by the developers to create applications for real-time embedded systems. Some object constructs are: tasks which are objects created by a developer to handle a distinct topic, queues which are used for task-task communication, semaphores and mutexes which are used for the synchronisation between the tasks and the interrupts.

Services: Operations performed by the kernels such as task management, inter task communication and synchronisation, interrupt handling, and resource management. The task management includes operations such as task creation, task deletion, task suspension and changing task priority. Inter task communication and synchronisation are mechanisms that enable information to be transmitted from one task to another such as: message queues, pipes, semaphores and mutexes. Interrupt handling are software routines used to handle interrupts; an interrupt is a signal to the microprocessor indicating that an event needs immediate attention. Resource management are some kernel functions used to manage system resources such as the CPU, memory and time.

3.2.1 FreeRTOS

FreeRTOS [11, 10] is an open source, mini real time kernel. It was primarily developed by Richard Barry and written in C with few assembler codes, with the result that it can be modified and extended as required. FreeRTOS code is freely available under GPL license and supports different platforms such as ARM7 and ARM9, MicroBlaze, MSP430, Coldfire V1,V2, and V85078K0R. FreeRTOS has one scheduler that schedules the highest priority thread first and this can be configured for preemptive or cooperative operation.

Data transfer is established by means of queues, semaphores, and mutexes. FreeRTOS is characterised by its simplicity of design, its portability, and scalability.

There are two architectural layers in FreeRTOS design which are the “hardware independent” layer and the “portable” layer. The “hardware independent” layer is responsible for performing the operating system functions and contains four C files which are Task.c, queue.c, list.c and co-routines.c. Tasks.c provides functionality for task management, queue.c provides functionality for the task communication of any synchronisation mechanism, list.c contains the implementation of list data structure that is used by the scheduler, and co-routines.c contains the co-routine macros and function prototypes. The second layer, on the other hand, contains hardware specific processing and includes some files such as heap.c that provides memory allocation functionality. Tasks are implemented using circular doubly linked lists, queues are implemented using circular buffers and memory management schemes are implemented using arrays and singly linked lists.

3.3 FreeRTOS Models

FreeRTOS is a complex system composed of different components. We consider that the complexity of developing FreeRTOS can be reduced by dividing FreeRTOS into different independent components and later compose these individual components. This can be achieved by considering each component of FreeRTOS independently, and collecting requirements associated only to that component. Requirements that show the interaction between the components can be treated as composition requirements and used later to compose the individual components.

The main components of FreeRTOS are task, queue, and memory. Thus, each of these components was handled separately. We gathered requirements from different tutorials [10, 11] and FreeRTOS code [10]. We then classified these requirements into two classes: the first class has three sets of requirements: task requirements, queue requirements, and memory requirements. Each of these requirements handles a particular component of FreeRTOS. The second class is composition requirements that show the interaction between task, queue, and memory. The requirement document are presented in appendix A.

The modelling activity initially was carried out by modelling each set of requirements independently. The specification in Event-B starts by an abstract model followed by a number of refinement steps. Each refinement augments more details than the previous model. The following steps outline an overview of the FreeRTOS specifications:

Task management

An abstract specification- Some Basic Functionality of Task Management and the Kernel

We begin with an abstract model of task management and the kernel focusing on task creation, task deletion, interrupt handling and context switch. The interrupt handling mechanism handles events that are usually signalled by the interrupts by executing its interrupt service routine (ISR). Context switch is the mechanism used for swapping the tasks.

First Refinement-Scheduler States

The first refinement level specifies the scheduler states. The scheduler can exist in one of the following states: not started, running, and suspended.

Second Refinement-Task States

The second refinement specifies the task states. A task can be in one of the following states: ready, blocked/delayed, or suspended state.

Third Refinement- Hardware clock and timing properties

This refinement level specifies the hardware clock and timing properties associated with a delay task such as: the sleep time and the wake-up time.

Fourth Refinement- Delay Operations

This refinement level distinguishes two kinds of delay operations: "delay" and "delay until". The "delay" operation places the calling task into the blocked state for a fixed number of tick interrupts without consideration of the time at which the last task left the blocked state, whereas "delay until" is an alternative operation that delays a task until a specific time has passed since the last execution of that operation. Thus "delay until" allows a frequent execution of a task so it is suitable for the periodic tasks (arriving at fixed frequency).

Fifth Refinement- Clock Overflow

This refinement specifies clock overflow. It introduces a collection of overflow-delay tasks to store tasks whose wake-up time has overflowed.

Sixth Refinement- Priority

This level introduces priority. FreeRTOS uses a highest priority first scheduler. This means that the higher priority task runs before the lower priority task. The scheduler then uses this priority to schedule the task with the highest priority.

Seventh Refinement- Contexts

This level specifies task contexts; the task context represents the state of the CPU registers required when a task is restored. If the scheduler switches from one task to another, the kernel saves the running task context and uploads the context of the next task to run. The context of the previous running task is restored the next time the task runs. Therefore, the kernel resumes the task execution from the same point where it had left off.

Queue management

An Abstract Specification-Queue types

The initial model formalises three types of queues: queues, binary semaphores and counting semaphores which are used for communication and synchronisation between the tasks, or between the tasks and interrupts.

First Refinement-Waiting Events

This level of refinement considers additional requirements pertaining to the addition of a task to event lists; $TaskWaitingToSend$ variable is introduced to store the tasks that failed to send items to a queue because the queue was full, and $TaskWaitingToReceive$ variable is introduced to store the tasks that failed to receive items because the queue was empty.

Second Refinement-Lock Mechanism

This level defines lock mechanism. Lock mechanism is used to prevent an ISR from updating the event lists $TaskWaitingToSend$ or $TaskWaitingToReceive$ while a task is being copied to the event lists.

Third Refinement - Mutexes

Our third refinement specifies the mutex; mutex is a type of binary semaphore that uses priority inheritance mechanism to reduce priority inversion. Priority inversion is a problem which occurs when a high priority task awaiting a mutex has been blocked by a low priority task which holds that mutex. Priority inheritance works by raising the priority of the lower priority task that owns the mutex to that of the highest priority task that is attempting to obtain the same mutex.

Fourth Refinement -Recursive Mutex

This level introduces a recursive mutex. Recursive mutex is a type of mutex that enables a token to be "taken" repeatedly by its owner. Its only becomes available again when the owner unblocks the mutex the same times it has locked it.

Memory management

An Abstract Specification - Memory Blocks and Addresses

There is two abstract models for memory. The first abstract model defines scheme1. It includes variables for blocks and addresses. The blocks are classified into a number of allocated blocks and one free block. This is because scheme1 fills the heap from the bottom and leave the top of the heap to be free.

The second abstract model defines scheme2. It includes variables for blocks and addresses. The blocks are classified into a number of allocated blocks and a number of free blocks. Unlike scheme1, memory in scheme2 can be freed once it has been allocated.

The proof statistics of the FreeRTOS models are given in following table:

Task Management (TM)			
Machines	Total POs	Automatic	Interactive
M0	32	30	2
M1	10	10	0
M2	72	68	4
M3	24	24	0
M4	39	38	1
M5	28	25	3
M6	41	32	9
M7	34	29	5
Total (TM)	280	256	24
Queue Management (QM)			
M0	53	46	7
M1	28	24	4
M2	109	70	39
M3	70	42	28
M4	19	10	9
Total (QM)	279	192	87
Memory Management-Scheme1 (MM1)			
M0	32	24	8
Memory Management-Scheme2 (MM2)			
M0	32	22	10
Overall	623	494	129

Table 3.1: Proof statistics of the FreeRTOS models.

3.4 Experiences and Challenges

This section serves as an overview of the overall thesis. It highlights some of the challenges that were faced throughout the development of FreeRTOS and suggests solutions to overcome these challenges. Some of these challenges are left as future work. The actual detail contributions will be discussed in subsequent chapters of the thesis.

– **Handling Large and Complex System Functionality**

The method we provide for structuring requirements into two classes consists of a set of requirements for each individual component and another set of requirements for composition purposes. In that way, we provide means of handling large and complex systems by dividing a problem into different independent smaller components and later composing these individual components. This method identifies the system based on compositional strategy in which the overall specification emerges from composing its sub-systems. In addition, with this method, the decomposition techniques are not used while modelling as the identified components do not require further breakdown. The compositional design strategy seems to be useful in designing reusable models specially with redundant modelling components. This is because the constructed formal models using compositional design strategy are

loosely-coupled models and are easier to adapt with different developments than the formal models that share several properties and variables.

In contrast to the compositional strategy, an alternative design strategy of complex system is to start with a single model that captures the main features of FreeRTOS problem and then decomposes that model into several sub-models using shared event/shared variable decomposition techniques to handle each aspect separately. This design approach seems to be useful when it is important to reason earlier about the initial model to ensure that it adequately models the desired system, but it does not help to construct reusable modelling components since it results in constructing models that share properties and variables.

– **Building Event-B Models from Requirements and Retaining Traceability to Requirements in Event-B Models**

The ability of building links between the requirements and the formal modelling holds crucial in system development. It enables bridging the gap between the informal requirements and formal models, and also verifies if all the requirements are fulfilled or not. Moreover, with this ability, formal models can be verified to meet changes in the requirements. During the modelling activity of FreeRTOS, it was a difficult task to go from requirements to build FreeRTOS models. The informal nature of the requirements makes the movement from requirements to models more cumbersome. We were not quite sure whether all the requirements were covered or not, and also it was difficult to validate the model against the requirements and maintain changes in the model when requirements are changed. We attempted to address this issue by providing an approach that uses intermediate languages represented in UML-B diagrams and AD diagrams to assist in the construction of traceable Event-B models. The UML-B provides a 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 proposed approach brings the natural language requirements and Event-B models together, and links each requirement directly to the Event-B constructs through UML-B and AD diagrams in three main steps: The first step is based on classifying requirements according to the Event-B constructs, the second step uses an intermediary (semi-formal) notation using UML-B and AD diagram to represent requirements and bridge the gap between the requirements and the Event-B models. The AD diagrams are also used in this step to develop refinement strategy. The third step uses UML-B tool and the AD tool to generate Event-B models. We applied the approach to specify queue management model. Details of the approach are given in Chapter 4 and Chapter 5.

– **Deriving Requirements/Specification from the Source Code to Assist with Code Maintenance and Evolution**

The textual requirements that are collected from authoritative resources are important for building the formal models. However, finding all necessary information in such resources can be difficult. Incomplete, inconsistent and ambiguous requirements are often encountered during this process. Another important source of information is the systems code. Extracting design requirements from the systems code is a difficult task. For instance, FreeRTOS is implemented roughly into four C files and the process of inspecting FreeRTOS code to understand details implementation would require devoting great time and efforts. Most FreeRTOS materials are usually devoted to explain FreeRTOS from the application developer's point of view. The design details of FreeRTOS are hidden in the C code and are usually not mentioned in the FreeRTOS materials. Therefore, there is a clear need to investigate and learn C code to better understand the implementation of FreeRTOS. A possible technique to understand the C code is to reverse engineer FreeRTOS code. Reverse engineering refers to the process of constructing high level representations of an implementation. Reverse engineering is considered as a good solution for handling legacy code than developing software from the original requirements. As a future plan, we will attempt to address this difficulty by investigating an approach that assists in the reverse engineering of structured programs to Event-B models. By focusing on structured programs, we find that structured programs are often composed of hierarchical program flow structures such as selection and repetition. On the other hand, the AD approach in Event-B provides a set of patterns that supports flow in Event-B models. We believe that it might be useful to use the AD approach to translate the programme flow structures to the corresponding AD flow structures and generate the Event-B specifications. This idea is not explored yet in this thesis and is considered as a future of our work.

– **Constructing Modelling Guidelines to Aid Building Event-B Models**

The ability to reuse modelling components for recurrent concepts are demanded to save time and efforts. The RTOS kernels are good example of this. RTOS kernels share similar components such as task and queue, scheduler, and a set of services such as task management, queue management and memory management. The derivation of modelling guidelines for recurrent concepts of RTOS is a good means to support reusability and assistance in the development of RTOS using formal methods. One of the objectives of this thesis is to come up with a set of modelling guidelines that can aid modelling of the RTOS kernels in Event-B. We dedicated time to extract general modelling guidelines for RTOS kernels. The modelling

guidelines were extracted based on our experience with modelling Event-B. We believe that these guidelines can be of benefit to assist the development of RTOS kernels. Full details about these guidelines are given in Chapter 8.

– **Supporting Reuse to Facilitate the Construction of Event-B Models**

Reusing Event-B models to resume the development of other developments is crucial to save time and efforts especially with complex systems that involve several recurrent components. During the modelling activity of FreeRTOS, we found out that there is a great need of reusing existing models to resume the development of FreeRTOS specification. In particular, we found that the abstract "set" in the task, queue and memory models must be data refined into a circular doubly/singly linked lists. Tasks states: running, ready, delay, suspend, and blocked in the task model are specified as " set" and need to be data refined into a circular doubly linked list. The process of refining each set to circular doubly linked lists demands a significant modelling and proving efforts. Therefore, developing an approach that aids reusing of models to resume other developments is necessary to save time and reduce modelling and proof efforts. We attempted to construct a single model of circular doubly linked list model "pattern" which refines a "set" and reuses it to data refine task sets. To achieve this, we found that it is necessary to have a renaming mechanism that renames the pattern to suitable names each time it is reused for the development. We also found out that it is necessary to have a mechanism to incorporate the pattern into the development. Luckily, both mechanisms are supported in Event-B. Generic instantiation supports the instantiation of the refinement pattern including proofs and allows renaming the pattern components (types, constants, variables, and events) to suitable names for the problem specification. Composition (shared-event style) enables the incorporation of the instantiated pattern in the development. Thus, we investigated an approach based on the combination of generic instantiation and composition techniques to facilitate the reuse of existing Event-B models to resume the development of other models. More details about the proposed approach can be found in Chapter 7.

– **Modelling Complex Data Structures**

The main data structure of FreeRTOS is linked lists. Linked lists are complex data structures which consist of a group of nodes and pointers. Linked lists are widely used data structures which can be used to implement many important data structures such as stacks, queues, and graphs. Linked lists can be used to implement several systems such as the internals of a file system to maintain available blocks of storage and directory structure to maintain a directory of names. There is a little support for modelling linked lists in Event-B, however, the theory feature in Event-B provides a mechanism to enhance the extensibility of Event-B toolset. It allows construction of new operators and new proof rules to suit users

needs. Thus, we make use of the theory feature to develop theories for linked lists to aid modelling of linked lists in Event-B. The developed linked list theories include sets of polymorphic operators to update a pointer structure when an item is added or removed from the linked list. It also includes polymorphic theorems and rewrite rules that assist in discharging proof obligations. The developed linked list operators are reusable and can contribute to supply a library of different theories in the future that would be of benefit to Event-B specifiers.

– **Verifying Linear Temporal Properties**

The verification of temporal properties is considered very important. Some requirements such as the requirements that describes liveness properties are hard to trace to components of Event-B model but simple to be formulated as temporal logic expressions. An example of a requirement that describe liveness property is the following requirement:

LIV	A task having been scheduled will eventually be executed.

It is crucial to find a method that specifies and verifies temporal properties for any modelled system such as the real-time kernels. Encoding temporal properties in Event-B and using Rodin tool to ensure that the encoded temporal properties are provable is a challenging task. In future work, we are going to study the possibility of formalising temporal properties using AD diagrams. The important motivation for selecting the AD approach to formalise the temporal properties in Event-B is that the AD approach explicitly shows the relationship between the abstract and refined events. The AD diagrams are translated into Event-B models as invariants, guards, and actions that can be verified using Rodin tool. Although, this topic is not covered in the thesis, we identify it as an issue and we plan to investigate the possibility of using the AD approach to support the verification of temporal properties in the future.

3.5 Two Approaches Supporting Formal Developments in Event-B

We have investigated two approaches that combine the existing Event-B techniques to support formal modelling. The first approach assists in reusing the Event-B formal models to resume the development of other models, whereas the second approach assists in building traceable Event-B models from requirements. The following subsections are briefly described both the approaches.

3.5.1 An Approach for Reusing Event-B Models

We present an approach for reusing the *modelling patterns* in Event-B through the process of constructing a new model in such a way that preserves the proofs. The reusability approach is based on the (de)composition and generic instantiation techniques. The decomposition technique is applied to extract part of the problem that matches with the pattern. The generic instantiation technique is then used to instantiate the pattern. Finally, the composition technique is applied to integrate the instantiated pattern into the problem. The reusability approach has the potential to reduce proof effort specially if the modelling patterns are integrated into building larger systems. The presented approach is applied to data refine the abstract FreeRTOS structures "set" into circular doubly linked list and singly linked list.

The architecture of the reusability approach is depicted in the following figure.

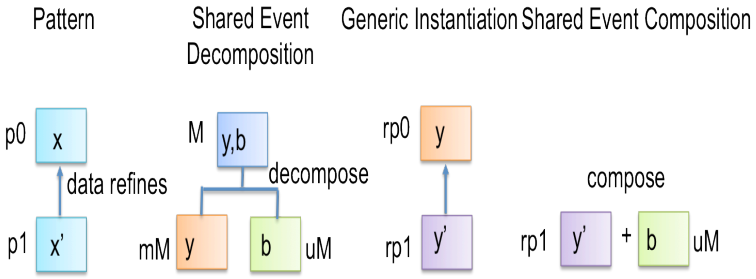


Figure 3.2: The architecture of the reuse approach.

We have a pattern P that consists of two machines, $p0$ with variable x and $p1$ that refines $p0$ and has the variable $x\prime$ that refines x. At the abstract machine M, in a particular specification with variables y and list of variables b, we figure out that a suitable continuation of the development would be to reuse the pattern P. In order to apply the pattern P to resume the development of M, we start by extracting the part that matches with the pattern in M using shared event decomposition approach. The part that matched the pattern of M is placed in machine mM, the remaining unmatched part of M is placed in machine uM. In the second step, we apply generic instantiation approach to instantiate the pattern with proofs and rename the pattern components to suitable names. The pattern P with machine $p0$ and $p1$ has been instantiated to rP pattern with machines $rp0$ and $rp1$. x has been replaced by y and $x\prime$ has been replaced by $y\prime$. In the third step, we composes uM with $rp1$ and show that the resulting composed machine is a refinement of M.

3.5.2 An Approach for Building Traceable Event-B Model from the Requirements

We present an approach for constructing a formal model from the informal requirements with the aim of retaining traceability to requirements in the models. In our approach, we make use of UML-B and Atomicity Decomposition (AD) approaches. The presented approach comprises of the following three stages:

- **Requirement Classification**

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

- **Construct Semi-Formal Artifacts and Develop Refinement Strategy**

  We use semi-formal artifacts described using UML-B, atomicity decomposition diagrams and structured English to represent the requirements. The UML-B is used to represent data-oriented requirements. The atomicity decomposition is used to represent flow-oriented requirements and 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 proper class (constraint or event-oriented). 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.

  We also combine atomicity decomposition diagrams and use them to assist the process of developing the refinement strategy.

- **Construct Formal Models**

  We 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.

3.6 Related work

This section examines some of the related work regarding the use of formal methods in operating systems.

Craig's work is one of the fundamental sources in this field [23, 24]. He focuses on the use of formal methods in OS development, and the work is introduced

in two books. The books contain formal specifications of simple and separation kernels along with the proofs written by hand. The first book is dedicated to specify the common structures in operating system kernels in Z [79] and Object Z [76], with some CCS [59] (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 book is devoted to the refinement of two kernels, a small kernel and a micro kernel for cryptographic applications. The books contain proofs written by hand, with some mistakes and some missing properties resulting due to manual proofs, some of which have been highlighted by Freitas [37].

Freitas [37, 80] 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 [37].

Furthermore, Déharbe et al [27] 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 [26].

The modelling work outlined in this chapter attempts to use a compositional strategy to build RTOS kernel modelling component using the Event-B. The compositional strategy allows us to specify each redundant modelling components in a separate model without considering the interaction between the components. The compositional strategy directs us to divide the requirements into several sets. Each requirement set is related to a specific component of FreeRTOS. Following that, we model each set of requirements separately without considering the connection between the different components. The composition of the separated models then becomes an explicit task. The compositional strategy helps in producing loosly-coupled models that can be reused easily than models that share several properties.

There is also an earlier effort by Neumann et al [60] to formally specify PSOS (Provably Secure Operating System) using a language called SPECIAL (SPECIfication and Assertion Language) [34]. 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. PSOS was designed at SRI international [64]. The work began in 1973 and the final design was presented in 1980 [60]. PSOS was focusing on the kernel design and it was unclear how much of it has been implemented [28]. Yet, there are other works inspired by the RSOS design such as Kernelized Secure Operating System (KSOS) [62] and the Logical Coprocessing Kernel (LOCK) [69].

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) [83] on the formalisation of the UCLA Unix security kernel. The work is developed at the University of California at Los Angeles UCLA 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 [42, 63]. 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 [50, 49] 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 [43] that provides an intermediate level that satisfies bottom-up and top-down approaches by providing a programming language for kernels developer and at the same time providing an artefact the 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 [61] shows that the implementation conforms with the abstract specification.

Chapter 4

Staged Approach for Constructing Models from the Requirements

Constructing traceable Event-B models from the 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. The objective of this chapter is to present a way based on the use of semi-formal structures to bridge the gap between the 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) approach. The 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 chapter is to harness the benefits of the UML-B and AD approaches to facilitate construction of the Event-B models from requirements and provide traceability between requirements and Event-B models.

4.1 Introduction

This chapter describes how FreeRTOS requirements are identified and also presents an approach for incrementally constructing a formal model from informal requirements with the aim of retaining traceability to requirements in models.

Decomposing the requirements into three components can be seen as a preparation step for a compositional development in which the overall specification emerges from composing its sub modules. There are a number benefits to this process.

Firstly, decomposing the problem into sub-problems promotes reusability; sub-problems can be treated as reusable component "patterns" and could be used in the development of other systems. Secondly, decomposing a problem into set of sub-problems supports team collaboration environment; where each developer or team can participate in handling a specific sub-problem separately from other sub-problems. Moreover, if the decomposition is made at the requirements level; there would be no need for adopting decomposition techniques and getting restricted by the model decomposition pitfalls such as the inability of refining shared variables in shared-variable decomposition technique.

We aim at providing clearly-defined requirements of FreeRTOS components. The architecture of the FreeRTOS has influenced our analysis of the FreeRTOS requirements. The division of the FreeRTOS into several layers: tasks, queue, etc enables the study of each layer separately from the others. Therefore, we began by studying each component "source code C File" separately and came up with a set of requirements for each component. We have three set of requirements: task management, queue management and memory management. Each set of requirements concerns only with a particular component of FreeRTOS and does not show how a component interacts with the other components. Therefore, we introduce another set of requirements namely composition requirement that shows the interaction of FreeRTOS components. The requirements of the FreeRTOS have been collected from the FreeRTOS codes and several other resources that discuss FreeRTOS [11, 10]. Following that, we present an approach that helps to bridge the gap between the requirements and the formal models. The approach helps to identify the modelling elements from the requirements, assists in 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 requirements document and allows missing requirements to be easily accommodated in the model.

In our approach, we make use of UML-B [77] and atomicity decomposition (AD) [15, 33] 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 the events. The visual view of the system provided by the 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 tools.

The presented approach comprises of three stages, shown in Figure 4.1.

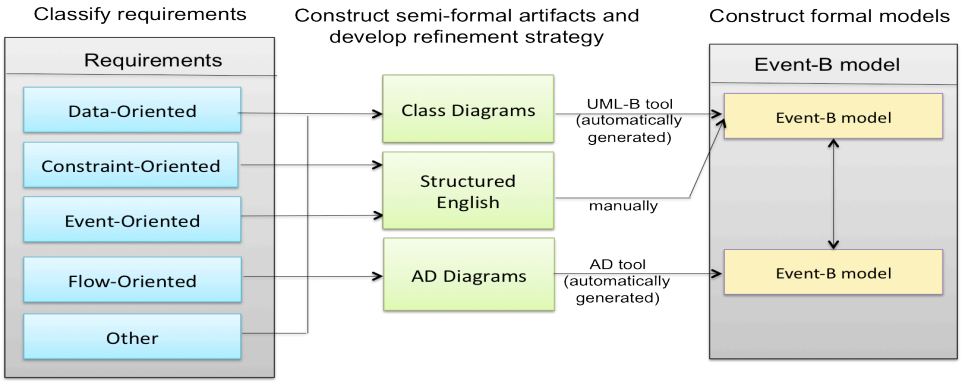


Figure 4.1: Steps for constructing a traceable formal models.

The first step in our approach is requirement classification. The requirements are classified based on Event-B components. The classification consists of five classes: data-oriented, constraint-oriented, event-oriented, flow-oriented, and others. The data-oriented requirements represent attributes and relationships between the attributes, the constraint-oriented requirements represent conditions that must remain true in the system, the event-oriented requirements represent the activities of the system and its components, the 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 main stages: Firstly, we use semi-formal artifacts described using UML-B, AD diagrams and structured English to represent the requirements. The UML-B is used to represent data-oriented requirements. The AD is used to represent the 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 proper class (constraint or event-oriented). The semi-formal artifacts serve as an intermediate representation that map the requirements to Event-B formalism. Representing requirements using semi-formal artifacts is reasonably simpler, 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 of the event components. Thirdly, we combine AD diagrams and use these diagrams to assist the process of developing the refinement strategy.

The third step of the proposed approach 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.

The remainder of this chapter is organized as follows: Section 4.2 outlines the requirement classification. The description of the presented approach is introduced in Section 4.3. Section 4.4 introduces some related works in requirement traceability. The conclusions are drawn in Section 4.5.

4.2 Requirements Classification

The way we construct the requirements document is based on the Abrial style [4], and there are two parts to this process. The first part is the text that explains the system to be modelled; this aids in understanding the problem at hand. The second part is a reference text that is short, precise and is labelled and numbered; this part is the one the modeler will reference while modelling. *TSK2* is an example of a reference text requirement. Large requirements can be decomposed into a set of shorter and understandable requirements. Good requirement documents list the requirements in order, starting from the requirements that describe a problem abstractly to more detailed ones. Ordering the requirements points the specifier to the right organisation of the model and the right order of the refinement levels. For instance, the requirements *TSK2* and *TSK14* show the need of modelling the collection of ready tasks after the scheduler so that the *idle* task is added to the collection of ready tasks when the scheduler starts (i.e the event responsible for running the scheduler is extended to add the idle task to the collection of ready tasks).

TSK2	When the scheduler is started, the idle task is created.
TSK14	When a task is created, it is added to the collection of ready tasks.

Ordering the requirements document and structuring refinement levels demand significant efforts. Thorough analysis of requirements are required to decide the most appropriate organisation of the refinement levels. We attempted to make use of the atomicity decomposition approach [33] to organise the refinement levels. Detail description on this issue is given in stage 3 of section 4.3.2.

Revising the requirement document is usually a continuous process, so after structuring the requirements, we embark on the modelling task. It is quite often that an ambiguous, inconsistent or missing requirement is experienced whilst modelling; therefore the requirements document must be revised continuously.

Requirements elicitation and requirements structuring are the main two phases of system analysis. Information that describe what the system should function can be gathered from several sources. Interviewing, survey, directly observing users, etc are example of requirements collecting methods. Requirement structuring phase focuses on augmenting methods to present the requirements such as data flow diagrams and entity-relationship diagram. Collection of FreeRTOS information carried out using different materials that describe FreeRTOS and from the source code. To reduce the complexity of structuring high volume of data, we classified the requirements into different sets. The obvious classification of FreeRTOS requirements is based on its components. Therefore, we classified FreeRTOS requirements into three classes. Task requirements set, queue requirements set, and

memory requirements set. The requirements *TSK3* and *QUE2* are examples of this classification. Requirements classification can vary according to the system architectures and the core features that the developers looking at. For instance, controlled system requirements can be classified into controlled requirements, monitored requirements, and commanded requirements.

In addition to task, queue, and memory classes of FreeRTOS requirements. We noticed that some requirements formed part of two independent requirements sets. We considered such requirements as composition requirement since each requirement of this class describes a possible connection between two independent components. Thus, we added another class called requirements composition set. The requirement $COMP3 - EVT$ is an example of composition requirements.

TSK3	Only one task can execute on a processor at a time.

QUE2	The length of the queue is identified when the queue is created.

COMP3-EVT	The running task can send/receive an item to/from a queue.

The requirements that represent the connections between the sub-problems need special attention. Each requirement of this type needs to be decomposed into several parts that can map to appropriate sub-problems.

The requirement *COMP3-EVT*, for instance is shared between the task sub-problem and the queue sub-problem. Therefore, it can be decomposed into two requirements: The first requirement labelled as *TSK10*, relates the first part of the requirement *COMP3-EVT* to the task sub-problem whereas the second requirement labelled as *QUE14* relates the second part of the requirement *COMP3-EVT* to the queue sub-problem. The requirements *TSK10* and *QUE14* fit the first class (i.e. requirements associated to a certain component), whereas *COMP3-EVT* is a composition requirement as it shows the connection between different components in different sub-models.

TSK10	The running task can hold an item or remove an existing item.
QUE14	A queue can be used to send and receive items.

The way the requirement *COMP3-EVT* is decomposed gives an insight into the possible composition techniques that could be used to show the interaction between the requirements *TSK10* and *QUE14*. In fact, both the requirements, *TSK10* and *QUE14*, are event-oriented requirements as explained in Section 4.3.1, and the

composition clearly could be obtained using shared event composition technique. The event driven by requirement *TSK10* could be combined with the event driven by the requirement *QUE14* to build the interaction between the task and queue components. An important point to note is that the composition requirements given in this thesis could be of two types:

- Composition requirements that connect the relevant events as shared events such as the requirement *COMP3-EVT*
- Composition requirements that specify the invariants linking variables of different sub-models such as the following requirement:

COMP1-INV	Tasks and queues are distinct.

In more complex cases, it is quite difficult to decompose some requirements; consider, for example the following requirement:

COMP9-EVT	The running task will be added to the collection of delay tasks and the collection of tasks waiting to send, if it attempts to send an item to a full queue.

According to the other requirements, we know that the collection of delay tasks stores the blocked tasks, so, the first part of the requirement *COMP9-EVT* belongs to the task sub-problem. We also know that each queue has a sending task-waiting list that stores tasks that fail to send items to that queue. The difficulty here lies in determining the appropriate requirement class for the sending task-waiting list. The sending task-waiting list is a strong mapping between the two separate components: task and queue and the identification of the correct class for the sending task-waiting list remains unclear.

Nevertheless, if we consider the second part of the requirement particularly checking out the availability of the queue, we might find that the queue requirements could be the appropriate class to define sending task-waiting list. Therefore, we can decompose the requirement *COMP9-EVT* into the following two requirements:

TSK21	The running task can be put in the collection of delay task.
QUE14	The running task is added to the collection of tasks waiting to send if it attempts to send an item to a full queue.

Likewise, the composition can be obtained by combining the event driven by the requirement *TSK21* with the event driven by the requirement *QUE14*.

4.3 Steps for Constructing Traceable Event-B Models

This chapter also presents an approach for constructing traceable Event-B models using UML_B and the AD approach. In this chapter, we do not address the question of how we arrive at the requirements. Requirements could have been sorted out by deploying the use cases [51]. In use cases, the systems functionality is described through structured stories in an easy-to-understand textual form, from which requirements could be drawn. Our objectives are to provide a link between the requirements and the formal models and to facilitate building traceable Event-B formal models from requirements. The following subsections describe the steps proposed to achieve these goals.

4.3.1 Step 1: Classify Requirements

Based on the structure of Event-B models, we classify the requirements into the following five classes: data-oriented requirements, constraint-oriented requirements, event-oriented requirements, flow requirements, and other requirements. Each requirement must be classified into at least one category. A detailed description of the requirement classification, with examples of lift controller requirements taken from [65], is given below.

Data-oriented requirements: are requirements that describe the attributes of the nouns and the relationships between them. Nouns represent objects and attributes represent the values of the object. Here are two 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 defined as data-oriented requirement since they describe different states of the lift.

Constraint-Oriented Requirements: are requirements that describe the properties of 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.

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

Event-oriented requirements: are requirements that describe behavioural characteristics of the system or its components (i.e. how an object acts). 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 the conditions under which an event can happen is called a guard requirement, whereas the part of an event-oriented requirement that describes how the state is going to change is called an action requirement.

Flow requirements: are requirements that describe the connection between the operations/events. This class describes the order in which operations/events are occurred. We can classify flow requirements generally into three types: sequence requirements which describe sequencing between the 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 4.1 provides several examples of the 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 be 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 4.1: Description of flow requirements.

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

Other Requirements: are those requirements that do not fit into the previous classes. This includes the requirements that are very hard to model in Event-B, such as the requirements that represent temporal properties or timing properties.

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

This step comprises of three stages described as follows:

4.3.2.1 Stage 1: Use Semi-Formal Artifacts (UML-B, AD, and Structured English)

In the first stage, the 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 the nouns are represented using UML-B associations, and transitions between the attributes are represented using the 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 into their requirement identifiers to facilitate the traceability of requirements.

  The structured English representation for the constraint-oriented requirements is of the following form:

  constraint : $< constraint\ \ requirement > \longrightarrow < REQ >$

  The structured English representation for event-oriented requirements is of the following form:

  event *name*
  guard : $< guard\ \ requirement > \longrightarrow < REQ >$
  action : $< action\ \ requirement > \longrightarrow < REQ >$

  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: 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. The description of the AD patterns is given in Section 2.6.

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

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

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

Figure 4.2: 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., 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 4.3, which shows two states, "open" and " close", and two transitions *OpenLiftDoor* and *CloseLiftDoor*.:

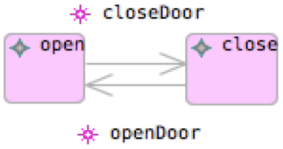


Figure 4.3: The state machine diagram for REQ2.

The data-oriented requirement ***REQ3*** is represented using the state machine in Figure 4.4, which shows two states, "stopped" and " moving", and two transitions *LiftStop* and *LiftMoving*.

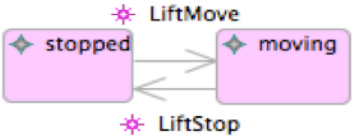


Figure 4.4: The state machine diagram for REQ3.

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

constraint : *The lift door of a moving lift must be closed* $\longrightarrow$ *REQ4*

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

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

The flow requirements ***REQ6***, ***REQ7*** and ***REQ8*** can be represented as follows:

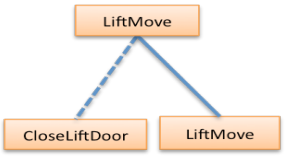


Figure 4.5: The ADD for REQ6.

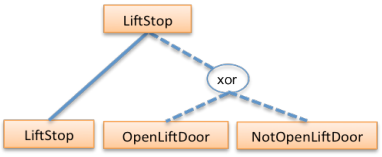


Figure 4.6: The ADD for REQ7.

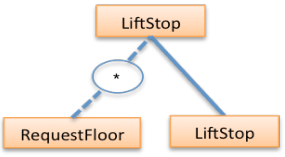


Figure 4.7: The ADD for REQ8.

Figure 4.5 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 4.6 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 4.7 is exhibited by executing the *RequestFloor* event multiple times followed by the *LiftStop* event.

4.3.2.2 Stage 2: Merging 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 any such requirements that refer to a single event.

