Study on Verifying the Conformance of the Design to Its Formal Specification

by

Vu Dieu Huong

submitted to
Japan Advanced Institute of Science and Technology
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy

Supervisor: Associate Professor Toshiaki Aoki

School of Information Science
Japan Advanced Institute of Science and Technology

March, 2015

Abstract

This work focused on development of reactive systems. A software development process begins with informal requirements which the target software is expected to meet. The informal requirements are translated into formal specifications to ensure their consistency. Then, system designs are developed as models for implementation. Finally, the implementation is done according to the designs using programming languages. In this development process, we should verify the fact that the designs satisfy the requirements described by formal specifications since incorrect designs likely lead to significant costs caused by back track of the developments. The specification captures the external behaviors including the results of the operations of the systems. Separately, the design represents the details of how to make the results. We consider that the formal specification languages such as Z, VDM, Event-B are appropriate to describe the specification because they provide rich notions (e.g., set, relation, and function) to facilitate describing the specification. They also provide tools to assure the consistency of the specification. Promela is an appropriate language for describing the design. In Promela, the design could be described in an imperative manner. Design decisions are straightforwardly described based on complex data structures (e.g, record type and array) and various control structures (e.g, selection and loop). Therefore, we intend to use Event-B and Promela for the specification and the design to facilitate describing them. Then, we propose a framework to verify the Promela design against the specification in Event-B. This framework is to verify the reactive systems.

The first problem we must deal with is that there exists a gap between the specification and the design. The specification defines what behaviors are produced, whereas the design defines the detail of how the behaviors are produced. Since there exists such a gap, we intentionally use different specification languages: Event-B for the specification and Promela for the design. This in turn leads to the second problem; that is, we have to deal with difference of specification languages used for the specification and the design. Actions in Event-B are performed in parallel, whereas actions in Promela may be performed step by step. Therefore, a state transition in the specification may be followed by multiple state transitions in the design. Another problem is that the reactive systems just operate if they receive stimulus from the outside, so-called environments. Therefore, the design must be verified in communication with the environment. The other problem is to assure the practicality of the framework. It must provide an ability to check important properties and detect typical bugs of the reactive systems. It is also possible for users to produce inputs of the framework. These must be demonstrated by some case studies including real systems.

The first contribution of the research is a new combination between Event-B and Promela/Spin included in a framework for the verification of reactive systems. This framework is to verify the conformance of the design to its formal specification where the design and the specification are described in different specification languages. Applying the framework, we can choose appropriate specification languages to describe the specification and the design for the purpose of verifying the design. With this combination

between Event-B and Promela/Spin, we can check the design against the consistent and the correct specification. This would drastically improve the reliability of model checking results because the specification is reliable. The second contribution of the research is to fill the gap between the specification and the design. The specification defines abstract data structures, whereas the design defines implementable data structure. Also, the specification defines results of operations, while the design defines details of how to make the results. In the framework, we relate the specification to the design by common semantics, LTSs, and correspondences between state transitions given by mappings from syntactic elements in the former to those in the latter. This makes it possible to systematically verify the conformance of the design to the specification. The third contribution refers to supports for applying the framework to verify real systems. As mentioned, the users must produce the formal specifications in Event-B and the proper bounds for the verification of the system design. We give guidelines for translation from the informal specifications into the formal specification in Event-B. These facilitate the validation of the formalism. We also give a procedure to give the proper bounds to direct the verification focus on the behaviors relevant to intended properties and bugs. Therefore, we could determine appropriate bounds to avoid the state explosion when applying model checking; the critical cases could not be missed because we use proper bounds for the verification.

To evaluate the applicability and the effectiveness of our framework, we conducted some case studies in which the target systems are the reactive systems ranging from the simple systems to complex systems. Specifically, we applied our framework to verify a real system, an operating system compliant with the OSEK/VDX standard. The results of the several experiments are shown to demonstrate that this approach can be practically applied in verification of important properties and detection of typical bugs of the target systems. This exhibits an ability to deal with the specifications and the designs which are described in different specification languages. Therefore, we can choose appropriate specification languages to describe the specification and the design for the purpose of verifying the design.

Keywords: formal specification, design, formal verification, simulation relation, OSEK/VDX OS.

Acknowledgments

This thesis could not be able to be completed without the support and contribution of many people. My greatest thanks goes to Assoc. Prof. Toshiaki Aoki who supervises me during my research. His encouragement, guidance and support always lead me ahead with my research. He creates a good research environment in the lab that I can study and enjoy my study time.

I highly acknowledge Prof. Kokichi Futatsugi for being my second supervisor. I learn from him many helpful things for my research.

My thanks will go to Prof. Kazuhiro Ogata for his acceptance as my minor research project supervisor. His kindly discussions are very useful for me to complete my research.

After completing the first draft of this dissertation, I received many comments from Prof. Kunihiko Hiraishi, Prof. Kazuhiro Ogata, Assoc. Prof. Masato Suzuki from JAIST and Assoc. Prof. Fuyuki Ishikawa from National Institute of Informatics, Japan. I would like to express my sincere thanks to them.

I would like to acknowledge the useful discussion and support from Dr. Yuki Chiba. His kindly guidance and discussions strongly contribute to the results of my research, especially, the expansion of my knowledge to logical aspects of my work.

I would like to thank Dr. Kenro Yatake for his support and discussion from the very beginning of my research in Japan. He gives many helpful comments on the experimental issues as well as writing the papers.

I would like to acknowledge the Vietnamese Government and the Ministry of Education and Training - Vietnam (MOET) for financing my study in Japan. Without this support, I could not complete the doctoral course in Japan.

I would also like to acknowledge University of Engineering and Technology (UET), Vietnam National University, Hanoi, and Japan Advanced Institute of Science and Technology (JAIST), who gave me chance to join the FIVE-JAIST program and helped me to complete the program. I would like to thank my co-supervisor in UET, Assoc. Prof. Nguyen Viet Ha, for his advice, kind support, and encouragement.

I would like also to express my sincere thanks to Ms. Nao S Aoki and Dr. John Blake. Their kindly discussions help me improve my English writing and speaking skills.

I express my sincere gratitude to the former and current members of the Aoki Lab for their support and sharing with me the studying time in the lab.

I would like to thank my family including my parents, sister, and brother for their endless love. They have been with me and have supported me since the day I was born. Finally, I would like to thank my husband and my son. They has always shared gladness and sadness with me, encouraged me to complete my work.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

A software development process begins with informal requirements which the target software is expected to meet. The informal requirements are translated into formal specifications to ensure their consistency. Then, system designs are developed as models for implementation. Finally, the implementation is done according to the designs using programming languages. In this development process, we should verify the fact that the designs satisfy the requirements described by formal specifications since incorrect designs likely lead to significant costs caused by back track of the developments.

We focus on the development of automotive systems. Automotive systems are widely used in industry and our daily life. Most of them are considered as safety-critical because their failure may result in loss of lives and assets (e.g. software systems for mobile vehicles). As the reliability of automotive systems is becoming a greater challenge in our community, increasingly more automotive companies are interested in applying formal methods to improve the reliability of automotive systems. We are working on a real design of an automotive operating system (OS, for short) conforming to OSEK/VDX standard [40] (OSEK OS, for short). Such operating system is considered as important components to ensure the reliability of the automotive systems. The aim of our work is to produce a high quality OS by applying automated formal verification.

An operating system is a reactive system responding to environmental stimuli. Reactive systems do not execute by themselves but in combination with their environments. Environments are the external systems which invoke the services of the target systems, e.g. software applications running on the operating systems. The specification of such a reactive system represents its externally visible behavior. That is, the specification represents what the system does in response to the invocations of its environments. Formal specification languages such as VDM[36], Z[39] and Event-B[1] allow us to formally describe the specification. On the other hand, the design represents the collaboration of internal components to realize observable behaviors described in the specification. It usually contains implementable data structures such as record types, flags, and hash tables. We consider that imperative specification languages like Promela/Spin [23] are appropriate to describe the design since the data structures and behaviors based on them can be straightforwardly described. Accordingly, we intentionally use Event-B for the specifications and Promela/Spin for the designs to facilitate describing them. The problem is how to verify the designs with respect to their specifications when they are described in

different specification languages.

One may say that some of the formal specification languages provide refinement and automatic generation of codes. In this way, we can describe the specification in an appropriate specification language; then, we derive the behaviors of the design from the higher-level specification as adopted in [46]. In order to represent highly optimized behaviors of reactive systems, we need to use various complex data structures and control structures. Therefore, directly deriving these behaviors from the abstract specification is generally very hard. Our idea is to describe the design in the specification language which is easy to represent the design. Then, we verify the design against the specification. Our approach provides another way to ensure that the design is consistent with the specification. Another question may arise here. The specification can be described in temporal logic if we describe the design in Promela/Spin. This approach is adopted in [10]. The advantage of using Promela is that it allows us to design the highly optimized behavior of the OS in an imperative manner using various data structures. However, we consider that temporal logic formulae, which allows us to describe properties about invariants on some variables and the relative order of event calls, are not adequate for describing the important properties of the OS. What we need to verify about the OS is the correctness of the scheduling which can be precisely described by specifying the pre-condition and the post-condition of each event. For example, when an activation event of a task is called, the task must become running and the currently running task must become ready in the states just after the event is called. To specify such a property in temporal logic, as discussed in [18], we need to explicitly define the execution steps of the events. This makes the formulae complex and prone to mistakes. Whereas, by using the rich notions (e.g. sets and relations) in the formal specification languages like Event-B[1][45], one could easily describe such properties. Therefore, we intentionally use Event-B to facilitate describing properties of the OS. In Event-B, one can describe the system as a set of events and the behavior of each event can be specified as pre-conditions and post-conditions using the rich notions. It also provides a facility to verify the consistency and the correctness of the properties. Thus, we think that dealing with the specification and the design based on the different specification languages is appropriate for systems in which there exist a big gap between the specification and the design like the operating systems. For these reasons, Event-B and Promela are intended to describe the specification and the design in our verification of the reactive systems. Consequently, we aim at an approach to ensure the conformance of design to its formal specification where the design and the specification are described in different specification languages. In such a combination between Event-B and Promela/Spin, the designs of the reactive systems could be verified with respect to reliable specifications.

1.2 Problem Statement

The first problem is that there exists a gap between the specification and the design. The specification defines abstract data structures and a collection of guarded events which represent effects of system services. The design defines implementable data structures and a collection of functions which describe highly optimized behaviors and design decisions using various control structures. Since there exists such a gap, we intend to use Event-B and Promela to facilitate describing the specification and the design. This in turn

leads to the second problem; that is, we have to deal with difference of specification languages used for the specification and the design. Actions in Event-B are performed in parallel, whereas actions in Promela may be performed step by step. Therefore, a state transition in the specification may be followed by multiple state transitions in the design. For these, we define the semantics of the specifications and the designs commonly as labelled transition systems (LTSs). We also define a simulation relation [31][34][43] between them and verify that for all execution sequences of the specification and the design, this simulation relation holds. In this approach, we generate a labeled transition system from the formal specification; and, from each state, verification conditions which must be met by the corresponding state of the design are generated. Then, we apply model checking [5] to the design to check the verification conditions. In this way, we can check the correspondence of state transitions, or simulation relation, between the specification and the design. This shows that the design conforms to the specification.

The third problem is verifying a design of OSEK OS. We have to verify essential behaviors of OSEK OS, especially the correctness of scheduling. As an automotive industry standard of operating system specification, the OSEK/VDX OS specification [40] is widely applied in the process of designing and implementing the operating system for automotive systems. For applying our framework to verify whether an OS design conforms to the OSEK/VDX specification, we consider two sub-issues. One of sub-issues is that the OSEK/VDX OS specification is described in natural language. It is not a direct input of a formal verification. We need to formalize this specification in Event-B and apply the framework to verify whether the design conforms to the Event-B specification replacing for the informal specification provided by OSEK/VDX group. Since there exists the OSEK/VDX OS specification provided by OSEK/VDX group, we cannot change it but faithfully formalize it. In this case, we need to validate the formal specification against the informal specification. This validation cannot be checked by the machine; however; it is important that the validation can be accepted by the clients or the stakeholders. Therefore, we give a procedure to facilitate the validation of formalization. It is presented in the form of guidelines to faithfully formalize informal specifications in Event-B. The other sub-issue is caused by the fact that when checking the simulation relation between the specification and the design, we need to generate the LTS from the specification. The problem is that generating all possible execution sequences from the specification makes the size of the LTS so large that it has a tendency to cause the state explosion [12] when we apply model checking. To avoid this, we explore only behaviors relevant to the properties and the bugs to be checked. Consequently, we must direct the verification focus on the behaviors relevant to intended properties and bugs. We restrict the behaviors to be checked depending on the corresponding properties. We define such restriction as bounds of the verification. Thus, we need to give proper bounds for the verification of essential behaviors. From results of verifying the design of OSEK OS, we have to evaluate the practicality of the framework. We consider the practicality of the framework according to some perspectives such as possibility for users to produce inputs of the framework, ability to verify important properties and detect typical bugs of real systems, and whether the scalability is reasonable for real systems.

1.3 Contribution and Impact

The first contribution of the research is a new combination between Event-B and Promela/ Spin included in a framework for the verification of reactive systems. This framework is to verify the conformance of the design to its formal specification where the design and the specification are described in different specification languages. The framework is precisely defined with a formal model whose essential parts are the model of the specification, the design, the environment, the communication between the design and its environment, the bounds, and the simulation relation of two LTSs within the bounds. Applying the framework, we can choose appropriate specification languages to describe the specification and the design for the purpose of verifying the design. The new combination between Event-B/Rodin and Promela/Spin is that: the proving in Event-B for the reliability of the specification and the model checking with Promela/Spin to make sure that operations described in the design preserve the pre-conditions and the post-conditions as described in the specification. With such a combination, we can check the design against the consistent and the correct specification. This would drastically improve the reliability of model checking results because the specification is reliable.

The second contribution of the research is to fill the gap between the specification and the design. The specification defines abstract data structures, whereas the design defines implementable data structure. Also, the specification defines results of operations, while the design defines details of how to make the results. In the framework, we relate the specification to the design by common semantics, LTSs, and correspondences between state transitions given by mappings from syntactic elements in the former to those in the latter. This makes it possible to systematically verify the conformance of the design to the specification.

The third contribution refers to supports for applying the framework to verify real systems. As mentioned, the users must produce the formal specifications in Event-B and the proper bounds for the verification of the system design. We give guidelines for translation from the informal specifications into the formal specification in Event-B. These facilitate the validation of the formalism. We also give a procedure to give the proper bounds to direct the verification focus on the behaviors relevant to intended properties and bugs. Therefore, we could determine appropriate bounds to avoid the state explosion when applying model checking; the critical cases could not be missed because we use proper bounds for the verification.

1.4 Dissertation Organization

The dissertation is organized as follows.

The first chapter (this chapter) is the introduction about the research. This chapter sets the context, describes the main contributions, and presents the structure of the remaining part of the dissertation.

Chapter 2 is about the background of the research. This chapter presents basic concepts referred in the dissertation. This chapter also includes an overview of formal methods for modelling and analysis which are used in our framework: Promela/Spin and Event-B/Rodin. In addition, this chapter summarizes the OSEK/VDX OS Specification.

Chapter 3 presents our framework to verify the conformance of the designs to their formal specification. This framework is for verification of the reactive systems. In this

chapter, we use a sample system to analyze a gap between the specifications and the designs of the reactive systems. We also present models of the specifications and the designs. After that, we present a particular simulation relation between the specification and the design and present an approach which is based on such a simulation relation to ensure the conformance of the designs to their specification. In this part, we present a formal model of the framework and our generator to automate steps of the framework. We also present applications of our framework to simple systems to discuss on generality and practicality of the framework.

Chapter 4 presents a case study to verify an OSEK OS design using its formal specification. We present a workflow to apply the framework in verification of the OS design using its specification in Event-B. After that, we present our process of formalizing the OSEK/VDX OS specification in Event-B. We apply the the framework flexibly: various ranges are considered to give appropriate bounds for the verification; and they need to be defined depending on the properties to be checked. In the case study, we mainly explain how we defined the appropriate bounds to check our desired properties. We also show the results of the experiments and evaluate the effectiveness and the practicality of the framework.

In chapter 5, we discuss on some aspects of the framework and evaluate advantages and disadvantages of our approach. In chapter 6, we present related works. Chapter 7 is the conclusion of the dissertation and a short description of our future works.

In this dissertation, Chapters 3, 4 deliver the main works of the research.

Chapter 2

Background

This chapter briefly introduces background of the dissertation. The first section of the chapter is about definitions of the simulation relation. Promela/Spin and Event-B/Rodin are separately introduced in Section 2.2 and Section 2.3. Our work aims at a combination of these two techniques and its application. The final section presents the main part of the OSEK/VDX standard, that is the OSEK/VDX OS specification.

2.1 Simulation Relation

The simulation relations (simulations, for short) have been considered in many earlier works. The simulations are generally defined based on in state-based formalisms. Typically, [31] defines the simulation between two automata. [34, 22, 51] define the simulation between two LTSs. We present here the simulation between two automata of [31] and the simulation between two LTSs of [51] because our simulation, which will presented in Chapter 3, is close to these definitions.

We begin with a definition of an automaton and the simulation between two automata of [31]. An automaton A consists of:

- a set Q of states
- a nonempty set $I \subseteq Q$ of start states
- a set Σ of actions that includes internal action τ,
- a set $\delta \subseteq Q \times \Sigma \times Q$ of transition steps

Note: τ denotes the internal action. If α is a sequence of actions then $\widehat{\alpha}$ is the sequence obtained by deleting all τ actions from α.

Refinement. A forward simulation from A to B is a relation f over Q_A and Q_B that satisfies:

- If $s \in I_A$ then $r(s) \in I_B$
- If $s \xrightarrow{a} s' \in \delta_A$, then $r(s) \xrightarrow{\widehat{a}} f(s') \in \delta_B$

The refinement is the simplest kind of simulation of [31].

Forward simulation. A forward simulation from A to B is a relation f over Q_A and Q_B that satisfies:

- If $s \in I_A$ then $f(s) \cap I_B \neq \emptyset$
- If $s \xrightarrow{a} s' \in \delta_A$ and $u \in f(s)$, then there exists a state $u' \in f(s')$ such that $u \xrightarrow{\widehat{a}} u' \in \delta_B$

The forward simulation is generalization of the refinement that allows a set of states of B to correspond to a single state of A. We write $A \preceq_b B$ if there exists a forward simulation from A to B.

Backward simulation. A backward simulation from A to B is a relation b over Q_A and Q_B that satisfies:

- If $s \in I_A$ then $b(s) \subseteq I_B$
- If $s \xrightarrow{a} s' \in \delta_A$ and $u' \in b(s')$, then there exists a state $u \in f(s)$ such that $u \xrightarrow{\widehat{a}} u' \in \delta_B$

We write $A \preceq_b B$ if there exists a backward simulation from A to B.

We now present a definition of LTS and the simulation between two LTSs of [51]. An LTS is a tuple $T = (Q, \Sigma, \longrightarrow)$ that consists of:

- A set Q of states,
- A set Σ of labels,
- A transition relation $\longleftrightarrow \subseteq Q \times \Sigma \times Q$.

Simulation. Let $T1 = (Q_1, \Sigma_1, \longrightarrow_1)$ and $T2 = (Q_2, \Sigma_2, \longrightarrow_2)$ be two transition systems over the same label set Σ. The relation $S \subseteq Q_1 \times Q_2$ is called a simulation if for all $(q1, q2) \in S$ and $q1 \xrightarrow{a} q1'$, then there exists $q2' \in Q_2$ with $q2 \xrightarrow{a} q2'$ and $(q1', q2') \in S$.

According to [51], the simulation between LTSs are used to formally define when one transition system implements another.

2.2 Promela/Spin

Promela (Process Meta Language)[38] is a specification language, which is used to describe an abstraction of the system. Promela allows to model concurrently executing processes; communication between these processes is via message channels. Spin[23] is a model checker for analyzing the correctness of systems described in Promela. It is so-called a verifier for the temporal logic properties [19] of models described in Promela. Spin provides a simulation mode and a verification mode. In the simulation mode, Spin outputs traces of system behaviors. In the verification mode, Spin explicitly explores reachable states of the system and checks whether each state violates the desirable property. If so, Spin outputs a trace for the bugs in the model.

2.2.1 Promela

A Promela model may consist of processes, message channels, and variables. Figure 2.1 show samples of Promela models.