4.3.2.3 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 architecture that the refinement is aiming towards, and 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 visually illustrate 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 the events given by the AD diagrams help us 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 properties 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 in deep refinement levels, the developer can introduce complex properties that are used by the real-time OS to handle the processes.

Figure 4.8 shows the refinement levels for the lift controller case study.

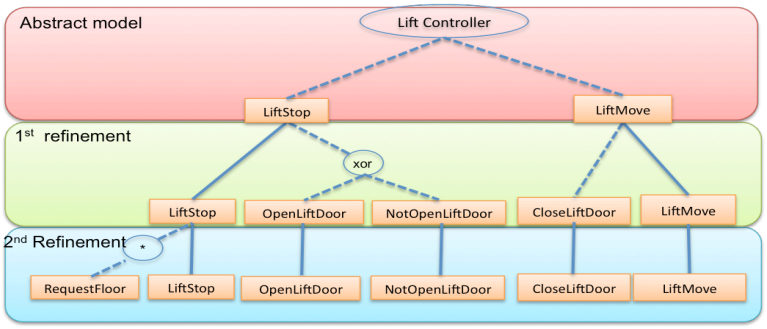


Figure 4.8: The combined AD diagrams for the lift controller.

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 the root *AbstractLiftStop*, that corresponds to the AD diagram in Figure 4.6 and the tree structure with root *AbstractLiftMove* that corresponds to the AD diagram in Figure 4.5. Finally, we make use of the AD diagram given in Figure 4.7 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 the event names. The generated gluing invariants show the relationship between the abstract flags and concrete flags. Therefore, the flag names for the abstract events are required to be different from the flag names for the refined events.

4.3.3 Step3: Construct Formal Models

In stage 3, we organised the refinement levels based on the combined AD diagram and then we structured the hierarchy of the class diagrams according to the combined AD diagram. So, we have three UML-B contexts: *c0*, *c1* that sees *c1*, and *c2* that sees *c1*. We also have three UML-B machines, the abstract machine *m0*, the first machine *m1* that refines *m0*, and the second machine *m2* that refines *m1*. Similarly, the AD gives rise to three contexts: *c0*, *c1* and *c3* and three machines: *m0*, *m1*, and *m2*. In this subsection, we use the UML-B and AD tools to convert the diagrams of step 2 to Event-B models. We also manually convert the structured English representation into Event-B. The Event-B model for the class and state machine diagrams are generated using UML-B. Since UML-B support the refinement concepts. Each UML-B machine gives rise to both an Event-B context and an Event-B machine. Similarly, each AD machine gives rise to an Event-B context and an Event-B machine.

Here, we give some examples on the Event-B specifications arises from the UMl-B and the AD tool.

The Event-B specification of the requirements *REQ1*, *REQ2*, and *REQ3*, generated from the class and state machine diagrams are as follows:

Requirements *REQ1*, *REQ2*, and *REQ3*:

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 Event-B model of *REQ1*, *REQ2*, and *REQ3* is:

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 4.9: Sets, constants and axioms generated from the class and state machine diagrams.

Figure 4.2 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 *Button* class are defined as 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 4.2 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 4.3 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 *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 4.4 is translated in a similar manner to the state machine in Figure 4.3.

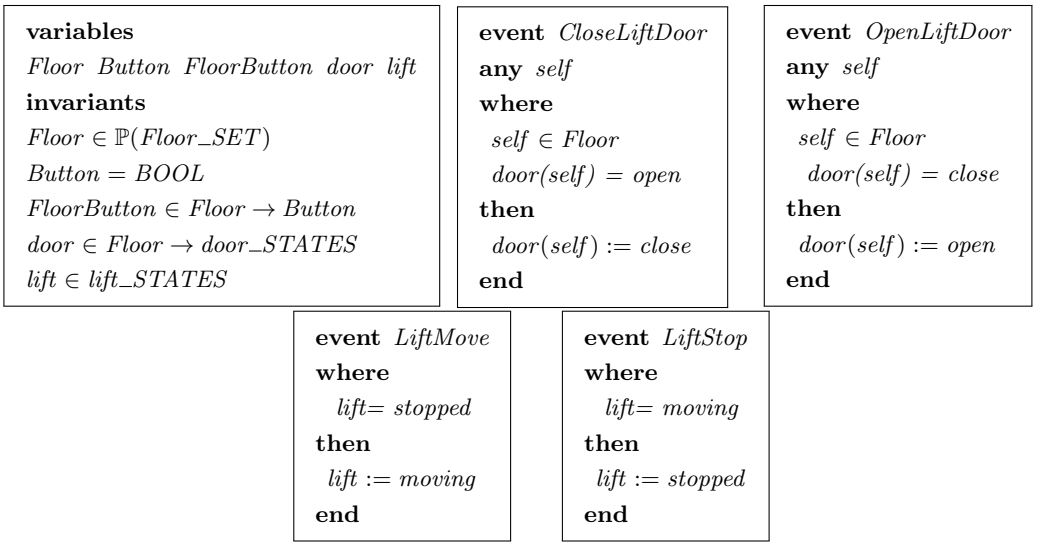


Figure 4.10: Variables, invariants and events generated from the class and state machine diagrams.

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

Requirement *REQ4*:

REQ4	The lift door of a moving lift must be closed.

The Event-B model of *REQ4*:

$$\forall f.f \in dom(door) \wedge lift = moving \Rightarrow door(f) = close$$

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

Requirement *REQ5*:

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

The Event-B model of *REQ5*:

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

Figure 4.11: 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 as follows.

Requirement *REQ8*:

REQ8	There might be more than one external floor request in a particular floor, the lift will respond to them (stop) only once.

The Event-B model of *REQ8*:

event *RequestFloor*
where
$LiftStop2 = FALSE$
end

event *LiftStop*2 **refines** *LiftStop*1
where
$LiftStop2 = FALSE$
then
$LiftStop2 = TRUE$
end

Figure 4.12: 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 number of 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 once or more number of times before the execution of the 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 4.7:

$$\boxed{LiftStop2 = LiftStop1}$$

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. In this step, we obtain two generated Event-B models: The first one is the Event-B model generated from the UML-B as well as the manual work resulting from converting the structured English representation into Event-B model, and the second one is the Event-B model generated from the AD. The generated Event-B models from AD diagrams and UML-B diagrams can be combined using shared-event composition [74]. Shared-event composition merges the variables and the invariants of each of Event-B machine. In each composed machine, events of Event-B model generated by UML-B are synchronised with events of the Event-B model generated by AD tool. For instance, event *openDoor* in the UML machine *m1* is synchronised with event *OpenLiftDoor* in the AD machine *m1*: $UML-B.m1.openDoor \parallel AD.m1.OpenLiftDoor$. Therefore, we obtained three contexts and three composed machines resulted from composing Event-B machines and contexts from the UML-B and the AD. The contexts:

$$Cxtcmp0 : UML-B.c0 \parallel AD.c0$$
$$Cxtcmp1 : UML-B.c1 \parallel AD.c1$$
$$Cxtcmp2 : UML-B.c2 \parallel AD.c2.$$

The composed machines:

$$Mcmp0 : UML-B.m0 \parallel AD.m0$$
$$Mcmp1 : UML-B.m1 \parallel AD.m1$$
$$Mcmp2 : UML-B.m2 \parallel AD.m2.$$

4.4 Related Work

This section presents different works in the area of requirements structuring, requirements traceability, requirements consistency, and requirements completeness.

There are several works regarding requirements structuring and requirements classification, we will present four of them. SOFL (Structured Object-Oriented Formal Language) [57] 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 the user and the developer. One of the main differences between our work and [57] is that our work makes use of the structured English and graphical notations represented in UML-B and the AD approach to bridge the gap between the requirements and Event-B models, whereas the semi-formal artifacts used in the SOFL approach are used to document the requirements.

Gandhi [38] proposes a method for visual requirements analytics framework to characterise its key components and relationships. The framework consist of five stages: User phase that represents a stakeholder using VA (Visual Analytic) approaches, methods, and tools to tackle RE (Requirement Engineering) tasks. Data phase that focuses on extracting the appropriate requirements information from raw date for further automatic and visual analysis using different methods such as stemming, stop word removal, and other data cleaning and normalisation procedures. Model phase that covers the definition of the entities and relationships of data. This phase can augment use cases, features, problem frames, and social networks of stakeholders. Visualisation phase that covers data analysis using requirement visualisation (a type of information visualisation). Finally, knowledge phase which is an iterative process that augments users knowledge discovery driven by the visual view of requirements and leads to decision making. The [38] platform helps different stakeholders such as requirements engineers, analysts, and decision makers to gain insights from the high volumes of data whereas our approach does not capture how the requirements were arrived at; it rather focuses on mapping requirements and Event-B models and enables the construction of traceable Event-B model from requirements.

Behaviour trees [30] are formal, graphical modelling language to represent natural requirements. Its originally developed by Dromey [30] and refined by University of Queensland, Griffith University research group. 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 our approach is that both approaches allow to express the system in detail and at an abstract level. The behaviour trees allow behaviour to be easily partitioned and separated out and the atomicity decomposition allows to visualise the system at different abstract levels. Behaviour trees and atomicity decomposition diagrams are composable, however, behaviour trees can expose behavioural defects whereas atomicity decomposition approach used to generate the flow and show relationships between abstract and refined events.

Yeganefard and Butler [88] described an approach for structuring the requirements of the 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 builds up on layering requirements by modelling one feature at a time in each refinement level where the developer chooses which feature to model at each refinement level. They suggest modelling the main role of the system with a minimum set of requirement in the very abstract model. The third step is based on revising the requirements document and the formal model to investigate any inconsistent, ambiguous or missing requirements. Comparing our work with [88], the approach used in [88] is specific to control systems whereas the approach presented in this chapter 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 better in deciding which feature to be modelled at every refinement level. It is possible to combine the strategy used in our approach and the approach of [88] to obtain more effective guidelines for developing traceable Event-B models for the control systems.

Requirements traceability is an important topic that have been studied a lot, in this section we present three works that deal with requirements traceability. Jastram [47] identifies a number of steps for mapping the requirements to B models. The first step is based on extracting a data structure (data types and events) from a set of requirements. They suggest the use of UML-B to map the data structure into Event-B models for complex system development. The second step is to map the safety requirements to invariants. The third step is to use LTL formulas to describe liveness properties. The fourth step is performed by structuring events. Finally, the last step aims to improve the model repeatedly and deal with changes in requirements. The main difference between our approach and Jastram's [47] is that besides the advantage of producing traceable Event-B models, our approach also supports the development of a refinement strategy through the use of AD diagrams. The two approaches can in fact complement each other: the use of LTL formulae in [47] can be augmented to the steps of our approach to obtain a more comprehensive approach.

Jastram et al [46, 45] presented another approach to achieve requirement traceability. They structured 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 the 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 the traceability between the 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 approach has tool support with Rodin called ProR [45]. The main difference between our approach and the approach of [46, 45] is that the latter focuses more on the traceability and uses intermediate constructs based on WRSPM to provide traceability between the requirements and Event-B models. On the other hand, the intermediate constructs which we use are based on the requirements classification derived from the Event-B components. In effect, the process of converting the semi-formal artifacts into an Event-B model are straightforward, as this is done using the UML-B and AD tools. Another difference is that, a requirement in [46, 45] work may be traced in a forward or backward direction whereas a requirement in our work is traced in forward direction. Moreover, our approach focuses not just on traceability but also on building Event-B models.

Chanda et al [19] propose a formal model for UML use case, activity and class diagrams. The work uses context free grammar to formally define UML diagrams. A set of verification criteria composed of traceability rules are defined. The defined traceability rules are based on the relationships among the diagrams. The traceability rules ensure that use case events map to activity events in the analysis phase and finally to class methods in the design phase. Comparing to [19] approach, our approach uses AD and UML diagrams as intermediate artefacts to facilitate traceability and no formal traceability rules are imposed in our approach.

Assuring consistency and completeness of requirements is important in the area of requirement engineering. Although, our guidelines do not deal with consistency and completeness of requirements, we aim to do more research later on this topic. There are several works cover consistency and completeness of requirements. In this section, we mention three of them. Boehm [13] identifies three fundamental characteristics for requirements completeness which are: (1) no information is left undefined, (2) all objects or entities of the information are defined, (3) no information is missing from the document. The first two characteristics are typically

referred to as internal completeness of the document. The third characteristic is referred to as external completeness of the document. External completeness is the process of ensuring that all of the information required for problem definition is found within the requirement specification [89]. Capturing the external completeness of the document is hard process [89]. There is however, some techniques that could assist on that such as domain modelling [89]. Requirement completeness is out of the scope of the thesis.

Zowghi et al [90] proposes a method to check consistency in natural language requirements. The method combines a simple engine for natural language parsing called Cico and the reasoning engine of CARET. Cico parses the natural language requirements and translate them into symbolic logic acceptable to CARET . The translated logical statements of requirements can be analysed for inconsistency using the reasoning engine of CARET. The algorithm for checking any inconsistency between requirements is based on well-known theories of classical logic, nonmonotonic reasoning and belief revision.

In [25], the authors present controlled language described by use cases scenarios and safety properties for analysing requirements consistency against constraints (safety properties) with B method. The controlled language is used to translate use case scenario sentences and some properties to B specifications. Following that, ProB is used to check the formal specification to find out inconsistencies such as missing a precondition constraint of an operation.

4.5 Conclusions

The first part of this chapter concentrated on categorising FreeRTOS requirements into two classes. The first class focuses on providing a set of requirements for each FreeRTOS component. Each set of requirements is specific to a particular FreeRTOS component and does not provide information about the interactions between the different components. The second class focuses on the composition requirements to show the interaction between different components and give complete prospective of the behaviour of FreeRTOS components. The categorisation of the requirements is not specific to FreeRTOS case study and it could be used as a preparation step for a compositional development in which the overall specification emerges from composing its sub modules.

The second part of this chapter introduced an approach which facilitates constructing Event-B models and provides a clear traceability between the requirements and the Event-B model. The approach is based on the use of the UML-B and AD tools. The 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 the 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 models. 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 modelled at 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 insight into the overall visualisation of the refinement structure and demonstrate the relationship between the events even before any model is written.

Most of the requirements were classifiable according to the classification scheme. Each requirement must be assigned to at least one class. It is also possible that a requirement fits several classes such as the requirement *REQ5*. The nouns "people", "floor" , "button" and "lift" are identified as data-oriented requirements, whereas the verb "request" denotes that *REQ5* is of event-oriented type. The intermediate semi-formal artifacts represented in UML-B, AD diagrams and structured English are applied easily for each of the requirements class. However, as described in the sub-section 4.3.3, more patterns are needed to cover several kinds of flow requirements. In addition, converting each semi-formal artefact to Event-B is straightforward. The UML-B and AD tools convert diagrams automatically to Event-B and we only need to focus on translating the structured English to Event-B model. The process of translating the structured English to Event-B model is an easy task. This is because all the variables and data-types and some events that are generated from the state machines diagrams are already generated from the UML-B tool. Therefore, it is easier to complete an existing model than starting the model from the beginning. Moreover, the combined AD diagrams assist in developing the refinement strategy, and therefore it is clear in which modelling level, the events need to be specified.

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

REQ9	The lift can make a 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 the future research. 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.

Chapter 5

The Application of the Staged Approach for Constructing Queue Management Model

This chapter describes the application of the staged approach presented in Chapter 4 to queue management case study. Throughout the presented case study, some useful conclusions were drawn and we suggest further evaluation of the approach through more case studies.

5.1 Introduction

Queues are mechanisms used to serve communication for a task-to-task or a task-to-interrupt service routine (ISR) [10]. Interrupt service routine is a software routine invoked in response to an interrupt (a signal to the CPU that requires immediate attention). In this chapter, we apply the staged approach presented in Chapter 4 to construct queue management Event-B model from the requirements given in Table 5.1. The application of a particular technique to a case study has the advantage of illustrating the use of the approach and validating its effectiveness. A number of lessons can be learnt from this process and some guidelines can be drawn to facilitate the application of the technique to a variety of applications.

This remaining of the chapter is organised as follows: Section 5.2 outlines the application of the staged approach to construct queue management Event-B models from the requirements. The conclusions are presented in Section 5.3.

5.2 The Application of the Staged Approach to Construct Queue Management Event-B Model from Requirements

The following sub-sections present the result of applying the staged approach to queue management case study through three steps.

5.2.1 Step1: Classify Requirements

Queue management requirements presented in Table 5.1 have been collected from different materails and from the FreeRTOS's source code [11, 10]. The requirements that are collected describe several functions such as create queues, send messages on queues, receive messages on queues and wait for messages. In order to classify the queue management requirements, we first need to classify the requirements based on data-oriented class, event-oriented class, and constraint-oriented class. Its possible that a single requirement classified based on more than one categorisation. After that, we classify the requirements based on flow-oriented requirements. The separation of flow requirements from other requirements is because the 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 the other requirement classes.

Table 5.1 categorises the queue management requirements based on data-oriented, event-oriented, and constraint-oriented classes.

Label	Requirements	Classification
TSK1	Tasks can be created and deleted.	Event-oriented and Data-oriented
QUE1	Queues can be created and deleted.	Event-oriented and Data-oriented
TSK2	Only one task is running at a time.	Constraint-oriented and Data-oriented
TSK3	Tasks are assigned priority when created.	Event-oriented and Data-oriented
QUE2	The length of the queue is identified when the queue is created.	Data-oriented and Event-oriented
QUE3	Task can only send item to a queue when there is enough room in the queue. Similarly, task can only receive an item from a queue when the queue is not empty.	Event-oriented and Data-oriented
QUE4	A queue contains a limited number of items	Constraint-oriented and Data-oriented.
QUE5	Each queue has two collections of waiting tasks: tasks waiting to send and tasks waiting to receive.	Data-oriented
QUE6	Tasks fail to send an item to a queue because the queue is full are placed into the collection of tasks waiting to send. Similarly, tasks fail 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
QUE7	Every task is mapped at most to one collections of waiting tasks.	Constraint-oriented and Data-oriented
QUE8	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.	Event-oriented and Data-oriented
QUE9	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.	Event-oriented and Data-oriented
QUE10	The queue must be locked when the running task failed to send or receive an item to a queue.	Event-oriented and Data-oriented
QUE11	All tasks that wait to receive an item from a locked queue are removed and and vice versa and the queue is unlocked.	Event-oriented and Data-oriented
QUE12	When all tasks that wait to receive/send an item from/to a locked queue are unblocked, the queue is unlocked.	Event-oriented and Data-oriented

Table 5.1: Data-oriented, Event-oriented, and Constraint-oriented Requirements classification.

Table 5.2 categorises the queue management requirements based on flow-oriented requirements

Label	Requirements	Classification
QUE3	Task can only send an item to a queue when there is enough room in the queue. Similarly, task can only receive an item from a queue when the queue is not empty.	Flow1
QUE6	Tasks fail to send an item to a queue because the queue is full are placed into the collection of tasks waiting to send. Similarly, tasks fail to receive an item from a queue because the queue is empty are placed into the collection of tasks waiting to receive.	
QUE3	Task can only send an item to a queue when there is enough room in the queue. Similarly, task can only receive an item from a queue when the queue is not empty.	Flow2
QUE8	When a queue becomes available (there is an item in the queue to be received), then the highest priority task waiting for an item to arrive on that queue (if any) will be removed from the collection of tasks waiting to receive.	
QUE10	The queue must be locked when the running task fails to send or receive an item to a queue.	Flow3
QUE11	All tasks that wait to receive an item from a locked queue are removed and and vice versa.	
QUE12	When all tasks that wait to receive/send an item from/to a locked queue are unblocked, the queue is unlocked.	

Table 5.2: Flow Oriented Requirements Classification.

In Table 5.2, 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 is revealed 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 must be locked when a task has failed to send or receive an item to a queue. The running task will then be added to the collection of waiting-tasks because it is blocked. Following that, the queue must be unlocked.

5.2.2 Step2: Construct Semi-Formal Artifacts and Develop Refinement Strategy

5.2.2.1 Stage1: Use Semi-Formal Artifacts (UML-B, AD, and Structured English)

TSK1

TSK1	Tasks can be created and deleted.	Event-oriented and Data-oriented.

The verbs *"created"* and *"deleted"* identify that *TSK1* requirement is of type event-oriented requirement, therefore, we represent it using structured English. The noun *Tasks* identifies that *TSK1* is a data-oriented requirement. Therefore, the tasks are represented using the class diagram as shown in Figure 5.1.

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

event : *DeleteTask*
action : *delete an existing task*

QUE1

QUE1	Queues can be created and deleted.	Event-oriented and Data-oriented.

Like the requirement *TSK1*, *QUE1* requirement is classified as an event-oriented requirement and data-oriented requirement, therefore, we represent it using structured English and the class diagram shown in Figure 5.3.

event : *CreateQueue*
action : *new queue is added to the pool of queues*

event : *DeleteQueue*
action : *delete an existing queue*

TSK2

QUE1	Only one task is running at a time.	Constraint-oriented and Data-oriented.

QUE1 requirement is a type of constraint-oriented and data-oriented requirement, therefore, we represent it using structured English. *Running* state identifies a possible state of a task, therefore we represent it using the class diagram shown in Figure 5.1.

Invariant : $<$ *Only one task is running at a time* $>\longrightarrow<$ *QUE*1 $>$

Figure 5.1: The class diagram for QUE1.

TSK3

TSK3	Tasks are assigned priority when created.	Event-oriented and Data-oriented.

The verb *"assigned"* denotes that *TSK3* requirement is of type event-oriented. Therefore, we represent it using structured English. The noun *"priority"* is identified as data-oriented requirement and is represented as an attribute in the class diagram shown in Figure 5.2.

event : *CreateTask*
action : *priority of a new task is set*

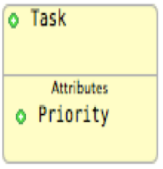


Figure 5.2: The class diagram for TSK3.

QUE2

QUE2	The length of the queue is identified when the queue is created.	Data-oriented and Event-oriented

The verb *"identified"* denotes that *QUE2* requirement is of type event-oriented. Therefore, we represent it in structured English. The noun *"length"* is identified as a data-oriented requirement and is represented as an attribute of queue class diagram.

event : *CreateQueue*
action : *The length of the queue is identified*

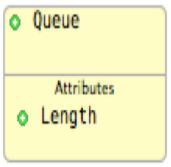


Figure 5.3: The class diagram for QUE2.

QUE3

QUE3	Task can only send an item to a queue when there is enough room in the queue. Similarly, task can only receive an item from a queue when the queue is not empty.	Event-oriented and Data-oriented

The verbs *"send"* and *"receive"* are identified as event-oriented requirements. Therefore, *QUE3* is represented as events using structured English representation. *"Queue item"* and *"task item"* identify the relationships between the nouns *"task"* and *"Queue"*, therefore, we represent them using the UML-B associations.

event : *TaskQueueSend*
guard : *there is enough room in the queue*
action : *Task can send item to a queue*

event : *TaskQueueReceive*
guard : *the queue is not empty*
action : *Task can receive item from a queue*

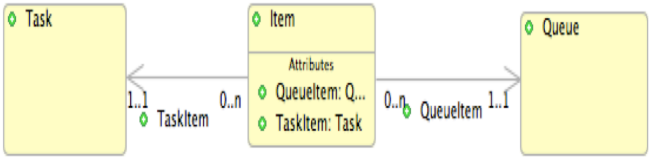


Figure 5.4: The class diagrams for QUE3.

QUE4

QUE4	A queue contains a limited number of items	Constraint-oriented and Data-oriented

QUE4 requirement is a type of constraint-oriented requirement, therefore, we represent it in the following structured English. *"Queue items"* is identified already as

a data-oriented requirement and is represented as an attribute in the class diagram of Figure 5.4.

Invariant : $< A \ queue \ contains \ a \ limited \ number \ of \ items > \longrightarrow < QUE4 >$

QUE5

QUE5	Each queue has two collections of waiting tasks: tasks waiting to send and tasks waiting to receive.	Data-oriented

The attributes *"tasks waiting to send"* and *"tasks waiting to receive"* are identified as a data-oriented requirement. Therefore, we represent the requirement *QUE5* using the following class diagram.

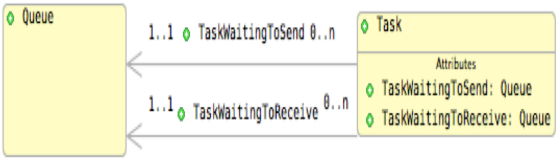


Figure 5.5: The class diagrams for QUE5.

QUE6

QUE6	Tasks fail to send an item to a queue because the queue is full are placed into the collection of tasks waiting to be sent. Similarly, tasks fail to receive an item from a queue because the queue is empty are placed into the collection of tasks waiting to be received.	Event-oriented and Data-oriented

The requirement *QUE6* is identified as an event-oriented requirement as it represents the action of placing tasks that have failed to send/receive to the collections of waiting-tasks. Therefore, *QUE6* is represented using structured English form. The nouns *"tasks"*, *"queue"*, *"item"*, *"task waiting to send"*, and *"task waiting to receive"* classify *QUE6* as a data-oriented requirement. The class diagrams that represent these nouns are already given in Figure 5.4 and Figure 5.5.

event : *PlaceOnTaskWaitingToSend* **guard** : *the queue is full* **action** : *stores the task into the collection of tasks waiting to send an item to the queue*	**event** : *PlaceOnTaskWaitingToReceive* **guard** : *the queue is empty* **action** : *stores the task into the collection of tasks waiting to receive an item from the queue*

QUE7

QUE7	Every task is mapped at most to one of the collections of the waiting tasks.	Constraint-oriented and Data-oriented

The requirement *QUE7* describes property about the task and is represented via structured English. The collection of *waiting tasks* classifies *QUE7* requirement as a data-oriented requirement and is already captured using the class diagram given in Figure 5.5.

Invariant : $<$ *Every task is mapped at most to one of the collections of the waiting tasks* $>$ $\longrightarrow <$ *QUE7* $>$

QUE8

QUE8	When a queue becomes available (there is an item in the queue to be received) then the highest priority task waiting for an item to arrive on that queue (if any) will be removed from the collection of tasks waiting to receive.	Event-oriented and data-oriented

The requirement *QUE8* is identified as an event-oriented requirement as it represents the action of removing tasks from the collection of tasks waiting to receive when the queue becomes ready (there is an item in the queue to be received). Therefore, *QUE8* is represented in structured English format. The nouns *"tasks"*, *"queue"*, *"item"*, *"task waiting to receive"*, and *"priority"* classify *QUE8* as a data-oriented requirements. The class diagrams that represent these nouns are already given in Figures 5.2, 5.4 and 5.5.

event : *RemoveFromTaskWaitingToReceive*
guard : *there is an item in the queue to be received*
action : *the highest priority task waiting for item to arrive on that queue will be removed from the collection of tasks waiting to receive item from that queue*

QUE9

QUE9	When a queue becomes available (i.e. when there is a 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.	Event-oriented and Data-oriented

The requirement *QUE9* is identified as an event-oriented requirement as it represents the action of removing tasks from the collection of tasks waiting to send when the queue becomes ready (i.e. there is a room in the queue). Therefore, *QUE9* is represented using structured English format. The nouns *"tasks"*, *"queue"*, *"item"*, *"priority"*, and *"task waiting to send"* classify *QUE9* as a data-oriented requirement. The class diagrams that represent these nouns are already given in Figures 5.2, 5.4 and 5.5.

event : *RemoveFromTaskWaitingToSend*
guard : *there is room in the queue*
action : *the highest priority task waiting for an item to arrive on that queue will be removed from the collection of tasks waiting to send an item to that queue*

QUE10

QUE10	The queue must be locked when the running task fails to send or receive an item from a queue	Event-oriented and Data-oriented

The requirement *QUE10* is identified as an event-oriented requirement as it represents the action of locking queue. Therefore, we represent it via structured English. It also shows that queue can be in a locked state and therefore is a data-oriented requirement represented through the class diagram. The class diagrams for the nouns *"queue"*, *"task"* are already captured in Figure 5.4.

event : *LockQueue*
guard : *the QueueFlag is False*
action : *the QueueFlag is True*

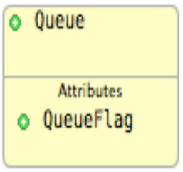


Figure 5.6: The class diagram for QUE10.

QUE11

QUE11	All tasks that wait to receive an item from a locked queue are removed and and vice versa.	Event-oriented and Data-oriented

The requirement *QUE11* is identified as an event-oriented requirement as it represents the action of removing tasks from a locked queue. Thus, we represent it

using structured English. The class diagrams for the nouns *"queue"*, *"task"*, and the collection of waiting tasks are already captured in Figure 5.4, and Figure 5.5.

event : *RemoveTaskFromTaskWaitingToSend2* **guard** : *queue q is locked* **action** : *remove tasks from the collection of task waiting to send*	**event** : *RemoveTaskFromTaskWaitingToReceive2* **guard** : *queue q is locked* **action** : *remove all tasks from the collection of task waiting to receive*

QUE12

QUE12	When all tasks that wait to receive/send an item from/to a locked queue are unblocked, the queue is unlocked.	Event-oriented and Data-oriented

The requirement *QUE12* is identified as an event-oriented requirement as it represents the action of un-locking the queue. Therefore, we represent it in structured English. The nouns *"queue"*, *"task"*, the collection of waiting tasks and the unlocked state of a queue are all data-oriented requirements as captured in Figures 5.4, 5.5 and 5.6.

event : *UnLockQueue* **guard** : *the QueueFlag is True* **action** : *the QueueFlag is False*

Flow1

QUE3	Task can only send an item to a queue when there is enough room in the queue. Similarly, task can only receive an item from a queue when the queue is not empty
QUE6	Tasks fail to send an item to a queue when the queue is full are placed into the collection of tasks waiting to send. Similarly, tasks fail to receive an item from a queue when the queue is empty are placed into the collection of tasks waiting to receive.

In this case study, we are interested to represent flows as it described in FreeRTOS. FreeRTOS combines several functions and uses different structures such as "if-then-else" or loops to manage the order of execution of these functions. The requirements *QUE3* and*QUE6* describes the "if-then else" structure. A task can successfully send/receive an item to/from a queue if the queue is ready or otherwise the task is placed into the collections of waiting tasks. We identified *QUE3* and*QUE6* requirements as flow requirements and therefore represent the flow using the following AD diagram. The "xor" pattern is used to represent "if-then-else" structure.

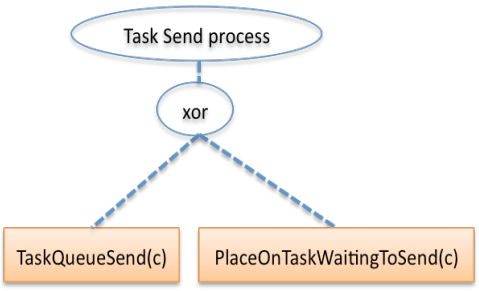


Figure 5.7: ADD for Flow1.

Flow2

QUE3	Task can only send an item to a queue when there is enough room in the queue. Similarly, task can only receive an item from a queue when the queue is not empty
QUE8	When a queue becomes available (i.e. there is an item in the queue to be received), the highest priority task waiting for the item to arrive on that queue (if any) will be removed from the collection of tasks waiting to receive.

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

We identified *QUE3* and*QUE8* requirements as flow requirements and therefore represent the flow using the following AD diagram. The "sequence" pattern is used to represent the sequence structure. The "xor" pattern is used 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.

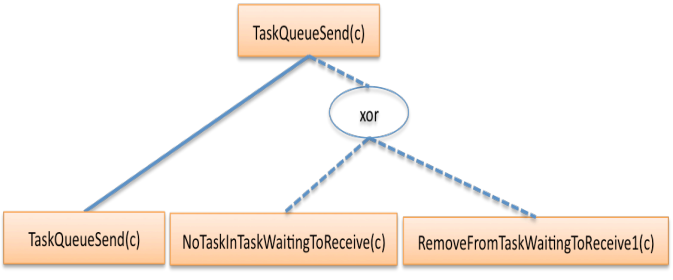


Figure 5.8: ADD for Flow2.

Flow3

QUE10	The queue must be locked when the running task fails to send or receive an item in a queue.
QUE11	The queue should be unlocked when the running task has been added to the collection of waiting tasks.
QUE12	When all tasks that wait to receive/send an item from/to a locked queue are unblocked, the queue is unlocked.

The requirements *QUE10,11* and *QUE12* show sequence ordering between the following events when the current task failes to send an item to queue: *LockQueue*, *PlaceOnTaskWaitingToSend*, *RemoveFromTaskWaitingToReceive*, and *UnLockQueue*. It also show a sequence ordering between the following events when the current task failes to receive an item from a queue: *LockQueue*, *PlaceOnTaskWaiting-ToReceive*, *RemoveFromTaskWaitingToSend*, and *UnLockQueue*. In this chapter, we consider the first case that shows the sequence ordering between events occur when the current task failes to send an item to queue as shown in Figure 5.9.

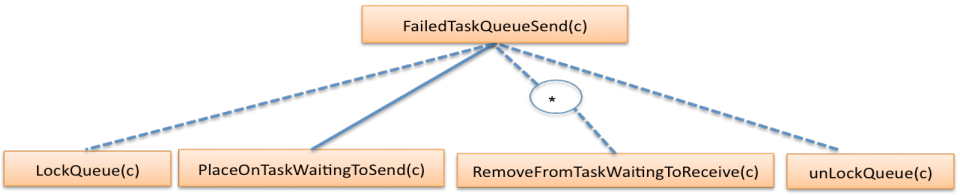


Figure 5.9: ADD for Flow3.

Stage2: 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.

TSK1, TSK3

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

QUE1, QUE2

event : *CreateQueue*
action : *new queue is added to the pool of queues*
action : *The length of the queue is identified*

5.2.2.2 Stage3: Develop Refinement Strategy

In this step, we combine AD diagrams and develop the refinement strategy. 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 RTOS level including adding tasks that failed to send item to queue to the collection of tasks waiting to send and queues locking. The abstract model includes *TaskQueueSend* event and *FailedTaskQueueSend* event (an abstract event of *PlaceOnTaskWaitingToSend* event). The introduction of *FailedTaskQueueSend* event in the most abstract level reduces the complexity and allow us to defer the introduction of *PlaceOnTaskWaitingToSend* event, *RemoveFromTaskWaitingToSend* event, *PlaceOnTaskWaitingToReceive* event and *RemoveFromTaskWaitingToReceive* to the first refinement level, where the atomicity of the *FailedTaskQueueSend* event is broken down into the first refinement level as *PlaceOnTaskWaitingToSend* event, *RemoveFromTaskWaitingToSend* event and *RemoveFromTaskWaitingToReceive* event.