A description of process contains data declarations (e.g., `byte tmp;`) and statements (e.g., `count:=count+1`). Basic data types in Promela include `integer` data types, `bits`,

```
#define cname  cvalue
type vname1, vname2;
chan qname = [size] of {type}

proctype A(){
        qname!vname1 ...
}
proctype B(){
        qname?vname2 ...
}
init{
        run A(); run B();
}
```

```
byte state = 1; byte count;
proctype A(){
    byte tmp;
    (state==1) -> tmp = state; tmp = tmp+1; state = tmp
}
proctype B(){
    do      :: (count != 0) ->  If   :: count = count + 1
                                     :: count = count – 1
                                fi
            :: (count == 0) -> break
    od
}
init{
    run A(); run B()
}
```

Figure 2.1: Promela models

and `boolean`. Users can define complex data structures such as `record types`, `arrays` (one-dimensional arrays). Promela is an imperative language with C-like syntax. The statements may be assignment of values/expressions to variables, control structures, i.e. `atomic` sequences, conditional (`if-statement`), repetition (`do-statement`), and unconditional jumps (`go to`). The `if-statement` may have multiple choices (guards). If there are at least two choices executable, it is executable and the guard is chosen nondeterministically.

Message channels are used to describe the transfer of data from one process to another. They are declared either locally or globally. The statement `qname!vname1` sends the value of `vname1` to channel `qname`: the value of `vname1` is appended to the tail of the channel. The statement `qname?vname2` receives the value from the head of the channel, and stores it in variable `vname2`. The channels pass messages in First-In-First-Out[30] order. In these sample cases, only a single value is passed through the channel. There may be more than one value to be transferred per message. If so, multiple values must be separately listed: each separated by a commas.

2.2.2 Description of properties to be checked

The properties could be described according to various types: basic assertions, LTL formulas (Linear Temporal Logic, a branch of temporal logic), end-state labels, progress-state labels, accept-state labels, never claims, trace assertions. The basic assertions, the LTL formulas and the never claims are usually used.

The basis assertions are inserted inside process declarations with the following syntax: `assert(<boolean expression>)`,e.g., `assert(a > b)`. The expression is evaluated each time the assert statement is executed. If the expression is false, an assertion violation is outputted. The assertions are usually used to define properties that should be hold in specific reachable states. To assert that an expression is true in every reachable state, one must define it in a independently process from the scope of the system.

The LTL formulas are specified globally, that is outside all process declarations, with the following syntax: `ltl [ name ] { formula }`. An LTL formula is made up from a finite set of propositional variables, the logical operators ! (negation), && (and), || (or), and the temporal modal operators X (neXt) and U (until). In addition, there are additional logical and temporal operators as follows:

- $\rightarrow$: implication

- $\Leftrightarrow$: equivalence
- []: always
- <>: eventually
- W: weak until

[18] presents patterns to define some typical properties in the LTL formulas. For example, **Precedence** is a pattern that defines "a state/event P must always be preceded by a state/event Q within a scope" as $(<> P \rightarrow (!P\ U\ (Q\ \&\&\ !P)))$ and some extensions; **Response** is the one that defines "a state/event P must always be followed by a state/event Q within a scope" as $([]P \rightarrow <> Q)$ and some extensions.

Promela does not contain syntax to specify temporal logic formulae; however, SPIN has a parser to read and translate temporal logic formulae into Promela syntax. In particular, LTL formulas are translated into never claims. Therefore, temporal logic is a part of the language that is accepted by SPIN. However, we use temporal logc to specify only correctness requirements but not models of target systems.

The never claims also can either be written manually or they can be generated mechanically from LTL formulas. Structure of the never claims is as below:

```
never /* [] p */ {
    do
        :: p
    od
} /* where p is a boolean expression in Promela */
```

It can be hard to specify intended properties by directly formalizing them in the never claims. One usually formalizes them in LTL formulas and then translate LTL formulas into the never claims because LTL formulas are more understandable than the never claims. However, as discussed in [38], temporal logic formulas are also strictly less expressive.

2.2.3 Spin

Spin is an on-the-fly model checker: constructing the state space and checking the error at the same time. The main advantage of on-the-fly techniques is that the algorithms execute until an error is found. Thus, the entire state space need not be explored and errors, if any, are detected quickly. In the worst case, if the system model is correct, the entire state space is explored. Therefore, on-the-fly techniques are suitable for systems detecting errors in the initial stages of design. If errors are found, counter-examples are generated to help the user with correcting them.

During analysis, Spin stores all data representing reachable system states, such as local process states, local and global variables, and channel contents, in a data structure called "state vector". In default method of storing the state space in Spin, all states are explicitly stored, as shown in Figure 2.2. Looking up is fast because of hash function. Total amount of memory used to store the state space is `n*m bytes + hash table`, where `n` is the total number of states and `m` is the size of a state vector.

Spin uses the search algorithms such as the depth-first search and the bread-first search to explore the complete state space and check the correctness properties of Promela models. Some optimized search algorithms are also integrated into Spin to deal with large state spaces. For example, the partial order reduction techniques are included to reduce

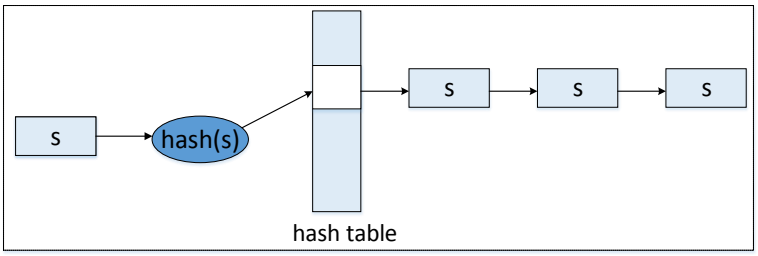


Figure 2.2: Storing states in Spin

the size of the state space; and, the bit-state hashing is applied to reduce the total memory required for storing each state.

Depth-first search.

Depth-first search (DFS) is a search algorithm on state transition graphs. The algorithm begins with the initial state and developed as far as possible according to each branch of the graph. It traverses the depth of any particular path before exploring its breadth. Typically, the search process is developed until it finds the first child node that meets some conditions or until it finds a node without child node. The algorithm then backtracks previously visited nodes. The search process on a node finishes only when all children have finished. The following is pseudo code for the DFS algorithm where the state space is implemented as a stack.

```
procedure dfs(s: state)
    if error(s) then report error fi
    add s to Statespace /* states are stored in hash table */
    foreach successor t of s do
        if t not in Statespace then dfs(t) fi
    od
end dfs
```

Figure 2.3 illustrates the DFS algorithm. Applying the DFS algorithm to the graph to find G, the order of nodes visited are A, B, E, C, F, G. Also, to find F, the order of nodes visited are A, B, E, C, F.

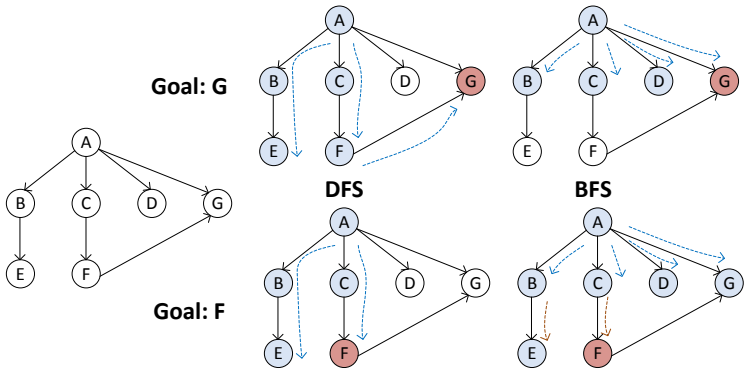


Figure 2.3: A state transition graph

Breadth-first search.

Breadth-first search (BFS) is a search algorithm on state transition graphs. The algorithm begins with the initial state and visits all neighbour nodes until it finds a node that satisfies some conditions. The following is pseudo code for the BFS algorithm. The state space is implemented as a queue.

```
procedure bfs()
    remove s from Statespace
    if error(s) then report error fi
    foreach successor t of s do
        if t not in Statespace then
            add t to Statespace
            bfs()
        fi
    od
end bfs
```

Figure 2.3 also illustrates the BFS algorithm. Applying the BFS algorithm to the graph to find G, the order of nodes visited are A, B, C, D, G. The BFS algorithm requires to store 5 states whereas the DFS algorithm requires to store 6 states. In this example, it seems that the DFS algorithms requires much more memory to store states than the BFS algorithm does. However, if we consider an example at the bottom of the figure, to find F by the BFS, the order of nodes visited are A, B, C, D, G, E, F. The BFS algorithm requires to store 7 states whereas the DFS algorithm requires to store 5 states. Generally, the BFS algorithm requires more memory that the DFS algorithm; but, it could find the goal faster than the DFS algorithm.

Verification with Promela/Spin.

The system models are described in Promela and the properties to be checked are described in various forms such as the LTL formulas and the assertions. A typical output of a verification is illustrated in Figure 2.4.

In the output, the plus indicates that the search checked for violations of user specified assertions, and the presence of acceptance. The plus indicates that the search did not check for LTL formulas and invalid end states. The complete storage of a global system state required 36 bytes of memory (per state). The longest depth-first search path is 6 (number of transitions from the root of the tree, i.e., from the initial system state).

- 1 errors were found in this search; accordingly, a counter-example trace is generated in a ".trail" file
- 69 unique global system states were stored in the state space.
- 69 transitions were explored in the search
- Total memory usage was 129.022 Megabytes, including the stack, the hash table, and all related data structures.

The verification executes until an error is found as shown in the sample of Figure 2.4. Thus, the entire state space need not be explored and errors, if any, are detected quickly.

```
pan:1: assertion violated 0 (at depth 1192)
pan: wrote 2t1e_error.pml.trail

(Spin Version 6.4.2 -- 8 October 2014)
Warning: Search not completed
      + Partial Order Reduction

Full statespace search for:
      never claim            - (none specified)
      assertion violations   +
      acceptance   cycles    - (not selected)
      invalid end states     +

State-vector 560 byte, depth reached 1192, errors: 1
       69 states, stored
        0 states, matched
       69 transitions (= stored+matched)
     1125 atomic steps
hash conflicts:         0 (resolved)

Stats on memory usage (in Megabytes):
    0.039   equivalent memory usage for states (stored*(State-vector + overhead))
    0.520   actual memory usage for states
  128.000   memory used for hash table (-w24)
    0.534   memory used for DFS stack (-m10000)
  129.022   total actual memory usage
```

2t1e_error.pml.trail

```
BEGIN2:ActivateTask(0, 2)
END
BEGIN2:WaitEvent(2, 1)
END
BEGIN2:ActivateTask(0, 3)
END
BEGIN2:TerminateTask(3)
END
BEGIN2:SetEvent(3, 2, 1)
END
BEGIN2:WaitEvent(2, 1)
END
BEGIN2:TerminateTask(2)
END
BEGIN2:WaitEvent(2, 2)
```

Figure 2.4: Output of Spin Verifier

2.3 Event-B/Rodin

Event-B and the Rodin platform [1][45] are a formal specification language and an environment for system modeling and analysis. Event-B is developed from B method [49]. It is intended to model event-driven reactive systems by incorporating action system formalism[4] into B method. The Rodin platform is easy to use and extend. It provides support for refinement and mathematical proof. In order to analyze consistency of an Event-B model, the Rodin tool generates verification conditions in terms of proof obligations. Some of the proof obligations could be discharged automatically by the Rodin tool, the others require interactive proof.

2.3.1 Event-B

Event-B is a formal specification language based on set theory, arithmetic, and the first order logic. Typing is expressed through set membership.

- First Order Logic: connective, quantifiers,
- Set theory: sets, relations, functions,
- Arithmetic (on Z, N, N1)

The main concept of formal development in Event-B is a model. An Event-B model may include several contexts and machines. Within a context, the clause “sets” and the clause “constants” are defined. Machine defines dynamic behavior of system by defining events that could occur. Machine models state of system and conditions that must apply if an event occurs and the changing of system state. A machine may see one or more contexts. The later machine may refine the former machine. We say that these two machines have a refinement relationship. Figure 2.5 demonstrates structures of the contexts and the machines in Event-B.

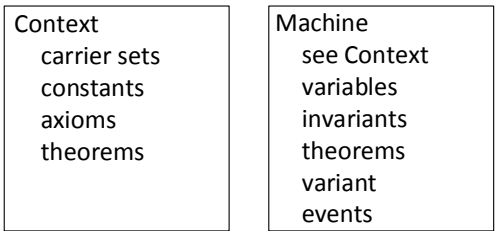


Figure 2.5: Structure of Context and Machine in Event-B

A context contains declarations of carrier sets, constants and axioms about these constants and sets. Axioms define the main properties of the constants. Theorems denote properties to be proved from the axioms. The contexts can be extended.

In a machine, the constants and the axioms are imported from contexts. Each variable has some domain of values. States of the machine are described by variable values. The consistency of the model is defined by a set of invariants. They are constraints that the variable values must satisfy in every state. Theorems are provable from invariants and seen axioms. A set of events describe the possible transitions of the machine's state. Descriptions of the events contain the condition under which the events may fire and changes of the states made by the events. It takes the form of a set of (parallel) assignments or substitutions. Any abstract machine must define a special event called initialisation for giving initial values to the variables of the machine. Events (other than initialization) are described in the form of:

- Guards: define the condition that the event can be triggered,
- Actions: reassign some variables of the machine

Each event has the form $evt = any\ x\ where\ G(x, v)\ then\ A(x, v, v')\ end$, where x are parameters of the event, $G(x, v)$ is the guard and $A(x, v, v')$ is the action. The guard of an event is the necessary condition for the event to be enabled. The action of an event comprises several assignments, each has one of the following forms:

- $v := E(x, v)$: deterministic substitution, that updates variable v to be a value $E(x, v)$.
- $v :\in E(x, v)$: non-deterministic substitution, that updates variable v to be either some member of a set $E(x, v)$.
- $v :| Q(x, v, v')$: non-deterministic substitution, that updates variable v to satisfy a before-after predicate $Q(x, v, v')$.

Actions are supposed to be performed in parallel.

Event-B provides an approach to model the system step by step by a technique, that is *refinement*. Refinement is a technique that supports modeling the system gradually from a high level of abstraction to lower levels. In Event-B, a machine can be refined by adding new variables, new invariants, new events or more details for the existing events. Figure 2.6 shows a refinement relationship between two machines, namely `M0` and `M1`.

We say `M0` is refined by `M1`; or, `M1` is a refinement of `M0`. In `M1`, invariant `I(v,w)` establishes a link between variables in `M0` and `M1`. If in the refinement we add new events, which are considered as internal, a variant `V` is introduced in such a way that it must

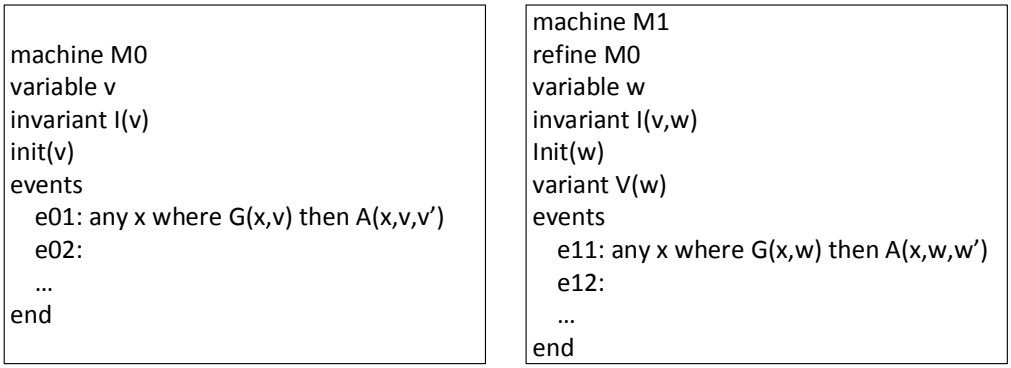


Figure 2.6: A machine and a refinement of it

be decreased by the new events. This is to ensure that the new events are not executed forever.

The refinement facilitates modeling the systems because we can postpone dealing with some features of the systems to later refinement steps.

2.3.2 Rodin

The Rodin tool supports not only describe but also verifying Event-B models. In particular, it has a capable of checking the internal consistency. This capability is provided by Proof Obligation Generator and Prover [1], which are included in the Rodin tool. The generator generates verification conditions as proof obligations. Table 2.1 shows types of proof obligations verified for consistency of the Event-B models. We divide types of proof obligations into two groups: (I) conditions for the consistency of elements in the same machine; and (II) conditions for the consistency of elements in different machines, which have a refinement relationship to each other. The former makes sure that the invariants are true before the event, and if the event's guard is true then the invariant must be true after the event. The latter makes sure that the guards of the concrete event are stronger than those of the abstract event; and, the effect of the concrete event and that of the abstract event are not contradictory.

The prover supports for discharging some proof obligations generated. In case of non-automatic proofs, it helps to interact with the tool. Consequently, the internal consistence in the whole of formal model could be ensured.

2.4 OSEK/VDX Operating Systems

The OSEK/VDX international standard [40] is widely applied in the development process of automotive systems. The OSEK/VDX OS specification (OSEK specification, for short) is a part of OSEK/VDX standard which offer big advantages for automotive systems such as portability, reliability and integration. The OSEK specification defines the operating system of automotive systems. Increasingly more organizations have been implementing operating systems based on this specification.

The OSEK specification mainly specifies the behaviour of automotive operating systems from the viewpoint of service users. Besides describing the concepts of operating

Table 2.1: Some types of proof obligation

Group	Name	Verification condition
I	Initialization/inv/INV $\texttt{init(v)} \vdash \texttt{I(v)}$	Initial values must satisfy all invariants
I	Event/grd/WD $\texttt{G(x,v)} \vdash \texttt{I(v)}$	Guard conditions are not inconsistent with invariant expressions
I	Event/inv/INV $\texttt{G(x,v)} \wedge \texttt{A(x,v,v')} \vdash \texttt{I(v')}$	Invariant is preserved by each event
II	Event/grd/GRD $\texttt{I(v)} \wedge \texttt{I(v,w)} \wedge \texttt{G(x,w)} \vdash \texttt{G(x,v)}$	Ensure that the concrete guard in the refining event are stronger than the abstract one
II	Event/act/SIM $\texttt{I(v)} \wedge \texttt{I(v,w)} \wedge \texttt{G(x,w)} \wedge \texttt{A(x,w,w')} \vdash \exists \texttt{v'}(\texttt{A(x,v,v')} \wedge \texttt{I(v',w')})$	Ensure that the concrete guard in the refining event are stronger than the abstract one

systems, the OSEK specification defines a set of system services. The system services are interfaces between the application software and the OSEK operating system. These services are visible to the users and invoked from outside of OSEK OS. These services are also called 'events' that are triggered by the application software.

Some main concepts of OSEK OS are as follows:

- **Task**

  A complex software can be subdivided in tasks. Tasks are assigned by the priorities. Tasks have to change between some states. OSEK OS makes task state transitions using some system services. Two types of tasks are described in OSEK specification: the basic tasks and the extended tasks. The extended tasks have four states. They are **sus** as suspended, **rdy** as ready, **run** as running and **wait** as waiting. The basic tasks have three states. They are **sus** as suspended, **rdy** as ready, and **run** as running.

- **Resource**

  Resources are entities that are typically shared among several tasks. The resources are assigned priorities, which are determined at the system generation. The priority of each resource are determined as follows: it must at least equal to the highest priority of all tasks that access the resource; and it must be lower than the lowest priority of all tasks that do not access the resource. For this reason, it is a so-called priority ceiling.

- **Ready queue**

  Ready queues are used to store instances of tasks that are currently in ready state. There are several ready queues for different task priorities. The ready queues work

according to the First In First Out rule [30].

The service functions of OSEK OS are classified into groups: task management, resource management, event control, interrupt management and alarms.

Task management

Task management includes activation, termination of tasks, and task switching.

Task state model. A task has to change between some states. Extended tasks have four task states: suspended, ready, waiting, and running. Table 2.2 describes state transitions of the extended tasks.

Table 2.2: State transitions for extended tasks

Transition	Former state	New state	Description
activate	suspended	ready	A new task is set into the ready state by a system service. An identifier of the task is added into the ready queue.
start	ready	running	A ready task starts at the first instruction. The scheduler decides which task in the ready queues is executed based on the priority and the activation order of tasks.
wait	running	waiting	A task transfers from running to waiting to wait for an event. This is caused by a system service.
release	waiting	ready	A task have just been set at least one event; it becomes ready and it identifier is added into the ready queue.
preempt	running	ready	The scheduler decides to start another task if the task priority is higher than the priority of the running task. At the same time, the running task transfers to the ready state.
terminate	running	suspended	The running task causes its transition into the suspended state by a system service.

The state model of basic tasks is similar to the state model of extended task; however, it contains transitions between three states: suspended, ready, and running. Table 2.3 describes state transitions of the basic tasks.

Activating a task. Tasks are activated by the operating system services including ActivateTask or ChainTask. After being activated, the task will start at the first instruction. Multiple activation requests are possible for the basic task but not the extended task. If the maximum number of multiple requests of a basic task has not been reached, the request for that task is added into the ready queues. The activation requests of tasks are queued according to the task priority and the activation order.

Task priority. The scheduler decides on the task priority and the activation order which task in the ready queues is executed next. The priority of a task is defined statically, i.e., at the time of system generation.

Table 2.3: State transitions for basic tasks

Transition	Former state	New state	Description
activate	suspended	ready	A new task is set into the ready state by a system service. An identifier of the task is added into the ready queue.
start	ready	running	A ready task starts at the first instruction. The scheduler decides which task in the ready queues is executed based on the priority and the activation order of tasks.
preempt	running	ready	The scheduler decides to start another task if the task priority is higher than the priority of the running task. At the same time, the running task transfers to the ready state.
terminate	running	suspended	The running task causes its transition into the suspended state by a system service.

Tasks with the same priority are started depending on their order of activation. A preempted task is put into the first position in the ready queue of its current priority. It is treated as the oldest activated task. A task becoming ready from the waiting state is put into the last position of the ready queue of its priority. It is regarded as the newest activated task. The scheduler uses multiple ready queues to manage the tasks in the ready state, each for a distinct priority. The ready queues work according to the First In First Out rule. Inserting an item in the queues follows the order of activation. Retrieving an item from the queues follows the scheduling.

Scheduling based on the priority and the activation order:

- Searching for the set of tasks in the ready queue with the highest priority,
- Finding the oldest activated task within the set of tasks returned from the previous step.

Scheduling policy. Two scheduling policies are available: full preemptive scheduling and non preemptive scheduling. Full preemptive scheduling allows a task with higher priority than the currently running task to stop the running task. In this situation, the preempted task is put into the ready state and resumed later when the higher priority task has finished its execution. The task context is saved so that the preempted task can be continued at the location where it was preempted. Rescheduling is performed when activating a task. Non preemptive scheduling allows a running task with lower priority to delay the start of a task with higher priority until the next point of rescheduling. Rescheduling is not performed when activating a task but terminating a task or having an explicit call from the scheduler.

Termination of tasks. In the OSEK operating system, a task terminates by itself. Termination of a task is caused by service TerminateTask or ChainTask. When TerminateTask is called, the calling task become suspended. Also, when ChainTask is called, the calling task terminates and the succeeding task is activated at the same time. The identifier of the activated task is put into the last position of the priority queue.

Interrupt processing

In addition to the task, another entity, which also competes for CPU is Interrupt Service Routine (ISR or interrupt, for short). ISRs are also managed by the OSEK OS. ISRs have higher priorities than the tasks. ISRs can interrupt the execution of the tasks. Service functions for interrupt processing includes SETINTR and RESETINTR.

Event mechanism

Events are entities managed by the operating system. Events are assigned to extended tasks. Each extended task can owns some events. Every task can set events for the extended task; only the owner can clear its event. When a running task requires an event, it transfers to waiting state to wait for that event. When a task in waiting state has already received an event that it is waiting for, it transfers to ready state.

Resource management

Resources include scheduler, memory areas, hardware, and program sequences. They may be occupied by the tasks and ISRs. The resource management is used to arrange accesses of several tasks and ISRs to the resources. Constraints on the resource management are as follows:

- Two tasks or ISRs cannot occupy the same resource at the same time.
- Priority inversion cannot occur.
- Deadlocks do not occur by use of these resources.
- TerminateTask, ChainTask, WaitEvent cannot be called by a task that is currently occupying a resource.
- An ISR cannot be finished when it is occupying a resource.
- One task can occupy multiple resources. In this case, resources must be requested and released following the Last In First Out principle [30].

The priority inversion is a case that some task with the lower priority delays the execution of a task with the higher priority when the higher priority task is waiting for a resource that is already occupied. The deadlock is a case in which two tasks are waiting for two separate resources; however, the first task is waiting for the resource that is already occupied by the second task; and the second task is waiting for the resource that is already occupied by the first one. With the priority ceiling of the resources, we could prevent the priority inversion and the deadlock situation.

2.5 Summary

In this chapter, we have presented two formal methods for modelling and analysis using Promela/Spin and Event-B/Rodin. As discussed in [2], Event-B is preferred to describe the system at the high level of abstraction. Whereas, Promela is intended to describe

the system at the low level of abstraction. With Event-B/Rodin, one could describe the system from the high level to the low level of abstraction by applying the refinement technique. However, this requires much interactive proof to confirm the conformance between two levels. Promela/Spin provides a completely automatic verification of the system model against the correctness properties. Nevertheless, it does not facilitate ensuring the consistency of the properties to be checked. Therefore, our work benefits Event-B/Rodin and Promela/Spin in a verification framework for the reactive systems. This framework is presented in the remaining chapters.

Chapter 3

A Framework for Verifying a Design against Its Formal Specification

Reactive system is a system that continuously interact with its environment by responding to external stimuli. For example, vending machines, elevator, and operating systems are reactive systems. The environment of a vending machine includes customers who want to buy products restocked in that vending machine; and, the environment of an operating system includes application softwares running on that operating system. Several reactive systems are considered as safety-critical because their errors cause highly destructive effects on the human life and the assets (e.g., operating systems for mobile vehicles). Reactive systems do not operate by only themselves but in communication with their environment. Therefore, reactive systems cannot be verified without considering how they communicate with their environment.

Specifications of reactive systems generally describe the desirable properties and the external behaviors of the systems based on the mathematical data structures using notions of set, relation, and function. Designs must be close to the implementation. The mathematical data structures must be replaced by the data structures implementable on a computer and underspecified design decisions must be introduced. Generally, designs of reactive systems describe implementation of functions which realize the observable behaviors appearing in the specifications. In the earlier sections of this chapter, we presents the specifications and the designs of reactive systems described in appropriate specification languages and their formal semantics. To verify the design of reactive systems, environment models are important; they describe possible entities and behaviors in communication with the target system. The environment model of reactive systems are also presented here. We use a simple example, a vending machine, to demonstrate the specifications, the designs, and the environments of reactive systems. After that, we present our framework to verify whether a design conforms to its formal specification. We give an intuitive description then we give a formal model of the framework. We also present the architecture of our generator which is used to automate the main steps of the framework. In the last sections of this chapter, we present some case studies to demonstrate how applying the framework to practical systems and discuss on the generality and the applicability of the framework.

3.1 Preliminary

Transition systems are usually used to describe the behavior of systems. They represent states and transitions between states, which may be labeled with actions obtained from a set. A state describes information about a system at a time. A state transition represents a change from one state to another by an action triggered.

Our work focuses on verifying the correctness of service functions of reactive systems. The properties to be checked generally refer to the conditions under which the service functions are called and the effects of these service functions. Our approach is to check a simulation between the design of the service functions against its specification. Therefore, our framework uses the LTSs to represent behaviors of components in our verification, where actions refer to the service functions and succeeding states represent the effect of these service functions. This section presents basic notions of the LTS. In our definition of the LTS, we may use different names for notions; however, we keep the consistency of meaning compared with the same notions presented in Chapter 2.

Definition 1 *(LTS). An LTS M is a quadruple $\langle\ Q, \Sigma, \delta, I\ \rangle$ where Q is a non-empty set of states, Σ is a set of actions, $\delta \subseteq Q \times \Sigma \times Q$ is a transition relation, we write $(p, a, p') \in \delta$ as $p \xrightarrow{a} p' \in \delta$, and $I \subseteq Q$ is a set of initial states.*

An LTS is deterministic if $|\ I| = 1$ and $(p, a, p1), (p, a, p2) \in \delta$ implies $p1 = p2$. In this case, we say all actions are deterministic. An LTS is finite if Q and Σ are finite sets.

We are going to define a simulation between the specification and the design based on semantics of their LTSs in section 3.5. This definition of the simulation relation is similar to the refinement of [31] and the simulation of [34][43]; however, in our definition, a one-step transition in M_1 may correspond to an n-step transition in M_2. This is appropriate for a simulation from a specification to its design because the behaviors in the design are usually more concrete than those in the specification. We now define n-step transition relation.

Definition 2 *(n-step Transition Relation). Let $M = \langle\ Q, \Sigma, \delta, I\ \rangle$ be an LTS and Σ^+ be the set of non-empty strings of Σ. Then p_n is reachable from p_0 with respect to a string $a_1a_2...a_n$ by δ (denoted $p_0 \xrightarrow{a_1a_2...a_n} p_n \in \delta^+$) if there exist states $p_1, p_2, ..., p_{n-1} \in Q$ such that $p_0 \xrightarrow{a_1} p_1 \in \delta$, $p_1 \xrightarrow{a_2} p_2 \in \delta$, ..., and $p_{n-1} \xrightarrow{a_n} p_n \in \delta$.*

In our framework, LTS associated with the specification in Event-B is obtained by executing this specification. An execution sequence of the specification is an alternating sequence of states and actions. An LTS represents several execution sequences.

Definition 3 *(Finite execution sequence). A finite execution sequence (denoted ρ) represented in an LTS is a finite (non-empty) sequence $\sigma_S^0 \xrightarrow{a_1} \sigma_S^1$, $\sigma_S^1 \xrightarrow{a_2} \sigma_S^2$, ..., $\sigma_S^{n-1} \xrightarrow{a_n} \sigma_S^n$ of transitions.*

We write $\rho_1 \cdot \rho_2$ to denote a concatenation of ρ_1 and ρ_2.

3.2 Specifications of Reactive Systems

A vending machine is a machine which dispenses items such as snacks, beverages, cigarettes, lottery tickets, etc. to customers automatically, after the customer inserts currency or credit into the machine. The specification of vending machines describes their external behaviors including (SF1) switching the machine on, (SF2) switching the machine off, (SF3) inserting credit into the machine, (SF4) returning credit, (SF5) restocking an item, and (SF6) dispensing an item. Each of them is a so-called service function. The essential properties of the vending machine refer to pre-conditions and post-conditions of the service functions. They are shown in Tab.1.

Table 3.1: Properties of Vending Machines

Description of Properties	**Reference**
Pushing a button shall vend a soda of the type corresponding to that button	SF6
The machine shall retain exactly item cost for each item vended	SF3, SF6
The machine shall return all deposited money in excess of item cost	SF3, SF6
The machine shall flash the light for a selected item while vending is in progress to indicate acceptance of a selection to the buyer	SF6
The machine returns the soda after the customer inserts money in excess of soda cost and selects the soda to be bought	SF3, SF6

The properties in the table could be defined in the LTL formulas and checked by Promela/Spin; however, the LTL formulas have a tendency to be complicated. For example, applying the patterns of [18] to define the last property in the table, the LTL formula may be defined in the following form: $<> dispense \rightarrow (!dispense\ U\ (insert\ \&\&\ !dispense\ \&\&\ (!dispense\ U\ select))$. This form of the LTL formula is complicated and prone to mistakes. Our idea is to describe the specification of the vending machine in Event-B and generate verification conditions from the Event-B specification, which represents desirable behaviors at a highly abstracted level.

Event-B is appropriate to describe the specification of the vending machine. Service functions are specified in terms of events with high-level operational definition of state changes by guarded substitutions. An event is made of two elements: (1) a guard that states the necessary conditions for the event to occur, and (2) a substitution that defines the state transition associated with the event. The semantics of the events define the overall results of the executions; therefore, represent pre-conditions and post-conditions of the service functions. Figure 4 demonstrates a specification of the vending machine in Event-B. Variable `stock` defines a set of items that are currently available to be dispensed. It has an abstract data type namely `PRODUCT`. Variable `cred` defines the total of money deposited so far and available to make a purchase. Variable `state` defines the state of the vending machine. Variable `card` defines the size of `stock`. External behaviors are specified in terms of events in Event-B namely `switchon`, `switchoff`, `insert`, `restock`, and `dispense`. Set operations (e.g. union, set minus) are used to describe what the system behaves when an item is restocked or dispensed. A mechanism to add an item into set `stock` and remove the corresponding item from the set has not been described. Also, the specification describes what happens when the customers insert cash or credit;

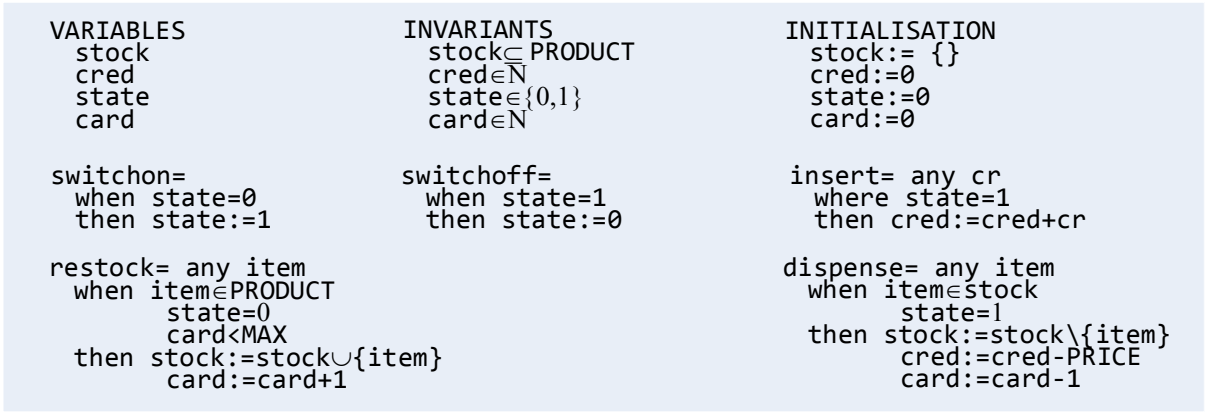


Figure 3.1: Specification

however, how to recognize them and compute the total deposited money is postponed to describing the design.

We present a model of specifications based on Event-B. $\mathcal{V}$ is the set of *variables*. $\mathcal{D}$ is the *domain*, which is the set of values. Exp is the set of expressions in the specifications. An *expression* may contain variables in $\mathcal{V}$, values in $\mathcal{D}$, arithmetic operators, logical operators, and set operators. BExp is the set of boolean expressions (BExp $\subset$ Exp)[1]. A *substitution* $a : \mathcal{V} \to$ Exp is a mapping from $\mathcal{V}$ to Exp. We note that value assignments are also substitutions because $\mathcal{D} \subseteq$ Exp. ACT is the set of substitutions for specifications. A *guard* is a boolean expression. GRD is the set of guards. An *event* is a pair $\langle g, a \rangle$ of a guard g and a substitution a. $\mathcal{E}$ is the set of events. If $e = \langle g, a \rangle$ then we write $grd(e) = g$ and $act(e) = a$. A *state* is a value assignment. $[exp]_\sigma$ denotes the interpretation of the value of an expression *exp* in a state σ. We say a guard g holds in a state σ iff $[g]_\sigma = tt$. Init is the set of special initialization events that have no guard.
We denote $\sigma \xrightarrow{e} \sigma'$ for an event $e = \langle g, a \rangle$ and states σ and σ' if $\sigma(g)$ holds and $\sigma' = \{v \mapsto [a(v)]_\sigma \mid v \in V\}$.

Definition 4 *(Specification models). A* specification model *is a tuple* $S = \langle \mathcal{V}_S, \mathcal{D}_S, \Sigma_S, \text{Init}_S, Inv \rangle$ *where* $\mathcal{V}_S \subseteq \mathcal{V}$ *is the set of variables used in* S, $\mathcal{D}_S \subseteq \mathcal{D}$ *is the domain,* $\Sigma_S \subseteq \mathcal{E}$ *is the set of events,* $\text{Init}_S \in$ Init *is the initialization of* S*, and* $Inv \in$ BExp *is the invariant of* S*. An LTS derived from the specification model* S *is defined as* $M_S = \langle Q_S, \Sigma_S, \delta_S, I_S \rangle$ *where* $Q_S = \{\sigma \mid \sigma : \mathcal{V}_S \to \mathcal{D}_S\}$ *is a non-empty set of states,* $\delta_S = \{\sigma \xrightarrow{e} \sigma' \mid \sigma, \sigma' \in Q_S,\ e \in \Sigma_S\}$ *is a transition relation, and* $I_S = \{act(e) \mid e \in \text{Init}_S\}$ *is a set of initial states.*

In Event-B, a substitution can be deterministic or non-deterministic. We regard a non-deterministic substitution as multiple deterministic substitutions. Therefore, we assume that the LTS is deterministic.

Correspondence between an Event-B description and the definition of specification models is as follows: Each event e in Event-B descriptions has the general form 'e : any u where $g(x, u)$ then $a(x, u)$', where e is name of the event, u is a set of event parameters, x is a set of variables, $g(x, u)$ is a predicate over u and x, and $a(x, u)$ is a substitution. In the execution of a Event-B description, each individual event is instantiated by

[1]Definition and evaluation of expressions and boolean expressions are presented in the appendix

a set of possible values of event parameters. If an individual event e is enabled when its parameter obtains any value in the set $\{w_1, w_2, \cdots, w_n\}$ then we regard this case as n events $e(w_1), e(w_2), \cdots, e(w_n)$ are enabled. Therefore, the transitions are labeled explicitly with the form $e(w)$ in the specification model associated to the Event-B description. For example, the event *restock* in Figure 3.1 follows the general form, where:

- e: *restock*,
- u: $\{item\}$,
- x: $\{stock, state, card\}$,
- $g(x, u)$: $item \in PRODUCT \wedge state = 0 \wedge card < MAX$, and
- $a(x, u)$: $\{card = card + 1, stock = stock \cup \{item\}\}$.

Suppose $PRODUCT = \{a, b, c\}$. Instances of the event *restock* may be $restock(a)$, $restock(b)$ and $restock(c)$ where:

- $restock(a) = \langle a \in PRODUCT \wedge state = 0 \wedge card < MAX, \{card \mapsto card + 1, stock \mapsto stock \cup \{a\}\}\rangle$,
- $restock(b) = \langle b \in PRODUCT \wedge state = 0 \wedge card < MAX, \{card \mapsto card + 1, stock \mapsto stock \cup \{b\}\}\rangle$,
- $restock(c) = \langle c \in PRODUCT \wedge state = 0 \wedge card < MAX, \{card \mapsto card + 1, stock \mapsto stock \cup \{c\}\}\rangle$.

We note that there are infinitely many variables and events in general. For example, if *tasks* is the set of natural numbers in the example above, the set of events contains the infinite set $\{activateTask(t) \mid t \in \mathbb{N}\}$.

Definition 5 *(The set of reachable states). The set of reachable states associated to S (denoted $Q_S^{\rightarrow}$) is the smallest set satisfying: $I_S \subseteq Q_S^{\rightarrow}$ and if $\sigma \in Q_S^{\rightarrow}$ and $\sigma \xrightarrow{e} \sigma'$ for some $e = (g, a) \in E_S$ then $\sigma' \in S_S^{\rightarrow}$.*

An Event-B model is consistent if for all reachable state σ, $[Inv_S]_\sigma = tt$.

3.3 Designs of Reactive System

Figure 3.2 shows an architecture design of the vending machine. The system consists of two sensors Coin_in sensor and Button_item sensor, a controller, and four actuators including Coin_out, Button_Light, Vend and VendMotor. The internal behaviors of the vending machine are as follows. When a coin is inserted, the Coin_in sensor detects the coin to be inserted and then sends an appropriate electrical signal to the Controller. The Controller computes the total of deposited money based on the inserted coin evaluation. When an item is selected, the Button_item sensor detects the item to be selected and then sends a corresponding signal to the Controller. The Controller commands the Button_Light to flash. The Controller compares the item cost with the total of the deposited money. If the item cost is less than the total, the Controller commands the VendMotor

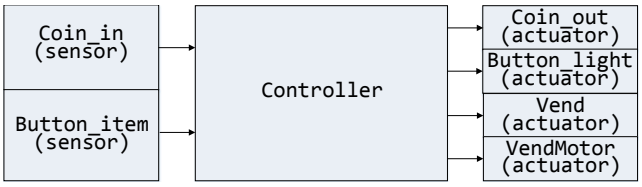


Figure 3.2: Architecture Design

and Vend to remove the corresponding item from the set of available items and dispense it. The Controller commands the Button_Light to stop flashing and commands the Coin_out to return the correct change.

Designs of the vending machine can be straightforwardly described in Promela. The abstract data structures are replaced by the implementable data structures, e.g. array, record type. Service functions of reactive systems can be described by using inline functions. In the description of the functions, the behaviors are explicitly defined using statements of Promela, e.g. expressions, assignment statements, and various control structures. The execution of the statement may change the value of variables. Additional variables and constants may be introduced to explicitly describe statements that must be performed to detect cash and credit for computing the total deposited money. Figure 3.3 demonstrates a detailed design of the vending machine. In the example, variables having abstract types, e.g. `stock` $\subseteq$ `PRODUCT`, are replaced by variables having concrete types, e.g. `ITEM stock[1000]`. New constants are introduced, e.g. `CENT` is used in the case that a one cent coin is inserted. (Design decisions for how to add a new item into the order set and to remove one from the corresponding position are explicitly described based on the implementable data structures and the control structures, e.g. loop and selection structures).