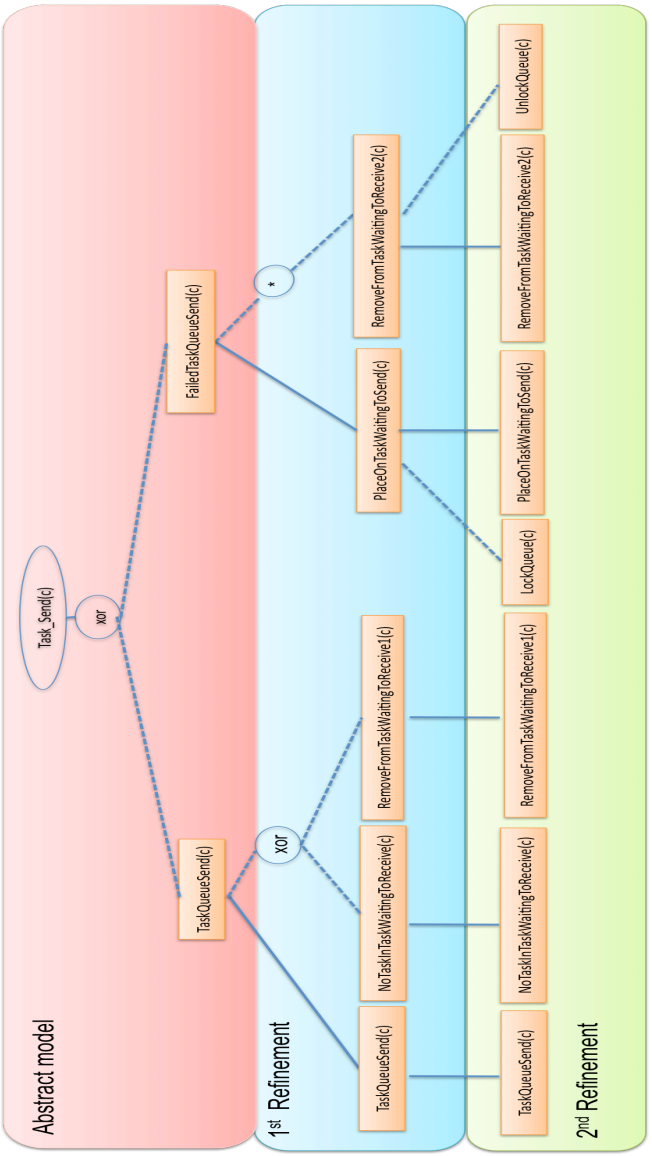


Figure 5.10: The combined ADD for task-send.

The most abstract level in the Figure 5.10 demonstrates task-send process. The tree with the root *Task_Send* corresponds to the AD diagram shown in Figure 5.7. The behaviour of the *Task_Send* process is exhibited 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. In the first refinement level, the atomicity of *TaskQueueSend* which corresponds to the AD diagram shown in Figure 5.8 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 *NoTaskIn-TaskWaitingToReceieve* event or *RemoveFromTaskWaitingToReceive* event. Similarly, the abstract *FailedTaskQueueSend* event, which corresponds partially to the

AD diagram shown in Figure 5.9, 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 class diagrams that correspond to the refinement structure of the combined AD diagrams in Figure 5.10 are:

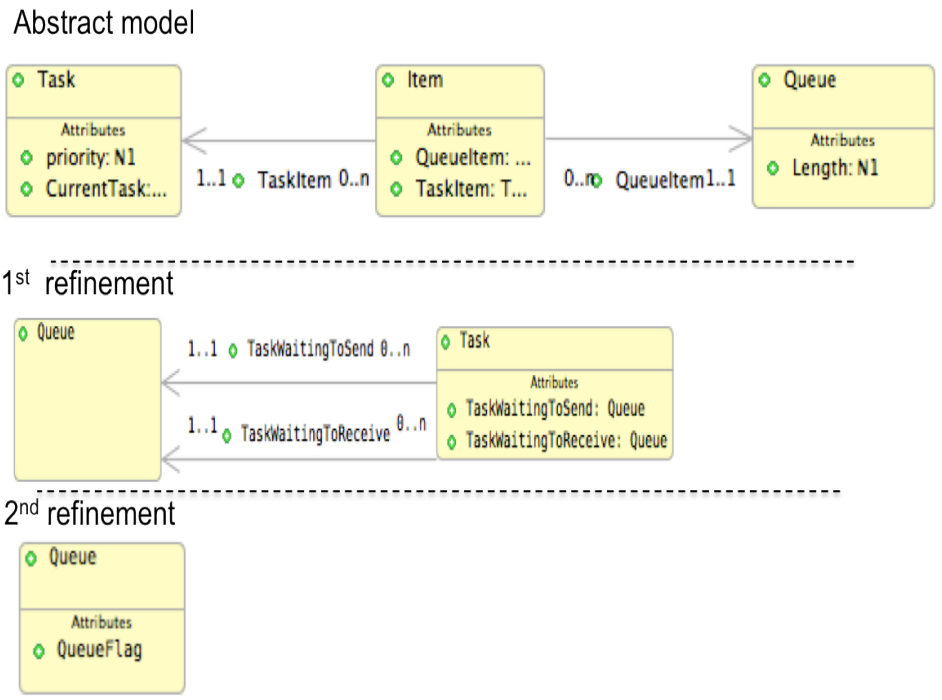


Figure 5.11: Queue management class diagrams.

The UML-B class diagrams are layered through the refinement based on the refinement strategy decided based on the combined AD diagrams in Figure 5.10. 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.

5.2.3 Step3: Construct Formal Models

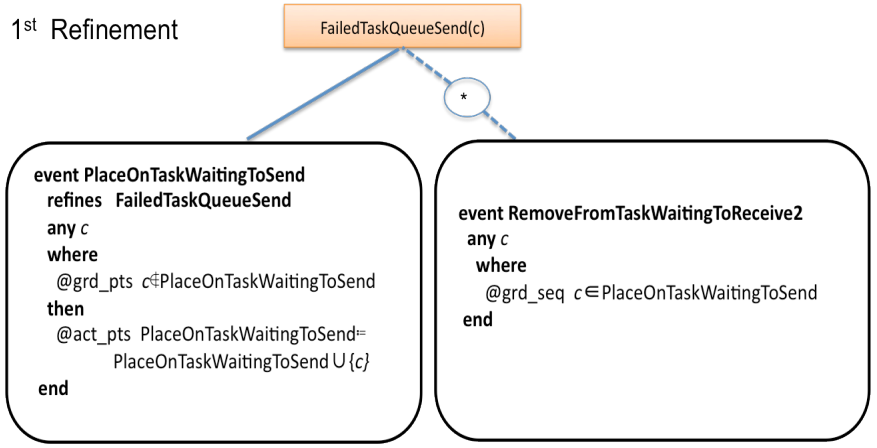
In stage 3 of the subsection 5.2.2, we showed that refinement strategy are organised based on the combined AD diagrams and the hierarchy of class diagrams are structured according to the combined AD diagrams. Therefore, we obtained three machines in the combined AD diagrams and three machines of the class diagrams.

In this sub-section, we will show some parts of the generated Event-B models from the class diagrams and AD diagrams.

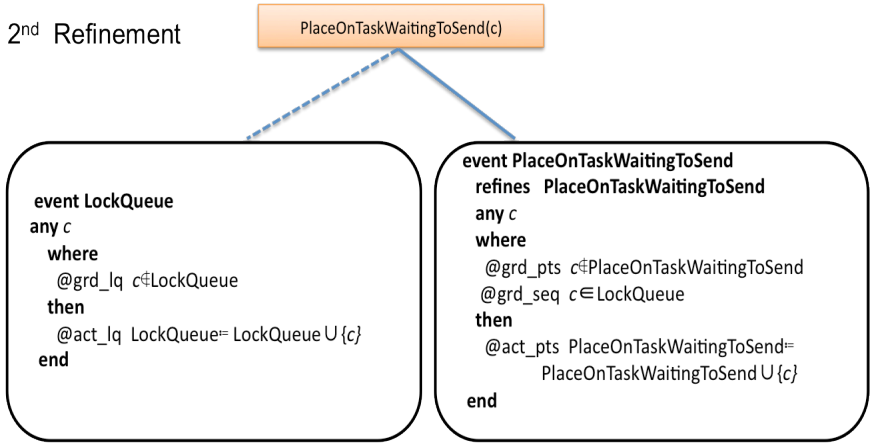
The Event-B model corresponds to the class diagrams shown in Figure 5.5:

machine $M1$
refines $M0$
sees $Cntxt$
variables
$Task$ $Queue$ $TaskWaitingToSend$ $TaskWaitingToReceive$
invariants
$TaskWaitingToSend.type$ $TaskWaitingToSend \in Queue \rightarrow Task$
$TaskWaitingToReceive.type$ $TaskWaitingToReceive \in Queue \rightarrow Task$
events $INITIALISATION$
then
$TaskWaitingToSend.init$ $TaskWaitingToSend := \varnothing$
$TaskWaitingToReceive.init$ $TaskWaitingToReceive := \varnothing$
end
end

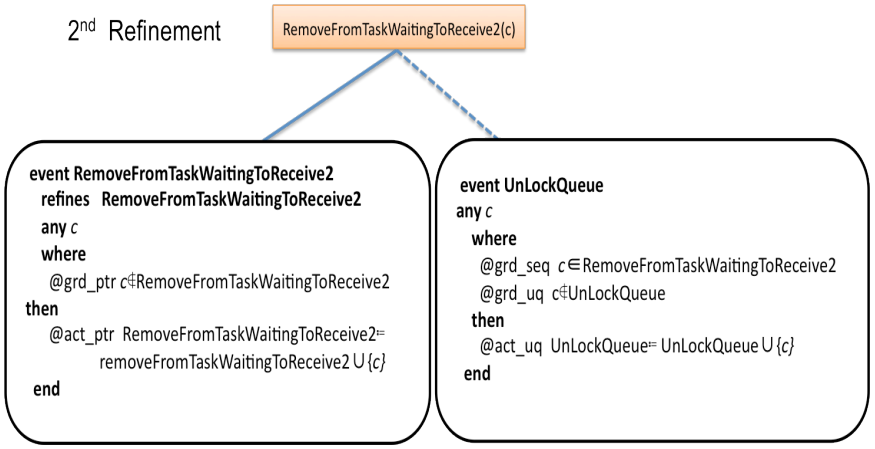
The Event-B model generated by AD tool that corresponds to ***Flow3*** is:



The guard $c \in PlaceOnTaskWaitingToSend$ in $RemoveFromWaitingToReceive2$ event assures that $RemoveFromWaitingToReceive2$ event is executed after $PlaceOnTaskWaitingToSend$. The solid line indicates that $PlaceOnTaskWaitingToSend$ event refines $FaildTaskWaitingToSend$ event.



The sequence pattern shown here indicates that *PlaceOnTaskWaitingToSend* is the refined event that executed after *LockQueue* event.



RemoveFromTaskWaitingToReceive2 is a refined event that get executed before *UnlockQueue* event.

The Event-B model corresponds to the structured English of *PlaceOnTaskWaitingToSend* is:

event *PlaceOnTaskWaitingToSend*
refines *FailedTaskQueueSend*
any $c\ q\ i$
where
$grd1\ \ c \in CurrentTask$
$grd2\ \ q \in Queue$
$grd3\ \ i \in TaskItem^{-1}[\{c\}]$
$grd4\ \ Length(q) = card(QueueItem^{-1}[\{q\}])$
then
$act1\ TaskWaitingToSend := TaskWaitingToSend \cup \{q \mapsto c\}$
end

The complete generated Event-B models are given in appendix B.

Since we have three Event-B machines generated from the AD diagrams and also three Event-B machines generated from UML-B diagrams. We need to combine

them to obtain one Event-B model consists of three machines that integrate Event-B machines of the AD diagrams and Event-B machines of the UML-B diagrams. Shared Event composition tool is used to integrate Event-B models generated from UML-B diagrams and Event-B models generated from AD diagrams. The Event-B components of the structured English is added manually to the composed machines. More discussions about the integration steps of UML-B, AD and refinement are given in Chapter 4.

5.3 Conclusions

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*, *QUE2*, and *QUE7*. We think it is useful to define clearly data-oriented requirement and separate them from event/constraint-oriented requirements. We can consider that data-oriented requirement always only describe that attributes of the system, constraint-oriented requirements describe properties about the system and event-oriented requirements describe the activities of the system. We ignore all the nouns and attributes mentioned in the constraint/event -oriented requirements. Therefore, the requirement *TSK3* can be restructured as follows:

TSK3-1	Each task has a priority associated with it.
TSK3-2	Tasks are assigned priority when created.

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 5.2. Thirdly, 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. Finally, 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 considered to evaluate the presented approach and to explore more useful patterns for managing the flows.

Chapter 6

Linking Composite Requirements with Composed Model

We adopted a compositional strategy and structured the requirement document into two main categories: The first category covers the requirements for each component (task, queue, memory), and the second category covers the composition requirements that link the separated components. As a result, we ended up with three sub-models which are task management, queue management, and memory management specified separately. Each sub-model defines a particular idealised projection of the behaviour of the whole system. The separated sub-models must be composed into a whole. The sub-model interactions must be verified to comply with the desired behaviour semantics of the system. The interaction usually occurs by means of a shared state, shared event, or a combination of both. The compositional strategy we adopted for building FreeRTOS models has confirmed our decision on using the shared event composition approach to show the sub-model interactions. In the shared event composition approach, sub-models interact via synchronisation over shared events and shared variables are not allowed. In Chapter 4, each composition requirement is split into sub-requirements, each of which belongs only to a particular component. The sub-requirements are specified as events in the sub-models as they are event-oriented requirements. We did not introduce shared variables [40] in the specified sub-models; rather used carrier sets as a means of common entity to provide links between the sub-models and facilitate applying shared event composition approach. In this chapter, we use composition requirements to select the events to be composed and apply composition using shared event composition approach.

6.1 Introduction

We attempt to make use of a compositional strategy to support the management of modelling complex systems in Event-B. When the size of the model is increased, the refinement process becomes difficult and the number of POs are increased. One solution to manage the complexity of big models is to use decomposition approaches to break down the models into sub-models that are easy to manage. However, decomposition approaches in Event-B have some limitations; for instance, shared-variables can not be refined in the shared-variable decomposition approach. Moreover, with decomposition approaches, it is essential to have an abstract model with global properties and shared elements before applying the decomposition. As a consequence, the sub-models will somehow be linked together through shared elements and shared properties which makes reusability of a particular sub-model (component) with other similar systems difficult.

An alternative approach to manage the complexity of big systems is to use a compositional strategy to model the system. The compositional strategy encourages building separate components without the need of having an abstract model with global properties. Each separate model deals only with a particular component and has its own set of requirements which has the advantage of producing loosely-coupled sub-models. The sub-models of FreeRTOS case study were built using the compositional strategy. We did not include shared-variables rather used only shared entities (carrier sets). The global properties are postponed until the composition level.

This chapter is organised as follows: Section 6.2 outlines the task and memory models. Section 6.3 describes the approach of using the shared event composition to compose sub-models with shared entities. It also outlines the application of this approach to a small example and compares it with the shared event composition approach that composes sub-models with shared variables. Section 6.4 outlines the composition of FreeRTOS specifications. Conclusions are given in Section 6.5.

6.2 Task and Memory Event-B Models

Before embarking into the composition task, it is necessary to introduce the formal models of task and memory to give an insight into the models to be composed. This section outlines the task and memory formal models. The queue formal models are given in appendix B.

6.2.1 Task Management

The task management model includes two parts: The context part which defines carrier sets, constants and axioms, and the machine part, which defines variables,

invariants and events. The task model contains two contexts *c0* and *c1*. Part of the context *c0* is given below:

CONTEXT c0
SETS
SCHEDULER_STATE, OBJECT, ...
CONSTANTS
TASKS, INTERRUPTS, PRIORITY, NOT_STARTED, RUNNING, SUSPENDED, ...
AXIOMS
axm1 : finite(OBJECT)
axm2 : partition(SCHEDULER_STATE, {NOT_STARTED}, {RUNNING}, {SUSPENDED})
axm3 : PRIORITY $\in \mathbb{N}1$
axm4 : TASKS $\subseteq$ OBJECT
axm5 : INTERRUPTS $\subseteq$ OBJECT
axm6 : TASKS $\cap$ INTERRUPTS $= \phi$
⋮
END

OBJECT is a carrier set that represents tasks and interrupts. *SCHEDULER_STATE* represents the scheduler, where the constants *NOT_STARTED*, *RUNNING*, and *SUSPENDED* represent the scheduler states. *Axm6* shows that interrupts and tasks are disjoint objects.

The context *c1* defines some constants which represent the configuration items of the configuration file of FreeRTOS. There are several items that identify some features of FreeRTOS and these features are present only if their items have been configured. For instance updating the priority of a task feature is only available when *INCLUDE_PrioritySet* is set to TRUE. Part of the context *c1* is given below:

CONTEXT c1
EXTENDS c0
CONSTANTS
INCLUDE_PrioritySet , INCLUDE_PriorityGet, INCLUDE_TaskDelete, ...
AXIOMS
axm1 : INCLUDE_PrioritySet = TRUE
axm2 : INCLUDE_PriorityGet = TRUE
axm3 : INCLUDE_TaskDelete = TRUE
⋮
END

6.2.1.1 An Abstract Specification-Some Basic Functionality of Task Management and the Kernel

In this abstraction, we begin with an abstract model of task management focusing on task creation, task deletion, interrupt handling and context switch.

We identified five machine variables: *AllTask* represents the set of all created tasks. *idle* task is a special task used to run when there is no task is available to run. *CurrentTask* represents the current running task. *Interrupts* represents the set of interrupts. *ISR* represents the interrupt handling routines. *CurrentInterrupt* represents the current running interrupt. The invariants of the abstract specification are:

inv1 : $AllTask \subseteq TASKS \cup \{idle\}$
inv2 : $CurrentTask \subseteq AllTask$
inv3 : $card(CurrentTask) \leq 1$
inv4 : $Interrupts \subseteq INTERRUPT$
inv5 : $ISR \in Interrupts \rightarrow INTERRUPT\_HANDLER$
inv6 : $CurrentISR \subseteq ran(ISR)$
inv7 : $card(CurrentISR) \leq 1$

Since FreeRTOS is a single core kernel, *inv3,7* are added to indicate that *CurrentTask* and *CurrentISR* are singleton sets.

In addition, we introduced seven simple machine events: *CreateTask* event creates a new task, *CreateIdleTask* event creates the *idle* task, *ContextSwitch* event switches between tasks, *HandleInterrupt* event processes interrupt requests, *DeleteRunningTask* deletes the running task, *FinishInterrupt* deletes any interrupt that has been handled, and finally *Delete* deletes all created tasks and interrupts.

The specification of *ContextSwitch* and *HandleInterrupt* events are given below:

ContextSwitch
any t
where
$grd1$ $t \in AllTask \setminus CurrentTask$
$grd2$ $CurrentISR = \phi$
then
$act1$ $CurrentTask := \{t\}$
end

HandleInterrupt
any t
where
$grd1$ $t \in Interrupts$
then
$act1$ $CurrentISR := \{ISR(t)\}$
end

6.2.1.2 First Refinement- Scheduler States

The machine specifies the scheduler states. We introduced a machine variable namely *Scheduler* to represent the current state of the scheduler ($Scheduler \in SCHEDULER\_STATE$).

Several events were introduced at this level; *StartScheduler* event extends *CreateIdleTask* to ensure that there is always at least one task that is able to run when the scheduler is running, *TaskEndScheduler* event extends *Delete* event to ensure that all the tasks and interrupts are deleted when the scheduler is not started. *SuspendScheduler* and *ResumeAll* update the *Scheduler* status.

The specification of *SuspendScheduler* and *ResumeAll* events are given below:

SuspendScheduler
where
$grd1$ $Scheduler \neq NOT\_STARTED$
then
$act1$ $Scheduler := SUSPENDED$
end

ResumeAll
where
$grd1$ $Scheduler = SUSPENDED$
then
$act1$ $Scheduler = RUNNING$
end

6.2.1.3 Second Refinement- Task States

This machine specifies the task states. *ReadyTask, DelayTask, SuspendTask* are the set variables for readied, blocked, and suspended tasks respectively. *TaskWaitingTermination* represents tasks that have been deleted, *PendingTask* represents the readied tasks while the scheduler is suspended. They are subsets of the variable $AllTask \subseteq OBJECT$.

In addition, seven events are introduced at this level. *AddToReady* event adds a task to a ready set, *AddToPending* event adds delay tasks that have been readied while the scheduler is suspended to pending ready set, *AddToDelay* event adds a task to delay set, *AddToSuspend* event adds a task to suspend set, *SuspendRunningTask* adds the running task to the suspend set, *DeleteRunningTask* event deletes the running task, and *ResumeTask* event adds a suspended task to ready set.

The specification of *AddToSuspend* event is as follows:

AddToSuspend
any t
where
$grd1$ $t \in (ReadyTask \setminus \{idle\}) \cup DelayTask$
$grd2$ $INCLUDE\_TaskSuspend = TRUE$
$grd3$ $t \notin CurrentTask$
$grd4$ $Scheduler \neq NOT\_STARTED$
then
$act1$ $SuspendTask := SuspendTask \cup \{t\}$
$act2$ $ReadyTask := ReadyTask \setminus \{t\}$
$act3$ $DelayTask := DelayTask \setminus \{t\}$
end

Grd2 ensures that the constant *INCLUDE_TaskSuspend* is set to TRUE to indicate the availability of suspension feature to the user. In addition, *ResumeAll* event is refined by adding the action to process pending tasks. $ReadyTask := ReadyTask \cup \{t \mid t \in PendingReadyTask\}$ and the action $PendingReadyTask := \phi$. Finally, the *StartScheduler* event is extended by adding the *idle* task into the collection of ready tasks.

6.2.1.4 Third Refinement- Hardware Clock and Timing Properties

This refinement level specifies the hardware clock and timing properties associated with a delay task such as the sleep-time ($SleepTask \in AllTask \nrightarrow SLEEP$) that represents the amount of time the task should be delayed for and the wake-up time ($TimeToWake \in DelayTask \nrightarrow \mathbb{N}$) represents the time at which the task should be woken up. *SLEEP* is an integer set constant defind in the context *c0*. Any task can sleep for certain ticks and placed into the collection of delay tasks based on its wake-up time. The time at which the task should be woken is calculated by adding the delay time of a task to the current time pointed out by the clock time ($Tick \in \mathbb{N}$).

A machine event namely *IncrementTick* is introduced to represent the clock interrupt by incrementing the *Tick* counter. The *IncrementTick* event is specified as follows:

IncrementTick
any t
where
$grd1$ $Tick \geq 0$
$grd2$ $Scheduler = RUNNING$
$grd3$ $TimeToWake \neq \varnothing \Rightarrow min(ran(TimeToWake)) > Tick$
then
$act1$ $Tick := Tick + configTICK\_RATE\_HZ$
end

Grd3 is added to ensure that all the tasks with expired timeouts are already woken. The predicate: $min(ran(TimeToWake)) > Tick$ assures that the minimum wake time of delayed tasks is greater than the current tick time, meaning that no task has an expired timeout. $configTICK\_RATE\_HZ$ is a constant that identifies the length of the time slice of the timer. In addition, $AddToDelay$ event is extended by calculating the time at which the task should be woken up by the following action: $TimeToWake(c) := SleepTask(c) + Tick$, where $SleepTask$ identifies the delay time of a task and $Tick$ identifies the current time pointed out by the timer. The $AddToReady$ event is also extended by adding the guard $TimeToWake(t) \leq Tick$ to ensure that the woken time of the delayed task is expired so it can be added to the collection of readied tasks. There are some existing approaches that model timing properties. Sarshogh work [68] focuses on modelling discrete timing properties in Event-B including deadline, delay, and expiry through several levels of refinement. The use of timing patterns introduced by Sarshogh is out of the scope of the thesis. Further work in the future is necessary to add more timing constraints on the refined levels of the FreeRTOS model.

6.2.1.5 Fourth Refinement- Delay Operations

In this refinement, the $AddToDelay$ event is refined into two events: The first event $TaskDelay$ specifies the delay operation and the second event $TaskDelayUntil$ specifies the delay-until operation. Delay until operation delays a task until a specific time has passed since the last execution of that operation. That way "delay until" allows a frequent execution of a task making it suitable for periodic tasks (arriving at fixed frequency). Delay operation, on the other hand, does not care about when the task last left the blocked state.

The $TaskDelay$ event has the additional guard $INCLUDE\_TaskDelay = TRUE$ to ensure that delay operation is set to true so that it is available to the user. The $TaskDelayUntil$ event has an extra guard and action that calculates the time at which the task last left the blocked state, thus the task will be blocked until a specific time in ticks has expired since the time pointed by $PreviousWakeTime$. $PreviousWakeTime$ is a variable that stores the last time a task was unblocked: $PreviousWakeTime \in DelayTask \nrightarrow \mathbb{N}$.

The specification of the $TaskDelayUntil$ event is as follows:

TaskDelayUntil **extends** *AddToDelay*
any c
where
$grd3$ $INCLUDE\_TaskDelayUntil = TRUE$
$grd4$ $c \in CurrentTask$
$grd5$ $SleepTask(c) > 0$
$grd6$ $PreviousWakeTime(c) > 0$
then
$act5$ $TimeToWake(c) := PreviousWakeTime(c) + SleepTask(c)$
$act6$ $PreviousWakeTime(c) := TimeToWake(c)$
end

Grd3 is introduced to ensure that the constant *INCLUDE_TaskDelayUntil* is set to true for the delay until operation to be available. *Grd6* ensures that *PreviousWakeTime* has been set, the wake-up time is then updated for the next call in *act5*. The time at which a task should be woken up in case of delay until feature is calculated by adding the sleep time of a task to the previous wake time (the time at which the task last left the blocked state).

Moreover, since the scheduler can be suspended several times, the scheduler will get out of the suspended state by resuming the scheduler for every preceding call that has suspended it. For this, we introduce a counter *SchedulerSuspend* $\in \mathbb{N}$ and a boolean flag *ResumeScheduler* $\in$ bool. Each time an event wants to ask the scheduler to be resumed, it sets the flag *ResumeScheduler* to true. *ResumeAll* event is refined into two events: *DecreaseSchedulerSuspend* event which refines skip and *ResumeAll* event that refines the abstract event *ResumeAll* as shown in the figure:

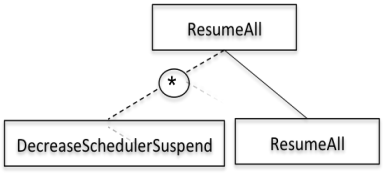


Figure 6.1: Refinement diagram of event *ResumeAll*.

DecreaseSchedulerSuspend is introduced to decrement the counter *SchedulerSuspend* by 1, each time an event set *ResumeScheduler* is set to TRUE. The *ResumeAll* event is refined by adding a guard that checks if the *SchedulerSuspend* counter is equal to 1, so that an action is introduced to assign the counter to zero, and set a flag *ResumeScheduler* to FALSE indicating that the scheduler is resumed.

6.2.1.6 Fifth Refinement- Clock Overflow

This machine specifies clock overflow. An increase in the tick count will eventually overflow and return to zero. A new delay variable is introduced to store the delay tasks whose wake-up time has overflowed. The wake-up time of a delay task is calculated by adding the sleep time of the task to the current tick count. If the

clock has overflowed, delay set should not have any task therefore, delay set and the new overflow delay set are swapped. *DelayTask* is partitioned into two disjoint sets which are *NormalDelayTask* to store tasks whose delay time does not overflow, and *OverflowDelayTask* to store tasks whose delay time overflows.

We further refine the *TaskDelay* event to *TaskNormalDelay* and *TaskOverflowDelay* events. *AddToOverflowDelay* event has an extra guard that checks whether the task time-out of the current task c is not overflowed: $TimeToWake(c) + SleepTask(c) < Tick$; so the task is added to *OverflowDelayTask* set by the action: $OverflowDelayTask := OverflowDelayTask \cup CurrentTask$. Similarly *TaskNormalDelay* has an extra guard and action to check if the tick has not overflowed. The refinement of *TaskDelayUntil* event is similar to the refinement of *TaskDelay* event. The following figure shows the structure of delay events in the fourth/fifth refinement levels.

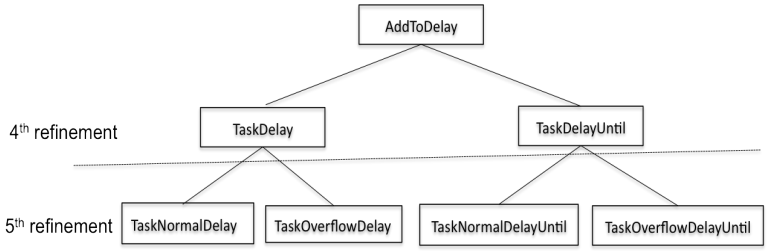


Figure 6.2: The structure of delay events in the fourth/fifth refinement levels.

Finally, the *IncrementTick* event is refined into *IncrementTickOverflow* and *IncrementTickNoOverflow* events. This is because we need to distinguish two cases; the first case, when the wake-up time of a task has overflowed, for instance, consider an 8 bit counter (i.e. the max value for counter is 255), if the tick counter is 240 and the task wants to sleep for 30 ticks, the wake up time will overflow the results in 15, the task then will be kept in the *OverflowDelayTask*, later when the tick counter overflows the two lists *NormalDelayTask* and *OverflowDelayTask* are swapped, so the overflowed tasks can be processed.

The *IncrementTickOverflow* has the guard: *Overflow(Tick,TickLength)* assesses if the tick counter is overflowed. It also has the action: $NormalDelayTask := OverflowDelayTask$ for swapping the delay sets and the action: $OverflowDelayTask := \phi$.

The *IncrementTickNoOverflow* event however, has the guard: *notOverflow(Tick,TickLength)* to ensure that tick counter is not overflowed.

Note that the use of the operators *Overflow* and *notOverflow* is to check if the clock has overflowed or not. Overflow is caused if the value exceeds the largest representable integer value. *Overflow* returns true if the value overflowed and false otherwise, the opposite takes place with *notOverflow*.

$$Overflow(x, y) \triangleq COND(x + y > 2147483647, \top, \bot)$$
$$notOverflow(x, y) \triangleq \neg Overflow(x, y)$$

6.2.1.7 Sixth Refinement- Priority

This level introduces priority. FreeRTOS uses a highest priority first scheduler which runs the higher priority task run before the lower priority task. Scheduler then uses this priority to schedule the task with highest priority. *Priority* variable represents the priority of the task that can be modified ($Priority \in AllTask \rightarrow PRIORITY$). *BasePriority* variable represents the base priority of the task which is always permanent ($BasePriority \in AllTask \rightarrow PRIORITY$). *InterruptPriority* represents the interrupt priority. *PRIORITY* is an integer constant set defined in the context *c0*.

It is frequently required to retrieve the priority of a certain task or set a new priority to it. Therefore, we introduce two more events for this purpose which are: *PriorityGet* to obtain a priority of a task and *PrioritySet* to set the priority of a task to a new priority.

6.2.1.8 Seventh Refinement- Contexts

This level specifies the task contexts. Task context represents the state of the CPU registers required when a task is restored. If the scheduler switches from one task to another, the kernel saves the running task context and uploads the context of the next task to run. The context of the previous running task is restored the next time the task runs. Therefore, the kernel resumes the task execution from the same point where it had left off. This level also specifies critical sections. FreeRTOS performs critical sections by disabling the interrupts, i.e. no interrupt takes place, until the critical section is exited. *TaskContext* variable represents task contexts, *InterruptContext* variable represents interrupt contexts and *Context* variable represents the physical context (processor context).

ContextSwitch **extends** *ContextSwitch*
where
*grd*8 $CurrentISR = \varnothing$
then
*act*6 $Context := TaskContext(t)$
*act*7 $TaskContext(c) := Context$
end

6.2.2 Memory Management

Among one of the most important features provided by FreeRTOS and other operating systems is memory management. FreeRTOS adopts three different schemes to allocate memory. The first scheme is targeted to small applications that do not require free memory. It uses malloc() function to allocate a fixed amount of memory to each object, memory de-allocation does not exist in this scheme. The second scheme adopts the best fit algorithm to re-allocate memory that have been freed. The third scheme is a wrapper calling the standard malloc() and calloc()

functions. FreeRTOS memory management schemes are RTOS independent. They can be used for managing memory for different RTOS; therefore, the development of FreeRTOS memory schemes provide generic memory management development patterns that can be used across different RTOS.

The memory management model consists of two abstract models with one context c. Part of the context c is given below:

CONTEXT c
SETS
BLOCK, OBJECT, ...
CONSTANTS
ADDR, StartAddressHeap, EndAddressHeap, ...
AXIOMS
axm0 : $StartAddressHeap \in \mathbb{N}$
axm1 : $EndAddressHeap \in \mathbb{N}$
axm2 : $StartAddressHeap \leq EndAddressHeap$
axm3 : $ADDR = StartAddressHeap \,..\, EndAddressHeap$

⋮

The *OBJECT* carrier set is shared between the three models: task, queue, and memory as shown in Figure 6.3. *StartAddressHeap* is a constant that represents the starting address of the heap structure. *EndAddressHeap* is a constant that represents the ending address of the heap structure.

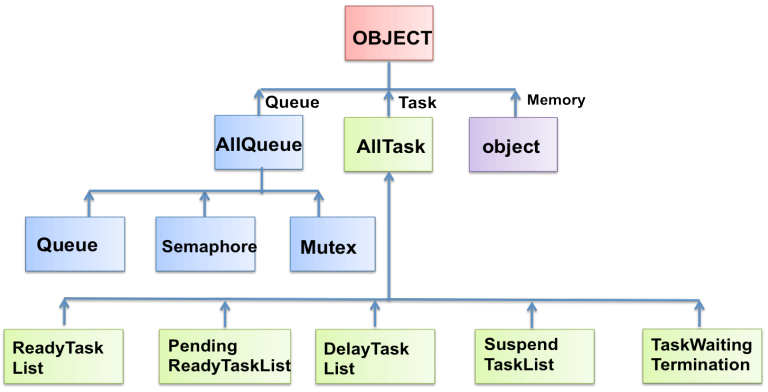


Figure 6.3: Shared entity "Object" between Task, Queue, and Memory models.

The abstract models of memory management for scheme1 and 2 are presented in the following sub-sections.

6.2.3 Scheme1

Scheme1 allocates memory of an adequate size to an object but it does not free an allocated memory. The abstract specification specifies memory blocks and their addresses.

6.2.3.1 An Abstract Specification- Memory Blocks and Addresses

This machine defines the memory structure including the blocks and addresses. We have four state variables: *ablocks* represents the allocated blocks , *fblock* represents the free block, *blockSize* represents the size of the blocks, and finally *blockAddr* represents the starting addresses of the blocks.

The definition of the state variables along with the invariants are given below:

inv1 : $blocks \subseteq BLOCK$
inv2 : $ablocks \subseteq blocks$
inv3 : $fblock \subseteq blocks$
inv4 : $fblock \cap ablocks = \phi$
inv5 : $blockSize \in (ablocks \cup fblock) \rightarrow \mathbb{N}$
inv6 : $blockAddr \in (ablocks \cup fblock) \rightarrow ADDR$
inv7 : $\forall b \cdot b \in blocks \Rightarrow blockAddr(b) \geq StartAddressHeap \wedge blockAddr(b) + blockSize(b) - 1 \leq EndAddressHeap$
inv8 : $\forall b1, b2 \cdot b1 \in blocks \wedge b1 \neq b2 \Rightarrow blockAddr(b1)\,..\,(blockAddr(b1) + blockSize(b1) - 1) \cap blockAddr(b2)\,..\,(blockAddr(b2) + blockSize(b2) - 1) = \varnothing$
inv9 : $malloc \in blocks \rightarrowtail OBJECT$
inv10 : $card(fblock) \leq 1$

Inv8 assures that the blocks are disjoint. *Inv10* states that the free-block is only one, this is because scheme1 fills the heap from the bottom and leave the top of the heap to be free.

In addition, this machine has two events that describe the allocation process. *Malloc1* event subdivides the large free-block in *grd2* into two blocks: the first block of the required size is allocated to the object, the second block returns a free-block.