We now define model of designs in Promela. The design model of reactive systems includes implementable data structures and a collection of inline functions. Syntactically, the function signature contains a function name and some parameters (function arguments). The functions are called from the environment. When a function is invoked, its parameters are instantiated by values specified from the environment. The body of the function consists of substitutions. We use $\mathcal{P}$ to denote a set of *parameters* (function arguments). In the design, an expression may contain constants, variables, parameters and arithmetic operators, therefore, a so-called *parameterized expression*. The set of parameterized expressions is denoted as PExp[2]. A function body is defined as a substitution. The substitution may contain the parameterized expressions. We use *p-substitution* to denote the substitution in the design. *p-substitution* is a mapping from $\mathcal{V}$ to PExp. The set of p-substitutions is denoted as PSubst. Id is the set of *identifiers* (used as function names). For the simplicity, we assume that functions have only one parameter. The design also includes an initialization function which assigns the initial values for the variables. Design models are defined as follows.

Definition 6 *(Design model). A* design model *is a tuple* $D = \langle \mathcal{V}_D, \mathcal{D}_D, \mathcal{P}_D, F, \Sigma_D, I_D \rangle$ *where* $\mathcal{V}_D \subseteq \mathcal{V}$ *is the set of variables used in* D, $\mathcal{D}_D \subseteq \mathcal{D}$ *is the domain of* D, $\mathcal{P}_D \subseteq \mathcal{P}$ *is a finite set of parameters for* D, F *is a set of function signatures defined as* $F = \{id(p) \mid id \in \mathrm{Id}, p \in \mathcal{P}_D\}$, Σ_D *is a relation such that* $\Sigma_D \subseteq F \times \mathrm{PSubst}$, *and* I_D *is a set*

[2]Definition of parameterized expressions is presented in the appendix

```
#define CENT 1;
#define x 10; /* number of vend slots */
#define y 20; /* number of availabe items in each slot */
#define MAX x*y;
typedef ITEM {byte id, pr, ...}; ITEM stock[1000];

inline insert(coin){
/* detecting coins to be inserted */
s= detect(coin);
/* computing the total money deposited so far */
if :: s=='c' -> credit=credit+CENT;
   :: s=='n' -> credit=credit+NICKEL;
   :: s=='q' -> credit=credit+QUARTER;
fi;
}

inline dispense(b){
/* detecting button to be pressed */
s= detect(b);
/* dispending the corresponding item */
remove(s);
credit=credit-stock[s].pr;
}

inline remove(s){
i=s*y + 1; /* the 1st item in slot s */
j= i; /* remove the 1st item in slot s */
/* repeating until j reaches (s+1)*x)
   { stock[j] = stock[j+1]; j++; }
stock[j] = 0;
}
```

Figure 3.3: Design

of value assignments of the initialization function such that $I_D \subseteq \{\sigma \mid \sigma : \mathcal{V}_D \to \mathcal{D}_D\}$.

We assume that the functions in the design are deterministic to have a unique successor state for each current state and each called function. This assumption is realistic for the implementation of the reactive systems like the automotive operating systems.
We can see a gap between the specifications and the designs. The observable behaviors appearing in the specifications are realized by the optimized behaviors appearing in the designs. The specifications can be described in a declarative manner whereas the design can be described in an imperative manner. Our objective is to verify the conformance between such specifications and designs by using a simulation relation between them.

3.4 Communication of Reactive System and Its Environment

Figure 3.4 illustrates the overall structure of the design (left) and the environment (right) of the reactive systems. We call them a *design model* and an *environment model*, respec-

tively. The design model defines data structures and a collection of inline functions; it cannot operate by itself. To operate it, we need an environment which calls the functions of the target system. Essentially, the reactive systems need to be verified in the combination with their environments. The environment model defines entities such as items, coins and a sequence of function calls to the target system. Generally, the environment invokes functions in the design in a non-deterministic manner in each state. This is shown in environment models.

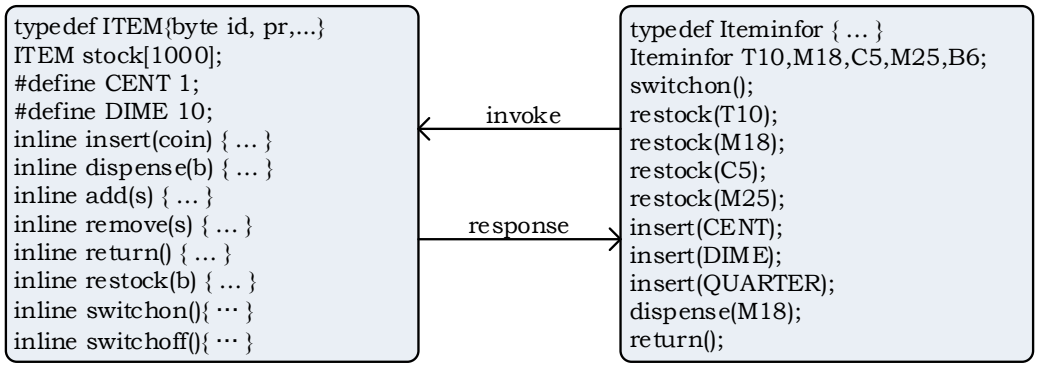


Figure 3.4: Design model and Environment model in Promela

Environment models are defined as follows.

Definition 7 *(Environment model). An* environment model for a design model D *is a tuple* $E = \langle \mathcal{V}_E, \mathcal{D}_E, \Sigma_E, I_E \rangle$ *where* $\mathcal{V}_E \subseteq \mathcal{V}$ *is a set of variables used in* E, $\mathcal{D}_E = \mathcal{D}_D$ *is the domain,* Σ_E *is a set of* invocations to D *such that* $\Sigma_E \subseteq \{id(v) \mid id \in \mathrm{Id}, v \in \mathcal{V}_E\}$, *and* I_E *is a set of value assignments from* $\mathcal{V}_E$ *to* $\mathcal{D}_D$.

As explained later, the environment is constructed from the specification, and combined with the design for the verification of the design against the specification.

By combining the design and the environment, we can make a closed system which can operate by itself. We call this a *combination model*. In terms of Promela, a combination model can be obtained by including the Promela code of the design into that of the environment model. A combination of a design and an environment describes the execution of the design according to the environment. An expression in the combination contains constants from $\mathcal{D}$, variables in $\mathcal{V}$, and arithmetic operators. The set of expressions in combinations is denoted as Exp'. A substitution for combinations is a mapping from $\mathcal{V}$ to Exp'. The set of substitutions for combinations is denoted as SubstDE. For a mapping π from $\mathcal{P}$ to $\mathcal{V}$ and a parameterized expression $pexp \in \mathrm{PExp}$, $pexp_\pi$ is the result of replacing each parameter p appearing in $pexp$ by $\pi(p)$. In other words, if $a(v)$ is an expression in D then $a(v)_\pi$ is an expression in the combination obtained by replacing each parameter p appearing in $a(v)$ by $\pi(p)$. Combination models are defined as LTSs as follows.

Definition 8 *(Combination model). Let* $D = \langle \mathcal{V}_D, \mathcal{D}_D, \mathcal{P}_D, F, \Sigma_D, I_D \rangle$ *be a design model and* $E = \langle \mathcal{V}_E, \mathcal{D}_E, \Sigma_E, I_E \rangle$ *an environment model.*

1. *We denote* $\sigma \xrightarrow{id(v)} \sigma'$ *for an invocation* $id(v) \in \Sigma_E$ *and states* σ *and* σ' *if there exist* $(id(p), a) \in \Sigma_D$ *and a mapping* $\pi : \mathcal{P}_D \to \mathcal{V}_E$ *such that* $\pi(p) = v$ *and* $\sigma' = \{v \mapsto [a(v)_\pi]_\sigma \mid v \in \mathcal{V}_D \cup \mathcal{V}_E\}$.

2. *The* combination model *of D and E (denoted as $D{\cdot}E$) is an LTS* $\langle Q_{D\cdot E}, \Sigma_{D\cdot E}, \delta_{D\cdot E}, I_{D\cdot E}\rangle$ *where* $Q_{D\cdot E} = \{\sigma |\ \sigma : \mathcal{V}_D \cup \mathcal{V}_E \to \mathcal{D}_D\}$ *is a set of states,* $\Sigma_{D\cdot E} = \Sigma_E$, $\delta_{D\cdot E} = \{\sigma \xrightarrow{id(v)} \sigma' |\ \sigma, \sigma' \in Q_{D\cdot E}, id(v) \in \Sigma_E\}$ *is a transition relation, and* $I_{D\cdot E} = I_D \cup I_E$ *is a set of initial states of D and E.*

3.5 Verification Framework

3.5.1 Overview of Framework

Our framework is to check the conformance of the design in Promela to its formal specification in Event-B. The verification technique is used in the framework is model checking. We check the conformance of two models based on the simulation relation between them. In particular, we check whether the design simulates the specification. As demonstrated in Figure 3.1, the specification defines state variables, invariants and events which trigger state transitions. Formally, the execution of the specification is represented as an LTS. Also, Figure 3.3 describes variables and functions appearing in the design in Promela. The variables represent information about the system (states) at certain moments. The execution of statements changes the values of variables. Therefore, the design can be interpreted as an LTS if we consider that the variables are states and each function call is a label to make transitions on the states. In previous section, we defined the semantics of the specification and the design commonly as LTSs. We now present a simulation relation between the specification and the design based on their LTS. Suppose that M1 and M2 are two LTSs. Informally, M2 simulates M1 if for each transition in M1 from state p to state p' and p relates to state q of M2, there exists state q' and a corresponding transition in M2 from q to q' such that p' relates to q'. In Figure 3.5, a line arrow connecting p to p' represents a one-step transition from p to p', and a dashed arrow connecting q to q' represents an n-step transition from q to q'.

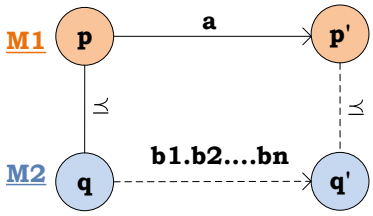


Figure 3.5: Simulation Relation

Figure 3.6 shows our framework to verify the simulation between a specification and a design using the Spin model checker.

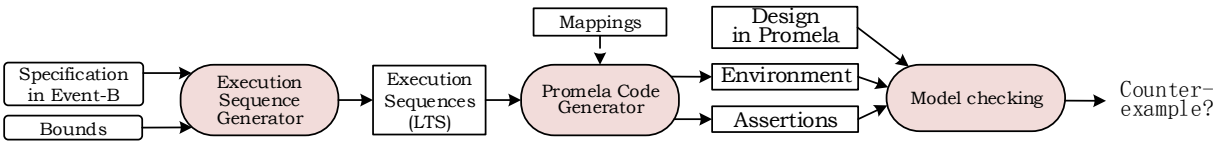


Figure 3.6: Checking simulation of design and its specification

Inputs of the framework include the Event-B specification, the bounds, the Promela design, and the mappings. The Event-B specification may contain abstract data types and

infinite types. For example, the specification of the vending machine described $cred \in N$ and $stock \subseteq PRODUCT$; N is infinite range of natural numbers and $PRODUCT$ is an abstract type whose range is not definite. The specification may also describe non-deterministic behaviors. The bounds are introduced to make sure that every data element, e.g., variable, constant, or event parameter, obtain values in the finite domain. We assume that the Promela design describes deterministic behaviors even though the specification may contain non-deterministic behaviors. The mappings are introduced to relate the specification to the design. We consider three kinds of mappings. They are mappings from variables in the specification to variables in the design, mappings from values in the specification to values in the design, and mappings from events in the specification to functions in the design. Generally, the mappings may be either one-to-one or many-to-one.

Intermediate outputs of the framework includes LTSs of the specification, environments and assertions. The LTSs are finite since they are generated from specification within the bounds. The LTSs are deterministic because we regard one non-deterministic substitution in Event-B as multiple deterministic substitutions. As mentioned, the design does not execute by itself; we need environments to trigger functions in the design. We assume that the environments invoke functions in the design in a non-deterministic manner in each state. Assertions define conditions for the simulation between the specification and the design. The final output of the framework is the statistic of model checking which shows whether the design simulates the specification within the bounds.

The framework consists of three steps. Firstly, bounds for the verification are given and an LTS is generated from the Event-B specification within the bounds. Next, the LTS is in turn used to generate the environment, which invokes service functions described in the design. The verification then amounts to checking the validity of certain relations between variables of the Promela design and variables of the Event-B specification in every reachable state. This is done using Spin assertions which are generated from states of the LTS and the given relations represented as mappings between date elements in the specification to those in the design. Finally, the design is combined with the environment; and, the combination model is analyzed by Spin against the assertions.

3.5.2 Bounds

Model checking does an exhaustive check of the system. It needs a representation of the system as a finite LTS. We know that there may exist infinitely many elements of basic types in Event-B. For example, the specification of the vending machine described $cred \in N$ and $stock \subseteq PRODUCT$; N is infinite range of natural numbers and $PRODUCT$ is an abstract type whose range is not definite. When the variables may obtain values in an infinite domain, the state space of the system may be an infinite set. Similarly, when the event parameters may obtain values in an infinite domain, there may be infinitely many events applicable in each state. This results an infinite LTS. Therefore, in order to apply the model checking technique, we restrict the state space and the set of applicable events in each state. We define such restrictions as bounds of the verification. The effect of such bounding process is that the state space and the set of applicable events in each state becomes finite. We formally define the bounds and the effect of such bounds.

Definition 9 *(Bounds).* Bounds for LTS $\langle Q, \Sigma, \delta, I \rangle$ *are defined as a pair* $B = \langle G, H \rangle$ *of mappings* G *and* H *where* $G : 2^Q \to 2^Q$, $G(Q) \subseteq Q$, *and* $Q' \subseteq Q''$ *implies* $G(Q') \subseteq G(Q'')$

and $H : Q \times \Sigma \to \{tt, ff\}$ *and for any state* $p \in Q$*, there exist finitely many actions* $a \in \Sigma$ *such that* $H(p, a) = tt$*.*

As we mentioned, the bounds are introduced to obtain a finite LTS from the Event-B specification. A finite LTS is obtained from an infinite LTS when we restrict the state space and the set of actions that trigger the state transitions within the bounds. The bounded LTS is defined as follows:

Definition 10 *(Bounded LTS). An LTS obtained by restricting an LTS* $M = \langle Q, \Sigma, \delta, I\rangle$ *within bounds* $B = \langle G, H\rangle$ *is defined as* $M{\downarrow_B} = \langle \widehat{Q}, \widehat{\Sigma}, \widehat{\delta}, \widehat{I}\rangle$*, where* $\widehat{Q} = G(Q)$*,* $\widehat{\Sigma} = \{a \mid \forall p \in Q, a \in \Sigma, H(p, a) = tt\}$*,* $\widehat{\delta} = \{p \xrightarrow{a} p' \in \delta \mid H(p, a) = tt\}$*, and* $\widehat{I} = G(I)$*.*

We now implement the bounds for the Event-B specification. Firstly, abstract types in Event-B must be replaced by concrete types, e.g., $PRODUCT$ is replaced by a definite set $\{a, b, c, d, e\}$. Then, types having infinite ranges of values like Int and Nat must be restricted as small ranges by giving the minimum value and maximum value of the ranges, e.g., $cred \in [MIN..MAX]$. These bounding process is demonstrated in Figure 3.7. The top part is the original Event-B specification, where exists abstract data types and infinite types. The bottom part is a bounded specification, where every variable/constant/event parameter obtain values in a finite domain. This bounding process reduces the size of execution sequences explored from the Event-B specification and produces a finite LTS associated to the restricted specification.

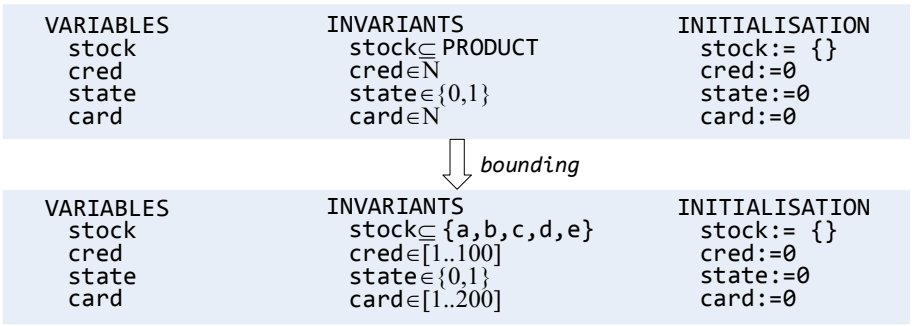


Figure 3.7: Bounding process is applied to Event-B model

We introduce mappings X_V and X_P to represent the implementation of the bounds for the Event-B specification: X_V is a mapping from the variables to a finite set of values; and X_P is a mapping from the event parameters to a finite set of values. When mappings X_V and X_P are applied to the specification, both the state space and the set of applicable events in each state become finite sets. We use $ES_X(\sigma)$ to denote the set of events which are applicable to state σ and satisfy the restriction defined by mappings X_V and X_P. It is obvious that $ES_X(\sigma)$ is a finite set for each σ. (Algorithm 1) presents how to compute $ES_X(\sigma)$ from the specification with mappings X_V and X_P.

Suppose $S = \langle \mathcal{V}_S, \mathcal{D}_S, \Sigma_S, \mathrm{Init}_S, Inv\rangle$ be a specification model and $\langle Q_S, \Sigma_S, \delta_S, I_S\rangle$ an LTS derived from S. With the mappings X_V and X_P, we define mappings G and H as follows: $G(Q_S) = \{\sigma \in Q_S \mid \forall v \in \mathcal{V}_S.\sigma(v) \in X_V(v))\}$, $G(I_S) \subset G(Q_S)$, and $H(\sigma, e) = tt$ iff $e \in \mathrm{ES}_X(\sigma)$.

Algorithm 1 Compute $ES_X(\sigma)$

1: $ES = EMPTY$,
2: **for** each $e \in \Sigma_S$ **do**
3: **if** $\forall u \in PARAMETER$, $\sigma(u) \in X_P(u)$ and $[g(u)]_\sigma = tt$ **then**
4: **if** $\forall v \in V_S, [(act(e))(v)]_\sigma \in X_V(v)$ **then**
5: $ES = ES \cup \{e\}$
6: **end if**
7: **end if**
8: **end for**
9: $ES_X(\sigma) = ES$
10: return $ES_X(\sigma)$

3.5.3 Exploring Execution Sequences

In order to explore the execution sequences, or LTS, from the specification and bounds, the LTS Explorer computes all possible transitions and reachable states. Every value used in the computation must be within the bounds. Starting at the initialization, the explorer enumerates all possible values for the constants and variables of the specification that satisfy the initialization to compute the set of initial states. To compute all possible transitions from a state, the explorer finds all possible values for event parameters of an individual event to evaluate the guard of that event. Although the guard may be logical predicates consisting of existential quantification or universal quantification, it is feasible to enumerate all possible values for event parameter because the range of values for every variable and event parameter has been bounded since the previous step. If the guard holds in the given state, the explorer computes the effect of the event based on substitution of that event. Also, the substitution may be non-deterministic; we can regard an event with non-deterministic substitution as a finite number of events with deterministic substitution. When new states are generated, we repeat this process to these states until no new states are generated.

Figure 3.8 (left and right) illustrates a bounded specification of the vending machine and an LTS generated from the specification. In the right part, the rectangles represent the states and the labeled arrows represent the events that are enabled in each states. For example, two events `restock(e)`, `switchon()` are enabled in state $s0$. In our framework, the states are defined as the value assignments; however, we show them here as values for readability. For example, `s2:(on, {a,b,c,d}, 0)` describes that the machine is on; there are 4 items `a`, `b`, `c`, and `d` available for buying; and the currently deposited money is 0, in state labeled `s2`.

The algorithm to compute execution sequences from a specification model is presented in (Algorithm 2). Inputs of the algorithm are a specification model $S = \langle \mathcal{V}_S, \mathcal{D}_S, \Sigma_S, \text{Init}_S, Inv \rangle$, and bound $B = \langle G, H \rangle$ which is implemented by X_V and X_P. Output is a finite LTS. The algorithm uses two data structures: $QUEUE$ storing reachable states, and $VISITED$ storing visited states. It uses two routines to access $QUEUE$: Push($QUEUE$,$\langle\sigma\rangle$) adds state σ as an element into $QUEUE$, Pop($QUEUE$) returns the head of $QUEUE$. In each step of **while** loop, one state is removed from $QUEUE$, and reachable states from the state are computed.

Termination. We use N to denote size of the state space within the given bounds. The

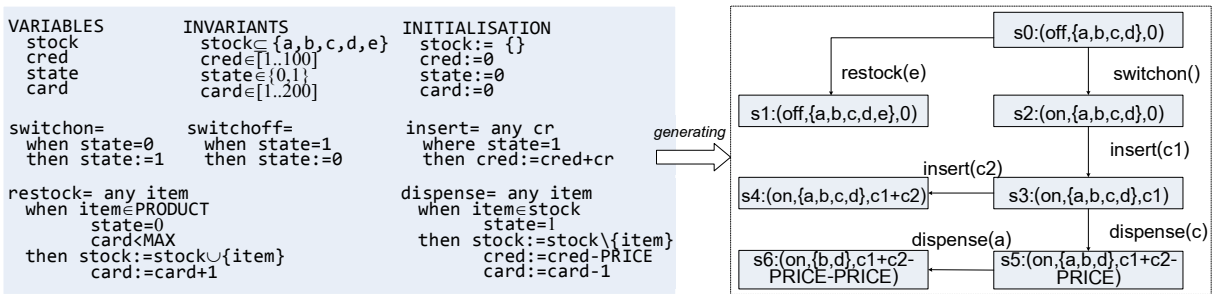


Figure 3.8: Generating LTS from bounded Specification

Algorithm 2 Generating $S{\downarrow_B} = \langle \widehat{Q}, \widehat{\Sigma}, \widehat{\delta}, \widehat{I} \rangle$ from $S = \langle \mathcal{V}_S, \mathcal{D}_S, \Sigma_S, \mathrm{Init}_S, Inv \rangle$ and X_V, X_P

1: $QUEUE = empty$
2: $VISITED = empty$
3: $\widehat{Q} = \widehat{\Sigma} = \widehat{\delta} = \widehat{I} = empty$
4: **for** each $\sigma_0 \in \{act(e) \mid e \in \mathrm{Init}_S\}$ **do**
5: **if** $\forall v \in V_S, \sigma_0(v) \in X_V(v)$ **then**
6: Push$(QUEUE, \langle \sigma_0 \rangle)$
7: $\widehat{Q} = \widehat{Q} \cup \{\sigma_0\}$
8: $\widehat{I} = \widehat{I} \cup \{\sigma_0\}$
9: **end if**
10: **end for**
11: **while** $QUEUE \neq empty$ **do**
12: $\langle \sigma \rangle = \mathrm{Pop}(QUEUE)$
13: $VISITED = VISITED \cup \{\sigma\}$
14: $\widehat{E} = \{e \mid e \in \mathrm{ES}_X(\sigma)\}$
15: **if** $\widehat{E} \neq empty$ **then**
16: **for** each event $e = (g, a) \in \widehat{E}$ **do**
17: $\sigma' = \{v \mapsto [(act(e))(v)]_\sigma | v \in \mathcal{V}_S\}$
18: **if** $\sigma' \notin VISITED$ **then**
19: Push$(QUEUE, \langle \sigma' \rangle)$
20: $\widehat{Q} = \widehat{Q} \cup \{\sigma'\}$
21: **end if**
22: $\widehat{\Sigma} = \widehat{\Sigma} \cup \{e\}$
23: $\widehat{\delta} = \widehat{\delta} \cup \{\sigma \xrightarrow{e} \sigma'\}$
24: **end for**
25: **end if**
26: **end while**
27: return $S{\downarrow_B}$

algorithm terminates when $N - |VISITED| = 0$. Initially, $N - |VISITED| > 0$ because $VISITED$ is empty. Lets consider $N - |VISITED|$ at each step of the **while** loop. Since range of values for every variable has been limited, the state space of is a finite set. N is a definite value. In each step of **while** loop, one state is added into $VISITED$. Consequently, $|VISITED|$ increases; $N - |VISITED|$ decreases. Thus, the algorithm

must then terminate. When the algorithm terminates, $\widehat{Q}$ stores reachable states and $\widehat{\delta}$ contains possible state transitions within the bounds.

Correctness. We use $S1{\downarrow_B} = \langle \widehat{Q_1}, \widehat{\Sigma_1}, \widehat{\delta_1}, \widehat{I_1} \rangle$ to denote the bounded LTS, that is defined in Definition 10. We use $S2{\downarrow_B} = \langle \widehat{Q_2}, \widehat{\Sigma_2}, \widehat{\delta_2}, \widehat{I_2} \rangle$ to denote the bounded LTS, that is computed by Algorithm 2. We remind here that $Q_S^{\rightarrow}$ denotes the set of reachable states associated to S. We now consider whether $S1 \downarrow_B$ equals to $S2 \downarrow_B$:

- $\widehat{I_1} = \widehat{I_2}$,
- For all $\sigma \in \widehat{Q_1^{\rightarrow}} \cap \widehat{Q_2^{\rightarrow}}$ such that $H(\sigma, a) = tt$ and $\sigma' = \{v \mapsto [a(v)]_\sigma | v \in \mathcal{V}_S\}$, we have $\sigma \xrightarrow{a} \sigma' \in \widehat{\delta_1} \cap \widehat{\delta_2}$. Thus, $\widehat{Q_1^{\rightarrow}} = \widehat{Q_2^{\rightarrow}}$,
- $\widehat{\delta_1} \cap (\widehat{Q_1^{\rightarrow}} \times \widehat{\Sigma_1} \times \widehat{Q_1}) = \widehat{\delta_2} \cap (\widehat{Q_2^{\rightarrow}} \times \widehat{\Sigma_2} \times \widehat{Q_2})$
- $\widehat{\Sigma_1} = \widehat{\Sigma_2}$

From the consideration above, we can show that, even though $\widehat{\delta_1}$ may not equal to $\widehat{\delta_2}$, $\widehat{Q_1^{\rightarrow}} = \widehat{Q_2^{\rightarrow}}$, $\widehat{\Sigma_1} = \widehat{\Sigma_2}$, $\widehat{\delta_1} \cap (\widehat{Q_1^{\rightarrow}} \times \widehat{\Sigma_1} \times \widehat{Q_1}) = \widehat{\delta_2} \cap (\widehat{Q_2^{\rightarrow}} \times \widehat{\Sigma_2} \times \widehat{Q_2})$, and $\widehat{I_1} = \widehat{I_2}$. Therefore, our algorithm is sufficient to compute the set of reachable states of the specification within the given bounds.

3.5.4 Generating Environments

The environments trigger specific behaviors of the target system; therefore, it is essential to construct such comprehensive environments that representing all possible behaviors in the specification. In the previous step, we explored the execution sequences as an LTS of the specification. In this step, we generate the environment by translating the LTS into Promela such that the enabled events in LTS are translated to the corresponding function calls in Promela.

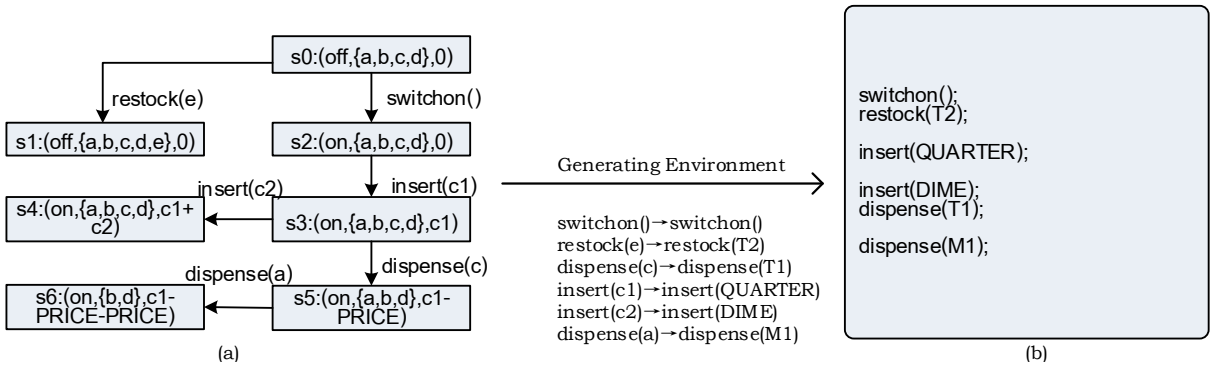


Figure 3.9: Generation of environment from LTS

Figure 4.14(a) illustrates an LTS generated to the specification within the bounds. The LTS is translated into Promela to generate the environment, from (a) to (b) of Figure 4.14. For this generation, we give a mapping from the enabled events in the LTS to the function calls in the environment. We present the mappings in Table 3.2.

In general, it may be a one-to-one mapping or a many-to-one mapping. In the sample case of the figure, it is a one-to-one mapping. For example, event `restock(e)` in the LTS is mapped to function call `restock(T2)` in the environment. The environment is then

Table 3.2: Mappings f from enabled events to function calls

Enabled Events	**Function calls**
restock(a)	restock(M1)
restock(b)	restock(M2)
restock(c)	restock(T1)
restock(d)	restock(B1)
restock(e)	restock(T2)
insert(c1)	insert(QUARTER)
insert(c2)	insert(DIME)
dispense(c)	dispense(T1)
dispense(a)	dispense(M1)
...	...

combined with the design to make a combination model, which will be input into the model checker in the last step of the framework.

We formally define how the environment is generated from the LTS of the specification model. Let $S = \langle \mathcal{V}_S, \mathcal{D}_S, \Sigma_S, \text{Init}_S, Inv \rangle$ be a specification model and $M_S = \langle Q_S, \Sigma_S, \delta_S, I_S \rangle$ be the LTS derived from S. Based on the given mapping $f : \Sigma_S \to \Sigma^+_{D \cdot E}$ from the events in the LTS to the function calls in the environment, mapping $R' : \mathcal{V}_S \to \mathcal{V}_E$ and mapping $C : \mathcal{D}_S \to \mathcal{D}_D$, the environment model $E = \langle \mathcal{V}_E, \mathcal{D}_E, \Sigma_E, I_E \rangle$ with $\mathcal{D}_E = \mathcal{D}_D$ is generated such that $\Sigma_E = \bigcup_{e \in \Sigma_S} \{f(e)\}$ and $I_E = \bigcup_{e \in I_S} \{f(e)\}$.

3.5.5 Simulation Relation between Specification and Design

In Section 3.5.1, we briefly introduced a simulation between two LTSs. In this section, we formally define the simulation based on a relation between states of the specification and the design. The states are value assignments which are mappings from the variables to the values. Therefore, the relation on states of M1 and those of M2 are established based on mappings R and C where R is the mapping from variables of M1 to those in M2, C is the mapping from values in M1 to those in M2. Figure 3.10 (left) shows a relation between state p of M1 and state q of M2. p relates to q based on R and C because $u = c1$ in state p corresponds to $v = QUARTER$ in state q with mappings $R(u) = v$ and $C(c1) = QUARTER$. The relation between states is formally defined as follows:

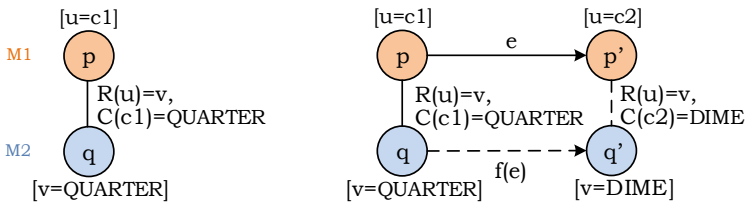


Figure 3.10: Simulation Relation

Definition 11 *(Relation between states). Let* $S = \langle \mathcal{V}_S, \mathcal{D}_S, \Sigma_S, \text{Init}_S, Inv \rangle$ *be a specification model,* $M_S = \langle Q_S, \Sigma_S, \delta_S, I_S \rangle$ *the LTS derived from* S*,* $D = \langle \mathcal{V}_D, \mathcal{D}_D, \mathcal{P}_D, F, \Sigma_D, I_D \rangle$ *a design model,* $E = \langle \mathcal{V}_E, \mathcal{D}_E, \Sigma_E, I_E \rangle$ *an environment model, and* $D \cdot E = \langle Q_{D \cdot E}, \Sigma_{D \cdot E}, \delta_{D \cdot E}, I_{D \cdot E} \rangle$

the combination model of D and E. We say a state $\sigma_{D\cdot E} \in Q_{D\cdot E}$ relates to a state $\sigma_S \in Q_S$ based on mappings $R : \mathcal{V}_S \to \mathcal{V}_D$ and $C : \mathcal{D}_S \to \mathcal{D}_D$ (denoted $\sigma_S \preceq_{R,C} \sigma_{D\cdot E}$), if for any $x \in \mathcal{V}_S$ and $y \in \mathcal{V}_D$, $R(x) = y$ implies $C(\sigma_S(x)) = \sigma_{D\cdot E}(y)$.

We omit R, C from $\preceq_{R,C}$ if they are clear from the context.

We now present a formal definition of the simulation relation between two LTSs by extending the given relation on the states. In general, a one-step transition in the specification is followed by an n-step transition in the design. In Figure 3.10 (right), a line arrow connecting p to p' represents a one-step transition from p to p', and a dashed arrow connecting q to q' represents an n-step transition from q to q'. In the definition, Σ^+ denotes the set of non-empty strings of Σ, δ^+ denotes an n-step transition relation, and $p \xrightarrow{a_1 a_2 \ldots a_n} p' \in \delta^+$ denotes an n-step transition from state p to state p'.

Definition 12 *(Simulation relation). Let $M_1 = \langle Q_1, \Sigma_1, \delta_1, I_1 \rangle$ and $M_2 = \langle Q_2, \Sigma_2, \delta_2, I_2 \rangle$ be LTSs, and $f : \Sigma_1 \to \Sigma_2^+$ a function from Σ_1 to Σ_2^+. Suppose a relation $\preceq \subseteq Q_1 \times Q_2$ is given. $M2$* simulates *$M1$ with respect to $\preceq$ if for all $q_1, q_1' \in Q_1$, $q_2 \in Q_2$, $a \in \Sigma_1$ such that $q_1 \preceq q_2$ and $q_1 \xrightarrow{a} q_1' \in \delta_1$, there exist $q_2' \in Q_2$ such that $q_1' \preceq q_2'$ and $q_2 \xrightarrow{f(a)} q_2' \in \delta_2^+$. If $M2$ simulates $M1$ with respect to $\preceq$, we denote $M1 \preceq M2$.*

M2 simulates M1 if for each transition in M1 from state p to state p' and p relates to state q of M2, there exists state q' and a corresponding transition in M2 from q to q' such that p' relates to q'. In the example of Figure 3.10 (right), to check whether M2 simulates M1, we check if there exists a reachable state q' from q such that $v = DIME$ corresponding to $u = c2$ in p' with mappings $R(u) = v$ and $C(c2) = DIME$.

3.5.6 Generating Assertions

Verification conditions represent constraints on the simulation relation between the specification and the design. They are also generated from the LTS and encoded as assertions in Promela/Spin. From states of the LTS, assertions, which must be met by the corresponding states of the designs, are generated. This is based on the mapping R and C from the variables, the values in the specification to those in the design. For example, Figure 3.11(a) illustrates an LTS generated to the specification within the bounds. As shown in the figure, when event `dispense(c)` fires in state $s3$, the system reaches successor $s5$. From value assignments in state $s5$ of the LTS, with mappings $R(cred) = credit$, $R(state) = state$, $C(on) = 1$, $C(a) = M1$, $C(b) = M2$, $C(c) = T1$, and $C(d) = B1$, the generator outputs an assertion ($state = 1$ && $credit = 15$ && $card = 3$). This assertion is to validate the value assignments in the state of the design that is reachable from state $s3$ by calling the function `dispense(T1)`.

Formally, the relation on states between the LTS of the specification model and the combination model is given based on the mappings $R : \mathcal{V}_S \to \mathcal{V}_D$ and $C : \mathcal{D}_S \to \mathcal{D}_D$. Verification conditions are generated as follows:

- For initialization of the combination, the assertion is:
  $$\bigwedge_{x \in \mathcal{V}_S, y \in \mathcal{V}_D, y = R(x)} (\sigma^0_{D\cdot E}(y) = C(\sigma^0_S(x))),$$
- For all (reachable) states $\sigma_S, \sigma_S' \in Q_S$ and $\sigma_{D\cdot E}, \sigma_{D\cdot E}' \in Q_{D\cdot E}$ such that $\sigma_S \xrightarrow{e} \sigma_S' \in \delta_{S\downarrow_B}$, $\sigma_{D\cdot E} \xrightarrow{f(e)} \sigma_{D\cdot E}' \in \delta^+_{D\cdot E}$, and $\sigma_S \preceq_{R,C} \sigma_{D\cdot E}$,

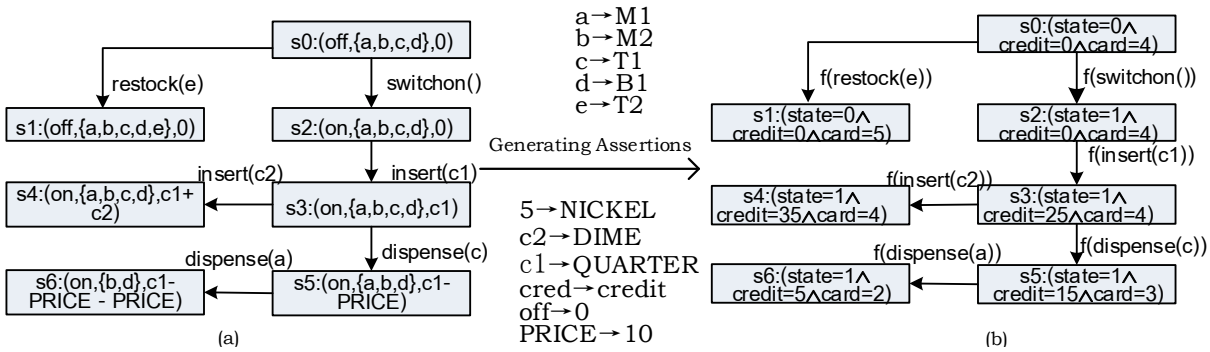


Figure 3.11: Generation of assertions from LTS

the assertion is:

$$\bigwedge_{x\in\mathcal{V}_S, y\in\mathcal{V}_D, y=R(x)} (\sigma'_{D.E}(y) = C(\sigma'_S(x))).$$

3.5.7 Translating Environments and Assertions into Promela

In this framework, the design is combined with the environment. The combination model and the assertion are input into Promela/Spin to check the simulation relation of the specification and the design. In terms of Promela, a combination model can be obtained by including the Promela code of the design into that of the environment model. Figure 3.12 demonstrates the design, the environment, the assertion and the combination model with the assertions. The states and transitions in the LTS are represented by labels and if-statements in the combination model. There may be more than one function call applicable in each state. For example, `insert(DIME)` and `dispense(T1)` are applicable in state $s3$; which function call actually applied is non-deterministic. The assertion is inserted at appropriate positions of the combination model. For example, position of the assertion, which is to validate the value assignments in the state reachable from state $s3$ by calling the function `dispense(T1)`, is underlined in the figure.

3.5.8 Checking of Simulation Relation.

In the last step, we verify the combination model against the assertions by using the Spin model checker to confirm the simulation relation of the specification and the design. Even though there exists a gap between the specification and the design, our framework can verify the correspondence between state transitions, or simulation relation, of the specification and the design. Specifically, each state transition in the specification leads to a function call, which in turn triggers multiple state transitions in the design; after these state transitions, the design reaches a state where the verification conditions are asserted.

We verify the simulation relation between the specification and the design within the given bounds. We now define the simulation relation between two LTSs within bounds.