Malloc1
any $s \quad b \quad c \quad o \quad p$
where
*grd*1 $s \in \mathbb{N}_1$
*grd*2 $b \in fblock$
*grd*3 $p \in portBYTE\_ALIGNMENT\_MASK$
*grd*4 $c \in BLOCK \setminus (ablocks \cup fblock)$
*grd*5 $blockSize(b) > allign(s, p)$
*grd*6 $o \in OBJECT \setminus ran(malloc)$
then
*act*1 $blockAddr := blockAddr \lhd\!\!- \{b \mapsto blockAddr(b) + allign(s, p), c \mapsto blockAddr(b)\}$
*act*2 $blockSize := blockSize \lhd\!\!- \{b \mapsto blockSize(b) - allign(s, p), c \mapsto allign(s, p)\}$
*act*3 $ablocks := ablocks \cup \{c\}$
*act*4 $malloc := malloc \cup \{c \mapsto o\}$
end

allign used in *grd5* and *act1,2* is an operator defined to ensure that blocks are always aligned to the required number of bytes p. It is defined as follows:

$$(((s - 1) / p) + 1) * p.$$

Where s is the required size and p is the $portBYTE\_ALIGNMENT\_MASK$. $portBYTE\_ALIGNMENT\_MASK$ is defined as a constant in the context c.

Malloc2 event is used when the free-block is of adequate size to the object as captured by the guard *grd4*:

Malloc2
any s b c o p
where
*grd*1 $s \in \mathbb{N}_1$
*grd*2 $b \in fblock$
*grd*3 $p \in portBYTE\_ALIGNMENT$
*grd*4 $blockSize(b) = allign(s, p)$
*grd*5 $o \in OBJECT \setminus ran(malloc)$
then
*act*1 $fblock := \varnothing$
*act*2 $ablocks := ablocks \cup \{b\}$
*act*3 $malloc := malloc \cup \{b \mapsto o\}$
end

6.2.4 Scheme2

This scheme uses the best-fit algorithm which works by searching for all free blocks and then placing an object in the block of adequate size to which it will fit; if no block of adequate size is found and there is a block larger than what is required, it is split into two, one block of the requested size is assigned to the object and the second block is added to the list of free blocks.

The initial model of scheme2 is similar to scheme1 with some changes related to freeing the allocated memory. The state variables are the same, but we do not include the invariant $card(fblock) \leq 1$, as we have several free blocks. In addition, there are three events in this level, *Malloc1,2* events are used for allocation and *Free* is used for de-allocation. *Malloc1* event is used to allocate a block of adequate size to the created object as captured by *grd5*:

Malloc1
any c s p o
where
*grd*1 $c \in fblocks$
*grd*2 $o \in OBJECT \setminus ran(malloc)$
*grd*3 $s \in \mathbb{N}_1$
*grd*4 $p \in portBYTE\_ALIGNMENT$
*grd*5 $blockSize(c) = allign(s, p)$
then
*act*1 $malloc := malloc \cup \{c \mapsto o\}$
*act*2 $fblocks := fblocks \setminus \{c\}$
*act*3 $ablocks := ablocks \cup \{c\}$
end

Malloc2 event is used when the free blocks are larger than what is required as captured by *grd4*, and hence the free-block is split into two blocks: one block of

the requested size is assigned to the object and the second block is added to the set of free blocks.

Malloc2
any c s $b1$ p o
where
$grd1$ $c \in fblocks$
$grd2$ $s \in \mathbb{N}_1$
$grd3$ $p \in portBYTE\_ALIGNMENT$
$grd4$ $blockSize(c) > allign(s, p)$
$grd5$ $\forall k.k \in blockSize[fblocks] \land k < blockSize(c) \Rightarrow k < allign(s, p)$
$grd6$ $b1 \in BLOCKS \setminus (fblocks \cup ablocks)$
$grd7$ $blockSize(c) - allign(s, p) > heapMINIMUM\_BLOCK\_SIZE$
$grd8$ $o \in OBJECT \setminus ran(malloc)$
then
$act1$ $malloc := malloc \cup \{c \mapsto o\}$
end $act2$ $blockSize := blockSize \leftarrow \{c \mapsto allign(s, p), b1 \mapsto blockSize(c) - allign(s, p)\}$
$act3$ $blockAddr := blockAddr \cup \{b1 \mapsto blockAddr(c) + allign(s, p)\}$
$act4$ $ablocks := ablocks \cup \{c\}$
$act5$ $fblocks := (fblocks \setminus \{c\}) \cup \{b1\}$

Grd7 ensures that the differences between the size of the block found and the required size is larger than $heapMINIMUM\_BLOCK\_SIZE$. This condition must hold to allow the block found to be split into two blocks: $heapMINIMUM\_BLOCK\_SIZE$ is a constant that defines the minimum acceptable size of a memory block.

Free event allows the previously allocated blocks to be freed.

Free
any b
where
$grd1$ $b \in ablocks$
then
$act1$ $ablocks := ablocks \setminus \{b\}$
$act2$ $fblocks := fblocks \cup \{b\}$
end

6.3 Description of the Approach

Shared entities are shared carrier sets between sub-models which we use instead of shared variables to link sub-models together. This section outlines the generic structure for two machines of different sub-models that include shared entities instead of shared variables. We attempt to use shared entities to provide links between the individual sub-models without the need of replicating shared properties on shared variables across the sub-models. Shared entities seem to produce loosely-coupled models and it helps to avoid the constraints imposed by having shared variables across the sub-models such as the inability of the data refinement of shared variables. This section also outlines an example that compares the application of the shared event composition to sub-models that include shared entities and sub-models with shared variables.

Assume that we have two sub-models: *model1* with machine *M1*, and *model2* with the machine *M2*. Both sub-models have the context *C*. The carrier set D is shared between *model1* and *model2*. Machine *M1* of *model1* consists of variable *d1* that is defined in terms of the carrier set D and possibly some other variables/sets K (*d1:(D, K)*), list of variables $\vec{v}$, and invariants $I(\vec{v})$. It also has event *e1* that modifies *d1*, and list of events *e2* that modify $\vec{v}$. Machine *M2* of *model2* consists of variables *d2* which are defined in terms of the carrier set D and some other variables/sets H (*d2:(D, H)*), list of variables $\vec{t}$, and invariants $L(\vec{t})$. *M2* has event *e1* that modifies *d2* and the list of events *e2* that modify $\vec{t}$. Both sub-models do not include shared-variables. The variables *d1* and *d2* are localised in each sub-model.

Context C
Sets D

Figure 6.4: Context C of *model1* and *model2*.

Machine $M1$ **sees** C
variables $d1$ $\vec{v}$
invariants
$d1 : (D, K)$
$I(\vec{v})$
events
$e1 \triangleq$ **any** k
where
$k \in D$
$G(k, d1)$
then
$d1 := E(d1, k)$
end
$e2 \triangleq$ **any** $\vec{y}$ **where**
$J(\vec{v}, \vec{y})$
then
$\vec{v} := (\vec{v}, \vec{y})$
end

Machine $M2$ **sees** C
variables $d2$ $\vec{t}$
invariants
$d2 : (D, H)$
$L(\vec{t})$
events
$e1 \triangleq$ **any** k
where
$k \in D$
$R(k, d2)$
then
$d2 := L(d2, k)$
end
$e2 \triangleq$ **any** $\vec{q}$ **where**
$M(\vec{t}, \vec{q})$
then
$\vec{t} := (\vec{t}, \vec{q})$
end

Figure 6.5: Generic machine *M1* of *model1* and *M2* of *model2*.

We compose *M1* and *M2* using the shared event composition. We are interested here to show the composition between *e1* in *M1* and *e1* in *M2* as both events update variables that are defined in terms of the shared entity "D". Therefore, we have the composed event *cmp.e1* which combines event *e1* from machine *M1* and event *e1* from machine *M2*.

Since *d1* and *d2* are localised in each sub-model, it was not possible to specify properties (invariants) that connect these variables together in one of the sub-models. Therefore, we add the invariant $I(d1, d2)$ to specify properties that relate the variable *d1* from machine *M1* and the variable *d2* from machine *M2*. The shared event paramater(s) k represents a key part of the interface between *M1* and *M2*, therefore, it must have the same names in the events to be composed and

must not disappear during the refinement of the individual models, *model1* and *model2*.

```
COMPOSED MACHINE cm
INCLUDES
  M1
  M2
INVARIANTS
  I(d1, d2)
Events
  cmp.e1
  Combines Events M1.e1 || M2.e1
...
END
```

Figure 6.6: The composed machine *cm*.

6.3.1 An Example of Shared Event Composition with Shared Entities

We use shared entities instead of shared variables as a means to link individual models together. Shared entities assist in providing loosely coupled models. It allows to avoid replicating constraints and properties on shared variables across the individual models. Shared properties are only introduced during the composition of the individual models.

We here outline a small example of two sub-models to illustrate the application of the shared event composition approach to shared entities sub-models.

The example consists of two sub-models: memory model (scheme2) and queue model. Machine *m1* in memory model, allocates RAM to each object via two events: *Malloc1* and *Malloc2*. *Malloc1* allocates memory of adequate size to the object. *Malloc2* is used if no block is available with adequate size for the object. It subdivides the large free block into two blocks: the first block of the required size is allocted to the object and the second block returns a free block. On the other hand, machine *m1* in the queue model deals with the creation of three different types of queues: general queues, binary semaphores, and counting semaphores. We need to show the composition between the allocation events in the memory model and the creation events in the queue model. The requirement *COMP1-EVT* shows that there is a connection between the allocation events in the memory model and the creation events in the queue model. Therefore *COMP1* is a composition requirement. As described in Chapter 4, composition requirements describe the connection of the relevant events as shared events, shown in *COMP1-EVT* and can also specify invariants that link variables of different sub-models such as the requirement *COMP1-INV*. The composition requirement that describes events are subdivided into a number of sub-requirements, each of which describes an event in a single sub-model. In fact, the sub-requirements fit the first class of the requirement classification given in Chapter 4 (i.e. requirements associated to a certain

component). The requirement *QUE1* is a sub-requirement in the queue model that is used to introduce queue creation event. The requirement *MEM1* is a sub-requirement in the memory model that is used to introduce the allocation events. The requirement *MEM2* is a sub-requirement in the memory model that is used to describe *Malloc1* and *Malloc2*.

The creation events of the queue model that correspond to requirements *TSK1*, *QUE1* and *QUE13* are:

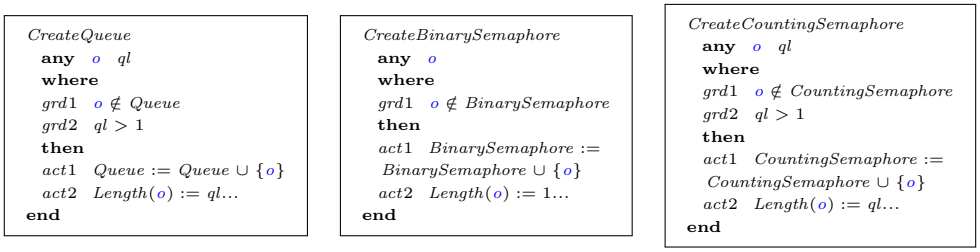
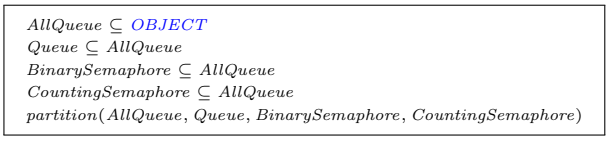


Figure 6.7: *CreateQueue event*, *CreateBinarySemaphore* event and *CreateCountingSemaphore* event in the queue model.

CreateQueue event creates a new queue of length grater than one. *CreateBinary-Semaphore* creates a binary semaphore of length one. *CreateCountingSemaphore*creates a counting semaphore of length greater than one. where *Queue*, *BinarySemaphore* and *CountingSemaphore* are defined as follows:



We recall that the allocation events of the memory model described in section 6.2.4 which correspond to requirements *MEM1* and *MEMS8* are:

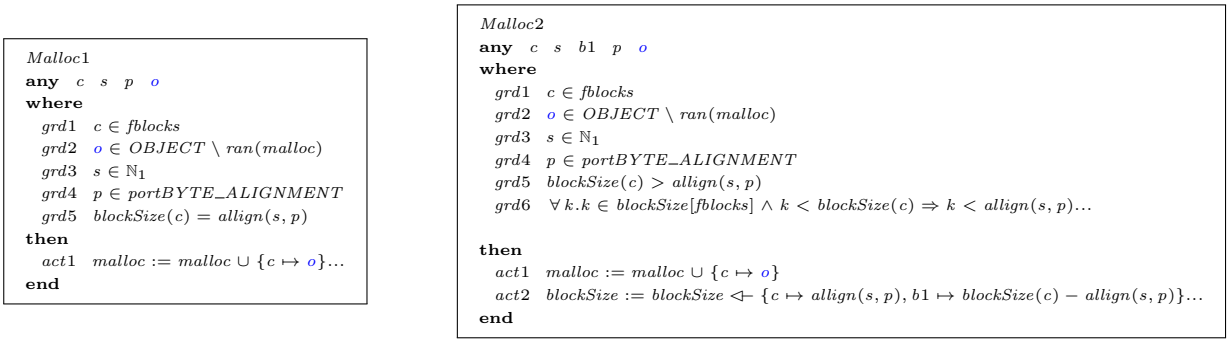


Figure 6.8: *Malloc1* event and *Malloc2* event of memory model.

Where *malloc* is defined as follows:

$$malloc \in blocks \rightarrow OBJECT$$

The carrier set *OBJECT* is shared between the memory and queue models. The parameter *o* in the creation events and allocation events is a shared event parameter. It links the object instance from the *queue* model with the object instance from the*memory* model. The shared parameter *o* must have the same name in both models and must not disappear during the refinement of *memory* and *queue* models.

To visually demonstrate the interaction between the sub-models via synchronisation over events, we introduce a diagram called composition diagram. The composition diagram of the example given in this sub-section is shown in the following figure:

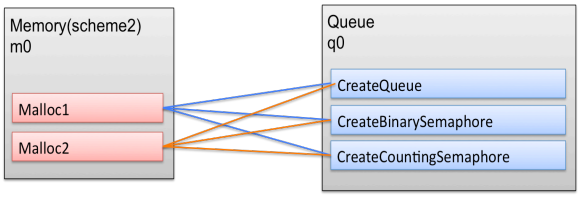


Figure 6.9: The composition diagram of sub-models with shared entities.

Each machine is represented into a block diagram that describes the sub-model names and the machine names. The small rectangles within each block represents the events to be composed. The number of lines gives the number of composed events. Thus, Figure 6.9 demonstrates the interaction between *Malloc1* and *Malloc2* in scheme1 model at machine*m0* with *CreateQueue*, *CreateBinarySemaphore*, and *CreateCountingSemaphore* in queue model at machine *q0*. The number of the composed events are six events.

In order to show the interaction between memory and queue models, we apply the shared event composition approach. The structure of the composed machine is shown in Figure 6.10:

COMPOSED MACHINE *cm*
INCLUDES
q0
m0
INVARIANTS
ran(malloc) = AllQueue
Events
MallocQueue1
Combines Events
q0.CreateQueue || m0.Malloc1
MallocQueue2
Combines Events
q0.CreateQueue || m0.Malloc2
MallocBinarySemaphore1
Combines Events
Q0.CreateBinarySemaphore || m0.Malloc1
MallocBinarySemaphore2
Combines Events
q0.CreateBinarySemaphore || m0.Malloc2
MallocCountingSemaphore1
Combines Events
q0.CreateCountingSemaphore || m0.Malloc1
MallocCountingSemaphore2
Combines Events
q0.CreateCountingSemaphore || m0.Malloc2
END

Figure 6.10: The composed machine *cm* of sub-models with shared entities.

Composed machine *cm* includes machines *q0* and *m0*. An invariant connecting the two machines $ran(malloc) = AllQueue$ is added to restrict the range of *malloc* function. We add this invariant in the composed machine because it connects *malloc* function to *AllQueue*. Events of *cm* are identified as the parallel composition (interaction) of *q0* events and *m0* event.

6.3.2 An Example of Shared Event Composition with Shared Variables Sub-models

We analyse Figure 6.7 and 6.8 to compare shared event composition with shared variable sub models. The variables *Queue*, *BinarySemaphore*, and *CountingSemaphore* become shared between *queue* model and *Memory* model. The allocation events in the *memory* model contains six events: *Malloc1Queue* event, *Malloc2Queue* event, *Malloc1BinarySemaphore* event, *Malloc2BinarySemaphore* event, *Malloc1CountingSemaphore* event, and *Malloc2CountingSemaphore* event.

The specification of *Malloc1Queue* event and *Malloc2Queue* event in the modified memory model (with shared variables) that correspond to requirements *MEM1* and *MEMS8* can be seen as follows:

MallocQueue1
any c s p o
where
$grd1$ $c \in fblocks$
$grd2$ $o \in Queue1$
$grd3$ $s \in \mathbb{N}_1$
$grd4$ $p \in portBYTE\_ALIGNMENT$
$grd5$ $blockSize(c) = allign(s, p)$
then
$act1$ $malloc := malloc \cup \{c \mapsto o\}...$
end

MallocQueue2
any c s p o s
where
$grd1$ $c \in fblocks$
$grd2$ $o \in Queue1$
$grd3$ $s \in \mathbb{N}_1$
$grd4$ $p \in portBYTE\_ALIGNMENT$
$grd5$ $blockSize(c) > allign(s, p)$
$grd6$ $\forall k.k \in blockSize[fblocks] \wedge k < blockSize(c) \Rightarrow k < allign(s, p)$
then
$act1$ $malloc := malloc \cup \{c \mapsto o\}$
$act2$ $blockSize := blockSize \leftarrow\!\!\!- \{c \mapsto blockSize(c) - s, c \mapsto s)\}...$
end

Figure 6.11: *Malloc1Queue* event and *Malloc2Queue* event of the modified machine *m0* of memory model.

In the above figure, *Queue1*, *BinarySemaphore1*, and *CountingSemaphore1* are defined in the memory model:

$AllQueue1 \subseteq OBJECT$
$Queue1 \subseteq AllQueue1$
$BinarySemaphore1 \subseteq AllQueue1$
$CountingSemaphore1 \subseteq AllQueue1$
$partition(AllQueue1, Queue1, BinarySemaphore1, CountingSemaphore1)$
$malloc \in blocks \rightarrow AllQueue1$

The variable *malloc* connects the memory and the queue models. The variables *Queue1*, *BinarySemaphore1*, and *CountingSemaphore1* are also shared between the queue and memory models. They are renamed differently in the modified memory model because of a limitation in the shared event composition tool which could be fixed. The shared event composition tool does not allow composing models with variables having similar names. We cope with this limitation for now by adding some additional invariants in the composed machine to show that *Queue1*, *BinarySemaphore1*, and *CountingSemaphore1* variables are the same as *Queue*, *BinarySemaphore*, and *CountingSemaphore* variables.

The composition diagram in this case can be as follows:

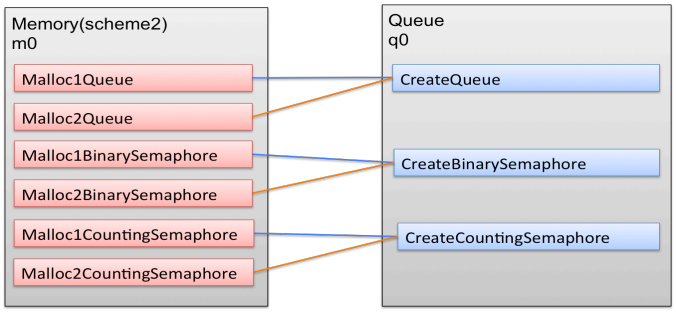


Figure 6.12: The composition diagram of sub-models with shared variables.

The structure of the composed machine is shown as follows:

COMPOSED MACHINE *cm*
INCLUDES
q0
m0
INVARIANTS
AllQueue1=AllQueue
Queue1=Queue
BinarySemaphore1=BinarySemaphore
CountingSemaphore1=CountingSemaphore
Events
Malloc1Queue
Combines Events
q0.CreateQueue || *m0.Malloc1Queue*
Malloc2Queue
Combines Events
q0.CreateQueue || *m0.Malloc2Queue*
Malloc1BinarySemaphore
Combines Events
queue0.CreateBinarySemaphore || *m0.Malloc1BinarySemaphore*
Malloc2BinarySemaphore
Combines Events
q0.CreateBinarySemaphore || *m0.Malloc2BinarySemaphore*
Malloc1CountingSemaphore
Combines Events
q0.CreateCountingSemaphore || *m0.Malloc1CountingSemaphore*
Malloc2CountingSemaphore
Combines Events
q0.CreateCountingSemaphore || *m0.Malloc2CountingSemaphore*
END

Figure 6.13: The composed machine *cm* of sub-models with shared entities.

As discussed in this section, the composition can be achieved successfully with two approaches shared entities and shared variables. However, it seems that connecting the sub-models using shared entities encourages modularity of building models and promotes reusability. The sub-models are required to be built separately of others and the properties associated to the inter-connection between the sub-models can be introduced only in the composition level through additional invariants. The sub-models can be incorporated to build different systems. For instance, the memory model can be incorporated to provide memory management functionality to different systems such as FreeRTOS, UCOS, etc. The memory model provides general functionality for memory allocation and de-allocation and does not include specific properties that connect particular objects such as task or queue to memory blocks. As a limitation of using shared entities to connect the sub-models, shared properties that link sub-models cannot be localised within sub-models; they can only be added during the composition level. In addition, shared parameters must have the same name in the events to be composed, and must not disappear during refinement because they form the interface between the included models (this limitation imposed by shared-event composition tool which could be fixed in the next releases of the tool). In addition, the properties associated to the inter-connection between the sub-models are introduced in the composition level through adding extra invariants. The generated proof obligations for these invariants are discharged automatically.

As for the second approach (shared variables), the properties associated to the sub-models are localised within the sub-models that have shared variables. On the other hand, when sub-models have shared variables, they must have different names in the sub-models since the shared event composition tool prohibits having shared variables in the included machines. Consequently, it becomes necessary at the composition level to add additional invariants to show that two variables names refer to the same variable (this limitation imposed by shared-event composition tool which could be fixed in the next releases of the tool). Although, the composition can be achieved successfully when sub-models have shared-variables; shared variables and properties are replicated across the sub-models. Thus, reusability of the sub-models becomes more difficult, and the individual sub-models become more targeted to a particular system.

6.3.3 Simplifying the Connection Between Sub-models with Abstraction

Another important point to note here is that, it is possible to simplify the connection between the queue and memory models and reduce the number of composed events composed events given in Section 6.3.1 from six to two. In order to do this, we need to return to the abstract machine *q0* in queue model and makes it more abstract. We can only define one variable in the modified abstract machine *q0*, which is *AllQueue* with a single event named *CreateAllQueue* that is specified as follows:

CreateAllQueue
  any o
  where
  $grd1$ $o \notin Queue$
  $grd1$ $ql > 1$
  then
  $act1$ $AllQueue := AllQueue \cup \{o\}$
end

Figure 6.14: The abstract *CreateAllQueue* event of the modified machine *q0* in the queue model.

After that, we can create another machine *q1* that refines the modified machine *q0*. The machine *q1* includes the variables *Queue*, *BinarySemaphore*, and *CountingSemaphore* with three events: *CreateQueue* event, *CreateBinarySemaphore* event, and *CreateCountingSemaphore* event which refine *CreateAllQueue* event.

As a consequence, the composition in this case can be applied between *Malloc1,2* events and the abstract event *CreateAllQueue*. The number of composed events in this case is reduced from six events to two events as shown in the following composition diagram:

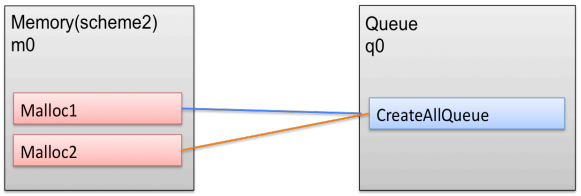


Figure 6.15: The composition diagram after the abstraction of *q0* of the queue model.

6.4 The Composition of FreeRTOS Specifications

This section outlines the composition requirement of FreeRTOS and the specification of some of the individual events and the composition diagrams that show the connection between the task management, queue management, and memory management of FreeRTOS case study.

6.4.1 Requirement COMP1

The composition requirement *COMP1-EVT* connects the task model, the queue model, and the memory model through event sharing.

COMP1-EVT	The kernel has to allocate RAM each time a task, queue, semaphore or mutex is created.
COMP1-INV	Tasks and queues are distinct.

The sub-requirements of *COMP1-EVT* are:

MEM1	Kernel allocates RAM to each created object.
MEMS8	The created object in scheme2 is placed in the block of adequate size in which it will fit (if any), if no block of adequate size is found and there is a block larger than what is required, it will be split into two; one block of the requested size is assigned to the object and the second block is added to the list of free blocks.
TSK1	Tasks can be created.
QUE1	Queues can be created.
QUE13	Queues are of three types: queues, semaphores, and mutex.

The composition diagram in Figure **??** illustrates the connection between the creation events in the the queue model and the allocation events in scheme2 model. The connection between the creation events in the the queue model and the allocation events in scheme1 model is similar to Figure **??**.

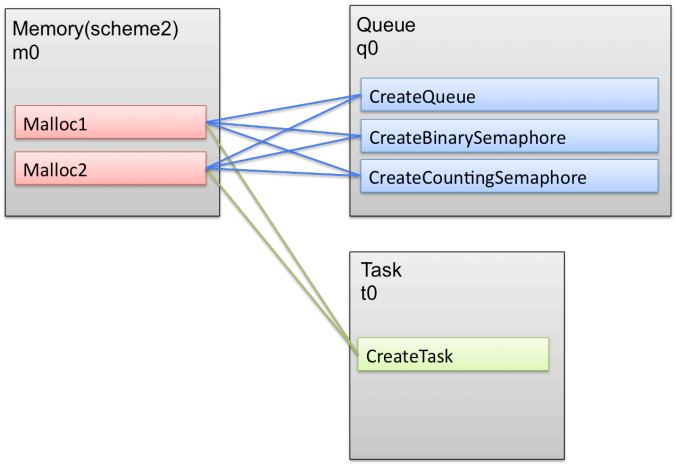


Figure 6.16: The composition diagram of the requirement COMP1-EVT.

The composition requirement *COMP1-INV* indicates the addition of the following invariants to the composition level to show that *AllTask* and *AllQueue* are disjoint: $partition(ran(malloc), AllTask, AllQueue)$ and $AllQueue \cap AllTask = \phi$

The composition diagram that shows the connection between the creation events in the task and queue sub-models and the allocation events in scheme2 model are similar to the above diagram.

6.4.2 Requirement COMP2

The composition requirement *COMP2* connects the task model, queue model, and memory model through event sharing.

COMP2-EVT	The kernel has to free RAM each time a task, queue, semaphore or mutex is deleted.

The sub-requirements of *COMP2* requirements are:

MEM10	Memory in scheme2 can be freed once it has been allocated.
TSK1	Tasks can be deleted.
QUE1	Queues can be deleted.
QUE13	Queues are of three types: queues, semaphores, and mutex.

The following diagram illustrates the connection between the deletion events in the task and queue models and the freeing memory events in scheme1 model.

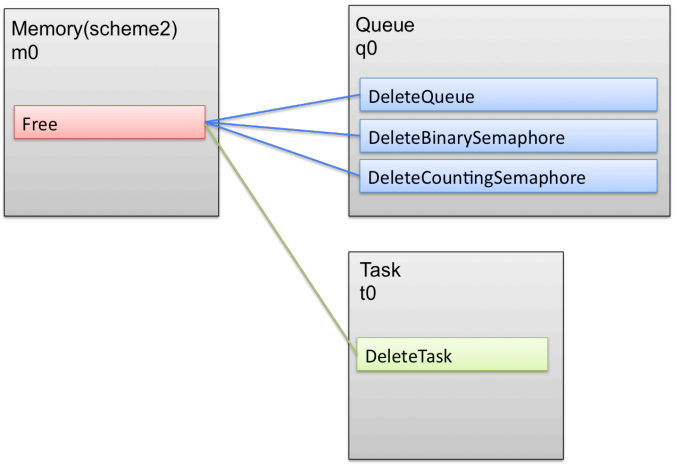


Figure 6.17: The composition diagram of the requirement COMP2-EVT.

The composition diagram that shows the connection between the deletion events of the task and queue models and the freeing memory events of scheme2 model is given in the figure.

6.4.3 Requirement COMP3

The composition requirement *COMP3* connects the task model and the queue model through event sharing.

COMP3-EVT	The running task can send/receive an item to/from a queue.
COMP3-INV	Task items and queue items are distinct.

The sub-requirements of *COMP3-EVT* requirement are:

TSK10	The running task can obtain an item or remove an existing item.
QUE14	A queue can be used to send and receive items.

The specification of *TaskRemoveItem* event and *ObjectQueueSend* event are:

TaskRemoveItem
any c i
$grd1$ $c \in CurrentTask$
$grd2$ $i \in TaskItem^{-1}[\{c\}]$
then
$act1$ $TaskItem := TaskItem \setminus \{i \mapsto c\}$
end

ObjectQueueSend
any q i
where
$grd1$ $q \in AllQueue$
$grd2$ $i \in ITEM \setminus dom(QueueItem)...$
then
$act1$ $QueueItem := QueueItem \cup \{i \mapsto q\}$
end

Figure 6.18: *TaskRemoveItem* event of task model and *ObjectQueueSend* event of queue model.

The following composition diagram illustrates the connection between the task send/receive events and queue send/receive events

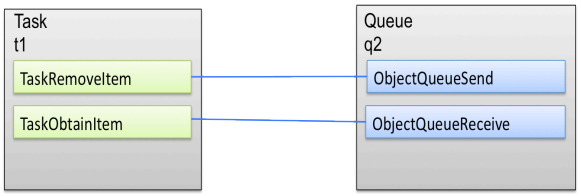


Figure 6.19: The composition diagram of the requirement COMP3-EVT.

The following invariant is added to the composition level:

$dom(TaskItem) \cap dom(QueueItem) = \phi$

6.4.4 Requirement COMP4

The composition requirement *COMP4* connects task model and queue model through event sharing.

COMP4-EVT	A task that is removed from the collection of waiting tasks is placed into the collection of pending-ready tasks if the scheduler is suspended or placed into the collection of ready tasks if the scheduler is running.
COMP4-INV1	A task that is added to the collection of waiting tasks must also be added to the collection of delay tasks.

The sub-requirements of *COMP4-EVT* requirements are:

QUE14	An object can be removed from the collections of waiting tasks.
TSK14	A task can be put into the collection of ready tasks if the scheduler is running.
TSK20	A task can be put into the collection of pending-ready tasks if the scheduler is suspended.

The specification of *AddToReady* event and *RemoveFromTaskWaitingToSend* event are:

AddToReady
any t
where
$grd1$ $t \in DelayTask$ ...
then
$act1$ $ReadyTask := ReadyTask \cup \{t\}$
$act2$ $DelayTask := DelayTask \setminus \{t\}$
end

RemoveFromTaskWaitingToSend
any q t
where
$grd1$ $q \in AllQueue$
$grd2$ $t \in TaskWaitingToSend^{-1}[\{q\}]$...
then
$act1$ $TaskWaitingToSend := TaskWaitingToSend \setminus \{t \mapsto q\}$...
end

Figure 6.20: *AddToReady* event of task model and *RemoveFromTaskWaitingToSend* event of queue model.

The following figure shows the composition diagram of the requirement *COMP4*

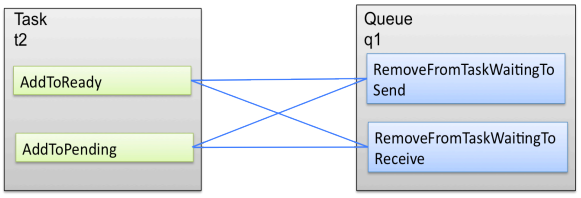


Figure 6.21: The composition diagram of the requirement COMP4-EVT.

The following invariant is added to the composition level based on the composition requirement *COMP4-INV1*:

$dom(TaskWaitingToSend) \subseteq DelayTask$

6.4.5 Requirement COMP5

The composition requirement *COMP5* connects task model and queue model through event sharing.

COMP5-EVT	The blocked running task with block-time that is greater than zero and less than the value determined by the constant *PortMAX_DELAY* is added to the collection of waiting tasks and also the collection of delay tasks.

The sub-requirements of *COMP5-EVT* requirement are:

QUE14	An object can be added to the collection of waiting tasks if its block-time is greater than zero and less than the constant *PortMAX_DELAY*.
TSK21	The running task can be put into the collection of delay tasks.

The composition diagram of *COMP5-EVT* requirement is:

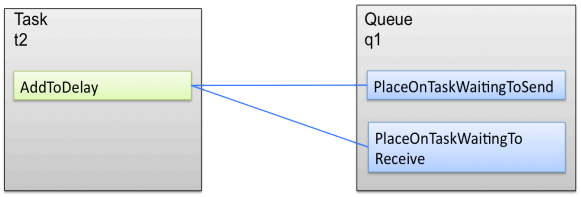


Figure 6.22: The composition diagram of the requirement COMP5-EVT.

6.4.6 Requirement COMP6

The composition requirement *COMP6* connects task model and queue model through event sharing.

COMP6-EVT	The blocked running task with block-time that is equal to the value determined by the constant *PortMAX_DELAY* is added to the collections of waiting tasks and also the collection of suspend tasks.

The sub-requirements of *COMP6-EVT* are:

QUE15	An object can be added to sending/receiving collections of waiting tasks if its block-time is equal to the constant *PortMAX_DELAY*.
TSK22	The running task can be put in the collection of suspend tasks.

The composition diagram of *COMP6-EVT* requirement is:

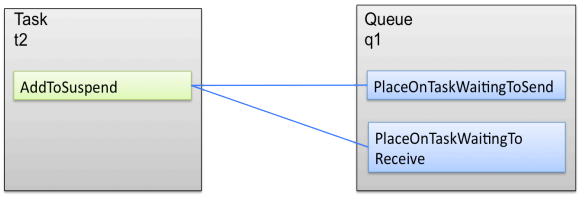


Figure 6.23: The composition diagram of the requirement COMP6-EVT.

6.4.7 Requirement COMP7

Priority inheritance mechanism is a method for minimising priority inversion. It allows a task having taken a mutex to run the highest priority task amongst the blocked tasks waiting for that mutex. The events responsible for modifying a

task-priority are defined in the task sub-model whereas the events responsible for creating mutex is defined in the queue sub-model.

The composition requirement *COMP7-EVT* connects the task model and queue model through event sharing.

COMP7-EVT	The priority inheritance is used to raise the priority of the mutex-holder task whenever the higher priority task (running task) attempts to hold that mutex.

The sub-requirements of *COMP7-EVT* requirement are:

TSK56	A task with lower priority can be raised by the priority of the running task.
QUE16	The object that holds a mutex can be obtained.

The specification of *PrioritySet2* in the task model and *GetMutexHolder* event in the queue model are:

PrioritySet2
any t c np
where
$grd1$ $t \in AllTask$
$grd2$ $np \in PRIORITY$
$grd3$ $c \in CurrentTask$
$grd7$ $TaskPriority(t) < TaskPriority(c)...$
then
$act1$ $TaskPriority(t) := TaskPriority(c)...$
end

GetMutexHolder
any q t
where
$grd1$ $q \in Mutex$
$grd2$ $t = MutexHolder(q)$
end

Figure 6.24: *PrioritySet2* event of task model and *GetMutexHolder* event of the queue model.