Definition 13 *(Simulation relation of two LTSs within bounds). Let M_1 and M_2 be two LTSs, and B be bounds. The simulation relation of M_1 and M_2 within bounds B is defined as $M_1 \preceq_B M_2$ if $M_1{\downarrow_B} \preceq M_2$. If $M_1 \preceq_B M_2$ holds, we say M_2 simulates M_1 within B.*

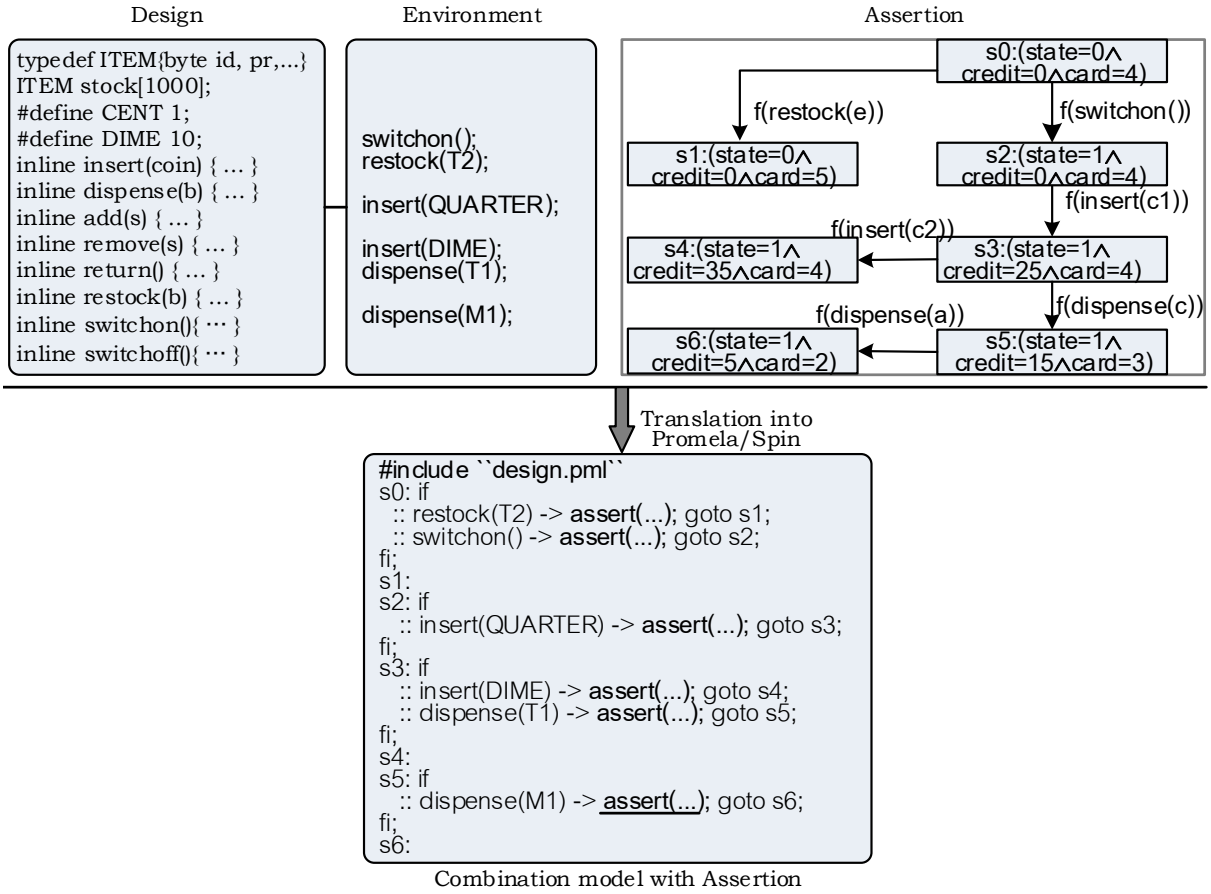


Figure 3.12: Translation into Promela/Spin

In the end, the verification of simulation between the design and the specification has been completed within the bounds. If no counter-example is found, the design conforms to the formal specification within the input bounds.

If an error is found when applying our framework to verify the design against the bounded specification, there actually exists a state transition in the bounded specification that is not followed by the design. It is obvious that this state transition is also included in the original specification; thus, the design does not conform to the original specification. Formally, $M_1 \not\preceq_B M_2 \Rightarrow M_1 \not\preceq M_2$. It indicates that the detected error really exists in the design.

In our framework, Spin can check the simulation relation between the specification and the design, where they are described in Event-B and Promela, within the given bounds.

3.6 Generator

We implemented a generator that produces the environments and the assertions from the specification. The architecture of our tool is shown in Figure 3.13. The core of our tool consists of three modules: Emulator Generator, Explorer and Translator. They are all implemented in the C++ programming language. The initial input is the specification in Event-B. The emulator generator performs the lexical and syntactic analysis to emulate the behaviors of the specification in C++. The explorer implements (Algorithm 2); it invokes functions which emulate events in Event-B and use the given bounds to outputs

the execution sequences of the specification within the bounds. The execution sequences are represented as an LTS of the specification. The enabled events appearing in the LTS are sources to generate sequences of invocations in the environments. The states appearing in the LTS are used to generate the assertions. The translator uses mappings between elements of the Event-B specification and those of the Promela design to output the environments and the assertions in Promela code.

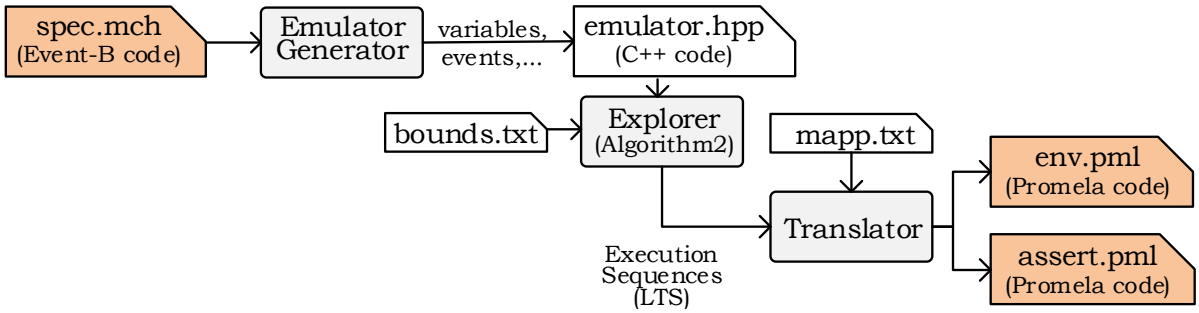


Figure 3.13: Architecture of Generator

Generating Description of Specification in C++. The emulator generator analyzes syntactic structures of the Event-B specification. They are variables, types, events, guards, substitutions, expressions, set operators, arithmetic operator, etc. These structures are translated into C++ by following the correspondences presented in Table 3.3.

Table 3.3: From Event-B to C++

Event-B	**C++**
Variable	Variable
Enumerated types	Enumerated types
Initialization	Function namely `init()`
Events	Functions
Event parameters	Arguments of function
Guards	Conditional structures
Substitutions	Assignment statements
Arithmetic operators	Arithmetic operators
Set operators (e.g., $\cup, \setminus$)	Library functions (e.g., `add`, `remove`)

The emulator generator outputs C++ codes in the targeted file with ".hpp" extension, which simulate the behaviors of the specification in the form of functions. It is in turn included in the source code of the Explorer. The Explorer invokes the functions of the specification to execute the specification and generate the LTS associated to the specification.

User Guides for Tool. Inputs produced by the users include:

- The Event-B specification file with ".mch" extension, as shown in Figure 3.1
- The bounds described in the file with ".txt". The bounds of vending machines are demonstrated in Figure 3.14.

- The mappings (from elements in the Event-B specification to those in the Promela design) described in the file with ".txt". Each correspondence from the source to the target is presented in a distinct row; the source is separated from the target by a tab character, as shown in Figure 3.15.

```
PRODUCT {a1,b2,c1,d3,e1,k4,d2,x5,u7,e3,p9,s1}
nitem   200
cred    [1..100]
```

Figure 3.14: Bounds used in verification of vending machines ("bounds.txt")

```
Variables   Variables
state   state
card    card
credit  credit
stock   stock;

Values  Values
a   M1
b   M2
c   T1
d   B1
e   T2
c3  NICKEL
c2  DIME
c1 QUARTER
off 0
on 1


Enabeld events  Invocations
restock(a)  restock(M1)
restock(b)  restock(M2)
restock(c)  restock(T1)
restock(d)  restock(B1)
restock(e)  restock(T2)
insert(c1)  insert(QUARTER)
insert(c2)  insert(DIME)
dispense(c) dispense(T1)
dispense(a) dispense(M1)
```

Figure 3.15: Mappings used in verification of vending machines ("mapp.txt")

The Explorer and the Translator are integrated into single module namely Explore_Translator. The Explore_Translator generates a targeted file with ".pml" extension. This includes the environment scripts and assertions.

1. From the Windows command line, type `emulator_generator  spec.mch`
2. Make sure that a targeted file with ".hpp" extension generated in the current folder. The emulator of the specification of vending machines is demonstrated in Figure 3.16.
3. From the Windows command line, type `explorer_translator bounds.txt mapp.txt`
4. Make sure that a targeted file with ".pml" extension generated in the current folder

```
int state;
int card;
int credit;
int nitem;
int stock[1000];
```

```
int switch_on(){
    if(state ==0){
        state=1;
        return 1;
    }
    else return 0;
}
```

```
int switch_off(){
    if(state ==1){
        state=0;
        return 1;
    }
    else return 0;
}
```

```
int insert_credit(cr){
    if(credit<10 && state ==1){
        credit=credit+cr;
        return 1;
    }
    else return 0;
}
```

```
int restock(int i)
{
    if(state==0 && card<nitem){
        card=card+1;
        add(stock,i);
        return 1;
    }
    else return 0;
}
```

```
int add(a,i){
    a[i]=1;
    return 1;
}

int remove(a,i){
    a[i]=0;
    return 1;
}
```

```
int dispense(int i){
    if(card>0 && state==1 && credit>=2){
        card=card-1;
        remove(stock,i);
        credit=credit - 2;
        return 1;
    }
    return 0;
}
```

Figure 3.16: Behaviors of specification of vending machine are emulated in C++ ("emulator.hpp")

3.7 Case studies

The purpose of the case studies is to evaluate the generality and the applicability of our framework to verify the reactive systems. The target systems used in these case studies are simple systems: vending machines (VM), controlling car on a bridge (CC), and elevator controllers (EC). We applied the framework to verify whether the designs of the target systems conform to their specifications. As we mentioned, the inputs produced by the users include the Event-B specification, the bounds, the mappings, and the design in Promela. In this section, we briefly describes the target systems. We also present the ranges to be restricted with their sizes as the bounds used for separate experiments.

3.7.1 Vending Machine

In Section 3.2, we presented a description of the vending machines. We also illustrated the Event-B specification and the design in Promela partially in Figures 3.1 and 3.3. In the framework, bounds are set for the verification by introducing finite ranges of variable values in the Event-B specification. In practical applications of the vending machines, the maximum number of available items is given for each machine. In the specification, constants `MAX` and `PRICE` define the maximum number of items, that can be contained, and the price of items. They are definite values. Variables `nitem` and `cred` may obtain values in infinite ranges. They define the number of items restocked and amount of money deposited so far. Range of values for `nitem` must be restricted to a finite set such that the highest value of the range must be less than or equal the maximum number to be given. Also, range of values for `cred` must be restricted to prevent the customers from inserting a large amount of money. Bounds are introduced to define such restriction. The mappings between elements of the specification and the design are illustrated in Figure 4.14. For example, `a` is mapped to `M1` and `cred` is mapped to `credit`.

3.7.2 Controlling car on a bridge

The main function of this system is to control cars on a bridge connecting the mainland to an island [3]. The number of cars on the bridge and the the island is limited. The bridge

[3]We refer the specification and the design of the target system in [1]

is one-way or the other, not both at the same time. In the specification, we compound the bridge and the island together to obtain the bridge_island. External behaviors are represented as events namely `FromML` and `ToML`. They correspond to cars leaving the mainland and entering the mainland. This is visualized in Figure 3.17(left) and formalized in Figure 3.17(right).

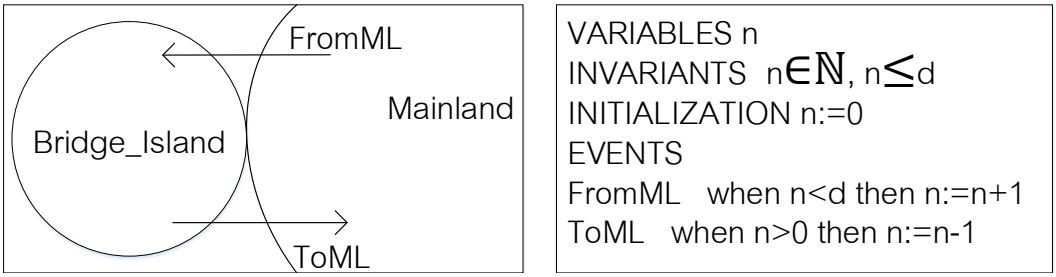


Figure 3.17: The mainland and the bridge_island

Figure 3.17(right) is a formal specification in Event-B. Variable `n` defines the number of cars on the bridge_island; and, constant `ncar` defines the maximum value for the number of cars on the bridge_island.

To describe the design, we separate the bridge from the island. New variables are introduced: `a` - the number of cars on the bridge going from the mainland; `b` - the number of cars on the island; and `c` - the number of cars on the bridge going from the island. The behaviors are `FromML`, `ToIL`, `FromIL`, and `ToML`. They correspond to cars leaving the mainland and entering the bridge, leaving the bridge and entering the island, leaving the island and entering the bridge, and leaving the bridge and entering the mainland. This is visualized in Figure 3.18(left) and formalized in Figure 3.18(right).

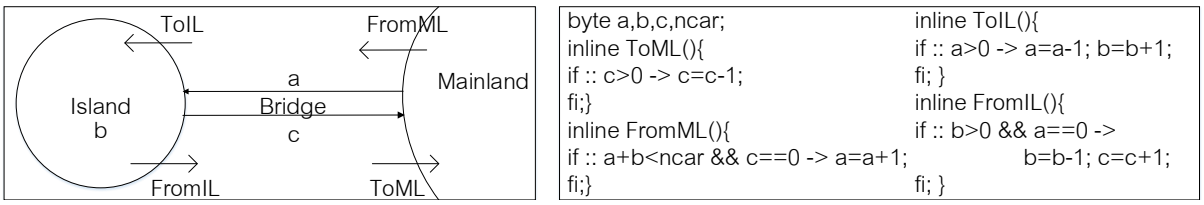


Figure 3.18: The mainland, the bridge, and the island

No complex data structure is required to describe the design of such target system; the design could be easily described in Event-B and apply the refinement facility to verify the conformance. However, to obtain a completely automatic verification, we described the design in Promela and applied our framework to verify the design in Promela against the specification in Event-B. The environment of the target system includes the cars moving between the mainland and the island through the bridge. Bounds are introduced to define the maximum number of cars on bridge and island. Therefore, the range of values for `ncar` must be restricted.

3.7.3 Elevator

The elevator has a definite number of floors and each floor has its request buttons (one for up and another for down) and a control light. A sensor is located at every floor. We can use these sensors to locate the current position of the elevator cab. The cab itself

consists of several parts: A door, which can be opened and closed by a motor. Two sensors inform the control system about the door status. A light sensor can detect objects while the door is closing. The elevator cab engine moves the cab up or down. Informally, the elevator behavior is defined as follows. If a passenger presses a request button at a floor, the request light is switched on. The cab moves to this floor within finite time. When the floor is reached, the door opens, and the request lamp is turned off. The door stays open for some time (10 seconds) to allow passengers to enter or exit the cab. After this time, the door closes. The cab is moved by the elevator motor. If the cab should stop at a certain floor, the motor is stopped immediately after the reception of a signal from the corresponding floor sensor. The control system should send an additional signal to the motor, if the motor should stop the cab at the next floor. The motor uses this signal to reduce the cab speed. This enables the motor to stop the cab at the exact floor position. The requests from the individual floors can be served using different strategies. The most important requirement for such a strategy is fairness. Every request must be served in finite time.

We summarize main elements of the specification and the design of an elevator controller in Table 3.4. The specification in Event-B describes services which response to the stimulus from its environment. In this case study, we consider that the environment of the elevator includes the passengers and the floor sensors. Services are described in form of events in Event-B: `RequestBPressed`, `DestinationFlPressed`, and `DestinationFlDetected` responding to the stimulus when a passenger presses a request button, when a passenger presses a destination floor button, and when a floor sensor detects a destination floor, respectively. The design contains functions with the same names as those in the specification, in an implementable level. Additionally, the design describes internal behaviors of the elevator. For example, strategies to place a request in the queue and serve the request are described in `put_req_inQ` and `remove_req_fromQ`.

Table 3.4: Specification and Design of Elevator Controller

Specification		Design	
Abstract data	Visible Events	Data structures	Functions
request_set	RequestBPressed	request_array	RequestBPressed
nfloor	DestinationFlPressed	nfloor	DestinationFlPressed
	DestinationFlDetected		put_req_inQ
			remove_req_fromQ
			DestinationFlDetected

The number of floors should be a finite value. Therefore, the range of value for `nfloor` must be restricted.

All experiments are conducted on an Intel(R) Core(TM) i7 Processor at 2.67GHz running Linux. Verification results outputted by Spin are shown in Table 3.5. Here, values in column "Size of Ranges" express bounds of the verification. Column "LTS Generation" shows statistics of the execution sequence generator. Columns "#State", and "#Trans" present the number of distinct states and that of transitions appearing in the execution sequences, each transition corresponds to a function call; column "Time" present the time taken (s) for the generation. Column "Model Checking" presents statistics of the model

Table 3.5: Experiment Outputs

Target Sys.	Bounds	LTS Generation			Model Checking		
VM	nitem ∈ [0..200]	#State	#Trans	Time(s)	Mem(Mb)	Time(s)	Result
No.1	50	151	252	1.2	129.2	1.0	√
No.2	200	604	1004	50.2	130.4	5.0	√
CCB	ncar ∈ [0..20]	#State	#Trans	Time(s)	Mem(Mb)	Time(s)	Result
No.3	10	121	220	1.1	129.2	1.0	√
No.4	20	441	840	60.2	129.2	1.0	√
EC	nfloor ∈ [0..20]	#State	#Trans	Time(s)	Mem(Mb)	Time(s)	Result
No.3	10	865	1820	1.1	129.2	1.0	√
No.4	20	2441	6450	60.2	129.2	1.0	√

checker including total actual memory usage, the time taken (s), and the verification result in which √ indicates the verification has been completed.

In verification of the vending machine, range of values for `nitem` is a finite set `0..200`. Case No.1 is conducted with 50 available items; this allows to restock 10 slots of products and 5 products in each slot. Case No.2 corresponds 20 slots and 10 products in each slot. This range is appropriate in practical applications of the vending machines.

In verification of the elevator, range of values for `nfloor` is a finite set `0..20`. Case No.1 is conducted with 10 floors; case No.2 corresponds 20 floors. This range is appropriate for some practical applications of the elevator.

This result offers a high degree of confidence on the conformance of designs with respect to their specifications within input bounds. We found that the framework could be straightforwardly applied to verify various reactive systems where the designs described in Promela and their formal specifications described in Event-B. This shows applicability of our framework in verification of the reactive systems.

3.8 Summary

This chapter has presented the specification and the design of the reactive systems and presents a framework to verify the conformance between them. In the earlier sections, we characterize the specification independently on the design to show a gap between them. This gap motivates to use different specification languages for the specification and the design. In the next sections, our verification framework is explained step by step using simple examples for readability. In addition, this framework is precisely defined by formal definitions. The core of our approach is base on a simulation relation between the specification and the design. By verifying such a simulation relation, we could deal with (i) the difference of the specification languages used for two models and (ii) the gap between two models; and we could check the correspondence between state transitions of two models. This shows that the design conforms to the specification. Some case studies are presented in the last section. The results of these case studies shows that the framework can be straightforwardly applied to verify various reactive systems.

Chapter 4

Verifying OSEK/VDX OS Design

As an automotive industry standard of operating system specification, OSEK/VDX OS specification [40] is widely applied in the process of designing and implementing the operating system (OS) for automotive systems. In order to obtain a high-reliability OS, a design model is often developed in advance, and some verification techniques like model checking [5][11] are employed to check whether the design conforms to the OSEK/VDX OS specification. If the design model has been ensured then the prototype can be developed following the design model. We are working on a design of the OS compliant with the OSEK/VDX standard. The aim of this work is to provide a high quality OS by applying automated formal verification.

In previous chapter, we presented a framework to check the design models of the reactive systems against their specifications based on a simulation relation [31][34]. The framework includes three main steps. Firstly, a labeled transition system (LTS) is generated from the specification. Next, from each state appearing in the LTS, verification conditions which must be met by the corresponding state of the design are generated. Finally, the design in combination with the LTS is input into a model checker to check the verification conditions. In this way, we can check the correspondence of state transitions, or the simulation relation, between the specification and the design. This ensures that the design conforms to the specification.

In this chapter, we present a case study of applying the framework to the verification of an OS design compliant with the OSEK/VDX standard which is described in Promela/Spin. In the earlier section of this chapter, we briefly present a workflow to apply to verify the OS design according to the proposed framework. There exists an informal specification of such operating system, called OSEK/VDX OS Specification. This specification is not an input accepted by formal verification since it is described in natural language. In order to formally verify the OS design against the specification, we faithfully formalize the OSEK/VDX OS specification in Event-B. Then, we apply our framework to verify the OS design against the formal specification. In the case study, we mainly explain how we defined the appropriate bounds to check our desired properties of the OS. We also show the results of the experiments and evaluate the effectiveness and the practicality of the framework.