The composition diagram of *COMP7-EVT* requirement is:

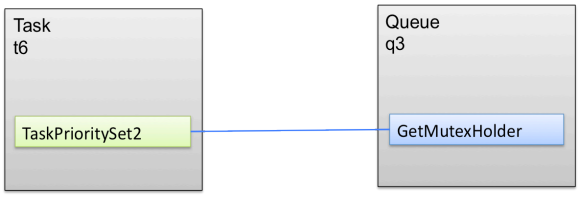


Figure 6.25: The composition diagram of the requirement COMP7-EVT.

6.4.8 Requirement COMP8

The composition requirement *COMP8-EVT* is similar to *COMP7-EVT* requirement. It connects the task model and queue model through event sharing.

COMP8-EVT	The priority disinheritance is used to set the active priority of the task holder back to its original priority.

The sub-requirements of *COMP8-EVT* requirement are:

TSK44	The active priority of a task can be replaced by its original priority.
QUE16	The object that holds a mutex can be obtained.

The composition diagram of *COMP8-EVT* requirement is:

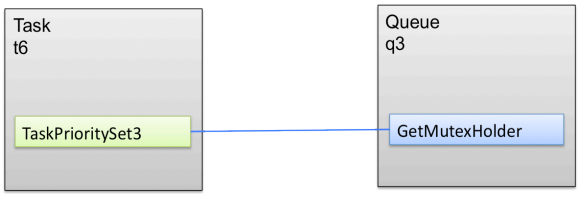


Figure 6.26: The composition diagram of the requirement COMP8-EVT.

6.5 Conclusions

This chapter has contributed to support the process of composing the separate sub-models: task management, queue management, and memory management to give a complete perspective of FreeRTOS.

Shared event composition is used to compose the three sub-models. Shared entities "carriers sets" were used as a means of connection between the sub-models instead of shared variables. The sub-models are linked together through shared entities, and the properties that are shared across the sub-models must only be introduced in the composition level through additional invariants. This approach of building models seem to promote reusability since the models to be composed are loosely-coupled and do not include shared properties.

Composition can also be obtained using shared variables. The properties can be shared across the sub-models and therefore do not require to be introduced at the composition level. Having shared variables between the sub-models results in more coupled models and therefore shared variables approach is less interesting when the reusability of models is important.

This chapter also provides an example of the composition resulted from composing the sub-models with shared variables using the shared event composition approach as illustrated in Section 6.3.2.

Moreover, Section 6.3.3 has demonstrated by an example the possibility of simplifying the connection between the sub-models with an abstraction. It shows that the more abstract the sub-models are composed, the less is the effort and the number of events we need to compose.

Chapter 7

Reusing Data Refinement Patterns through Generic Instantiation and Composition

The benefits of applying reuse at different stages of the software development cycle are widely recognised. The essence of reuse is the use of existing artifacts during the construction of a new one. Reusing in formal modelling incorporates the reuse of models through the process of constructing a new model in such a way that proofs are preserved. The main objective of this chapter is to present an approach to facilitate the reuse in Event-B formal method through the use of generic instantiation and composition techniques. An example of data refinement of the abstract " set" of FreeRTOS model to linked lists is presented to illustrate the use of this approach and to show the **significant** benefits of reusing models in formal modelling.

7.1 Introduction

Software reuse is a promising area for better quality and increased productivity in software [67]. Reuse is known in the area of formal methods: a model can be constructed by combination of several modelling patterns. Besides the advantages of applying patterns such as reducing time, cost, and proof effort, there is also an opportunity for the reused patterns to go through a review process and this will lead to an improvement of the quality of the patterns.

A pattern may be developed as a general pattern applicable in several different modelling problems such as patterns for data structures (e.g. arrays, linked lists, queues, and stacks). A pattern also might be specified for a particular domain. Developing reusable patterns for recurring modelling concepts in given contexts is useful. For instance, real time operating systems meet a set of criteria that can

be formalised as a number of patterns to aid the formal modelling of operating systems. Consequently, this would increase the reuse of the formal modelling in the operating system domain.

The advantages of applying patterns can be achieved by having several reusable patterns available for the designer. A modelling reuse repository with various reusable modelling patterns should be developed to facilitate rapid adaptation of a pattern as per designer needs.

Our main contribution is an approach for reusing modelling *patterns* in Event-B [4] based on generic instantiation [72] and composition [73] techniques. Generic instantiation technique is used to create an instance of a pattern that consists of a refinement chain and allowing replacement of the pattern names (types, constants, variables, events) by names that suit the development at hand. The composition technique, on the other hand, enables the integration of several sub-models into a large model. Composition may also be applied in reverse in a top-down way by factorising a model into a composition of smaller models. We refer to this as decomposition. The techniques are well-established and widely used in modelling with Event-B. Detailed descriptions of generic instantiation and composition techniques are given in Chapter 2.

This chapter is mainly divided into two parts. The first part presents an approach that aids in reusing Event-B models and incorporating them to resume the specification of other development. The presented approach has been used successfuly to replace the abstract "set" of FreeRTOS specification to linked lists structure. The second part focuses on developing theories of linked lists involving set of operators and inference rules that support the development of linked lists models.

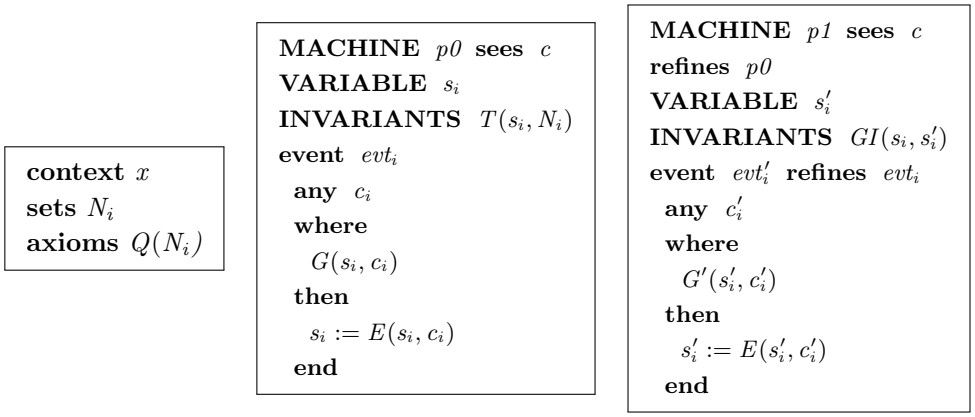
Section 7.2 presents an approach that supports reusing Event-B models. Section 7.3 introduces a theory of circular linked lists. Section 7.4 outlines the circular doubly linked list pattern. Section 7.5 outlines the application of the presented approach to FreeRTOS case study. Section 7.6 presents some related work to the presented approach. Conclusions are given in Section 7.7

7.2 Description of the Approach

This section describes our approach and shows how generic instantiation and composition techniques are used to support reuse in Event-B.

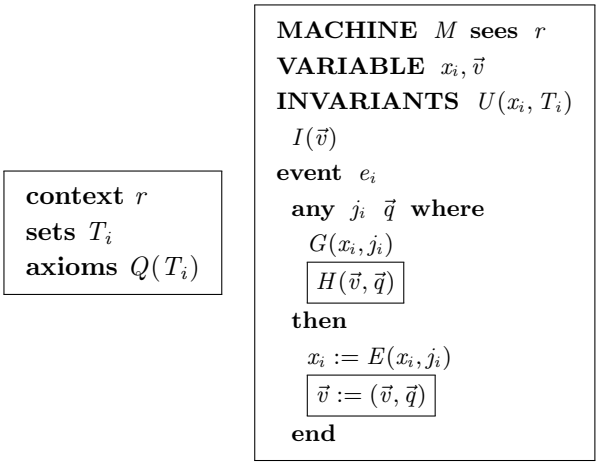
Assume that we have a pattern P consisting of a context x with a list of sets N_i and two machines, an abstract machine *p0* and its refinement *p1*. The abstract machine *p0* has a list of variables s_i, invariants $T(s_i, N_i)$, and evt_i that represents a list of events all of which operate on the same variables s_i. c_i in evt_i represents a list of parameters. Machine *p1* refines *p0* and has the variables s_i' that data refine s_i, the gluing invariants $GI(s_i, s_i')$, and the concrete events evt_i' that refine evt_i.

The pattern $P[\textbf{vars} : s_i, \textbf{sets} : N_i]$ has the following form:



We further assume that at some abstraction level of a particular specification M, we figure out that a suitable continuation of the development would be to reuse the pattern P. M contains a list of variables x_i, a list of variables $\vec{v}$, invariants $U(x_i, T_i)$ and $I(\vec{v})$ and a list of events e_i. j_i and $\vec{q}$ in e_i represent a list of parameters.

The model $M[\textbf{vars} : x_i, \vec{v}, \textbf{sets} : T_i]$ has the following form:



In order to resume the specification of the problem M using the pattern P, we need to check that pattern machine *p0* matches problem machine *m1*. This can be ensured by using one of the following approaches:

Approach 1: Syntactic matching: In the syntactic matching we need to check the following conditions:

- The suitable renaming of free variables of pattern formula f would lead to problem formula g. Meaning that, the elements of the abstract model of a pattern (variables, guards, actions, ..etc) are syntactically the same as the elements of the problem.
- Other elements in the problem (guards and actions) that are not matched with the pattern cannot modify a matched variable with the pattern.

Approach 2: Semantic matching

In the semantic matching, the pattern formula f matches problem formula g if suitable renaming of free variables of pattern formula f would lead to a formula that is semantically equivalent to problem formula g.

Following the first approach, we notice that variables s_i in the pattern are matched syntactically with variables x_i in the problem. The guards $G(x_i, p_i)$ and actions $x_i := E(x_i, p)$ of events e_i in the problem are the only guards and actions that modify x_i and they match with guards $G(s_i, c_i)$ and actions $s_i := E(s_i, c_i)$ of the pattern. Other guards and actions in the problem do not modify the variables x_i.

The refinement of the machine M can be obtained by combining generic instantiation technique and shared event composition technique. The generic instantiation technique is used to instantiate the refinement pattern including proofs and allows renaming of the pattern components (variables, types, and constants) to suitable names for the problem specification. Composition (shared-event style) enables the incorporation of the instantiated pattern in the development.

The structure of the new model that combines generic instantiation and shared-event composition techniques is an extension of Event-B that is not currently supported by Rodin. The outline of the new structure is shown as follows:

MACHINE *m2*
refines M
import $P.ip0[x_i, T_i]$
Events
e_i' **refines** e_i
Combines Events $ip0.evt_i[x_i] \parallel M.e_i \triangleq$ **any** $\vec{q} \ldots$ **end**

Machine *m2* is the created machine that refines the abstract machine M of the problem. The keyword ***import*** enables machines instantiation of a pattern or a set of patterns. An instance of the pattern machine is created by replacing (variables, types, and constants to specific names according to the instance). Here, the clause $P.ip0[x_i, T_i]$ is shorthand of generic instantiation. It demonstrates that the machine *p0* of the pattern P is instantiated using generic instantiation and the parameters (c_i, and M) of *p0* are replaced by the parameters (x_i, and T_i) in the new instance *ip0*. Events of the problem that modify a matched variable are refined based on the composition technique. The refined events are generated by combining two types of events which are:

- Selected event(s) of the created instance.
- Guards/actions that refer to unmatched variable of the event to be refined in the problem.

Therefore, events e_i of the problem are refined by combining evt_i of the instantiated machine (i.e. $ip0.evt_i[x_i]$: evt_i of the instance *ip0* where all occurances of c_i are replaced by x_i) with the unmatched guards/actions of event e_i in the problem.

The **main difference** between the syntactic matching and semantic matching is that, correctness of the matching in the syntactic matching is syntactically checked rather than proved. The modeller must check the model carefully before applying the pattern. No proof effort need to be performed, this is because generic instantiation preserves the correctness of the instantiated model [72] and also because shared event composition preserves refinement [74]. In the semantic matching, on the other hand, the modeller needs to prove that the instantiation of the abstract pattern is a correct refinement. In our case, we need to prove that events e_i of M are refined by the combination of the abstract evt_i with the residual guards and actions of e_i in case of the semantic matching.

7.2.1 Example of a Pattern and Its Instantiation into a Problem Refinement

The *Pattern* consists of two machines *p0* and *p1* that refines *p0*. The abstract model of the pattern *p0* consists of two disjoint sets a and b and two events *AddToA* and *RemoveFromA*. *AddToA* moves entities from set b to the set a whereas *RemoveFromA* moves entities from set a to the set b. In the data refinement of this model, we replace the two abstract variables a and b with *status* function. The new variable *status* is a total function from *Entity* to *STATUS*. *STATUS* is an enumerated type with distinct values *InA* and *OutA*. The refined events *AddToA*′ and *RemoveFromA*′ update the *status* function rather than modifying the *InA* and *OutA* variables.

The *Pattern* has the following form:

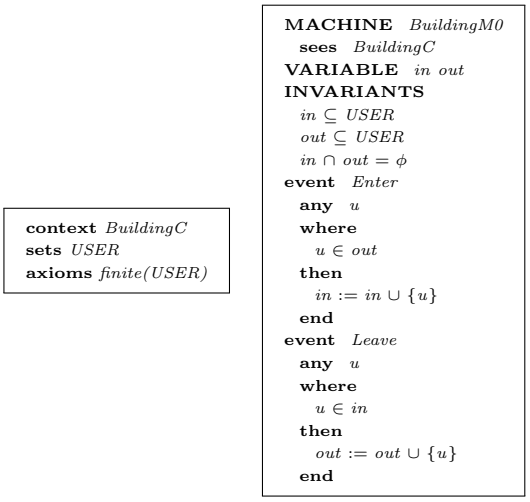
context *PatternC*
sets *Entity* *STATUS*
constants *InA* *OutA*
axioms *finite(Entity)*
partition(STATUS, {InA}, {OutA})

MACHINE *PatternM0*
sees *PatternC*
VARIABLE a b
INVARIANTS
$a \subseteq Entity$
$b \subseteq Entity$
$a \cap b = \phi$
event *AddToA*
any i
where
$i \in b$
then
$a := a \cup \{i\}$
end
event *RemoveFromA*
any i
where
$i \in a$
then
$b := b \cup \{i\}$
end

MACHINE *PatternM1*
sees *PatternC*
refines *PatternM0*
VARIABLE *status*
INVARIANTS
$status \in Entity \rightarrow STATUS$
event *AddToA*
refines *AddToA*
any i
where
$i \in Entity$
$status(i) = OutA$
then
$status(i) := InA$
end
event *RemoveFromA*
refines *RemoveFromA*
any i
where
$i \in Entity$
$status(i) = InA$
then
$status(i) := OutA$
end

The first model we need to refine using the *Pattern* is of a system for checking users in and out of a building. The abstract model has variables to represent the set of people who are in the building *in* and those that are outside the building *out*. It also has an invariant that shows that a user cannot be simultaneously inside and outside the building and two events to model users entering and leaving the building. *Enter* event allows a user outside the building to enter the building whereas *Leave* event allows a user inside the building to leave the building.

The *Building* model has the following form:



We notice that, by renaming the pattern variables a and b with the variables *in* and *out*, and the carrier set *Entity* with the carrier set *USER* we get a model that is syntactially same as *BuildingM0*. Therefore, we can apply the proposed approach to refine *BuildingM0* to *BuildingM1*.

The structure of the new model that refines the abstract model *BuildingM0* is shown as follows.

MACHINE *BuildingM1*
refines *BuildingM0*
import *Pattern.building*0[*in*, *out*, *USER*]
Events
Enter′ **refines** *Enter*
Combines Events *building*0.*AddToA*[*in*, *out*]
Leave′ **refines** *Leave*
Combines Events *building*0.*RemoveFromA*[*in*, *out*]

The second model specifies lift controller system. The abstract model of this speification contains variables *open* and *close*, an invariant that shows that floor door cannot be simultaneously *open* and *close* and events *OpenFloorDoor* and *CloseFloorDoor*.

The lift controller model has the following form:

context *LiftControllerC*
sets FloorDoor LIFTSTATUS
axioms finite(FloorDoor)
partition(LIFTSTATUS, {MOVING}, {STOPPED})

MACHINE *LiftControllerM0*
sees *LiftControllerC*
VARIABLE *open close LiftStatus*
INVARIANTS $open \subseteq FloorDoor$
$close \subseteq FloorDoor$
$open \cap close = \phi$
$LiftStatus \in LIFTSTATUS$
event *OpenFloorDoor*
any *fd*
where
$fd \in close$
$LiftStatus = STOPPED$
then
$open := open \cup \{fd\}$
end

event *CloseFloorDoor*
any *fd*
where
$fd \in open$
$LiftStatus = STOPPED$
then
$close := close \cup \{fd\}$
end

By considering the syntactic matching, we figure out that *LiftControllerM0* matches *PatternM0*. The replacement of the variables *a* and *b* of *Pattern* with variables *open* and *close* and the set *Entity* with *USER*, would lead to a model that is matched syntactically with *LiftControllerM0*. Thus, its possible to apply the proposed approach to refine the abstract model *LiftControllerM0* using the *Pattern*.

The structure of the new model that refines the abstract model *LiftControllerM0* is shown as follows.

MACHINE *LiftControllerM1*
refines *LiftControllerM0*
import *Pattern.liftController*0[*open*, *close*, *FloorDoor*]
Events
OpenFloorDoor′ **refines** *OpenFloorDoor*
Combines Events *liftController*0.*AddToA*[*open*, *close*] ‖
OpenFloorDoor ≙ **where** *LiftStatus=STOPPED* **end**
CloseFloorDoor′ **refines** *CloseFloorDoor*
Combines Events *liftController*0.*RemoveFromA*[*open*, *close*] ‖
CloseFloorDoor ≙ **where** *LiftStatus=STOPPED* **end**

It is important to point out how to deal with several cases while applying the patterns. If the model consists of several events that match with a single event in the pattern, then the pattern can be applied once to refine these events. The composition allows to combine the refined event of the instantiated pattern several times with several events of the problem. In addition, several variables can be data-refined at the same time if they have corresponding matched variables in the pattern. Finally, if there are several variables that need to be refined using a single or several patterns, then the patterns can be instantiated several times in one or more refinement levels to data-refine these variables, this case is described in details in section 7.5.

7.3 Theory of Circular Doubly Linked List

Linked lists are used for various applications especially applications which have to deal with an unknown number of objects. Linked lists are also used as a building block for many other data structures, such as stacks, queues and their variation. Before constructing a modelling pattern for circular doubly linked list (a kind of linked list), we need first to develop a theory for the circular doubly linked list data structure to overcome the lack of supportive operators on a circular doubly linked list in Event-B. The theory we developed includes operators for inserting an item to the end of a circular doubly linked list and operators for removing an item from a circular doubly linked list, as well as theorems and inference rules. The circular doubly linked list modelling pattern can be used to data refine "set" structure and makes it close to implementation.

Let us first take a moment to explain the circular doubly linked list structure. A circular doubly linked list consists of nodes, each of which contains a data and a pointer that references the next node and a pointer that references the previous node. The next pointer of the last node (tail node) is always pointing to the head node and the previous pointer of the head is pointing to the tail as illustrated in Figure 7.4.

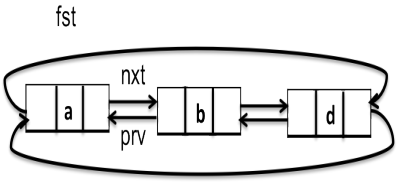


Figure 7.1: Circular doubly linked list structure.

We present two sets of operators; the first set of operators are used to insert an item at the end of a circular doubly linked list whereas the second set of operators are used to remove an item from a circular doubly linked list.

7.3.1 Operators for Inserting an Item to the End of a Circular Doubly Linked List

The operators presented here is used to insert an element at the end of a circular doubly linked list. Figure 8.2 illustrates how it works.

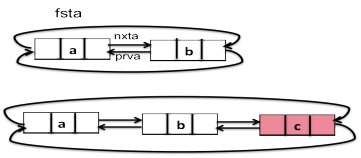


Figure 7.2: inserting an element at the end of the circular doubly linked list.

We need first to show how the structure of circular doubly linked is modelled using a set of nodes *nds*, a first node *fst*, a next pointer *nxt*, and a previous pointer *prv*. The definitions of these variables and the invariants are listed as follows.

$$
\begin{array}{ll}
inv1 & fst \in nds \cup \{null\} \\
inv2 & nxt \in nds \rightarrowtail\!\!\!\!\!\!\rightarrow nds \\
inv3 & prv = nxt^{-1} \\
inv4 & nds \neq \phi \Leftrightarrow fst \neq null \\
inv5 & nds = \varnothing \Leftrightarrow fst = null \\
inv6 & isCircular(nds, fst, nxt) \\
thm & card(nds) = 1 \Rightarrow nds = \{fst\} \wedge nxta(fst) = fst
\end{array}
$$

Inv6 states that all nodes are reachable from any node (reachability property). *isCircular* is predicate in the theory feature that is defined as follows $\forall\, n \cdot n \in nds \Rightarrow cls(nxt)[\{n\}] \cup \{n\} = nds$, where *cls* is transitive closure operator defined in the theory feature as follows: $cls(r) \triangleq fix(\lambda\, s \cdot s \in \mathbb{P}(S \times S) \mid r \cup (s;\ r))$. The explanation of the definition can be found in [4]. *Thm* states that nodes cannot point to themselves except in case where *fst* node is the only node in *nds*.

There are four important **well-definedness conditions** for the operators used to insert an element at the end of a circular doubly linked list which are:

- $finite(nds)$
- $fst \in nds \Leftrightarrow fst \neq null$
- $nds = \varnothing \Leftrightarrow fst = null$
- $isCircular(nds, fst, nxt)$

The operators used to insert a node to the end of a circular linked lists are three operators. The first operator named *getNxt_insertLast_cl* used to return the next pointer *nxt* after inserting a node *c* to the end of the circular linked list. The second operator named *getPrv_insertLast_cl* used to return the previous pointer after inserting a node *c* to the end of the circular linked list. The third operator *getFst_insertLast_cl* operator used to return the first node after inserting a node *c* to the end of the circular linked list.

The **definitions** of these operators are as follows:

$$
\begin{aligned}
getNxt\_insertLast\_cl(nds, fst, nxt, c) \triangleq nxt \leftarrow\!\!\!\!\!\!\leftarrow COND(fst = null, \{c \mapsto c\}, \\
\{nxt^{-1}(fst) \mapsto c\} \cup \{c \mapsto fst\})
\end{aligned}
$$

$$
getPrv\_insertLast\_cl(nds, fst, nxt, c) \triangleq getNxt\_insertLast\_cl(nds, fst, nxt, c)^{-1}
$$

$$getFst\_insertLast\_cl(nds, fst, c) \triangleq COND(fst = null, c, fst)$$

7.3.2 Operators for Removing an Item from a Circular Doubly Linked List

The following diagram illustrates the case where an element is removed from the end of the circular doubly linked list.

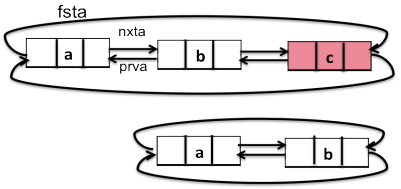


Figure 7.3: delete the last element from the circular doubly linked list.

The operators used to remove a node from a circular linked lists are three operators. The first operator named *getNxt_deleteNode_cl* used to return the next pointer after removing a node c from any position in the circular doubly linked list. The second operator named *getFst_deleteNode_cl* used to return the first node after removing a node c from any position in the circular doubly linked list. The third operator *getPrv_deleteNode_cl* used to return the previous pointer after removing a node c from any position in the circular doubly linked list.

The **definition** of these operators are as follows:

$$getNxt\_deleteNode\_cl(nds, fst, nxt, c) \triangleq COND(nxt(fst) = fst, \phi,$$
$$(\{c\} \lhd nxt) \lhd\!\!- \{nxt^{-1}(c) \mapsto nxt(c)\})$$

$$getFst\_deleteNode\_cl(nds, fst, nxt, c) \triangleq COND(c = fst,$$
$$getFst\_deleteFst\_cl(nds, fst, nxt), fst)$$

where *getFst_deleteFst_cl* returns the new first element after removing the head of the circular doubly linked list

$$getFst\_deleteFst\_cl(nds, fst, nxt) \triangleq COND(card(nds) > 1, nxt(fst), null))$$

Definition:

$$getPrv\_deleteNode\_cl(nds, fst, nxt, c) \triangleq getNxt\_deleteNode\_cl(nds, fst, nxt, c)^{-1}$$

Additional well-definedness condition for deletion node operators is necessary to check that the list must not be empty: $nds \neq \phi$.

7.3.2.1 Inference Rules

A sequent in Event-B is of the form $H \Rightarrow G$. The inference rules are used in Event-B proofs to show that a goal G is a consequence of the hypotheses H. When the infer clause of an inference rule matches the goal of a sequent, the proof is completed by considering the instantiated given clauses as the new sub-goals (backward reasoning). Alternatively, when given clauses of an inference rule matches the hypotheses of a sequent, the proof is completed by adding the instantiated infer clause as a hypothesis (forward reasoning). The following inference rules are added to allow the prover to proof that the theorem *thm* is sound.

insert_Last_inf

Given

$isCircular(nds, fst, nxt)$

$fst \in nds \cup \{null\}$

$nxt \in nds \rightarrowtail\!\!\!\!\!\rightarrow nds$

$n \in S \setminus nds$

Infer

$isCircular(nds \cup \{n\}, getFst\_insertLast\_cl(nds, fst, c), getNxt\_insertLast\_cl(nds, fst, nxt, n))$

delete_Node_inf

Given

$isCircular(nds, fst, nxt)$

$fst \in nds$

$nxt \in nds \rightarrowtail\!\!\!\!\!\rightarrow nds$

$n \in S \setminus nds$

Infer

$isCircular(nds \backslash \{n\}, getFst\_deleteNode\_cl(nds, fst, nxt, n), getNxt\_deleteNode\_cl(nds, fst, nxt, n))$

7.4 Circular Doubly Linked List Pattern

In this section we outline a pattern for refining operations on a set by operations on a circular doubly linked list. The circular doubly linked list pattern *CDLinkedList* consists of two machines, the abstract machine *p0* contains a set a defined as $a \subseteq T$, the concrete machine *p1* relates the abstract specification of the set a to a circular doubly linked list via data refinement.

The abstract machine *p0*: This level defines two abstract events. The *AddToA* event adds an element to a set a, whereas *RemoveFromA* event removes an element from the set a.

AddToA
any t
where
$grd1$ $t \in T \setminus \{a\}$
then
$act1$ $a := a \cup \{t\}$
end

RemoveFromA
any t
where
$grd1$ $t \in a$
then
$act1$ $a := a \setminus \{t\}$
end

The first machine *p1*:

This machine refines the abstract machine into a circular doubly linked list. Let us first take a moment to explain the circular doubly linked list structure. A circular doubly linked list consists of nodes, each of which contains a data and a pointer that references the next node and a pointer that references the previous node. The next pointer of the last node (tail node) is always pointing to the head node and the previous pointer of the head is pointing to the tail as illustrated in Figure 7.4.

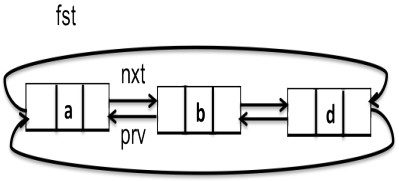


Figure 7.4: Circular doubly linked list structure.

In order to model this, pointer variables are introduced to replace the sets by circular linked list structure.

The pointer variables in this level are: *nxta*, *prva*, *fsta*. *nxta* represents the next pointer, *prva* represents the previous pointer, and *fsta* represents the first node in the list.

The definition of the variables and the invariants are listed as follows:

$inv1$ $fsta \in a \cup \{null\_a\}$
$inv2$ $nxta \in a \rightarrowtail\!\!\!\!\!\rightarrow a$
$inv3$ $prva = nxta^{-1}$
$inv4$ $fsta \in a \Leftrightarrow fsta \neq null\_a$
$inv5$ $a = \varnothing \Leftrightarrow fsta = null\_a$
$inv6$ $isCircular(a, fsta, nxta)$
thm $card(a) = 1 \Rightarrow a = \{fsta\} \wedge nxta(fsta) = fsta$

Two events that refine*AddToA* are introduced in this level. The first event *AddToStart*, adds an element to the beginning of the circular doubly linked list a.

The second one *AddToEnd*, adds an element to the end of the circular doubly linked list a.

Here is the specification of the *AddToEnd* event:

AddToEnd **refines** *AddToA*
any e
where
$grd1$ $e \in T \setminus a$
then
$act1$ $nxta := getNxt\_insertLast\_cl(a, fsta, nxta, e)$
$act2$ $prva := getPrv\_insertLast\_cl(a, fsta, nxta, e)$
$act3$ $fsta := getFst\_insertLast\_cl(a, fsta, e)$
$act4$ $a := a \cup \{e\}$
end

In addition, three events that refined *DeleteFromA* are introduced in this level. The first one *DeleteFst*, deletes the first element from the circular doubly linked list a. The second event *DeleteLst*, deletes the last element from the circular doubly linked list a. Finally, the third event *DeleteAny* deletes an arbitrary element from the circular doubly linked list a.

Here is the specification of the *DeleteAny* event:

DeleteAny **refines** *RemoveFromA*
any e
where
$grd1$ $e \in a$
then
$act1$ $nxta := getNxt\_deleteNode\_cl(a, fsta, nxta, e)$
$act2$ $prva := getPrv\_deleteNode\_cl(a, fsta, nxta, e)$
$act3$ $fsta := getFst\_deleteNode\_cl(a, fsta, nxta, e)$
$act4$ $a := a \setminus \{e\}$
end

7.5 Applying the Proposed Approach to Resume the Development of FreeRTOS Case Study

This section shows the application of the proposed approach to resume the development of FreeRTOS case study using *CDLinkedList* pattern .

The description of the case study is given in Chapter 3.

The variables of tasks in the abstract specification of task management case study are sets which need to be refined into circular doubly linked lists. Therefore, tasks sets: *ReadyTask*, *SuspendTask*, *DelayTask*, *TaskWaitingTermination*, *PendingTask*

need to be refined to circular doubly linked lists. The process of refining each set to circular doubly linked lists without having mechanism for reusing modelling patterns demands a significant modelling and proving efforts. The proposed approach assists in refining the development of FreeRTOS case study using only one modelling pattern: circular doubly linked list. In order to apply the *CDLinkedList* pattern to refine the *m1* development, we need to perform syntactic checking. We notice that, each variable of the abstract machine *m1*: *ReadyTask*, *DelayTask*, *SuspendTask*, *PendingTask*, *TaskWaitingTermination* is matched with variable *a* in the pattern. The syntactic checking need to be checked five times (once for each matched variable). For instance, variable *ReadyTask* is matched with variable *a* and the events that modify variable *ReadyTask* are also matched with the events that modify variable *a* in the pattern. Event *AddToReady* in *m1* is matched with *AddToA* in *p0* and event *AddToSuspend* in *m1* is matched with *RemoveFromA* in *p0*. The same scenario applies to other variables:*DelayTask*, *SuspendTask*, etc.

The structure of the new model that refines the abstract model *m1* is shown as follows.

MACHINE *m2*
refines *m1*
import $CDLinkedList.ready0[ReadyTask, TASK]$
$CDLinkedList.delay0[DelayTask, TASK]$
$CDLinkedList.suspend0[SuspendTask, TASK]$...
Events
AddToReady′ **refines** *AddToReady*
Combines Events *ready0.AddToEnd[Ready]* ‖ *delay0.DeleteAny[Delay]* ‖
AddToReady ≙ **where** *scheduler* = *RUNNING* **then** *skip* **end**
AddToSuspend′ **refines** *AddToSuspend*
Combines Events *suspend0.AddToEnd[Suspend]* ‖ *ready0.DeleteAny[Ready]* ‖
delay0.DeleteAny[Delay] ‖
AddToSuspend ≙ **any** *t* **where** $t \notin CurrentTask \wedge scheduler \neq NOT\_STARTED$ **then** *skip* **end** ...

The created model *m2* refines the abstract model *m1* of FreeRTOS specification. It instantiates machine *p0* of the *CDLinkedList* pattern five times. One instance for each set variable: *ReadyTask*, *SuspendTask*, *DelayTask*, *TaskWaitingTermination*, *PendingTask*. The refinement of *AddToReady* in the created model *m2* is generated by composing *AddToEnd* event of the instance of *ReadyTask* variable (*ready0*) and the *DeleteAny* event of the instance of *DelayTask* variable (*delay0*) and the unmatched guard *Scheduler=RUNNING* of *AddToReady* event. Similarly, the refinement of *AddToSuspend* event in the created model *m2* is generated by composing *AddToEnd* event of the instance of *SuspendTask* variable (*suspend0*), *DeleteAny* event of the instance of *ReadyTask* variable (*ready0*), *DeleteAny* event of the instance of *DelayTask* variable (*delay0*), and the unmatched guard $t \notin CurrentTask$ and $Scheduler \neq NOT\_STARTED$ of *AddToSuspend* event

The case study presented here is oversimplified. FreeRTOS makes use of about 8 task lists. It uses two delay lists for delay tasks; the first list *DelayTaskList* stores tasks that have not overflowed the current tick count and the second list *OverflowDelayTaskList* stores the tasks that have overflowed the current tick. It

also has two event lists to store tasks waiting for events which are *TaskWaitingToSendList* and *TaskWaitingToReceiveList*. *ReadyTaskList* and event lists in FreeRTOS are sorted based on priority, thus, a pattern that deals with prioritised tasks is required to data refine *ReadyTaskList* and event lists.

The overall POs of the pattern machine *p1* is 41 POs of which 17 were proved interactively. Reusing the pattern using the proposed approach saves proving efforts because POs that are originated from the pattern only need to be discharged once and not for all instantiations of the pattern.

7.6 Related Work

This section presents some works of supporting reuse in the area of formal method.

In Classical B [70, 2], reusing an existing machine in other development can be performed through the use of the clauses *INCLUDES*, *USES* and *IMPORTS*. *INCLUDES* is used to include a machine as part of another machine, *USES* clause is used to have a read access to a certain machine. The difference between *INCLUDES* and *USES* clauses is that *INCLUDES* has control to all relevant operations of the including machine but *USES* has only control to all the operations except the ones of the used machine [70]. *IMPORTS* clause, on the other hand, is used to import any number of abstract machines into the implementation machine [70]. Our approach differs from including/importing mechanisms in two ways: firstly, it is possible to incorporate refinement using our approach but this is not supported by including/importing mechanisms in Classical-B. Secondly, our approach allows the user to rename the pattern with specific names using generic instantiation technique while renaming mechanism in Classical-B annotate the pattern with certain attributes using prefixing mechanism.

Z formal method offers several operators of schema calculus to combine schemas such as conjunction operator that is used to join the declaration parts of schemes and the predicate parts of the schemes and disjunction operator that merges the declaration parts and disjoints the predicate parts [79]. It also possible to rename the components of a schema and moreover it is possible to use a generic schema as a pattern for a variety of schemes with different types [86]. The variables of the generic schemas are defined as polymorphic type using formal parameter X, hence, they can be instantiated later with any set [86]. Like Classical-B and in contrast to our approach, Z also allows only reusing specification and do not support reusing refinement.

In Event-B, Thai Son Hoang et al proposed an approach that supports reusing formal models as patterns in Event-B [39]. The approach is supported by tool and primarily depends on matching the specification of the pattern (variables and

events) with the problem. The steps required by this approach are: firstly, matching the problem and the specification. Then, checking the validity of the matching using syntactical checking mechanism. After that, renaming the variables and events of the pattern refinement. Finally incorporating the renamed refinement of the pattern to create the refinement of the problem. The main differences between our approach and [39] is that our presented approach combines existing techniques in Event-B (generic instantiation and composition) to support reuse whereas [39] approach provides different mechanisms to support reuse by performing renaming and incorporation mechanisms which are done automatically by a tool. Furthermore, before generating the problem refinement using [39] approach, the user is asked to enter the gluing invariant so the tool can check whether the variable in the development matches a disappearing variable in the pattern so it will also disappear or not.

7.7 Conclusions

Modelling patterns are a promising technique for supporting reuse in system development. They have the potential to reduce time, cost, proof effort and to improve the quality of the patterns through the review process.

We proposed an approach for supporting the process of reusing modelling patterns. Any repeated structure can be specified once as a modelling pattern and then reused whenever suitable throughout the combination of existing methodological approaches in Event-B (generic instantiation and composition). Generic instantiation technique is used to create an instance of a pattern allowing replacement of the pattern names (types, constants, variables, events) by names that suit the development at hand. Generic instantiation assures that proofs associated to a generic development remain valid in an instantiated development which helps to avoid unnecessary re-proof when reusing models. The composition technique, on the other hand, enables the integration of the instantiated patterns into a development. The flexibility offered by the combination of generic instantiation and composition facilitates the reusability of patterns. The structure of the new model allows to instantiate a pattern or set of patterns several times with several parameters in one or more refinement levels. The composition, on the other hand, allows events in the pattern or set of patterns to be matched to several events in the problem.

The proposed approach has been applied to refine the development of FreeRTOS case study and has been very effective in saving proof effort.

Further work is concerned with developing a tool to automate this approach. Another direction of further work is establishing a repository of modelling patterns

in order to attract more designers into depositing their own patterns to promote reuse of models.

Chapter 8

Guidelines for Modelling and Theory Development based on FreeRTOS Experiences

Drawing upon our experience with modelling FreeRTOS, this chapter presents a set of guidelines for modelling OS kernels for embedded real-time systems in Event-B and some tips for theory development. The presented modelling guidelines are intended to assist specifiers of real-time kernels with a set of modelling steps for the construction of formal models of real-time kernels. Each of these guidelines gives directions on how to model a certain aspect of RTOS kernels. The guidelines focus on the basic functionality of RTOS and represent the primary requirements of RTOS for an Event-B model. Design details that are RTOS-specific are left out from the guidelines. Design details of a specific-RTOS can be specified through a refinement of the abstract model driven by the guidelines. The identified modelling guidelines can be understood as a modelling pattern for the following RTOS features: task management, scheduling and context switch, interrupts, queue management, and memory management. The presented tips for theory development provide a guide to developing theories including managing complex theories and checking the validity of the developed operators.

8.1 Introduction

Engineering cookbooks and guidelines for modelling, refinement, and proofs are important in the context of formal methods. They can be used to systematise the modelling and refinement process and aid the proofs, therefore reducing the time and cost of the formal development of systems.

Both new specifiers and professionals can gain benefit from adopting such guidelines and modelling patterns for their specifications. The guidelines tell us how to

model a system most effectively and provide good examples and lessons that aid the formal development of several systems. The guidelines also save the specifier's time and make the formal methods approach more acceptable in industry. Moreover, these guidelines can shed light into the requirements of systems and draw attention towards some important properties of systems that might be ignored in the absence of the guidelines. The guidelines may also give an insight to the specifier regarding the requirements that should be modelled first and how the refinement steps can be organized; this is usually considered a source of difficulty in the process of modelling and refinement.

This chapter is divided into two parts, the first part provides a set of modelling guidelines for RTOS whereas the second part provides some tips for theory development. Although the guidelines provided in this chapter draw upon our experience with modelling FreeRTOS, it is believed that these guidelines can also be used for constructing formal models of different real-time operating system (RTOS) kernels. This is because various RTOS kernels share similar features and contain the same components [56]. Table 8.1 describes some of the basic features and concepts of four RTOSes: FreeRTOS [10], UCOS [58], eCos [31], and VxWorks [84].

Concept/Feature	FreeRTOS [10]	UCOS [58]	eCos [31]	VxWorks [84]
process TCB	yes	yes	yes	yes
Null process	yes	yes	yes	yes
process creation and termination	yes	yes	yes	yes
process priority	yes	yes	yes	yes
clock tick	yes	yes	yes	
process states	ready, running, blocked, suspended	dormant, ready, running waiting, ISR	running, sleeping, countsleep, suspended, creating, exited	ready, suspended, pended (blocked) and delayed
scheduler state	start the scheduler, lock the scheduler, and unlock the scheduler	lock the scheduler, and unlock the scheduler	start the scheduler, lock the scheduler, and unlock the scheduler	lock the scheduler, and unlock the scheduler
context switch	yes	yes	yes	yes
interrupts handling	yes	yes	yes	yes
priority manipulation	changes a process priority and returns the priority of a process	changes a process priority	changes a process priority	changes a process priority and returns the priority of a process
delay operation	yes	yes	yes	yes
scheduling	priority-based round robin scheduling	priority-based scheduling and round robin scheduling	priority based round robin and bitmap schedulers	priority-based scheduling and round robin scheduling
Queue creation and termination	yes	yes	yes	yes
process synchronization	queues, semaphores, mutexes	Semaphores, Message mailbox, Message queues, Tasks and Interrupt service routines (ISR)	message box, sema phore, queue	message queues, pipes, semaphores
waiting list for tasks waiting to retrieve/post message to a queue	yes	yes	yes	yes
MemoryManagement	memory allocation and deallocation	memory allocation and deallocation	memory allocation	memory allocation and deallocation

Table 8.1: General RTOS Concepts: FreeRTOS, UCOS, eCos, and VxWorks

The subsequent sections identify Event-B modelling guidelines for the general RTOS concepts summarised in Table 8.1. Section 8.2 to Section 8.7 identify the modelling concepts for task management, scheduler states, scheduling and context switching, interrupts, queue management, and memory management. Section 8.8 compares the presented modelling guidlines with Craige's models of operating system kernels. Section 8.9 presents some tips for theory development drawn from our experience with the theory extension.

8.2 Task Management

8.2.1 Process

A process is an independent thread of execution [10, 56]. RTOS can execute multiple processes concurrently [10, 56]. The processes appear to execute concurrently, however, the kernel interleaves the execution sequentially based on a specific scheduling algorithm [10, 56]. The term process is also called "thread" or "task" in some RTOSes.

To model a process, we define a new type in the context level *PROCESS* "carrier set", where each process within the kernel is an element of this set. We also add an axiom to indicate that the process set is finite. This is because the processor runs finite number of processes.

$$finite(PROCESS)$$

8.2.2 Process Table

A process table is a data structure consisting of a collection of elements to store the information of a process [56, 52]. Each process has its own control block that contains the process information such as the process id, the process priority, etc. A process table could consist of more than 10 elements. To deal with a process table, we introduce the following Event-B concepts:

Process set: A set of the possible processes available in the system.

Process elements functions: These functions map processes to the process elements, where each element has a total function.

Process creation event: An event used to create a process.

Formalising a process table is simple; process elements variables can be modelled by introducing a function for each element. For instance, let us use P to identify the set of the created processes, $P1, P2, ..., PN$, for process elements such as name, priority, etc.The process variables pe_1, pe_2,..., pe_n can be defined as follows:

$$pe_1 \in P \rightarrow P1,$$
$$pe_2 \in P \rightarrow P2 \ldots,$$
$$pe_n \in P \rightarrow PN$$

We use total functions because we assume that each process has $P1, P2, ..., PN$ elements. The functions corresponding to the process elements do not need to be introduced in one machine; some of the elements can be postponed until a later refinement depending on the features the specifier wants to model first and the features the specifier wants to postpone.

When a new process is created, the kernel instantiates the process block of the created process. To model this, we introduce a process creation event as follows:

$$\begin{aligned} \mathbf{PC} = \ & \mathbf{any}\ p \\ & \mathbf{where}\ p \in P \\ & \mathbf{then}\ pe1(p) := v1 \parallel pe2(p) := v2 \ldots \mathbf{end} \end{aligned}$$

A process creation event needs to be extended further in later refinement steps when a new element of a process is introduced.

The definition of a process table is similar to an approach to record in Event-B [32]. In fact, the carrier set P can be understood of as a record type. The attributes pe_1, pe_2,..., pe_n are defined using a projection function (function from P to some type $P1, P2, ..., PN$). **PC** event is used to modify the value of the attributes pe_1. It is possible to extend the record type P by adding more attributes in another refinement step.

8.2.3 Process Priority

In real-time kernels, each process has a priority as defined in the process block. Most RTOS imply a combination of base priority and active priority. The base priority is the original priority specified when the task is constructed. The active priority is the priority that can be possibly modified. One possible use of base and active priorities is in the priority inheritance protocol. A priority inheritance protocol takes place when a lower priority process blocks some higher priority processes; this problem is called a priority inversion. The priority inheritance protocol resolves this problem by raising the priority of the process that caused the problem against the highest priority process blocked by it to release the blocked processes. When the process that caused the problem releases the blocked processes, its priority then returns back to its base priority. For this reason, it is important to have a base priority that holds the original priority of the process and active priority that holds the priority that can be changed. To deal with this, we need to define two elements in the process block, one for active priority and one for base priority:

$$ActivePri \in P \rightarrow PRIORITY$$
$$BasePri \in P \rightarrow PRIORITY$$

We also need to introduce the following Event-B events that capture the most common operations related to the process priority- priority set *PriSet* and priority get *PriGet*:

$$\begin{aligned} PriSet = \ & \mathbf{any}\ p\ np \\ & \mathbf{where}\ p \in P\ \wedge\ \ np \in PRIORITY \wedge ActivePri(p) \neq np \\ & \mathbf{then}\ ActivePri(p) := np\ \mathbf{end} \\ PriGet = \ & \mathbf{any}\ p\ np! \\ & \mathbf{where}\ p \in P\ \wedge\ \ np! = ActivePri(p)\ \mathbf{end} \end{aligned}$$

We use the "**!**" convention to represent result parameters as shown in the *np!* parameter.

8.2.4 Process States

At any time, the process can be in any one state [10, 52]. There are different states of a process, for instance, in FreeRTOS a process can be in one of the following states: ready, suspended, blocked, and running.

State diagrams are always a useful way to show the transition of process states. Figure 8.1 shows a possible transitions among the states $RUNNING$, $READY$, $SUSPENDED$ *and* $BLOCKED$ during a process life.

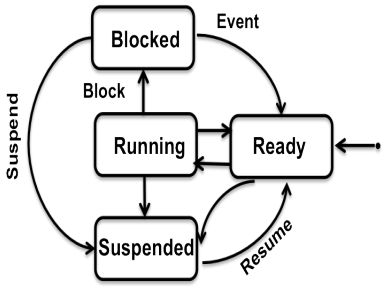


Figure 8.1: Process states.

To deal with process states, we identify the following Event-B modelling concepts:

Process states variables: These variables are defined for each possible state.

One way of defining process variables states in an Event-B model is to have a different set for each state; each set is disjoint from other sets indicating that a process must be in any one state at any given time. Each created process must belong to one of these sets. We use the partition operation to indicate that the collection of the sets is disjoint.

$$partition(P, READY, SUSPENDED, BLOCKED)$$

To check the state of a particular process, we check which set the process belongs to.

Running set, $RUNNING$ in our diagram corresponds to the runnable process and demands a special treatment. It is special process of ready set. It should be defined as a singleton set if we are dealing with one core processor.

$$RUNNING \subseteq READY$$
$$card(RUNNING) \leq 1$$

An alternative way to represent process states is to use a state function style as follows:

$$state \in Process \rightarrow STATE$$

A *process* is a carrier set that defines processes of the system and a *STATE* is a carrier set of all possible process states: partition(*STATE*,{*READY*}, {*SUSPENDED*}, {*BLOCKED*})

Process states event(s): These events are used to update the state of the process to a new one. Process states events model the transitions between the states. According to the Figure8.1, we have four process states events: *RunToReady, RunToBlock, RunToSuspend, ReadyToRun, ReadyToSuspend, BlockedToReady, BlockedToSuspend*, and *SuspendToReady*.

The formalism of *RunToReady* would be of the form:

$$\begin{aligned} RunToReady = \ & \textbf{any } p \\ & \textbf{when } \ p \in RUNNING \\ & \textbf{then } RUNNING := RUNNING \setminus \{p\} \parallel READY := READY \cup \{p\} \textbf{ end} \end{aligned}$$

The conditions around which the transitions is occured are considered as guards for the process states events.

In later refinement levels, process states variables "sets" are refined according to the appropriate data structure. For instance, ready state is usually implemented as a linked list for each priority. This level of detail can be left until data refinement levels. In data refinement levels sets are data refined to the appropriate data structure such arrays or linked lists depending on the data structure implementation adopted for the specified RTOS. For instance, the data structure for process states in FreeRTOS is a circular doubly linked list [10].

8.2.5 Null Process

A null or idle process is a special process that is run when there is no process available to run [10, 52]. A null process is permanently ready to run and is assigned to the lowest priority.

It is important process as the kernel needs always to execute a process or interrupt handler, and there might be a situation where there is no ready process available to run, in this case, the kernel switch control to the null process.

The definition of a null process is:

$$Null \in READY$$

8.2.6 Timing Behaviour

A scheduler interrupts at regular frequency to schedule tasks. An RTOS measures time using a tick count variable. A timer increments the tick count accurately.

Each time the tick is incremented the scheduler checks to see if a task needs to be woken. If this is the case, the scheduler executes the newly woken task.

An important operation used by many RTOSs is the *Delay* operation. It is used to suspend a process until a fixed time point in the future. The delay operation takes the process and the time as parameters and returns the process delayed until the specified delay time.

To deal with the timing behaviour, we identify the following Event-B modelling concepts:

Tick variable: A counter that measures the system time.

Tick length constant: A constant that defines the length of the tick depending on the hardware timer's design.

Sleep time function: A partial function that maps the processes to the sleep time. The partial function indicates that only some processes in the system have a delay time and not all processes.

TimeToWake function: The partial function describes the wake-up time of a delayed task.
The definitions of these variables can be written as follows:

$$Tick \in \mathbb{N}$$
$$Sleep \in P \pfun \mathbb{N}$$
$$PreviousTime \in P \pfun \mathbb{N}$$
$$TimeToWake \in BLOCKED \fun \mathbb{N}$$

where, *BLOCKED* identifies the set of delayed processes.

Since, all processes that have a wake-up time, also have sleep time, we add the following invariant:

$$dom(TimeToWake) \subseteq dom(Sleep)$$

IncrementTick: An event used to increment a tick counter with a strict accuracy.

$$\begin{aligned} IncrementTick = \ & \textbf{where } Tick \geq 0 \\ & \land\ TimeToWake \neq \varnothing \Rightarrow min(ran(TimeToWake)) > Tick \\ & \textbf{then } Tick := Tick + TickLength \textbf{ end} \end{aligned}$$

The guard, $TimeToWake \neq \varnothing \Rightarrow min(ran(TimeToWake)) > Tick$, ensures that all delayed processes have un-expired delay time. This is because each time a tick count is incremented, it must check if a delayed task needs to be woken.

delay: An event used to delay a process for a fixed time.

$$
\begin{aligned}
Delay = \ & \textbf{any } p \\
& \textbf{where } p \in RUNNING \wedge \ p \in dom(Sleep) \\
& \textbf{then } BLOCKED := BLOCKED \cup \{p\} \\
& \parallel \ TimeToWake(p) := Sleep(p) + Tick \\
& \parallel RUNNING := RUNNING \setminus \{p\} \\
\textbf{end}
\end{aligned}
$$

8.3 Scheduler States

The scheduler can exist in one of the different states [10, 52]. For instance, the possible scheduler states in FreeRTOS are: not-started, running, and suspended.

To deal with scheduler states, we have:

Scheduler states constants: Constants represent scheduler states.

Scheduler status variable: A variable whose value determines the state of the scheduler.

Scheduler events: These events are used to update the value of the scheduler variable to a new one.

Assuming that we have a scheduler with three states, s_1, s_2 and s_3. We use ss to represent the scheduler variable, s_1, s_2 and s_3 are constants corresponding to the scheduler states ss that are defined in the context level and SE is a scheduler event. To define scheduler states ss a partition operation can be used as follows:

$$\textbf{Partition}(ss, s_1, s_2, s_3)$$

A scheduler event would be of the form:

$$
\begin{aligned}
SE = \ & \textbf{when } \ ss = s_1 \\
& \textbf{then } ss := s_2 \ \textbf{end}
\end{aligned}
$$

The conditions under which the transitions occur are considered to be the guards for the scheduler states events.

There might be more than one event that changes the value of the scheduler variable and would take the aforementioned form.

Many operations in the RTOS occur when the scheduler is running or suspended. For instance, the *IncrementTick* event described in Section 8.2.6 must occur in FreeRTOS when the scheduler runs; therefore the *IncrementTick* event is extended by adding the guard $ss = s_1$ where s_1 denotes the scheduler running state.

8.4 Scheduling and Context Switching

8.4.1 Context Switch

A context switch occurs when the scheduler switches from one process to another. It is the technique of replacing the process being executed by another process that is ready to run [56, 52]. A context switch takes place when the kernel decides to switch control to another process; so it exchanges the registers' contents by saving the information of the current executing process to resume its execution later and loading the information of the new process to the processor registers.

There are many situations in which a context switch is performed. The most common context switch performed is the tick interrupt. At every tick, the scheduler checks if a new process should run. If such a process is found, the scheduler will save the current process information to resume it later and execute the new one [10].

The kernel schedule processes are based on a specific algorithm. There are different scheduling algorithms; a preemptive priority scheduling mechanism is one of the most common scheduling algorithms. In this algorithm, each process has a priority, and the higher readied priority process runs first. The preemptive priority scheduling algorithm can augment round robin algorithm by giving processes of the same priority an equal share of processor time.

To deal with a context switch, we introduce the following modelling concepts:

Context function: A total function that maps each task to its context (attribute in the process table).

$$ProcessContext \in P \rightarrow CONTEXT$$

Current context: The physical context that captures the current process that holds the processor.

$$physicalContext \subseteq CONTEXT$$
$$card(physicalContext) \leq 1$$

Context switch event(s): Event(s) that replace the current process by another one.

These guidelines deal with the preemptive priority scheduling algorithm. At every tick, a context switch is performed when there is a process with a priority higher than or with equal priority of the current one being executed (equal in this context allows time slicing between processes with same priorities). The guards or the conditions that constrain the context switch *CS* event rely on checking the priority of the readied processes. The context switch event can be of the form:

$$
\begin{aligned}
\mathit{Context\_Switch} = \ & \textbf{any } p \ cp \ ph \\
& \textbf{where } \ p \in \mathit{READY} \wedge cp \in \mathit{RUNNING} \wedge \ \mathit{activePri}(p) \geq \mathit{activePri}(cp) \\
& \wedge \ ph \in \mathit{physicalContext} \\
& \textbf{then } \mathit{RUNNING} := \{p\} \parallel \ \ \mathit{physicalContext} := \{\mathit{ProcessContext}(p)\} \parallel \\
& \mathit{ProcessContext}(cp) := ph \textbf{ end}
\end{aligned}
$$

The actual context switch is done by saving the state of the current context in the process context and setting the current context for the new running process.

Context switch flag: A boolean flag that is set to true to indicate the need of performing a context switch. We use *csFlg* for a context switch flag.

There are different cases in which a context switch may perform. For instance, context switch might perform when a certain process transmits to ready state or when a new process is created.This is because it is possible for these situations to introduce a process with a priority higher than the priority of the current executing one in which case the context switch should perform. Therefore, for every event that may require a context switch, we need to add an additional action that sets the context switch flag to true. And we add an additional guard to ensure that the context switch flag is true in the context switch *CS* event, and that finally the context switch flag is assigned back to false in the *CS* event. Thus, we extend *CS* event to be:

$$
\begin{aligned}
\mathit{Context\_Switch} = \ & \textbf{any } p \ cp \ ph \\
& \textbf{where } \ p \in \mathit{READY} \wedge cp \in \mathit{RUNNING} \wedge \ \mathit{activePri}(p) \geq \mathit{activePri}(cp) \\
& \wedge \ ph \in \mathit{physicalContext} \wedge \mathit{csFlg}{=}\mathit{TRUE} \\
& \textbf{then } \mathit{RUNNING} := p \parallel \mathit{physicalContext} := \{\mathit{ProcessContext}(p)\} \\
& \parallel \mathit{ProcessContext}(cp) := ph \parallel \mathit{csFlg}{:=}\mathit{FALSE} \textbf{ end}
\end{aligned}
$$

In order to prevent any event to be enabled when calling the context switch event, we add an additional guard, *csFlg=FALSE* to any events that might get enabled and affect the execution of the context switch.

Another important issue raised here is what if an event calls a context switch and sets the *csFlg* to true but not all the guards of the context switch *CS* event are satisfied. This situation occurs when the event that called a context switch does not introduce a readied process with higher or equal priority than the executing one. This means that the context switch event is not enabled and consequently the value of the *csFlg* is always true which leads to a deadlock. To prevent this situation, we suggest having two versions of context switch events, the first one performs the context switch if all the guards of *CS* event are satisfied, the second one, however, triggers only to set *csFlg* back to false when the guards of the

the context switch are not satisfied. This is to prevent a possible deadlock from occurring.Therefore, the revised CS events are:

$$
\begin{aligned}
Context\_Switch1 = \ & \textbf{any } p\ cp \\
& \textbf{where } p \in READY \wedge cp \in RUNNING \wedge \ activePri(p) \geq activePri(cp) \\
& \wedge\ ph \in physicalContext \wedge csFlg{=}TRUE \\
& \textbf{then } RUNNING := p \parallel \ physicalContext := \{ProcessContext(p)\} \\
& \parallel ProcessContext(cp) := ph \parallel \ csFlg := FALSE \textbf{ end} \\
Context\_Switch2 = \ & \textbf{any } cp \\
& \textbf{where } cp \in RUNNING \wedge \ max(activePri[READY]) < activePri(cp) \\
& \wedge\ csFlg := TRUE \\
& \textbf{then } csFlg := FALSE \textbf{ end}
\end{aligned}
$$

The above context switch specification is abstract and only represents the general operations performed in any context switching. Context switching is a hardware-dependant operation. In order to model context switch in details, the registers of the target processor needs to be modelled along with the other specific context switch details for the targeted architecture.

8.5 Interrupts and Interrupt Service Routines

Interrupts are hardware mechanisms used to inform the kernel that an event has occurred [10, 52]. A process can be interrupted by external interrupts raised by peripherals or software interrupts raised by executing a particular instruction. Interrupts are handled by ISRs which are stored in interrupt vector table. When an interrupt occurs, the kernel saves the current context of the process being interrupted and jumps to the ISR to handle that interrupt. After processing the ISR, the kernel returns to the process level and resumes the process that was interrupted [10, 52].

Interrupts may ready a blocked process for an external device when an event has occurred, so, the kernel might execute a different process before completing the preempted one.

To deal with interrupts, we introduce the following modelling concepts:

Interrupt data type: A new data type for interrupts. We will use *INTERRUPT* "carrier set" as interrupts data type. The scheduler runs a process or interrupt. So, we can define processes and interrupts in terms of one carrier set (e.g. *Obj*) and then distinguish between them by adding the following invariant:

$$P \cap INTERRUPT = \phi$$

Interrupt variable: A set of the raised system interrupts.

$$Interrupts \subseteq INTERRUPT$$

Interrupt handler function: A function that maps each interrupt to its ISR. We will use an *interrupt handler* (a total function that maps an interrupt to its corresponding interrupt service routine).

$$ISR \in Interrupts \rightarrow INTERRUPT\_HANDLER$$

Where *INTERRUPT_HANDLER* is a carrier set defined in the context.

Current interrupt: A variable that stores the current executing interrupt.

$$currentISR \subseteq ran(ISR)$$
$$card(currentISR) \leq 1$$

The invariant $card(currentISR) \leq 1$ is added in case of single kernel.

Handle interrupt event: An event used to handle interrupts.

$$\begin{aligned} Handle\_Interrupt = \ & \textbf{any } int \\ & \textbf{where } int \in Interrupts \\ & \textbf{then } currentISR := \{ISR(int)\} \textbf{ end} \end{aligned}$$

Complete interrupt event: An event used to discard completed interrupts

$$\begin{aligned} Complete\_Interrupt = \ & \textbf{any } int \\ & \textbf{where } int \in Interrupts \ \ \wedge currentISR = \{ISR(int)\} \\ & \textbf{then } currentISR := \phi \parallel \ Interrupts := Interrupts \setminus \{int\} \\ & \parallel \ ISR := \{int\} \lhd\!\!\!- ISR \textbf{ end} \end{aligned}$$

For later refinement steps, more features can be introduced to be used from within an ISR and we need to make a distinction between the features used by process level and the features used by interrupt level. For instance, if we were to introduce an event for sending an item to a queue that is used by ISRs and processes, we need to introduce two events: *QueueSendFromISR* is an event used by ISRs and *QueueSendProcess* is an event used by processes.

Interrupts have priority, an interrupt with lower priority can be interrupted by other interrupt of a higher priority. To deal with this, we identify the following Event-B modelling concepts:

Interrupt priority variable: A function that maps each interrupt to its priority

$$InterruptPriority \in Interrupts \rightarrow INTERRUPT\_PRIORITY$$

where *INTERRUPT_PRIORITY* is a constant set defined in the context level.

The event *Handle_Interrupt* and *Complete_Interrupt* can be extended as follows:

$$
\begin{aligned}
\mathit{Handle\_Interrupt} = \ & \textbf{any } \mathit{int}\ \ c\ \ \mathit{cint} \\
& \textbf{where } \mathit{int} \in \mathit{Interrupts} \\
& (\ c \in \mathit{currentISR} \wedge \mathit{cint} = \mathit{ISR}^{-1}(c) \\
& \wedge \mathit{InterruptPriority}(\mathit{int}) \geq \mathit{InterruptPriority}(\mathit{cint})) \vee \mathit{currentISR} = \varnothing \\
& \textbf{then } \mathit{currentISR} := \{\mathit{ISR}(\mathit{int})\} \textbf{ end} \\
\mathit{Complete\_Interrupt} = \ & \textbf{any } \mathit{int} \\
& \textbf{where } \mathit{int} \in \mathit{Interrupts}\ \ \wedge \mathit{currentISR} = \{\mathit{ISR}(\mathit{int})\} \\
& \textbf{then } \mathit{currentISR} := \phi \parallel\ \ \mathit{Interrupts} := \mathit{Interrupts} \setminus \{\mathit{int}\} \\
& \parallel\ \ \mathit{ISR} := \{\mathit{int}\} \lhd\!\!\!- \mathit{ISR} \\
& \parallel \mathit{InterruptPriority} := \{\mathit{int}\} \lhd\!\!\!- \mathit{InterruptPriority} \textbf{ end}
\end{aligned}
$$

The processor always gives priority to execute interrupts over tasks; the ISR must complete its execution without being interrupted by tasks. When the ISR is completed, the kernel dispatches the correct task. To deal with this, we identify the following Event-B modelling concepts:

Interrupt context: A function that maps each interrupt to its context.

$$\mathit{InterruptContext} \in \mathit{Interrupts} \rightarrow \mathit{CONTEXT}$$

In order to separate a process context from an interrupt context; we add the following invariant:

$$\mathit{ran}(\mathit{InterruptContext}) \cap \mathit{ran}(\mathit{ProcessContext}) := \phi$$

The *Handle_Interrupt* and *Complete_Interrupt* are extended as follows:

$$
\begin{aligned}
\mathit{Handle\_Interrupt} = \ & \textbf{any } \mathit{int}\ c\ \mathit{cint}\ \mathit{ph} \\
& \textbf{where } \mathit{int} \in \mathit{interrupts} \\
& ((\ c \in \mathit{currentISR} \wedge \mathit{cint} = \mathit{ISR}^{-1}(c) \\
& \wedge \mathit{InterruptPriority}(\mathit{int}) \geq \mathit{InterruptPriority}(\mathit{cint})) \\
& \vee \mathit{currentISR} = \varnothing) \wedge \mathit{ph} \in \mathit{physicalContext} \\
& \textbf{then } \mathit{currentISR} := \{\mathit{ISR}(\mathit{int})\} \\
& \parallel\ \ \mathit{physicalContext} := \{\mathit{InterruptContext}(\mathit{int})\} \\
& \parallel \mathit{InterruptContext}(c) := \mathit{ph} \textbf{ end} \\
\mathit{Complete\_Interrupt} = \ & \textbf{any } \mathit{int} \\
& \textbf{where } \mathit{int} \in \mathit{Interrupts}\ \ \wedge \mathit{currentISR} = \{\mathit{ISR}(\mathit{int})\} \\
& \textbf{then } \mathit{currentISR} := \phi \parallel\ \ \mathit{Interrupts} := \mathit{Interrupts} \setminus \{\mathit{int}\} \\
& \parallel\ \ \mathit{ISR} := \{\mathit{int}\} \lhd\!\!\!- \mathit{ISR} \\
& \parallel \mathit{InterruptPriority} := \{\mathit{int}\} \lhd\!\!\!- \mathit{InterruptPriority} \\
& \parallel\ \ \mathit{InterruptContext} := \{\mathit{int}\} \lhd\!\!\!- \mathit{InterruptContext} \\
& \parallel\ \ \mathit{physicalContext} := \phi \textbf{ end}
\end{aligned}
$$

An extra guard, the $currentISR = \phi$ is needed in context switch events to prevent any context switch while there is an ISR running.

Timer interrupt given in Section 8.2.6 is a perfect example of an interrupt. In every tick, the tick-ISR represented in *IncrementTick* event wakes up the blocked process that has an expired delay time. If the woken process has a priority higher than the current process, the ISR then will return control to the higher priority process.

8.6 Queue Management

Processes often need to communicate. There are several ways of communications that most RTOSs offer such as queues and semaphores. Queues are the primary object of process-process communications. Queues can be used to send messages between processes. Semaphores are a special kind of queue which are usually used for mutual exclusion and synchronisation (communication between processes and ISRs). Semaphores can be binary or counting. Binary semaphores are queues of length one, whereas counting semaphores are queues of length greater than one.

To define a queue, we define a new type in the context level $QUEUE$ "carrier set" where each created queue is an element of this set.

$$AllQueue \subseteq QUEUE$$

AllQueue identifies all possible created queues in the system.

We also add the following axiom to indicate that the queue set is finite

$$finite(QUEUE)$$

A queue has a length that defines the maximum number of messages the queue can hold.

$$Qlength \in AllQueue \rightarrow \mathbb{N}$$

In order to identify the message (data) that is stored in the queue, we define the following variable:

$$QMessage \in DATA \nrightarrow AllQueue$$

We use a partial function because the queue can be empty.

In order to identify the message (data) that is sent by an object *Process* or *ISR*, we define the following variables:

$$PMessage \in DATA \nrightarrow P$$
$$ISRMessage \in DATA \nrightarrow ISR$$

Again, we use a partial function since some processes or ISR have no data to send.

The following invariant is added to show that a process message is different from an ISR message:

$$dom(PMessage) \cap dom(ISRMessage) = \phi$$

Queue messages are usually of fixed size. To define this we add the following constraint invariant:

$$MessageSize \in AllQueue \rightarrow \mathbb{N}$$

Since there are different types of queues such as general queues, binary semaphore, counting semaphores, we can define them as follows:

$$partition(AllQueue, queue, binarySemaphore, countingSemaphore)$$

To define the length of each queue kind, we define the following:

$$\forall q \cdot q \in queue \Rightarrow QLength(q) > 1$$
$$\forall q \cdot q \in binarySemaphore \Rightarrow QLength(q) = 1$$
$$\forall q \cdot q \in countingSemaphore \Rightarrow QLength(q) > 1$$

There are three important operations supported by the queues. A queue creates an operation that is used to create a queue. Enqueue or process send operation that is used to send data by a process to another and a dequeue operation or process receive that is used by the other process to receive the data.

To deal with the queue creation operation, the queue send operation and the queue receive operation, we introduce the following Event-B concepts:

$Queue\_Create =$ **any** q
where $q \in QUEUE \setminus AllQueue$
then $AllQueue := AllQueue \cup \{q\} \parallel Qtype := Qtype \cup \{q\} \ldots$ **end**

$Qtype$ can be replaced by one of the variables: $queue$, $binarySemaphore$, or $countingSemaphore$. For each queue type, there should be one queue creation event.

The queue send event can be of the following form:

$Queue\_Send =$ **any** $p\ q\ m$
where $p \in Obj \ \wedge q \in Qtype \ \wedge card(QMessage^{-1}\{q\}) < Qlength(q)$
$\wedge\ m \in ObjMessage \setminus dom(QMessage)$
then $QMessage := QMessage \cup \{m \mapsto q\} \parallel ObjMessage := \{m\} \lhd\!\!\!- ObjMessage$ **end**

The guard, $card(QMessage^{-1}\{q\}) < Qlength(q)$ ensures that there is a room in the queue, so the running process can send a message to the queue q. Obj refers to the object that has sent a message to a queue, so it needs to be replaced by P or ISR.

ObjMessage denotes the message of the object, so it needs to be replaced by *PMessage* or *ISRMessage*. *Obj*, *Qtype* and *ObjMessage* need to be replaced by appropriately defined variables to show the object-sender and the type of the queue the object wants to send the message to.

The queue receive event can be of the following form:

$$
\begin{aligned}
Queue\_Receive = \ & \textbf{any } p\ q\ m \\
& \textbf{where } p \in Obj\ \wedge q \in Qtype\ \wedge m \in QMessage^{-1}[\{q\}] \\
& \textbf{then } QMessage := \{m\} \lhd\!\!\!- QMessage \parallel ObjMessage := ObjMessage \cup \{m \mapsto o\} \\
& \textbf{end}
\end{aligned}
$$

The guard, $m \in QMessage^{-1}[\{q\}]$ ensures that the queue is not empty, so that the process can receive a message from a queue.

Similarly, *Obj*, *Qtype* and *ObjMessage* need to be replaced by appropriately defined variables to show the object-receiver and the type of the queue the object wants to receive the message from.

An important point to note is that enqueue and dequeue operations usually require to be done in FIFO order. Therefore, one way to deal with this is to define an abstract sequence instead of a set that allows the queue to store messages or refine a set into a sequence to deal with this aspect. It is also possible that a queue message set is refined by a linked list to maintain the order between messages.

8.6.1 Waiting Messages

If the queue is full, the sending process will not be successful and has to be blocked and wait before sending its message until the queue has a room to receive the message. Similarly, if the queue is empty, the receiving task will not be successful and has to wait and block until a message is arrived.

To deal with this, we introduce the following Event-B concepts:

waiting to send queue: A function that store processes that are blocked to send item to a queue:

$$WaitingToSend \in P \nrightarrow AllQueue$$

waiting to receive queue: A function that store processes that are blocked to receive item from a queue:

$$WaitingToReceive \in P \nrightarrow AllQueue$$

waiting messages events: Two events are defined to add blocked processes to the appropriate waiting queues: waiting to send queue or waiting to receive queue and two events are defined to remove a process from waiting queues: waiting to send queue or waiting to receive queue.