4.1 Overview of Workflow

Our workflow to apply the framework in verification of the OS design using its specification in Event-B is shown in Figure 4.1. According to the proposed framework, checking the conformance of the OS design to the Event-B specification is based on a simulation relation between them. Here, the Event-B specification and the OS design are defined as LTSs. Our objective is to verify whether the design satisfies the formal specification. The technique we use for this verification is model checking. Firstly, we formalize the OSEK/VDX OS specification in Event-B. The consistency of the behaviors and the properties are ensured in this step. Secondly, to avoid the state explosion, we give reasonable bounds for the verification. In this step, we pick up behavior scenarios of the OS from the OSEK/VDX OS specification. The scenarios provide examples of the intended system behaviors which satisfy the desirable properties. Based on the behavior scenarios, we determine bounds for the verification so that when we apply them to the Event-B specification, the execution sequences of the specification within the bounds cover at least the behaviors under consideration and prevent the state explosion. Finally, we apply the proposed framework to check the OS design conforms to the Event-B specification within the bounds: generating the LTS of the specification; translating it into Promela to generate the environment and assertions; and applying model checking. When this check has been completed, one can say the OS design satisfies the desirable behaviors and properties within the bounds.

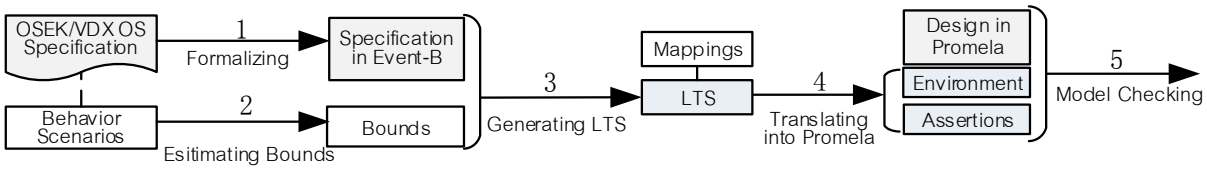


Figure 4.1: Model Checking Design using Formal Specification (workflow)

4.2 Formalizing OSEK/VDX OS Specification

In this section, we present formalization of the OSEK/VDX OS specification in Event-B. Our formalization process is shown in Figure 4.2. In the first step, we analyze the OSEK/VDX OS specification to capture requirements of OSEK OS. The second step is to formally define the requirements in Event-B and validate them. In this step, we benefit refinement and consistency proof provided by Event-B/Rodin to facilitate the validation.

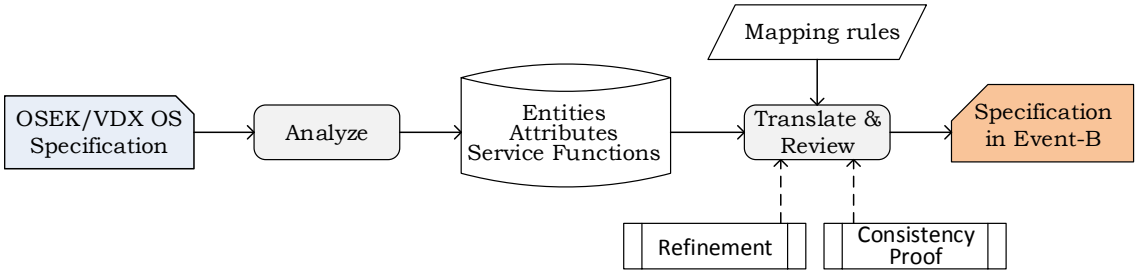


Figure 4.2: Formalizing OSEK/VDX OS Specification

4.2.1 Analysis of OSEK/VDX OS specification

OSEK specification describes not only the OSEK operating system but also the concepts of a real time operating system, capable of multitasking and changes from the old versions of this document. Our formalization focuses on the behavior of OSEK OS. In order to specify the behavior of OSEK OS, three classes of core features are singled out. They are entities that the operating system manages, service functions of the system and constraints on the behavior of the system. The purpose of this work is to formalize only features of OSEK OS that are described in OSEK specification. This work is not a translation of sentences one by one from a natural language to a formal language. Therefore, the first step in our formalization process is to identify features of OSEK OS from OSEK specification and retrieve their descriptions from that specification.

Entities. Entities/Objects managed by OSEK OS are easily identified from OSEK specification because they are emphasized in the document. They are task, resource, ready queue, alarm, event and interruption. When we identify the entities, we need to extract descriptions of their attributes and relationships and enumerate them as demonstrated in Table 4.1.

Table 4.1: Attributes and relationships of entities

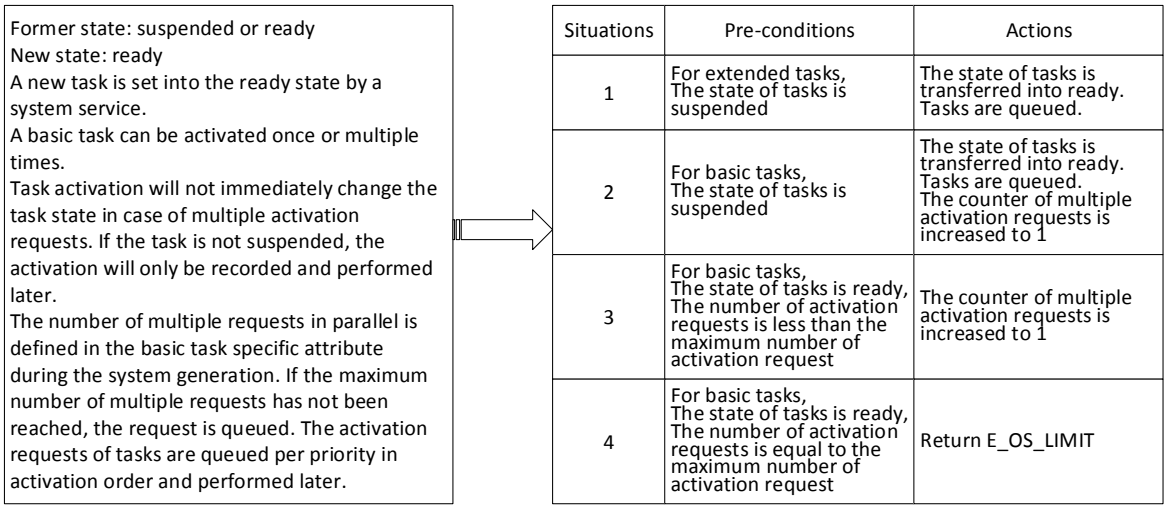
Entities	**Attributes**
tasks	state of task, priority of task
resources	state of resource, ceiling priority of resource
event	-
interruption	-
Relationships: tasks occupy resources, tasks own events	

Functions. Functions of OSEK OS are services that OSEK OS performs to manage and control the entities. Services represent the behavior of the system. When OSEK OS receives an invocation from the outside, it checks the current state of the related entities and performs the corresponding service to respond. The performance of service often makes some state transitions. Hence, it is necessary to analyze the conditions on the state of some objects before each service served, they are so-called pre-conditions or guards of each service. We also collect descriptions about actions that make the state transitions.

Descriptions of each service are usually scattered in the informal specification. Therefore, we must collect all descriptions of each service. For example, descriptions for service ActivateTask are collected and represented on the left side of Figure 4.3. From these descriptions, we see that the service ActivateTask may happen in four different situations. Each situation is described by a pair of pre-conditions and actionseffects. When analyzing the OSEK specification, we should separate these situations to facilitate the formalization. For example, the descriptions of service ActivateTask are rearranged and decomposed into four situations as shown on the right side of Figure 4.3. Two descriptions in Figure 4.3 have the same meaning but the right one facilitates not only modeling the service but also checking the meaning of its correspondence in the formal model. This is explained more in the latter part of this section.

Pre-conditions are also called local constraints because they need to be satisfied at the

time only before each service served. They differ from the global constraints which are satisfied all of the time.



Former state: suspended or ready
New state: ready
A new task is set into the ready state by a system service.
A basic task can be activated once or multiple times.
Task activation will not immediately change the task state in case of multiple activation requests. If the task is not suspended, the activation will only be recorded and performed later.
The number of multiple requests in parallel is defined in the basic task specific attribute during the system generation. If the maximum number of multiple requests has not been reached, the request is queued. The activation requests of tasks are queued per priority in activation order and performed later.

Situations	Pre-conditions	Actions
1	For extended tasks, The state of tasks is suspended	The state of tasks is transferred into ready. Tasks are queued.
2	For basic tasks, The state of tasks is suspended	The state of tasks is transferred into ready. Tasks are queued. The counter of multiple activation requests is increased to 1
3	For basic tasks, The state of tasks is ready, The number of activation requests is less than the maximum number of activation request	The counter of multiple activation requests is increased to 1
4	For basic tasks, The state of tasks is ready, The number of activation requests is equal to the maximum number of activation request	Return E_OS_LIMIT

Figure 4.3: Descriptions of service ActivateTask - before and after rearranging

Constraints. The constraints refer to the conditions that must hold in states. As mentioned, global constraints are conditions under which the system state must satisfy all of the time. For example, one constraint on the relation between tasks and resources is described in OSEK specification:
`Two tasks cannot occupy the same resource at the same time`.
This constraint means that for each resource, at the most one task can occupy at any point of time. It must hold at any state of the system.

The result of this step is a list of OSEK OS features retrieved from the OSEK specification. The meaning of informal descriptions in the feature list must be captured for the purpose of translation and validation in the next step.

4.2.2 Translation and Validation

Faithfulness of formalization is validated according to the following criteria: (1) every feature of OSEK OS is defined correctly in the formal specification and (2) no redundant feature is added to that specification. Every feature must be translated into Event-B and meaning of the formal description must be reviewed to make sure that it matches with meaning of the informal one. We know that the formalization could not be validated by machine; however, it could be checked intuitively. This is still accepted by the client and the stakeholder. Our objective is to facilitate the validation of the formalization so that the formalization could be accepted.

To facilitate the validation, we consider (i) the bi-directional traceability [13] between each feature described in the informal specification and its appearance in the formal specification and (ii) the feasibility of checking the meaning of an informal description with its formalization.

Traceability. The traceability according to the forward direction is used to trace forward from each feature i in the feature list (denoted L) to a correspondent element j in the

formal specification (denoted S). The traceability according to the backward direction is used to trace backward from each j in S to a correspondent feature i in L. For the forward direction, after tracing each feature i toward the correspondent element j, we need to review meaning of j against i. For the backward direction, apply this method, reversely.

To achieve the traceability, we define *mapping rules*. Each mapping rule represents a correspondence between a feature class in the list and a notation in the formal specification language. The mapping rule is constructed in the following form: $X \rightarrow Y$. Where X is a feature class enumerated in L and Y is the formal notation used to specify X. The meaning of mapping rule is defined that 'If an instance x of X appears in L then an instance y of Y must appear in S and vice versa. So, whenever a mapping rule is applied, a trace link [13] is established to associate each instance x of X with an instance y of Y. We define the mapping rules by matching feature classes in the list with notations of the formal specification language to find the correspondences between them.

As we mentioned, the formal specification described in Event-B is regarded as a highly abstracted level description of the systems. This description mainly consists of state variables, operations (events) on the variables, and state invariants. The variables are typed using set theoretic constructs such as sets, relations, and functions. The events are defined with their guard conditions and substitutions (so-called before and after predicates), which allow both deterministic and non-deterministic state transitions. Table 4.2 summarizes the correspondence between features and Event-B notions. We follow this correspondence to translate features of the OS into Event-B and establish the trace links.

Table 4.2: Matching Features and Notions in Event-B

Features	Notions in Event-B
Concepts (Entities)	Variables
System Services	Events
Pre-conditions	Guards (logical expressions)
ActionsEffects	Actions (substitutions)
Constraints	Invariants

Refinement and Consistency Proofs. As mention earlier, it is impossible to formally prove that an informal description matches a formal definition. The only way to check the meaning of a formal definition with an informal description is to review the meaning of the logical expression and compare with what is understood from the informal description, intuitively. For example, a simple description picked up from the OSEK/VDX OS specification:
`The task <id> is transferred from the suspended state into the ready state.`
This description is translated into Event-B in the form of a guard condition and a substitution as follows:
$WHERE\ tid \in tasks \wedge tstate(tid) = sus\ THEN\ tstate(tid) := rdy$
It is accepted that the meaning of the latter matches that of the former. However, the features of OSEK/VDX OS are generally complex. Formalization of complex features is hard and could be wrong. We use mechanisms provided by Event-B/Rodin to facilitate the formalization and the validation. Firstly, we decompose the complex features into the

simple ones (sub-features) and distribute them one by one in different levels of abstraction by applying refinement. We now present how formalizing the ready queues to explain this idea. The ready queue is used to store instances of tasks which are activated, and record the order of task activations. We decompose the description of the ready queues into two levels of abstraction. In one level of abstraction, the ready queue was defined as a non-order set of items. In another level of abstraction, descriptions of the order of items have been added to the definition of the ready queue. Finally, we obtained the definition of the ready queue which sufficiently takes the order into account. After decomposing features, a plan for distributing sub-features must be established to ensure the consistency of the overall model. We apply the refinement technique to distribute sub-features in different abstraction layers of the system model as illustrated in Figure 4.4 (left). Figure 4.4 (middle) shows formal descriptions of the ready queues as non-order sets. Figure 4.4 (right) shows formal descriptions of the ready queues with the order of items. The Event-B specification defines several ready queues. Each ready queue `rdyQuItem` corresponds to an individual task priority (`tpri(t)`). We reference a task in the ready queues by its priority and its order activation. Therefore, each `rdyQuItem` associates a position in the ready queues (`qsize`) to the task which is currently stored at that position. When a task t is activated, the size of the ready queue that corresponds to the priority of t is increased by 1, and an instance of t is pushed into the last position of this ready queue.

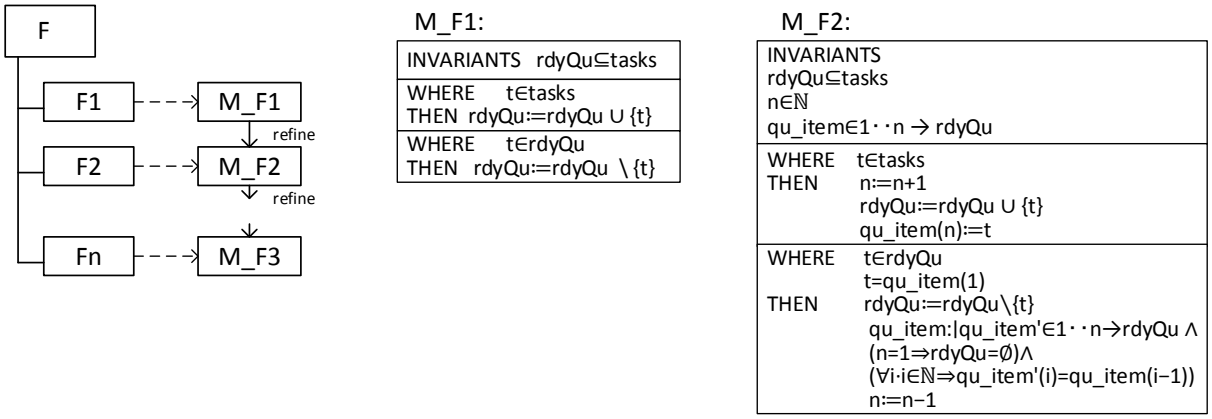


Figure 4.4: Two levels of specification of ready queues

Secondly, we benefit the consistency proofs provided by Event-B/Rodin to review the meaning of the formal descriptions. The consistency proofs should be completed at each step of the translation. It is stated that the final percentage of automatic proofs is a good indication of the quality of the model [1]. Furthermore, the proof results are guidelines for modeling and analyzing the model by showing some hints for where and what should be improved in the model. Assume that the existing formal descriptions have already been reviewed, when a new formal description is added, it should be confirmed that the new one is consistent with the existing ones. If the proof result shows any inconsistency then the new one needs to be revised based on hints from the proof results. The hints may show: (i) which invariant should be added; (ii) which guard should be added to strengthen the model; and (iii) which form of expressions should be specified to get automatic proof. This requires that we must prove the formal model at each step of the formalization.

Figure 4.5 illustrates the structure of the specification in Event-B. The variables such

as `tasks`, `res` represent all the created tasks and the managed hardware resources. The invariants represent constraints, e.g. at any time only one task is in running state. System services are formally defined as *guarded events* like `ActivateTask` with guard conditions, e.g. task `t` is in a suspended state, and actions that make the state transitions, e.g. transferring `t` to ready state and pushing it into the corresponding ready queue.

```
VARIABLES
    tasks, res, evt, isr, tpri, tstate, rdyQuItem, qsize, rdyQSet, acnt
INVARIANTS
    rdyQSet⊆ tasks
    qsize ∈ 0‥MAXPRI → ℕ1
    rdyQuItem ∈ 0‥MAXPRI × 0‥MAXQSIZE → rdyQSet
    tstate ∈ tasks → STATE
    ∀ta,tb·ta∈tasks∧tb∈tasks∧tstate(ta)=run∧tstate(tb)=run⇒ta=tb
EVENTS
ActivateTask  ≙
    any t where t∈tasks, tstate(t)=sus, acnt(t)<MAXACT
    then tstate:|tstate'∈tasks→STATE ∧ (tstate(t)=sus ⇒ tstate'(t)=rdy)
            rdyQSet:|rdyQSet'⊆ tasks ∧ rdyQSet' = rdyQSet ∪ {t}
            rdyQuItem(tpri(t)↦qsize(tpri(t))+1):=t
            qsize(tpri(t)):=qsize(tpri(t))+1
```

Figure 4.5: Formal Specification in Event-B

In summary, the main points of our approach are as follows: (1) Pre-processing of the original specification, that is, identification of the features and their decomposition to facilitate the review of the formal specification; (2) Introducing mapping rules to achieve bi-directional traceability; and (3) proving the internal correctness of the formal model at each step of the formalization to support for validating the formal model against the informal specification.

As a result, the specification in Event-B contains a list of state variables, a list of events which modify states (or state variables), and a list of invariants which are preserved by events (or transitions). Thus, the possible execution sequences of such specification can be represented as an LTS.

4.3 OS Design Model in Promela

OSEK OS is an open system. It does scheduling of the tasks if it gets stimulus such as system call invocations from its environment. The environment of the OS includes applications running on the OS and the hardware causing interruptions. The operating system does nothing if it does not get any stimulus. The OS design only defines a collection of service functions, it cannot operate by itself. To operate it, we need an environment which calls functions of the OS; the design must be verified in communication with the environment. As we explained later, the environment is constructed from the specification, and input to Spin to check the simulation relation.

In Promela, service functions of the OS can be described by using inline functions. Figure 4.6 (left) illustrates the whole structure of the OS design. We call this model a *design model*. It is constructed based on the OSEK/VDX OS specification and described in about 2800 lines of Promela code, according to the approach in [3]. It first defines data structures such as `tsk`, `res`, and `ready` which represent an array of tasks,

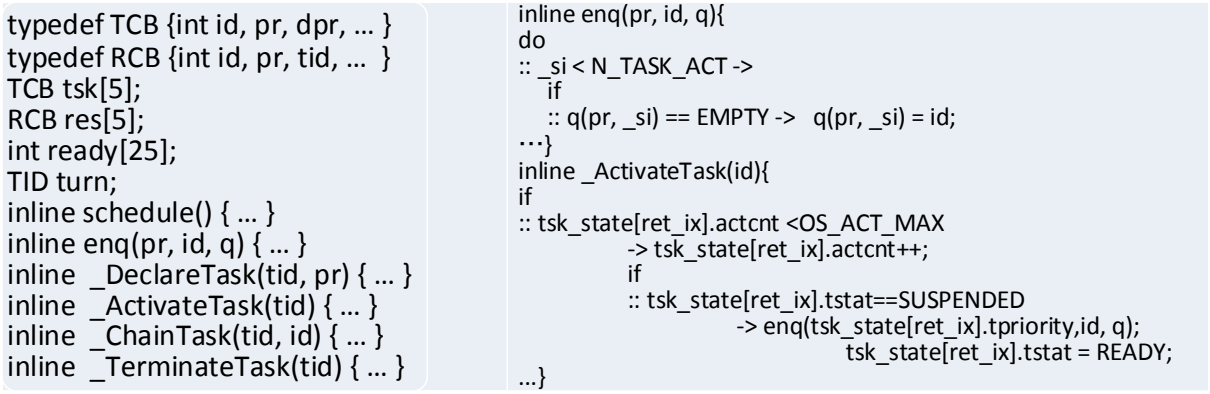


Figure 4.6: OS design in Promela

an array of resources, and ready queues, respectively. Following these data structures, a set of functions are defined. For example, `_ActivateTask` and `_TerminateTask` are the functions to perform activation and termination of tasks, respectively. Variable *turn* is used to store the identifier of the task which is currently in state *running*. The functions with "_" in the names like `_ActivateTask` are called from the outside of the OS. The functions without "_" in the names like `enq(pr,id,q)` are called internally. Figure 4.6 (right) illustrates the body of the functions which describes the implementation of the OS services. They mainly consist of statements that represent the behavior of system over the variables. The variables such as `tsk`, `res`, and `ready` represent information about the system (states). The execution of statements changes the values of variables. Therefore, the model in Promela can be interpreted as an LTS if we consider that the variables are states and each function call is a label to make transitions on the states.