The formalization of the operation of blocking a process waiting message events can be expressed as follows:

$$
\begin{array}{l}
Add\_ToWToS = \ \textbf{any}\ q\ c \\
\qquad\qquad \textbf{where}\ q \in Qtype\ \wedge c \in RUNNING\ \wedge QLength(q) = card(QMessage^{-1}[\{q\}]) \\
\qquad\qquad \textbf{then}\ WaitingToSend := WaitingToSend \cup \{c \mapsto q\} \\
\qquad \textbf{end}
\end{array}
$$

The guard $QLength(q) = card(QMessage^{-1}[\{q\}])$ ensures that the queue q is full, so that the process c must be blocked.

$$
\begin{array}{l}
Add\_ToWToR = \ \textbf{any}\ q\ c \\
\qquad\qquad \textbf{where}\ q \in Qtype\ \wedge c \in RUNNING\ \wedge card(QMessage^{-1}[\{q\}]) = 0 \\
\qquad\qquad \textbf{then}\ WaitingToReceive := WaitingToReceive \cup \{c \mapsto q\} \\
\qquad \textbf{end}
\end{array}
$$

The guard $card(QMessage^{-1}[\{q\}]) = 0$ ensures that the queue q is empty and so the process c must be blocked.

$$
\begin{array}{l}
Remove\_FromWToS = \ \textbf{any}\ q\ t \\
\qquad\qquad\qquad \textbf{where}\ q \in Qtype\ \wedge t \in WaitingToSend^{-1}[\{q\}] \\
\qquad\qquad\qquad \wedge\ card(QMessage^{-1}[\{q\}]) < QLength(q) \\
\qquad\qquad\qquad \textbf{then}\ WaitingToSend := \{t\} \lhd\!\!\!- WaitingToSend \\
\qquad\qquad \textbf{end}
\end{array}
$$

$$
\begin{array}{l}
Remove\_FromWToR = \ \textbf{any}\ q\ t \\
\qquad\qquad\qquad \textbf{where}\ q \in Qtype\ \wedge t \in WaitingToReceive^{-1}[\{q\}] \\
\qquad\qquad\qquad \wedge\ card(QMessage^{-1}[\{q\}]) > 0 \\
\qquad\qquad\qquad \textbf{then}\ WaitingToReceive := \{t\} \lhd\!\!\!- WaitingToReceive \\
\qquad\qquad \textbf{end}
\end{array}
$$

Locking queue mechanism and critical sections vary so much from one RTOS to another, it is hard to give universal guidance about how a locking mechanism is used by several RTOSes in any given situation. Therefore, this part has been left out from the presented guidelines.

8.7 Memory Management

The RTOS provides memory management techniques to assign memory to objects. The memory usually is divided into fixed size memory blocks which can be requested by objects. Each object in the system (process, queue, semaphore, ...etc) is assigned a private memory space.

To deal with memory management techniques, we introduce the following Event-B concepts:

block set: A set of all memory blocks:

$$Blocks \subseteq BLOCK$$

block-size function: A function that defines the size of each block:

$$BlockSize \in Blocks \rightarrow \mathbb{N}$$

blockAddr function: A function that defines the start address of each block:

$$BlockAddr \in Blocks \rightarrow ADDR$$

where, $ADDR$ is a constant set defined as: $ADDR = StartAddress..EndAddress$. $StartAddress$ and $EndAddress$ are constants representing the start and the end addresses of the heap structure and are defined in terms of $\mathbb{N}$.

object: A set of objects in the system:

$$object \in Blocks \rightarrow OBJECT$$

The memory blocks can be further divided into two sets, allocated blocks and free blocks as follows:

$$partition(Blocks, FreeBlocks, AllocBlocks)$$

To ensure that blocks are consecutive and guarantee that a block can not be allocated for two distinct objects, we introduce the following invariant:

$$\forall b1, b2 \cdot b1 \in Blocks \land b2 \in Blocks \land b1 \neq b2 \Rightarrow BlockAddr(b1)\,..\,(BlockAddr(b1) + BlockSize(b1) - 1) \cap BlockAddr(b2)\,..\,(BlockAddr(b2) + BlockSize(b2) - 1) = \varnothing$$

In order to model the process of memory allocation and deallocation, we introduce *Alloc1,2* events to allocate memory and assign it to an object and *Free* event to free memory blocks.

$$\begin{aligned} Alloc1 = \ & \textbf{any } s\ b\ c\ o \\ & \textbf{where } s \in \mathbb{N} \ \land\ b \in FreeBlocks \ \land\ s > 0 \\ & \land\ BlockSize(b){=}s \ \land\ c \in BLOCK \setminus Blocks \land\ o \in OBJECT \setminus ran(object) \\ & \textbf{then } FreeBlocks := FreeBlocks \setminus \{b\} \parallel AllocBlocks := AllocBlocks \cup \{b\} \\ & \parallel\ \ object := object \cup \{b \mapsto o\} \textbf{ end} \end{aligned}$$

Alloc1 is used to allocate enough memory to be assigned to the object. The size of the memory is equal to the size requested by the object. However, if all blocks are not of adequate size to the requested one, the following event can be used to divide the free memory block into two blocks. The first block is of equal size as

requested and is used to allocate to the object. The second block is added to the set of free blocks. This allocation process is know as the best-fit algorithm.

$$
\begin{aligned}
Alloc2 = \ & \textbf{any } s\ b\ c\ o\ k \\
& \textbf{where } s \in \mathbb{N} \ \wedge \ b \in FreeBlocks \ \wedge \ s > 0 \\
& \wedge\, BlockSize(b) > s \ \wedge (\forall\, k.k \in BlockSize[FreeBlocks] \wedge k < BlockSize(b) \Rightarrow k < s) \\
& c \in BLOCK \setminus Blocks \wedge \ o \in OBJECT \setminus ran(object) \\
& \textbf{then } BlockAddr := BlockAddr \lhd\!\!\!- \{b \mapsto BlockAddr(b) + s, c \mapsto BlockAddr(b)\} \\
& \parallel BlockSize := BlockSize \lhd\!\!\!- \{b \mapsto BlockSize(b) - s, c \mapsto s\} \\
& \parallel AllocBlocks := AllocBlocks \cup \{c\} \parallel object := object \cup \{c \mapsto o\} \textbf{ end}
\end{aligned}
$$

The guard $\forall\, k.k \in BlockSize[\{FreeBlocks\}] \wedge k < BlockSize(b) \Rightarrow k < s$ guarantees that the free-block b is the best-fit free block.

To free an allocated memory block, the following event is introduced:

$$
\begin{aligned}
Free = \ & \textbf{any } b \\
& \textbf{where } b \in AllocBlocks \\
& \textbf{then } FreeBlocks := FreeBlocks \cup \{b\} \parallel AllocBlocks := AllocBlocks \setminus \{b\} \\
& \textbf{end}
\end{aligned}
$$

The problem of fragmentation occurs when part of allocated blocks are unused. The fact that the system allocated memory is more than is requested can be dealt with in an abstract form. However, we have left this topic from the guidelines as FreeRTOS do not deal with fragmentation problems.

8.8 Comparison

This section compares our guidelines to Craig's models of operating system kernels. Craig carried out his development in Z, and Object Z with some CCS (Calculus of Communicating Systems) whereas we use Event-B formal method to develop our guidelines.

In our modelling guidelines, the main abstract data structure is a"set". Sets are simple data structures that can be easily refined to more complex data structures, such as sequence and linked lists. The approach we presented in Chapter 7 assists in reusing data refinement patterns to resume the development of our the abstract models' "sets". This way we save more time and proof efforts while building FreeRTOS models. Craig [23, 24] does not adopt any approach that helps to reuse data refinement models. He rather focuses on his second book [24] to carry out the refinement of the abstract models mentioned in the first book [23].

Our guidelines majorly cover the common concepts of any RTOS, such as process tables, queues, semaphores and memory. Craig's models, however, are richer and

cover different concepts that can be found in some complex kernels, such as virtual storage.

Some of the definitions and data structures used in Craig's models are different from the definitions we adopted in the presented guidelines. Here we are going to compare the definition of some of the modelling concepts of our guidelines and Craig's models for three aspects: process management, queue management, and memory management.

The definition of process management in Craig's models is similar to the one presented in the guidelines. Craig defines processes as set of process names and the attributes of the process are mappings from process identifiers to the various attribute types. Process status is defined as a function from process names to process state set and he also defines a number of operations to change a process from a state to another, such as: $SetProcessStatusToReady$, $SetProcessStatusToRunning$, $SetProcessStatusToWaiting$, etc. The context switch specified in Craig's book involves a number of variables for registers and stacks which we do not specify in our guidelines. The registers and stacks defined in Craig's models are: the general register set is $hwregset$, the stack is in $hwstack$, the instruction pointer (program counter) is $hwip$ and the status word is denoted by $hwstatwd$. $STATUSWD$ is an enumeration, for example: overflow, division by zero, carry set. He also defines a number of attributes for a process which map from process identifiers to registers and stacks which are process stack $pstatcks$, process status words $pstatwds$, process general registers $pregs$, and process instruction pointer $pips$. The context switch then happens saving all hardware registers used by the running process (i.e $hwtack'(pid?) = hwstack$, $pip's(ids?) = hwip$, etc) and restoring all hardware registers for the new process $(hwstack' = pstacks(p?), hwip = pips(p?))$.

Craig modelled FIFO queue as an injective sequence while we are abstracting the queues as sets. With our investigation on patterns, the abstract set of queue can easily be refined by sequence pattern or ordered linked list pattern depending on the implementation of queue.

We define two events to send and receive items from queues which are: $Queue\_Send$ and $Queue\_Receive$. Waiting messages are defined in our guidelines as a function from process set to queue set. We define two waiting messages events to store processes that failed to send/receive items to/from queues which are: Add_ToWToS and Add_ToWToR. Craig defines $Enqueue$ and $Dequeue$ operations using concatenation and extraction operations. Concatenation is used to add an element to a sequence, whereas extraction is used to remove an element from a sequence. Craig also defines waiting messages operations to store processes that failed to send an item to a queue or receive an item from a queue. The $AddWaitingSenders$ operation is defined to enqueue the failed process to the sequence of failed processes $waitingsenders$ whereas $AddWaitingReceivers$ operation is defined to enqueue the failed process to the sequence of failed processes $waitingReceivers$.

Memory management in Craig's model has a number of definitions. *ADDRESS* defines the address of memory. It ranges from 1 to maximum constant value. *MEMDESC* describes a region of store whose address is given by the first component and whose size is given by the second. The free space is called *Holes* and is defined as sequence of *MEMDESC*. The used memory is captured by *usermem* which is also defined as a sequence of *MEMDESC*. He defines a number of operations such as: *RSAllocateFromHole* performs storage allocation from free store. *FreeMainstoreBlock* is used for freeing of an allocated block *MergeAdjacentHoles* is used to merge djacent holes to form larger ones.

Our guidelines defines memory blocks as sets. We also define the address of a block as a function from blocks to address set. The size of blocks are defined as a function from blocks to set of natural number, $\mathbb{N}$. The guidelines cover two types of events: allocation events to allocate memory to the created object and free event to free allocated memory. Unlike Craig's models, the fragmentation issue is not addressed in the presented guidelines.

8.9 Some Tips on Developing Theories in Event-B

In this section, we provide some suggestions drawn from our experience of using the theory feature. The suggestions explain how to develop a theory using the theory plug-in and how to validate theories.

Defining Operators

Operators can be expressed in terms of predicates or expressions. A predicate resembles an operator that returns either true or false. A predicate is formed from a number of expressions. On the other hand, expressions can be formed by applying predicates to expressions. For instance, the addition formula over two integer numbers, ***Add****(3, 5)* is an expression and is called an expression operator. On the other hand, the comparison predicate over integers $bool(3 \geq 5)$ is called a predicate operator.

Well-definedness(WD) can be used to specify the preconditions over the parameters and to ensure their WD. For instance, if the defined operator has two parameters a and b, and the requirement is of a greater than or equal to b, then, this can be regarded as the well-definedness condition $a \geq b$.

Defining Recursive Operators

Operators can be defined in terms of themselves. Such operators are called recursive operators. Recursive definitions in theories are defined on inductive types.

Two steps are required to define a recursive operator: The first is to have an argument (of an inductive type) for the recursive definition and the second step is to formulate the recursive definition.

For instance, one may need to define summation operator. This can be achieved by:

- Constructing an inductive data type: $list \in List(\mathbb{N})$
- Writing the inductive definition of summation over the inductive data type:

Definition:

match list	
$sum(nil)$	0
$sum(cons(n, l))$	$n + sum(l)$

Note that *List* is an inductive data type defined in [18].

We define the recursive operator over a recursive data type (*List*). The result of recursive operator sometimes needs to be converted to a suitable data type. For instance, one may need to add a certain number to each element of a set, this requires the formalisation of a recursive definition over an inductive data type such as *List*, which adds a number to each element of the set resulting in an answer of the form *List*. In order to convert it to a set, another recursive definition is needed to convert a list to a set. The recursive operator *listToSet* transforms a list to a set. It can be defined as follows:

Parameter:

* $list \in List(T)$: List of any type (e.g. natural numbers)

Definition:

match list	
$listToSet(nil)$	ϕ
$listToSet(cons(n, l))$	$\{n\} \cup listToSet(l)$

Note that the theory plug-in does not support the definition of a recursive operator over two or more inductive arguments.

8.9.1 Defining Polymorphic Constants

Polymorphic constants are quite useful in formalising several operators.
getFirst_deleteFirst_sl can return the constant *null* if the removed node is the only node available in the list. The theory plug-in allows to define some polymorphic constants such as empty sequence. The polymorphic empty sequence can be defined as an operator with no argument as follows:

$$\varnothing : \mathbb{Z} \leftrightarrow A$$

Polymorphic constants that form an element of a set such as *nil* can not be defined by the theory. For all the operators that require such a constant, we need to regard that constant as an input parameter of the operator. An example of this is the aforementioned operator $getFirst\_deleteFirst\_sl(nds, nxt, fst, null)$

In this operator, we regarded *null* as an input parameter, since the theory does not allow *null* to be formalised as an operator with no argument.

8.9.2 Conditional Expressions

Case analysis can be performed by using conditional expressions. Conditional expressions allow a single assignment statement to choose between different values based on condition(s). Although, there is no operation that allows a specifier to directly perform a case analysis in a single action of an Event-B model, theory plug-in supports the use of ***COND*** for operators definitions. For instance, the definition $\boldsymbol{COND}(a < b, 5, 6)$ corresponds to the statement **if** $a < b$ **then** 5 **else** 6. For more complex statments such as **if** $a < b$ **then** 5 **else if** $b < c$ **then** 3 **else** 4. We can write $\boldsymbol{COND}\ (a < b, 5,\ \boldsymbol{COND}(b < c, 3, 4))$. The condition(s) written in ***COND*** can be of any type and not just natural numbers.

8.9.3 Composition of Operators

The theory feature supports combining operators into more complicated ones. We can build up complicated operators from simple ones, where the output of one operator becomes the input of another. For instance, ***listToSet*** operator is used in conjunction with the *List* operator as described in the following example: ***listToSet(cons(10, cons(23, cons (4,nil))))***. Operators can also be defined over mixed theories where it is possible for an operator to include operators from multiple theories. This requires importing the required theories into the one being used to define the wanted operator.

8.9.4 Decomposition of Operators

Decomposing operators is the reverse process of composing operators. It is often convenient to decompose a complicated operator into two or more operators. Breaking down the complex operator into smaller operators will facilitate the treatment of complex operators and simplify discharging of their proofs. For instance, let us define an operator that adds a node a based on its priority ap to the appropriate position of a descendingly ordered singly linked list. Before embarking into the definition of the operator, we need to understand the structure of singly linked list. Singly linked list is composed of nodes, each of which contains a data and a

pointer that references the next node. The following diagram shows the structure of singly linked list.

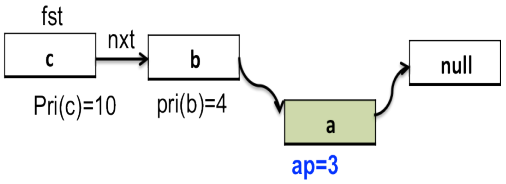


Figure 8.2: inserting a node based on its priority to descendingly ordered singly linked list.

The parameters of the operator are: a set of nodes *nds*, a first node *fst*, a next pointer *nxt*, priority of nodes *pri*, a new node *a*, and the priority of the new node *ap*. The definitions of these variables are listed as follows.

- $nds \in \mathbb{P}(S)$
- $fst \in nds \cup \{null\}$
- $nxt \in nds \setminus \{null\} \rightarrowtail\!\!\!\!\!\rightarrow nds \setminus \{fst\}$
- $pri \in nds \rightarrow \mathbb{N}$
- $a \in S$
- $ap \in \mathbb{N}$

The definition of the operator used for updating *nxt* pointer can be written as follows:

Definition:

$$getNext\_insert\_desc\_sl(nds, nxt, pri, fst, a, ap) \triangleq$$
$$nxt \leftarrow (\{a \mapsto fst \mid ap \geq pri(fst) \vee fst = null\} \cup \{j \mapsto a \mid \exists\, m \cdot j \mapsto m \in nxt \wedge$$
$$pri(j) > ap \wedge ap \geq pri(m)\} \cup \{a \mapsto m \mid \exists\, j \cdot j \mapsto m \in nxt \wedge$$
$$pri(j) > ap \wedge ap \geq pri(m)\} \cup \{nxt^{-1}(null) \mapsto a \mid ap < pri(nxt^{-1}(null))\})$$

The first union adds the new node at the beginning of the list if its priority is greater than or equal to the first node, the second and the third unions add the new node in the middle of the list, if the priority of the node to be inserted is less than a certain node j and greater than or equal the node m (the successor node of j), the last union adds the new node at the end of the list if its priority is less than the last node.

Although this definition is correct from the mathematical point of view, it is complex to understand and prove. Thus, it is very convenient to break this complex operator up into a composition of the following operators.

- $insertToFirst(nds, fst, pri, a, ap) \triangleq \{a \mapsto fst \mid ap \geq pri(fst) \lor fst = null\}$
- $insertToMiddle(nds, nxt, val, pri, a, ap) \triangleq$
  $\{j \mapsto a \mid \exists m \cdot j \mapsto m \in nxt \land pri(j) > ap \land ap \geq pri(m)\} \cup$
  $\{a \mapsto m \mid \exists j \cdot j \mapsto m \in nxt \land pri(j) > ap \land ap \geq pri(m)\}$
- $insertToLast(nds, va, pri, a, ap) \triangleq \{nxt^{-1}(null) \mapsto a \mid ap < pri(nxt^{-1}(null))\}$

The definition of *getNext_insert_desc_sl* can be re-written to:

$$\begin{aligned} getNext\_insert\_desc\_sl(nds, nxt, pri, fst, a, ap) \triangleq \\ nxt \lhd\!\!- \; insertToFirst(nds, fst, val, pri, a, ap) \cup \\ insertToMiddle(nds, nxt, val, pri, a, ap) \cup \\ insertToLast(nds, val, pri, a, ap) \end{aligned}$$

Decomposing complex operators into two or more operators introduces additional operators that are not of direct use to the users and mixing them together with other operators makes it difficult for a user to differentiate between the operators used in the definition of other operators and the operators used by the users. Thus, it is recommended to split the theory into two parts: the first one has all the operators developed for the users and the second one has all the operators used in the definition of other operators.

8.9.5 How to Check the Validity of the Operators

Discharging the POs of the formulae does not guarantee that the formulae is what the users intended to define. The definitions of the formulae can sometimes return unintended results. So it is important to check the validity of the formula to ensure that it always return the correct result. Verifying the validity of the definitions can be obtained by the use of rewrite rules, theorems, and inference rules to capture different execution cases. The definitions of rewrite and inference rules can be used to facilitate proofs and can also help in verifying the validity of the formulae.

For instance, assume that we have two operators: $getNext\_insertFirst\_sl(nds, n, nxt, fst)$ updates a next function after inserting a new node at the beginning of a singly linked list and $getNext\_deleteFirst\_sl(nds, fst, nxt)$ operator updates a next function after removing the head from a singly linked list. In order to assure the validity of both operators, we need to prove the following theorem:

$$\begin{aligned} getNext\_insertFirst\_sl(nds \setminus \{fst\}, fst, \\ getNext\_deleteFirst\_sl(nds, fst, nxt), \\ getFirst\_deleteFirst\_sl(nds, nxt, fst)) = nxt \end{aligned}$$

Chapter 9

Conclusions and Future Work

We aim to provide systematic approaches that support the development of complex systems in Event-B. We have attempted to support reusability and traceability mechanisms, along with the management of complexity in complex systems of Event-B. FreeRTOS was chosen as a case study to identify and address the general problems (complexity, reusability and traceability) in the formal development of complex systems.

We applied compositional design strategy to manage the complexity of FreeRTOS. With the compositional design strategy, FreeRTOS is divided into smaller and independent components that are easy to manage and are recomposed later. Thus, we categorise requirements into a set of requirements for each component and a set of composition requirements for the composition purposes. The models then are carried out separately without the need of using decomposition approaches, following the necessary composition of these separate models. We applied composition technique to link composite requirements with the composed model.

To facilitate building Event-B models from the requirements and retain traceability to requirements in Event-B models, we proposed a staged approach and evaluated the approach through a queue management case study.

To support reusability, we proposed an approach based on the generic instantiation and composition techniques to support the reusability of data refinement patterns.

Apart from the above mentioned main contributions of the thesis, we also have sub-contributions including the development of theories for the linked lists in Event-B, the construction of Event-B models for FreeRTOS that can contribute to the Verified Software Repository, and the derivation of a set of modelling guidelines for RTOS based on our experience with FreeRTOS models.

The following sections give more details about the main contributions and outline the future directions of this research.

9.1 Summary of Contributions

9.1.1 Building Traceable Event-B Model from Requirements

Bridging the gap between the requirements and the formal models to support requirements traceability is a key part of requirements validation. Our work presents an approach for incrementally constructing a traceable formal model from the informal requirements. The approach makes use of UML-B and atomicity decomposition (AD) approaches. UML-B provides the 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 presented approach comprises of three stages:

– **Classify Requirements**

In this stage, the requirements are classified based on the Event-B components. The classification of the requirements based on Event-B structures, helps to bridge the gap between the requirements and the Event-B models, and enables the use of existing Event-B techniques (UML-B and AD) to construct the Event-B models.

– **Construct Semi-Formal Artifacts and Develop Refinement Strategy**

We make use of UML-B, AD diagrams and structured English to represent the requirements as semi-formal artifacts. 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.

– **Construct Formal Models**

In this stage, we use the UML-B tool and the AD tool to generate the Event-B models and also manually write the corresponding Event-B from the structured English representation depending on the type of requirements/semi-formal artifacts. The composition technique is used to integrate the resulting Event-B.

We applied the approach to queue management case study, and drew some conclusions in Chapter 5. Further evaluation of the approach through larger case studies is needed to study the scalability of the graphical models and evaluate the requirements categorization.

9.1.2 Linking Composite Requirements with Composed Model

We adopt compositional strategy to manage the complexity of FreeRTOS development. The compositional strategy aims to divide the system into independent and smaller parts, and later recompose these parts. Thus, FreeRTOS requirements are

divided into two classes: the first class introduces a set of requirements for each individual component and the second class introduces composition requirements for linking the individual components together. Following that, each set of the requirements of the first class are used to construct an Event-B formal model. Thus, the models are constructed separately and later, we apply the composition technique to link the composite requirements with the composed model. During this work, we make use of carrier sets as a way of linking models together instead of shared variables. The use of carrier sets instead of shared variables helps to avoid replicating the shared variables/properties across the individual models, and provides us loosely-coupled models that could be reused easily than the models with shared variables/properties. Carrier sets also help to avoid restrictions imposed by shared variables such as the inability to perform data refinement of shared variables. Properties that are shared across the models can be described only in the composed models. Shared event composition is used to generate the composed models. The composite requirements point us clearly to the events to be composed. We found out that the composition could be simplified through abstraction; the more abstract the sub-models are composed, the lesser is the effort of composing the number of the events.

9.1.3 Reusing Data Refinement Patterns through Generic Instantiation and Composition

The benefits of applying reusability at different stages of the software development cycle are widely recognized. In this thesis, we attempted to present an approach to facilitate the reusability in Event-B formal method through the use of generic instantiation and composition techniques. Generic instantiation technique is used to create an instance of a pattern that consists of refinement chain and allows replacement of the pattern names (types, constants, variables, events) by the names that suit the development at hand. The composition technique, on the other hand, enables the integration of several sub-models into a large model. The combination of generic instantiation and composition techniques provides renaming and incorporation mechanisms to integrate the pattern into the problem. We applied this approach to data refine the abstract "set" of FreeRTOS to circular doubly linked list. The overall POs of the pattern machine is 41 POs, of which 17 were proved interactive. Reusing the pattern using the proposed approach saves proving efforts because POs that are originated from the pattern only need to be discharged once and not for all the instantiations of the pattern.

9.2 Future Work

9.2.1 Reverse Engineering of Structured Languages to Event-B

Systems source code is considered an important source of information that describes the actions to be performed by the system. Understanding how a program works is a difficult task specially for systems that are not very well documented. The process of constructing high level representations of an implementation is known as reverse engineering. During our work, we find that most of the resources of FreeRTOS describe the system from the application developer point of view and do not give more details about the RTOS level. We have spent time to understand FreeRTOS C codes to come up with design requirements. As a future goal, we plan to look at the effect of using AD diagrams to bridge the gap between the code and the Event-B formal models, and ultimately extract the requirements from the structured programs. In a structured language, such as C, a program is divided into blocks called functions or procedures. Each function is performed a specific task and isolates one block of code from other independent block of code. Structured programs are often composed of flow structures such as selection and repetition. The reason of choosing AD approaches to assist in the process of reverse engineering the structured program code into Event-B models is due to the availability of AD patterns that support flow structures of structured programming languages. AD diagrams offer patterns that support sequence, selection and repetition, which are common flow structures of structured programs. Sequence flow structures of structured programs cover ordering execution of statements. The selection flow structures of structured languages cover if-then-else conditions and the repetition flow structures of structured programs cover for/while loop...etc. The AD diagrams can be combined to show the overall structures of a piece of code, and this adds another value and facilitates the organization of the refinement strategy. There are some issues that should be explored more thoroughly such as the need of some guidelines that assist in extracting invariants from the programs.

9.2.2 Verifying Linear Temporal Properties

One important formal verification approach of the kernel of real-time operating systems (RTOS) is the verification of temporal properties such as liveness properties.

One of the main future direction is to study the use of AD approach to support the verification of linear temporal properties of Event-B formal models.

The strength of the AD approach is that it shows the relationships between the abstract and refinement levels. Expressing temporal properties using AD diagrams

can be translated easily into Event-B models. The proof of refinement guarantees that the formal model satisfies these temporal properties. We are going to study how to express liveness properties into Event-B formal models using AD diagrams and guarantee the preservation of these properties. We are also trying to investigate several AD patterns that support different kinds of temporal properties.

9.2.3 Evaluate the General Guidelines for Modelling RTOS Kernels

The devised guidelines given in Chapter 8 need to be evaluated through several case studies. As a direction for future research, we are going to choose an RTOS kernels such as UCOS, and evaluate the guidelines by applying them to build UCOS Event-B model. We expect to improve these guidelines further while applying them to different case studies. Moreover, applying the guidelines to different case studies helps in learning different lessons and have solid experience about building RTOS kernels using the formal methods. We also would like to extend the guidelines to cover timing properties. We will adopt Sarshogh patterns [68] to verify timing properties (expiry, delays and deadlines) throughout the refinement levels.

We also would like to utilise several case studies to clarify, evaluate and strengthen our approach for generating traceable Event-B models. More guidelines can be added to extract the appropriate Event-B formalism from requirements. We also would like to augment ProR tool [45] with our approach to add a capability of tracing requirements in backward direction.

Appendix A

FreeRTOS Requirements

A.1 Task management requirements

Label	Requirements
TSK1	Tasks can be created and deleted.
TSK2	Only one task can be executed on a processor at a time.
TSK3	Tasks assigned priority when created.
TSK4	The processor handles events that are usually signaled by interrupts by executing its interrupt service routine (ISR).
TSK5	Context switch is the mechanism used for swapping tasks.
TSK6	The scheduler can exist in one of the following states: *not started, running*, and *suspended*.
TSK7	When the scheduler is started, the *idle* task is created to ensure that there is always at least one task that is able to run.
TSK8	No context switch is performed when the scheduler is not in the running state.
TSK9	When the scheduler in the not started state all the created tasks are deleted.
TSK10	A task can hold an item, a task-item considers one field that descripes the information of the TCB of a task.
TSK11	The running task can send/receive an item.
TSK12	The interrupt can send/receive an item.
TSK13	A task can be in one of the following states: ready, blocked or delayed, or suspended state.
TSK14	A task can be put in the collection of ready tasks if the scheduler is running.
TSK15	When the idle task is created it will be added to the collection of ready tasks.
TSK16	The idle task is always ready and never blocks.

Label	Requirements
TSK17	A suspended task can be resumed by any task.
TSK18	The running task can resume a suspended task.
TSK19	A delayed task can get out of delay state and transfers to the ready state when the scheduler is running.
TSK20	A task can be put in the collection of pending tasks if the scheduler is suspended.
TSK21	A task can be put in the collection of delay tasks.
TSK22	A task can be put in the collection of suspended tasks.
TSK23	When the scheduler returns to the running state, any task in the collection of pending tasks will be moved to the collection of ready tasks.
TSK24	The scheduler should be suspended while adding the running task to the collection of delay tasks to prevent a context switch from occurring.
TSK25	INCLUDE_TaskSuspend must be set to TRUE to be available.
TSK26	A task can delete itself or any other task. This feature is available only when *INCLUDE_TaskDelete* is set to TRUE.
TSK27	Any task can sleep for certain ticks and placed into the collection of delay tasks based on its wake-up time. The time at which the task should be woken is calculated by adding the delay time of a task to the current time pointed by the timer.
TSK28	A delay task is moved to ready state only when its delay time is expired.
TSK29	The clock that generates interrupts at a regular rate to measure time.
TSK30	The length of the time slice is set by *configTICK_RATE_HZ* configuration constant.
TSK31	The scheduler runs itself at the end of each time slice to check if there is any task requires to be woken and to select the next task to run.
TSK32	Delay feature is only available if the configuration constant *INCLUDE_TaskDelay* is set to TRUE.

Label	Requirements
TSK33	Delay until feature is only available if the configuration constant*INCLUDE_TaskDelayUntil* is set to TRUE.
TSK34	Delay until feature delays a task until a specific time has passed since the given time reference whereas delay feature delays a task for the defined time from the time of call, thus, the time at which a task should be woken in case of delay until feature is calculated by adding the sleep time of a task to the previous wake time (the time at which the task last left the blocked state).
TSK35	The number of ticks have been passed while the scheduler is suspended are consider missed ticks.
TSK36	When the scheduler returns to the running state, missed ticks should be processed by by incrementing the tick counts once for each missed tick to check if any delay task requires waking.
TSK37	The scheduler can be suspended several times, the scheduler will get out of the suspended state by resuming the scheduler for every preceding call has suspended it.
TSK38	A blocked task is placed into either the collection of delay tasks if its wake-up time has not overflowed or the collection of overflow delay tasks if its wake-up time has overflowed.
TSK39	When the tick is incremented, the collection of delay tasks and the collection of overflow delay tasks are swapped if the tick count has overflowed.
TSK40	The maximum priority is set by the configuration constant *configMAX_PRIORITIES*.
TSK41	Base priority stores the original priority the task has when it is created, whereas active priority can be modified at any point in time.
TSK42	Higher priority readied tasks run before lower priority readied tasks, task within the same priority share CPU time in time slices.
TSK43	The idle task has the lowest possible priority (priority zero) to ensure it never prevents a higher priority task.
TSK44	The active priority of a task can be replaced by its original priority.
TSK45	Updating the priority of a task or returning the priority of a task operations are available only when *INCLUDE_TaskPriorityGet* and *NCLUDE_TaskPrioritySet* are set to TRUE.
TSK46	A context switch should perform if the priority being set by updating task priority operation is higher than the currently executing task.

Label	Requirements
TSK47	A context switch should perform when there is a ready task with higher priority than the current running one and the state of the scheduler is running but when the scheduler is suspended, the context switch will perform and schedule the highest priority task as soon as the scheduler is resumed.
TSK48	The are some other cases where a context switch can perform such as: incrementing the tick count, adding a new task to the collection of ready tasks, suspending the running task, deleting the running task, resuming a task, resuming the scheduler, moving delay tasks to the collection of ready tasks. All these cases can result in having a task with a priority higher than the priority of the current running task, thus a context switch is necessary to schedule the highest priority task.
TSK49	Interrupts also have priority,$configMAX\_PRIORITY$ defines the highest interrupt level available to interrupts.
TSK50	Higher priority interrupt run before lower priority interrupt.
TSK51	Lower priority task/interrupt can be interrupted by higher priority task/interrupt before the lower priority task/interrupt completed its execution.
TSK52	Each task has its own context.
TSK53	Context switch is performed by storing the context of the running task and loading the context of next task to run.
TSK54	Critical section protection is used to protect regions where mutual exclusion protection is needed, the protection is performed by disabling interrupts.
TSK55	The processor always gives priority to execute interrupts over tasks, the ISR must complete its execution without being interrupted by tasks.
TSK56	When the ISR is completed, the kernel dispatches the correct task.
TSK57	Interrupts must save their context to the stack.
TSK58	Tasks in the blocked state always have a 'timeout' period, after which the task will be unblocked.

A.2 Queue management requirements

Label	Requirements
QUE1	Queues can be created and deleted.
QUE2	The length of the queue is identified when the queue is created.
QUE3	Task can only send item to a queue when there is enough room in the queue. Similarly, task can only receive an item from a queue when the queue is not empty.
QUE4	A queue contains a limited number of items.
QUE5	Each queue has two collections of waiting tasks: tasks waiting to send and tasks waiting to receive.
QUE6	Tasks fail to send an item to a queue because the queue is full is placed into the collection of tasks waiting to send. Similarly, tasks fail to receive an item from a queue because the queue is empty is placed into the collection of tasks waiting to receive.
QUE7	Every task is mapped at most to one collections of waiting tasks.
QUE8	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.
QUE9	When a queue becomes available (there is a 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.
QUE10	The queue must be locked when the running task failed to send or receive an item to a queue.
QUE11	The queue should be unlocked when the running task has been added to the collection of waiting tasks.
QUE12	Unlock queue operation processes the waiting tasks by removing all tasks from the collections of waiting tasks.

A.3 Memory management requirements

Label	Requirements
MEM1	Kernel allocates RAM to each created object.
MEM2	Memory blocks are consecutive to each other and non-overlapping.
MEM3	Memory in scheme1 does not be freed once it has been allocated.
MEM4	The total size of the array is defined by *configTOTAL_HEAP_SIZE* constant.
MEM5	The scheduler ensures that blocks are always aligned to the required number of bytes.
MEM6	FreeRTOS provides different alignment masks based on architecture.
MEM7	Before allocation, the scheduler ensures that the size of the requested memory is valid (not zero) and that there is enough free memory to serve the request.
MEM8	The created object in scheme2 is placed in the block of adequate size in which it will fit (if any), if no block of adequate size is found and there is a block larger than what is required, it will be split into two; one block of the requested size is assigned to the object and the second block is added to the list of free blocks.
MEM9	In scheme2 if the block found is larger than *heapMINIMUM_BLOCK_SIZE*, then it can be split into two blocks.
MEM10	Memory in scheme2 can be freed once it has been allocated.

A.4 Composition requirements

COMP1-EVT	The kernel has to allocate RAM each time a task, queue, semaphore or mutex is created.
COMP1-INV	Tasks and queues are distinct.

The sub-requirements of *COMP1-EVT* are:

MEM1	Kernel allocates RAM to each created object.
MEMS8	The created object in scheme2 is placed in the block of adequate size in which it will fit (if any), if no block of adequate size is found and there is a block larger than what is required, it will be split into two; one block of the requested size is assigned to the object and the second block is added to the list of free blocks.
TSK1	Task can be created.
QUE1	Queues can be created.
QUE13	Queues are three types: queues, semaphores, and mutex.

COMP2-EVT	The kernel has to free RAM each time a task, queue, semaphore or mutex is deleted.

The sub-requirements of *COMP2* requirements are:

MEM10	Memory in scheme2 can be freed once it has been allocated.
TSK1	Task can be deleted.
QUE1	Queues can be deleted.
QUE13	Queues are three types: queues, semaphores, and mutex.

COMP3-EVT	The running task can send/receive an item to/from a queue.
COMP3-INV	Task items and queue items are distinct.

The sub-requirements of *COMP3-EVT* requirement are:

TSK10	The running task can hold an item or remove an existing item.
QUE14	A queue can be used to send and receive items.

COMP4-EVT	A task that is removed from the collection of waiting tasks is placed into the collection of pending-ready tasks if the scheduler is suspended or placed into the collection of ready tasks if the scheduler is running.
COMP4-INV1	A task that is added to the collection of waiting tasks must also be added to the collection of delay tasks.

The sub-requirements of *COMP4-EVT* requirements are:

QUE14	An object can be removed from the collections of waiting tasks.
TSK14	A task can be put into the collection of ready tasks if the scheduler is running.
TSK20	A task can be put into the collection of pending-ready tasks if the scheduler is suspended.

COMP5-EVT	The blocked running task with block-time that is greater than zero and less than the value determined by the constant *PortMAX_DELAY* is added to the collection of waiting tasks and also the collection of delay tasks.

The sub-requirements of *COMP5-EVT* requirement are:

QUE14	An object can be added to the collection of waiting tasks if its block-time is greater than zero and less than the constant *PortMAX_DELAY*.
TSK21	The running task can be put into the collection of delay tasks.

COMP6-EVT	The blocked running task with block-time that is equal to the value determined by the constant *PortMAX_DELAY* is added to the collections of waiting tasks and also the collection of suspend tasks.

The sub-requirements of *COMP6-EVT* are:

QUE15	An object can be added to sending/receiving collections of waiting tasks if its block-time is equal to the constant *PortMAX_DELAY*.
TSK22	The running task can be put in the collection of suspend tasks.

COMP7-EVT	The priority inheritance is used to raise the priority of the mutex-holder task whenever the higher priority task (running task) attempts to hold that mutex.

The sub-requirements of *COMP7-EVT* requirement are:

TSK56	A task with lower priority can be raised by the priority of the running task.
QUE16	The object that holds a mutex can be obtained.

COMP8-EVT	The priority disinheritance is used to set the active priority of the task holder back to its original priority.

The sub-requirements of *COMP8-EVT* requirement are:

TSK44	The active priority of a task can be replaced by its original priority.
QUE16	The object that holds a mutex can be obtained.

Appendix B

Queue Event-B Model

The context:

CONTEXT Cntxt
SETS
ReadyTask
Task_SET
Queue_SET
Item_SET
END

The abstract model:

MACHINE M0
SEES Cntxt
VARIABLES
TaskQueueSend
FailedTaskQueueSend
CurrentTask
Task
Queue
Item
priority
Length
QueueItem
TaskItem
INVARIANTS
Task.type : $Task \in \mathbb{P}(Task\_SET)$
Queue.type : $Queue \in \mathbb{P}(Queue\_SET)$
Item.type : $Item \in \mathbb{P}(Item\_SET)$
priority.type : $priority \in Task \rightarrow \mathbb{N}_1$
Length.type : $Length \in Queue \rightarrow \mathbb{N}_1$
QueueItem.type : $QueueItem \in Item \rightarrow Queue$
TaskItem.type : $TaskItem \in Item \rightarrow Task$
inv1 : $CurrentTask \subseteq Task$
inv2 : $card(CurrentTask) \leq 1$

inv_TaskQueueSend_type : $TaskQueueSend \subseteq CurrentTask$

inv_FailedTaskQueueSend_type : $FailedTaskQueueSend \subseteq CurrentTask$

inv_xor : $partition((TaskQueueSend \cup FailedTaskQueueSend), TaskQueueSend, FailedTaskQueueSend)$

EVENTS

Initialisation

begin

act_TaskQueueSend : $TaskQueueSend := \varnothing$

act_FailedTaskQueueSend : $FailedTaskQueueSend := \varnothing$

Task.init : $Task := \varnothing$

Queue.init : $Queue := \varnothing$

Item.init : $Item := \varnothing$

priority.init : $priority := \varnothing$

Length.init : $Length := \varnothing$

QueueItem.init : $QueueItem := \varnothing$

TaskItem.init : $TaskItem := \varnothing$

end

Event $CreateTask \mathrel{\widehat{=}}$

any

t

where

grd1 : $t \in Task\_SET \setminus Task$

then

act1 : $Task := Task \cup \{t\}$

end

Event $DeleteTask \mathrel{\widehat{=}}$

any

t

where

grd1 : $t \in Task$

then

act1 : $Task := Task \setminus \{t\}$

end

Event $CreateQueue \mathrel{\widehat{=}}$

any

q

l

where

grd1 : $q \in Queue\_SET \setminus Queue$

grd2 : $l \in \mathbb{N}_1$

then

act1 : $Queue := Queue \cup \{q\}$

act2 : $Length(q) := l$

end

Event $DeleteQueue \mathrel{\widehat{=}}$

any

q

where

grd1 : $q \in Queue$

then

act1 : $Queue := Queue \setminus \{q\}$

act2 : $Length := Length \rhd \{Length(q)\}$

end

Event *TaskQueueSend* $\widehat{=}$

any

c

q

i

where

grd_self : $c \notin TaskQueueSend$

grd_xor : $c \notin FailedTaskQueueSend$

grd1 : $c \in CurrentTask$

grd2 : $q \in Queue$

grd3 : $i \in TaskItem^{-1}[\{c\}]$

grd4 : $Length(q) < card(QueueItem^{-1}[\{q\}])$

then

act : $TaskQueueSend := TaskQueueSend \cup \{c\}$

act1 : $QueueItem := QueueItem \cup \{i \mapsto q\}$

act2 : $TaskItem := TaskItem \setminus \{i \mapsto c\}$

end

Event *FailedTaskQueueSend* $\widehat{=}$

any

c

q

i

where

grd_self : $c \notin FailedTaskQueueSend$

grd_xor : $c \notin TaskQueueSend$

grd1 : $c \in CurrentTask$

grd2 : $q \in Queue$

grd3 : $i \in TaskItem^{-1}[\{c\}]$

grd4 : $Length(q) = card(QueueItem^{-1}[\{q\}])$

then

act : $FailedTaskQueueSend := FailedTaskQueueSend \cup \{c\}$

end

END

The first refinement:

MACHINE M1
REFINES M0
SEES Cntxt
VARIABLES

`TaskQueueSend`
`FailedTaskQueueSend`
`NoTaskInTaskWaitingToReceive`
`RemoveFromTaskWaitingToReceive`
`PlaceOnTaskWaitingToSend`
`Task`
`Queue`
`Item`
`priority`
`Length`

QueueItem
TaskItem
CurrentTask
TaskWaitingToSend
TaskWaitingToReceive

INVARIANTS

inv_NoTaskWaitingToReceice_seq : $NoTaskInTaskWaitingToReceive \subseteq TaskQueueSend$
inv_RemoveFromTaskWaitingToReceive_seq : $RemoveFromTaskWaitingToReceive \subseteq TaskQueueSend$
inv_PlaceOnTaskWaitingToSend_type : $PlaceOnTaskWaitingToSend \subseteq CurrentTask$
inv_PlaceOnTaskWaitingToSend_gluing : $PlaceOnTaskWaitingToSend = FailedTaskQueueSend$
TaskWaitingToSend.type : $TaskWaitingToSend \in Queue \rightarrow Task$
TaskWaitingToReceive.type : $TaskWaitingToReceive \in Queue \rightarrow Task$

EVENTS

Initialisation

extended

begin

act_TaskQueueSend : TaskQueueSend $:= \varnothing$
act_FailedTaskQueueSend : FailedTaskQueueSend $:= \varnothing$
Task.init : Task $:= \varnothing$
Queue.init : Queue $:= \varnothing$
Item.init : Item $:= \varnothing$
priority.init : priority $:= \varnothing$
Length.init : Length $:= \varnothing$
QueueItem.init : QueueItem $:= \varnothing$
TaskItem.init : TaskItem $:= \varnothing$
act_NoTaskWaitingToReceice : $NoTaskInTaskWaitingToReceive := \varnothing$
act_RemoveFromTaskWaitingToReceive : $RemoveFromTaskWaitingToReceive := \varnothing$
act_PlaceOnTaskWaitingToSend : $PlaceOnTaskWaitingToSend := \varnothing$
TaskWaitingToSend.init : $TaskWaitingToSend := \varnothing$
TaskWaitingToReceive.init : $TaskWaitingToReceive := \varnothing$

end

Event *CreateTask* $\widehat{=}$
extends *CreateTask*

any

t

where

grd1 : t $\in$ Task_SET $\setminus$ Task

then

act1 : Task $:=$ Task $\cup \{$t$\}$

end

Event *DeleteTask* $\widehat{=}$
extends *DeleteTask*

any

t

where

grd1 : t $\in$ Task

then

act1 : Task $:=$ Task $\setminus \{$t$\}$

end

Event *CreateQueue* $\widehat{=}$
extends *CreateQueue*

any

q
l
where
grd1 : $\mathtt{q} \in \mathtt{Queue\_SET} \setminus \mathtt{Queue}$
grd2 : $\mathtt{l} \in \mathbb{N}_1$
then
act1 : $\mathtt{Queue} := \mathtt{Queue} \cup \{\mathtt{q}\}$
act2 : $\mathtt{Length}(\mathtt{q}) := \mathtt{l}$
end
Event *DeleteQueue* $\widehat{=}$
extends *DeleteQueue*
any
q
where
grd1 : $\mathtt{q} \in \mathtt{Queue}$
then
act1 : $\mathtt{Queue} := \mathtt{Queue} \setminus \{\mathtt{q}\}$
act2 : $\mathtt{Length} := \mathtt{Length} \rhd \{\mathtt{Length}(\mathtt{q})\}$
end
Event *TaskQueueSend* $\widehat{=}$
refines *TaskQueueSend*
any
c
q
i
where
grd_self : $c \notin TaskQueueSend$
grd_xor : $c \notin PlaceOnTaskWaitingToSend$
grd1 : $c \in CurrentTask$
grd2 : $q \in Queue$
grd3 : $i \in TaskItem^{-1}[\{c\}]$
grd4 : $Length(q) < card(QueueItem^{-1}[\{q\}])$
then
act : $TaskQueueSend := TaskQueueSend \cup \{c\}$
act1 : $QueueItem := QueueItem \cup \{i \mapsto q\}$
act2 : $TaskItem := TaskItem \setminus \{i \mapsto c\}$
end
Event *NoTaskInTaskWaitingToReceive* $\widehat{=}$
any
c
where
grd_self : $c \notin NoTaskInTaskWaitingToReceive$
grd_seq : $c \in TaskQueueSend$
grd_xor : $c \notin RemoveFromTaskWaitingToReceive$
grd1 : $TaskWaitingToReceive = \varnothing$
then
act : $NoTaskInTaskWaitingToReceive := NoTaskInTaskWaitingToReceive \cup \{c\}$
end
Event *RemoveFromTaskWaitingToReceive1* $\widehat{=}$
any
c
q

where

grd_self : $c \notin RemoveFromTaskWaitingToReceive_1$

grd_seq : $c \in TaskQueueSend$

grd_xor : $c \notin NoTaskInTaskWaitingToReceive$

grd1 : $q \in Queue$

grd2 : $q \mapsto c \in TaskWaitingToReceive$

then

act : $RemoveFromTaskWaitingToReceive_1 := RemoveFromTaskWaitingToReceive_1 \cup \{c\}$

act1 : $TaskWaitingToReceive := TaskWaitingToReceive \setminus \{q \mapsto c\}$

end

Event *PlaceOnTaskWaitingToSend* $\widehat{=}$

refines *FailedTaskQueueSend*

any

c

q

i

where

grd_self : $c \notin PlaceOnTaskWaitingToSend$

grd_xor : $c \notin TaskQueueSend$

grd1 : $c \in CurrentTask$

grd2 : $q \in Queue$

grd3 : $i \in TaskItem^{-1}[\{c\}]$

grd4 : $Length(q) = card(QueueItem^{-1}[\{q\}])$

then

act : $PlaceOnTaskWaitingToSend := PlaceOnTaskWaitingToSend \cup \{c\}$

act1 : $TaskWaitingToSend := TaskWaitingToSend \cup \{q \mapsto c\}$

end

Event *RemoveFromTaskWaitingToReceive2* $\widehat{=}$

any

c

q

where

grd_seq : $c \in PlaceOnTaskWaitingToSend$

grd1 : $q \in Queue$

grd2 : $q \mapsto c \in TaskWaitingToReceive$

then

act1 : $TaskWaitingToReceive := TaskWaitingToReceive \setminus \{q \mapsto c\}$

end

END

The second refinement:

MACHINE M2
REFINES M1
SEES Cntxt
VARIABLES

`TaskQueueSend`

`FailedTaskQueueSend`

`NoTaskInTaskWaitingToReceive`

`RemoveFromTaskWaitingToReceive`

PlaceOnTaskWaitingToSend
LockQueue
RemoveFromTaskWaitingToReceive2
UnlockQueue
Task
Queue
Item
priority
Length
QueueItem
TaskItem
CurrentTask
TaskWaitingToSend
TaskWaitingToReceive
QueueFlg

INVARIANTS

inv_LockQueue_type : $LockQueue \subseteq CurrentTask$
inv_PlaceOnTaskWaitingToSend_seq : $PlaceOnTaskWaitingToSend \subseteq LockQueue$
inv_RemoveFromTaskWaitingToReceive2_seq : $RemoveFromTaskWaitingToReceive2 \subseteq PlaceOnTaskWaitingToSend$
inv_UnlockQueue_seq : $UnlockQueue \subseteq RemoveFromTaskWaitingToReceive2$
QueueFlg.type : $QueueFlg \in Queue \rightarrow BOOL$

EVENTS

Initialisation

extended

begin

act_TaskQueueSend : TaskQueueSend := $\varnothing$
act_FailedTaskQueueSend : FailedTaskQueueSend := $\varnothing$
Task.init : Task := $\varnothing$
Queue.init : Queue := $\varnothing$
Item.init : Item := $\varnothing$
priority.init : priority := $\varnothing$
Length.init : Length := $\varnothing$
QueueItem.init : QueueItem := $\varnothing$
TaskItem.init : TaskItem := $\varnothing$
act_NoTaskWaitingToReceice : NoTaskInTaskWaitingToReceive := $\varnothing$
act_RemoveFromTaskWaitingToReceive : RemoveFromTaskWaitingToReceive := $\varnothing$
act_PlaceOnTaskWaitingToSend : PlaceOnTaskWaitingToSend := $\varnothing$
TaskWaitingToSend.init : TaskWaitingToSend := $\varnothing$
TaskWaitingToReceive.init : TaskWaitingToReceive := $\varnothing$
act_LockQueue : $LockQueue := \varnothing$
act_RemoveFromTaskWaitingToReceive2 : $RemoveFromTaskWaitingToReceive2 := \varnothing$
act_UnlockQueue : $UnlockQueue := \varnothing$
QueueFlg.init : $QueueFlg := \varnothing$

end

Event *CreateTask* $\widehat{=}$
extends *CreateTask*

any

t

where

grd1 : $t \in Task\_SET \setminus Task$

then

act1 : $Task := Task \cup \{t\}$

end

Event *DeleteTask* $\widehat{=}$

extends *DeleteTask*

any

t

where

grd1 : $t \in Task$

then

act1 : $Task := Task \setminus \{t\}$

end

Event *CreateQueue* $\widehat{=}$

extends *CreateQueue*

any

q

l

where

grd1 : $q \in Queue\_SET \setminus Queue$

grd2 : $l \in \mathbb{N}_1$

then

act1 : $Queue := Queue \cup \{q\}$

act2 : $Length(q) := l$

end

Event *DeleteQueue* $\widehat{=}$

extends *DeleteQueue*

any

q

where

grd1 : $q \in Queue$

then

act1 : $Queue := Queue \setminus \{q\}$

act2 : $Length := Length \rhd\!\!- \{Length(q)\}$

end

Event *TaskQueueSend* $\widehat{=}$

extends *TaskQueueSend*

any

c

q

i

where

grd_self : $c \notin TaskQueueSend$

grd_xor : $c \notin PlaceOnTaskWaitingToSend$

grd1 : $c \in CurrentTask$

grd2 : $q \in Queue$

grd3 : $i \in TaskItem^{-1}[\{c\}]$

grd4 : $Length(q) < card(QueueItem^{-1}[\{q\}])$

then

act : $TaskQueueSend := TaskQueueSend \cup \{c\}$

act1 : $QueueItem := QueueItem \cup \{i \mapsto q\}$

act2 : $TaskItem := TaskItem \setminus \{i \mapsto c\}$

end

Event *NoTaskInTaskWaitingToReceive* $\widehat{=}$

extends *NoTaskInTaskWaitingToReceive*

any

c

where

grd_self : $\mathtt{c} \notin \mathtt{NoTaskInTaskWaitingToReceive}$

grd_seq : $\mathtt{c} \in \mathtt{TaskQueueSend}$

grd_xor : $\mathtt{c} \notin \mathtt{RemoveFromTaskWaitingToReceive}$

grd1 : $\mathtt{TaskWaitingToReceive} = \varnothing$

then

act : $\mathtt{NoTaskInTaskWaitingToReceive} := \mathtt{NoTaskInTaskWaitingToReceive} \cup \{\mathtt{c}\}$

end

Event *RemoveFromTaskWaitingToReceive1* $\widehat{=}$

extends *RemoveFromTaskWaitingToReceive1*

any

c

q

where

grd_self : $\mathtt{c} \notin \mathtt{RemoveFromTaskWaitingToReceive1}$

grd_seq : $\mathtt{c} \in \mathtt{TaskQueueSend}$

grd_xor : $\mathtt{c} \notin \mathtt{NoTaskInTaskWaitingToReceive}$

grd1 : $\mathtt{q} \in \mathtt{Queue}$

grd2 : $\mathtt{q} \mapsto \mathtt{c} \in \mathtt{TaskWaitingToReceive}$

then

act : $\mathtt{RemoveFromTaskWaitingToReceive1} := \mathtt{RemoveFromTaskWaitingToReceive1} \cup \{\mathtt{c}\}$

act1 : $\mathtt{TaskWaitingToReceive} := \mathtt{TaskWaitingToReceive} \setminus \{\mathtt{q} \mapsto \mathtt{c}\}$

end

Event *LockQueue* $\widehat{=}$

any

c

where

grd_self : $c \notin LockQueue$

grd_xor : $c \notin TaskQueueSend$

then

act : $LockQueue := LockQueue \cup \{c\}$

end

Event *PlaceOnTaskWaitingToSend* $\widehat{=}$

refines *PlaceOnTaskWaitingToSend*

any

c

q

i

where

grd_self : $c \notin PlaceOnTaskWaitingToSend$

grd_seq : $c \in LockQueue$

grd1 : $c \in CurrentTask$

grd2 : $q \in Queue$

grd3 : $i \in TaskItem^{-1}[\{c\}]$

grd4 : $Length(q) = card(QueueItem^{-1}[\{q\}])$

then

act : $PlaceOnTaskWaitingToSend := PlaceOnTaskWaitingToSend \cup \{c\}$

act1 : $TaskWaitingToSend := TaskWaitingToSend \cup \{q \mapsto c\}$

end

Event $RemoveFromTaskWaitingToReceive2 \widehat{=}$

refines $RemoveFromTaskWaitingToReceive2$

any

c

q

where

grd_seq : $c \in PlaceOnTaskWaitingToSend$

grd1 : $q \in Queue$

grd2 : $q \mapsto c \in TaskWaitingToReceive$

then

act1 : $RemoveFromTaskWaitingToReceive2 := RemoveFromTaskWaitingToReceive2 \cup \{c\}$

act2 : $TaskWaitingToReceive := TaskWaitingToReceive \setminus \{q \mapsto c\}$

end

Event $UnlockQueue \widehat{=}$

any

c

where

grd_self : $c \notin UnlockQueue$

grd_seq : $c \in RemoveFromTaskWaitingToReceive2$

then

act : $UnlockQueue := UnlockQueue \cup \{c\}$

end

END

References

[1] VDMTools: advances in support for formal modeling in VDM. *SIGPLAN Not.*, **43**[2]:3–11, February 2008.

[2] J.-R. Abrial. *The B-book - assigning programs to meanings.* Cambridge University Press, 2005.

[3] J.-R. Abrial. Formal methods : Theory becoming practice. *Journal of Universal Computer Science*, pages 619–628, 2007.

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

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

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

[7] J.-R. Abrial and S. Hallerstede. Refinement, decomposition, and instantiation of discrete models: Application to Event-B. *Fundamenta Informaticae*, **77**:1–28, 2007.

[8] E. Alkhammash, A. Fathabadi, M. Butler, and C. Cristea. Building traceable Event-B models from requirements. In *Proceedings of Automated Verification of Critical Systems (AVoCS2013)*, 2013.

[9] F. Badeau and A. Amelot. Using B as a high level programming language in an industrial project: roissy VAL. In *Proceedings of the 4th international conference on Formal Specification and Development in Z and B*, ZB'05, pages 334–354, Berlin, Heidelberg, 2005. Springer-Verlag.

[10] R. Barry. The FreeRTOS project. http://www.freertos.org/, 2010.

[11] R. Barry. *Using the FreeRTOS Real Time Kernel- a Practical Guide.* Lulu, 2010.

[12] P. Behm, P. Benoit, A. Faivre, and J.M. Meynadier. Meteor : A successful application of B in a large project. In *In Wing et al*, pages 369–387, 1999.

[13] B. W. Boehm. Verifying and validating software requirements and design specifications. *IEEE Softw.*, **1**[1]:75–88, January 1984.

[14] M. Butler. *On the Verified-by-Construction Approach.* Facs Facts Newsletter Issue 2006-1, 2006.

[15] M. Butler. Decomposition structures for Event-B. In *Proceedings of the 7th International Conference on Integrated Formal Methods*, IFM '09, pages 20–38, Berlin, Heidelberg, 2009. Springer-Verlag.

[16] M. Butler. Using Event-B refinement to verify a control strategy. Technical report, University of Southampton, United Kingdom, 2009.

[17] M. Butler and S. Hallerstede. The rodin formal modelling tool. In *Proceedings of the 2007th internatioanal conference on Formal Methods in Industry*, FACS-FMI'07, pages 2–2, Swinton, UK, UK, 2007. British Computer Society.

[18] M. Butler and I. Maamria. Mathematical extension in Event-B through the rodin theory component. *University of Southampton*, 2010.

[19] J. Chanda, A. Kanjilal, S. Sengupta, and S. Bhattacharya. Traceability of requirements and consistency verification of uml use case, activity and class diagram: A formal approach. In *Methods and Models in Computer Science, 2009. ICM2CS 2009. Proceeding of International Conference on*, pages 1–4, Dec 2009.

[20] Jr. Clarke, M. Edmund, O. Grumberg, and D. Peled. *Model checking.* MIT Press, Cambridge, MA, USA, 1999.

[21] ClearSy. Atelier B. http://www.atelierb.eu/, 2013.

[22] J. Cliff. *Systematic Software Development Using VDM (2Nd Ed.).* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1990.

[23] I. Craig. *Formal models of operating system Kernels.* Springer, 2007.

[24] I. Craig. *Formal Refinement for Operating System Kernels.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.

[25] T. de Sousa, J. Almeida, S. Viana, and J. Pavón. Automatic analysis of requirements consistency with the b method. *SIGSOFT Softw. Eng. Notes*, **35**[2]:1–4, March 2010.

[26] D. Deharbe, S. Galvao, and A.M. Moreira. http://code.google.com/p/freertosb/source/browse, 2009.

[27] D. Deharbe, S. Galvao, and A.M. Moreira. Formalizing FreeRTOS: First steps. In Marcel Vinicius Medeiros Oliveira and Jim Woodcock, editors, *SBMF*, **5902** of *Lecture Notes in Computer Science*, pages 101–117. Springer, 2009.

[28] T. der Rieden. *Verified Linking for Modular Kernel Verification.* 2009.

[29] W. DeRoever and K Engelhardt. *Data Refinement: Model-Oriented Proof Methods and Their Comparison.* Cambridge University Press, New York, NY, USA, 1999.

[30] R. Dromey. Formalizing the transition from requirements to design. In *Mathematical Frameworks for Component Software Models for Analysis and Synthesis, World Scientific, Singapore*, 2007.

[31] eCos team. eCos. http://ecos.sourceware.org/, 2013.

[32] N. Evans and M. Butler. A proposal for records in Event-B. In *In Formal Methods 2006, McMaster, Canada,*, pages 221–235, 2006.

[33] A. Fathabadi, M. Butler, and A. Rezazadeh. A systematic approach to atomicity decomposition in Event-B. In *Proceedings of the 10th international conference on Software Engineering and Formal Methods*, SEFM'12, pages 78–93, Berlin, Heidelberg, 2012. Springer-Verlag.

[34] R. Feiertag and P . Neumann. The foundations of a provably secure operating system (PSOS). In *IN PROCEEDINGS OF THE NATIONAL COMPUTER CONFERENCE*, pages 329–334. AFIPS Press, 1979.

[35] J. Fitzgerald, P.G. Larsen, P. Mukherjee, N. Plat, and M. Verhoef. *Validated Designs for Objectoriented Systems.* Springer, New York (2005), 2005.

[36] International Organization for Standardization, International Electrotechnical Commission, Organisation International Electrotechnical Commission, and Organisation Internationale de Normalisation. *Information Technology- Programming Languages, Their Environments and System Software Interfaces- Vienna Development Method- Specefication Language.* ISO/IEC, 1996.

[37] L. Freitas. Mechanising data-types for kernel design in Z. In *SBMF*, pages 186–203, 2009.

[38] R.A. Gandhi and Seok-Won Lee. Visual analytics for requirements-driven risk assessment. In *Requirements Engineering Visualization, 2007. REV 2007. Second International Workshop on*, pages 6–6, Oct 2007.

[39] T.-S. Hoang, A. Furst, and J.-R. Abrial. Event-B patterns and their tool support. In *Proceedings of the 2009 Seventh IEEE International Conference on Software Engineering and Formal Methods*, SEFM '09, pages 210–219, Washington, DC, USA, 2009. IEEE Computer Society.

[40] T.S. Hoang, A. Iliasov, R.A. Silva, and W. Wei. *A Survey on Event-B Decomposition.* Electronic Commnications of the EASST. Eidgenossiche Technische Hochschule Zurich, 2012.

[41] C.A.R. Hoare. The verifying compiler: A grand challenge for computing research. *Journal of the ACM*, **50**:2003, 2003.

[42] C.A.R. Hoare and N. Wirth. An axiomatic definition of the programming language PASCAL. *Acta Informatica*, **2**[4]:335–355, December 1973.

[43] P. Hudak, J. Peterson, and J. Fasel. A gentle introduction to Haskell, haskell.org, 2000.

[44] M. Jackson. System development. England Cliffs, 1983. Prentice Hall.

[45] M. Jastram and A. Graf. Requirements, traceability and DSLs in Eclipse with the requirements interchange format (RIF/ReqIF). Technical report, Dagstuhl-Workshop MBEES 2011: Modellbasierte Entwicklung eingebetteter Systeme, 2011.

[46] M. Jastram, S. Hallerstede, M. Leuschel, and A. Russo, Jr. An approach of requirements tracing in formal refinement. In *Proceedings of the Third International Conference on Verified Software: Theories, Tools, Experiments*, VSTTE'10, pages 97–111, Berlin, Heidelberg, 2010. Springer-Verlag.

[47] M. Jastram, M. Leuschel, and J. Bendisposto. Mapping requirements to B models, 2009.

[48] C.B. Jones. *Systematic software development using VDM (2nd ed.)*. Upper Saddle River, NJ, USA, 1990.

[49] G. Klein, P. Derrin, and K. Elphinstone. Experience report: sel4: formally verifying a high-performance microkernel. *SIGPLAN Not.*, **44**[9]:91–96, August 2009.

[50] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. sel4: Formal verification of an OS kernel. In *ACM SYMPOSIUM ON OPERATING SYSTEMS PRINCIPLES*, pages 207–220. ACM, 2009.

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

[52] J. Labrosse. *Microc/OS-II*. R & D Books, 2nd edition, 1998.

[53] G. Leavens, J.-R. Abrial, D. Batory, M. Butler, A. Coglio, K. Fisler, E. Hehner, C. Jones, D. Miller, S. Peyton-Jones, M. Sitaraman, D. Smith, and A. Stump. Roadmap for enhanced languages and methods to aid verification. In *Proceedings of the 5th international conference on Generative programming and component engineering*, GPCE '06, pages 221–236, New York, NY, USA, 2006. ACM.

[54] lemma 1. ProofPower-Z. http://www.lemma-one.com/ProofPower/doc/usr011.pdf, 2011.

[55] M. Leuschel and M. Butler. ProB: A model checker for B. In *FME 2003: FORMAL METHODS, LNCS 2805*, pages 855–874. Springer-Verlag, 2003.

[56] Q. Li and C. Yao. *Real-Time Concepts for Embedded Systems*. CMP Books, 2003.

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

[58] Micrium. UCOS. http://micrium.com/, 2013.

[59] R. Milner. *Communication and concurrency*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[60] P. Neumann, R. Boyer, R. Feiertag, K. Levitt, and L. Robinson. A provably secure operating system: The system, its applications, and proofs. In *Technical Report CSL-116, SRI International*, 1980.

[61] T. Nipkow, M. Wenzel, and L. Paulson. *Isabelle/HOL: a proof assistant for higher-order logic*. Springer-Verlag, Berlin, Heidelberg, 2002.

[62] T. Perrine, J. Codd, and B. Hardy. An overview of the kernelized secure operating system (KSOS). In *In Proceedings of the Seventh DoD/NBS Computer Security Initiative Conference*, pages 146–160, 1984.

[63] G. Radha. *Pascal Programming*. New Age International (p) Limited, 1999.

[64] RayleighLord Rayleigh. Sri international. http://www.sri.com, 2008.

[65] K. Robinson. *Draft book on Event-B*. unpublished, 2010.

[66] M. Saaltink and O. Canada. The z/eves 2.0 user's guide, 1999.

[67] J. Sametinger. *Software engineering with reusable components*. Springer-Verlag New York, Inc., New York, NY, USA, 1997.

[68] R. Sarshogh and M. Butler. Specification and refinement of discrete timing properties in event-b. In *Proceedings of Automated Verification of Critical Systems (AVoCS2011)*, 2011.

[69] S. Saydjari, J. Beckman, and J. Leaman. Locking computers securely. In *In 10th National Computer Security Conference*, pages 129–141, 1987.

[70] S. Schneider. *The B method: An Introduction*. Palgrave, 2001.

[71] D. Sheppard. *Introduction to Formal Specifications with Z and VDM*. McGraw-Hill, Inc., New York, NY, USA, 1994.

[72] R. Silva and M. Butler. Supporting reuse of Event-B developments through generic instantiation. In *Proceedings of the 11th International Conference on Formal Engineering Methods: Formal Methods and Software Engineering*, ICFEM '09, pages 466–484, Berlin, Heidelberg, 2009. Springer-Verlag.

[73] R. Silva and 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, pages 122–141, Berlin, Heidelberg, 2011. Springer-Verlag.

[74] R. Silva and 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, pages 122–141, Berlin, Heidelberg, 2011. Springer-Verlag.

[75] R. Silva, C. Pascal, T.C. Hoang, and M. Butler. Decomposition tool for Event-B. *Softw., Pract. Exper.*, **41**[2]:199–208, 2011.

[76] G. Smith. *The Object-Z specification language*. Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[77] C. Snook and M. Butler. UML-B: Formal modeling and design aided by UML. *ACM Trans. Softw. Eng. Methodol.*, **15**[1]:92–122, January 2006.

[78] C. Snook and M. Butler. UML-B: A plug-in for the Event-B tool set. In *Proceedings of the 1st international conference on Abstract State Machines, B and Z*, ABZ '08, pages 344–344, Berlin, Heidelberg, 2008. Springer-Verlag.

[79] M. Spivey. *Z Notation - a reference manual (2. ed.)*. Prentice Hall International Series in Computer Science. Prentice Hall, 1992.

[80] A. Velykis and L. Freitas. Formal modelling of separation kernel components. In *Proceedings of the 7th International colloquium conference on Theoretical aspects of computing*, ICTAC'10, pages 230–244, Berlin, Heidelberg, 2010. Springer-Verlag.

[81] M. Verhoef. *Modeling and Validating Distributed Embedded Real-Time Control Systems*. Ph.D thesis, Radboud University Nijmegen (2008), 2008.

[82] M. Verhoef, P.G. Larsen, and J. Hooman. Modeling and validating distributed embedded real-time systems with VDM++. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *FM 2006: Formal Methods, 14th International Symposium on Formal Methods, Hamilton, Canada, August 21-27, 2006, Proceedings*, **4085** of *Lecture Notes in Computer Science*, pages 147–162. Springer, 2006.

[83] J. Walker, A. Kemmerer, and J. Popek. Specification and verification of the UCLA unix security kernel. *Commun. ACM*, **23**:118–131, February 1980.

[84] Wind River Systems. VxWorks. http://www.windriver.com/products/vxworks/, 2013.

[85] J. Woodcock. First steps in the verified software grand challenge. *IEEE Computer*, 2006.

[86] J. Woodcock and J. Davies. *Using Z: specification, refinement, and proof*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[87] J. Woodcock, P.G. Larsen, J. Bicarregui, and J. Fitzgerald. Formal methods: Practice and experience. *ACM Comput. Surv.*, **41**:19:1–19:36, October 2009.

[88] S. Yeganefard and M. Butler. Control systems: Phenomena and structuring functional requirement documents. In *ICECCS*, pages 39–48, 2012.

[89] D. Zowghi and V. Gervasi. The three cs of requirements: Consistency, completeness, and correctness. In *Proceedings of 8th International Workshop on Requirements Engineering: Foundation for Software Quality, (REFSQ'02*, 2002.

[90] D. Zowghi, V. Gervasi, and A. McRae. Using default reasoning to discover inconsistencies in natural language requirements. In *Proceedings of the Eighth Asia-Pacific on Software Engineering Conference*, APSEC '01, pages 133–, Washington, DC, USA, 2001. IEEE Computer Society.