4.4 Simulation relation between Specification and Design of OS

We give a simulation relation between the specification and the design of OSEK OS based on the relation of variables, values in two models. The relation of variables, values in the specification and the design are represented by mappings $R : V_S \to V_D$ and $C : D_S \to D_D$.

Figure 4.7(left) demonstrates an LTS, which is generated from the specification of OSEK OS. The LTS represents possible sequences of state transitions within the bounds. Here, the rectangles represent the states and the labeled arrows represent the events that are enabled in each state. For example, two events `AT(t1)`, `AT(t2)` are enabled in state $s0$, and two events `TT(t1)`, `AT(t2)` are enabled in state $s1$. In our framework, the states are defined as the value assignments; however, we show them here as values, e.g. `(sus, sus, sus)`, for readability. Similarly, Figure 4.7(right) demonstrates an LTS of the design. With the given mappings, we can see a particular simulation between the specification and the design: $\preceq = \{(p0, q0), (p1, q1), (p2, q2), (p3, q3), (p4, q4), (p5, q5)\}$.

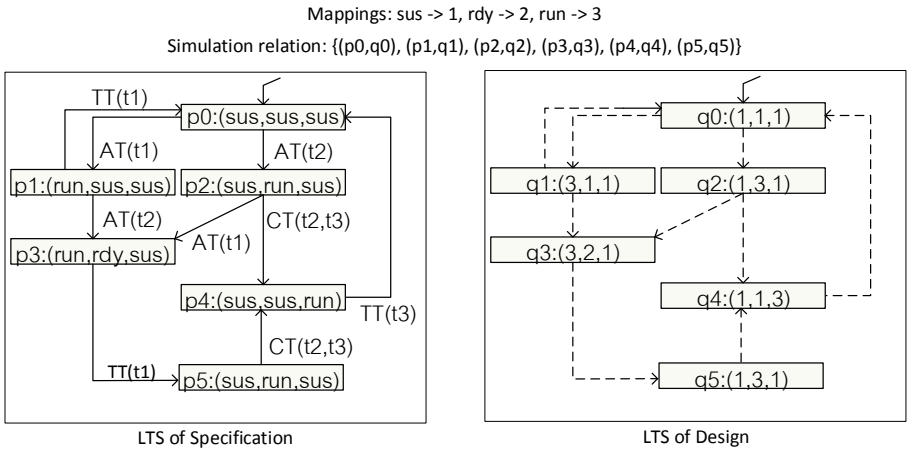


Figure 4.7: A Simulation between Specification and Design based on Mappings

4.5 Bounding Process

4.5.1 Implementation of bounds

As mention earlier, model checking does an exhaustive check of the system. It needs a representation of the system as a finite set of all possible states. In previous chapter, we presented one implementation of the bounds to restrict the range of values for every variable and parameter. Firstly, abstract types in the Event-B specification must be replaced by concrete types. Also, types having infinite ranges of values like Int and Nat must be restricted as small ranges. We consider that such implementation of the bounds is not sufficient enough to verify the OS design. In this chapter, we additionally introduce two implementations of the bounds to lead the verification to focus on the intended behaviors of the OS. Such implementations of the bounds are all to reduce the size of LTS explored from the Event-B specification.

The size of the LTS depends on ranges: the range of values for every variable and parameter (D_S); the range of operating system services which are defined as events in Event-B (Σ_S); and the depth of the execution of the Event-B specification (d).

As presented in the previous chapter, D_S is replaced with its subset $\widehat{D}_S$ by a mapping X_V and X_P such that every variable and parameter obtain the values in the finite domain. Within this restriction, the state space and the set of transitions of the LTS become finite sets. Such restriction is essential to apply the model checking technique. In addition, we could lead the verification to focus on the intended behaviors of the OS by replacing Σ_S with its subset $\widehat{\Sigma}_S$. In this way, we separate verifications to deal with distinct groups of system services. As a result, set of events that may be enabled in states is reduced; thus, set of transitions that may be triggered in states of the LTS is also reduced. Also, we could restrict the depth of the execution of the Event-B specification by giving a finite value for d. This is useful to check the properties among different groups of system services while avoiding the state explosion.

Figure 4.8 illustrates the execution sequences of the Event-B model are bounded by: (i) $\widehat{D}_S$; (ii) $\widehat{D}_S$ and $\widehat{\Sigma}_S$; and (iii) $\widehat{D}_S$ and d, respectively. Here, we restrict Σ_S to obtained limited sets of enabled events in states as $LimEvent(s0) = \{b, c\}$, $LimEvent(s2) = \{f, g\}$, $LimEvent(s3) = \{h\}$, $LimEvent(s5) = \{n\}$, $LimEvent(s6) = \{o\}$, $LimEvent(s10) =$

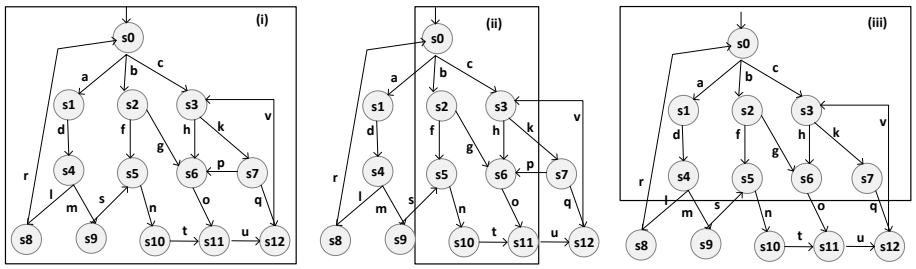


Figure 4.8: Behaviors within the Bounds

$\{t\}$, $LimEvent(s11) = \{\}$. We restrict the depth of the execution sequences by giving $d = 2$. The states and the transitions being outside of the boxes are excluded when the bounds are applied. We determine bounds by studying the properties and the behavior scenarios of the target system. This step is explained in Section 4.5.2.

As we mentioned, the bounds are given to restrict both of the state space and the set of triggered transitions. In chapter 3, we defined a general bound with a pair $\langle G, H\rangle$ where G is a mapping used to restrict the state space, H is a mapping used to restrict applicable transitions in each state. Such notion of the bounds could be applied for every LTS-based model. For the verification of the OS design, as we mentioned earlier, three implementations of the bounds are applied. Formally, they are denoted by $\langle G1, H1\rangle$, $\langle G2, H2\rangle$, $\langle G3, H3\rangle$ where:

- $\langle G1, H1\rangle$ defines the restriction of data elements including the range of values for the variables and the event parameters. This matches with the bounds defined in Chapter 3 and refers to case (i) of Figure 4.8.
- $\langle G2, H2\rangle$ defines the restriction of both the data elements and the events. This refers to case (ii) of Figure 4.8.
- $\langle G3, H3\rangle$ defines the restriction of both the data elements and the depth of the execution sequences. This refers to case (iii) of Figure 4.8.

In chapter 3, we also presented the implementation of the bounds to restrict the range of values for every data element in Event-B model by defining mappings X_V and X_P. Such implementation is also applied to $\langle G1, H1\rangle$ to verification of OSEK OS.

Suppose $M_S = \langle Q_S, \Sigma_S, \delta_S, I_S\rangle$ is a LTS associated to the specification. We define mappings $G1$ and $H1$ as follows:

- $G1(I_S) = \{\sigma \in I_S \mid \forall v \in \mathcal{V}_S.\sigma(v) \in X_V(v))\}$,
- $G1(Q_S) = \{\sigma \in Q_S \mid \forall v \in \mathcal{V}_S.\sigma(v) \in X_V(v))\}$ and
- $H1(\sigma, e) = tt$ iff $e \in ES_X(\sigma)$.

With these $G1$ and $H1$, we obtain a bounded LTS denoted $S{\downarrow_{\langle G1,H1\rangle}}$.

As for $\langle G2, H2\rangle$, we use $ES_{XE}(\sigma)$ to denote the set of events which are applicable to state σ and satisfy the restriction of both $\widehat{D}_S$ and $\widehat{\Sigma}_S$. We define mappings $G2$ and $H2$ as follows:

- $G2(I_S) = G1(I_S)$,
- $G2(Q_S) = G1(Q_S)$ and

- $H2(\sigma, e) = tt$ iff $e \in ES_{XE}(\sigma)$.

With these $G2$ and $H2$, we obtain a bounded LTS denoted $S{\downarrow_{\langle G2,H2\rangle}}$. Algorithm 2 in chapter 3 is extended by Algorithm 3 to obtain $S{\downarrow_{\langle G2,H2\rangle}}$ from S, $\widehat{D}_S$ and $\widehat{\Sigma}_S$. This extension is shown at line 12 of Algorithm 3, where $ES_X(\sigma)$ is replaced with $ES_{XE}(\sigma)$.

Algorithm 3 Main() (Generating $S \downarrow_{\langle G2,H2\rangle} = < \widehat{S}, \widehat{\Sigma}, \widehat{\delta}, \widehat{I} >$ from S, X_V, X_P), and E

```
1: Initially, QUEUE = VISITED = Ŝ = Σ̂ = δ̂ = Î = EMPTY
2: for each σ0 ∈ I_S do
3:    if ∀v ∈ V_S, σ(v) ∈ X_V(v) then
4:       Push(QUEUE, < σ0, 0 >)
5:       Ŝ = Ŝ ∪ {σ0}
6:       Î = Î ∪ {σ0}
7:    end if
8: end for
9: while QUEUE ≠ empty do
10:   < σ, i > = Pop(QUEUE)
11:   VISITED = VISITED ∪ {σ}
12:   Ê = ES_XE(σ)
13:   if Ê != empty then
14:      for each event e = (g, a) ∈ Ê do
15:         σ' = {v ↦ [(act(e))(v)]_σ | v ∈ V_S}
16:         if σ' ∉ VISITED then
17:            Push(QUEUE,< σ' >)
18:            Ŝ = Ŝ ∪ {σ'}
19:         end if
20:         Σ̂ = Σ̂ ∪ {e}
21:         δ̂ = δ̂ ∪ {σ -e-> σ'}
22:      end for
23:   end if
24: end while
25: return S ↓⟨G2,H2⟩
```

$ES_{XD}(\sigma)$ is used to denote the set of events which are applicable to state σ and satisfy the restriction defined by mapping X and within a depth defined by D. We define mappings $G3$ and $H3$ as follows:

- $G3(I_S) = G1(I_S)$,
- $G3(Q_S) = G1(Q_S)$ and
- $H3(\langle \sigma, d\rangle, e) = tt$ iff $e \in ES_X(\sigma)$ and $d \leq depth$.

With these $G3$ and $H3$, we obtain a bounded LTS denoted $S{\downarrow_{\langle G3,H3\rangle}}$. Algorithm 2 in Chapter 3 is extended by Algorithm 4 to obtain $S{\downarrow_{\langle G3,H3\rangle}}$ from S, $\widehat{D}_S$ and d. The extension is shown at line 12 and 13 of Algorithm 4, where $ES_X(\sigma)$ is replaced with $ES_{XD}(\sigma)$ and the search depth is limited by d.

Algorithm 4 Main() (Generating $S \downarrow_{\langle G3,H3\rangle} = < \widehat{S}, \widehat{\Sigma}, \widehat{\delta}, \widehat{I} >$ from S, X_V, X_P), and d

Initially, $QUEUE = VISITED = \widehat{S} = \widehat{\Sigma} = \widehat{\delta} = \widehat{I} = EMPTY$
for each $\sigma_0 \in I_S$ **do**
  if $\forall v \in V_S, \sigma(v) \in X_V(v)$ **then**
    Push($QUEUE, < \sigma_0, 0 >$)
    $\widehat{S} = \widehat{S} \cup \{\sigma_0\}$
    $\widehat{I} = \widehat{I} \cup \{\sigma_0\}$
  end if
end for
while $QUEUE \neq empty$ **do**
  $< \sigma, i > =$ Pop($QUEUE$)
  $VISITED = VISITED \cup \{\sigma\}$
  if $i \leq d$ **then**
    $\widehat{E} = ES_{XD}(\sigma)$
    if $\widehat{E}$! $= empty$ **then**
      for each event $e = (g, a) \in \widehat{E}$ **do**
        $\sigma' = \{v \mapsto [(act(e))(v)]_\sigma | v \in V_S\}$
        if $\sigma' \notin VISITED$ **then**
          Push($QUEUE, < \sigma' >$)
          $\widehat{S} = \widehat{S} \cup \{\sigma'\}$
        end if
        $\widehat{\Sigma} = \widehat{\Sigma} \cup \{e\}$
        $\widehat{\delta} = \widehat{\delta} \cup \{\sigma \xrightarrow{e} \sigma'\}$
      end for
    end if
  end if
end while
return $S \downarrow_{\langle G3,H3\rangle}$

4.5.2 Determining bounds according to properties

We focus on the verification of properties concerned with the correctness of scheduling. Scheduling is concerned with entities such as tasks, ready queues, resources, events, and interruption routines. We illustrate important properties of scheduling in Table 4.3.

Bounds are determined based on the properties and behavior scenarios of the target system. The scenarios provide examples of the intended system behaviors which satisfy the desirable properties. Each scenario represents a partial behavior of the system. In the following, we analyze the behaviors of full preemptive scheduling to illustrate how to determine appropriate bounds for verification of such behaviors.

Figure 4.9 visualizes a scenario which represents desirable behaviors of full preemptive scheduling. This scenario describes the behaviors that satisfy property (P1). Here, the state transitions of two tasks, `T1` with priority `2` and `T2` with priority `1`, caused by `ActivateTask(T1)`. Initially, two tasks are both in the suspended state. Next, task `T2` transfers to running state after `ActivateTask(T2)` and task `T1` is still suspended. Then, `T2` activates `T1`. Due to the higher priority of task `T1`, task `T2` is preempted by task `T1`.

Table 4.3: Properties

Prp.	Description
P1	A task with lower priority is preempted by a task with higher priority (full preemptive scheduling)
P2	A terminated or chained task goes into the suspended state
P3	An extended task in the waiting state is released to the ready state if at least one event for which the task is waiting has occurred
P4	If a task or interrupt routine requires a resource, and its current priority is lower than the ceiling priority of the resource, the priority of the task is raised to the ceiling priority of the resource
P5	If a task or interrupt routine releases the resource, the priority of this task is reset to the priority which was dynamically assigned before requiring that resource
P6	The index value is within the bounds of the array (boundary check)
P7	A task must not terminate or chain another task while holding resources
P8	Over activation of a task is prohibited

By counting objects appearing in the scenario, we see 2 tasks, 2 different values for the task priorities, and 3 enabled events including `ActivateTask(T1)`, `ActivateTask(T2)`, and `TerminateTask (T1)` used to describe such behaviors. The least configuration of the OS to check such behavior includes 2 tasks and 2 different values for the task priorities. We use `TASK` and `PRI` to denote the restricted ranges of values that variables `tasks` and `pri` may obtain. We use `S` to denote the restricted set of system services. Therefore, the bounds with `TASK = {T1,T2}`, `PRI = {1,2}`, and `S` including `ActivateTask`, and `TerminateTask` are applied to the Event-B specification for checking the behaviors under consideration.

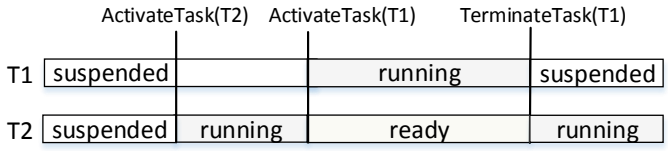


Figure 4.9: Scenario representing P1

Figure 4.10 visualizes a scenario which represents desirable behaviors for terminated or chained tasks. This scenario describes the behaviors that satisfy property (P2). Initially, two tasks are both in the suspended state. Next, task `T2` transfers to running state after `ActivateTask(T2)` and task `T1` is still suspended. Then, `ChainTask(T2,T1)` is applied; this terminates `T2` and activates `T1`. `T1` quickly transfers to running. Finally, `T1` terminates by itself. By counting objects appearing in the scenario, we see 2 tasks, 1 value for the task priority, and 3 enabled events including `ActivateTask(T2)`, `ChainTask(T2,T1)`, and `TerminateTask (T1)` used to describe such behaviors. The least configuration of the OS to check such behavior includes 2 tasks and 1 value for the task priority. Therefore, the bounds with `TASK = {T1,T2}`, `PRI = {1}`, and `S` including `ActivateTask`, `ChainTask` and `TerminateTask` are applied to the Event-B specification for checking the behaviors under consideration.

Figure 4.11 describes desirable behaviors in which the state transitions of two tasks, `T1` and `T2` with the same priority, caused by `SetEvent(T2,T1)`. This scenario describes the behaviors that satisfy property (P3). Initially, two tasks are all in the suspended

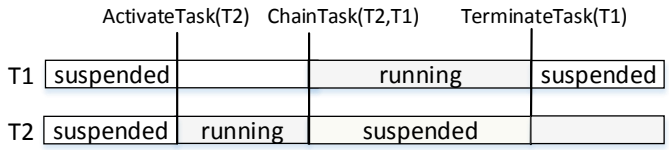


Figure 4.10: Scenario representing P2

state. Next, task `T1` transfers to running state after `ActivateTask(T1)` and task `T2` is still suspended. Then, `T1` activates `T2`. It is expected that `T2` transfers to ready state. After that, since `T1` waits for an event, `T1` transfers to waiting state and `T2` enters running state. Finally, `T2` set an event for `T1`, it is expected that `T1` enters ready state and `T2` still keeps running. The least configuration of the OS model to check such behavior includes 2 tasks and one value for the task priorities. Therefore, the bounds with `TASK = {T1,T2}`, `PRI = {1}`, and `S` including `ActivateTask`, `WaitEvent`, and `SetEvent` are applied to the Event-B specification for checking the behaviors under consideration.

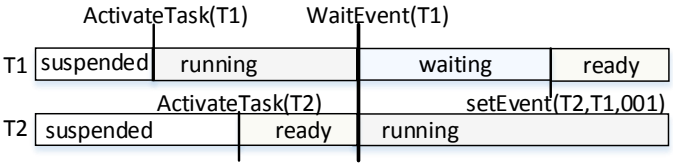


Figure 4.11: Scenario representing P3

Figure 4.12 describes desirable behaviors that satisfy properties (P4, P5). Task T1 is running and requests a resource shared with the interrupt service routine INR1. The priority of task T1 is raised to the ceiling priority of the resource. Task T1 activates the higher prior task T2. The ceiling priority of the resource is higher than the priority of T2; task T1 is still running. Interrupt INR1 occurs. Because of the ceiling priority of the resource, task T1 is still running, the interrupt INR1 is pending. Interrupt INR2 occurs. The priority of INR2 is higher than the ceiling priority of the resource of T1; INR2 interrupts task T1 and it is executed. After INR2 is done the task T1 is continued. The task T1 releases the resource. The interrupt service routine INR1 is executed, the task T1 is interrupted. After INR1 is done the T2 is running. After termination of task T2 the task T1 is continued. The least configuration of the OS model to check such behavior includes 2 tasks and 5 values for the priorities. Therefore, the bounds with `TASK = {T1,T2}`, `PRI = {1,2,3,4,5}`, and `S` including `ActivateTask`, `TerminateTask`, `GetResource`, `ReleaseResource`, `SETINR` and `RESETINR` are applied to the Event-B specification for checking the behaviors under consideration.

Property 6 is checked to make sure that the index value is within the bounds of the array. For example, as defined in the OS design, the bound of array `ready` is established by 72. To check whether the index value exceeds this bound. We can use 4 tasks, 2 multiple activation requests, and 10 values for the task priority. However, if we restrict only V with 4 tasks, 2 multiple activation requests, and 10 values for the task priority, the verification could run out of memory, while we only need to call functions including declaration of tasks and activation of tasks to check such property. Therefore, an appropriate bound is to restrict both V and E where V is restricted as presented above and E is defined as a restricted set containing `DeclareTask` and `ActivateTask`.

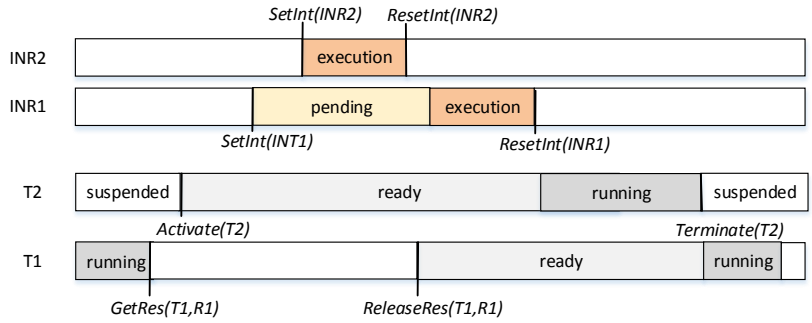


Figure 4.12: Scenario representing P4, P5

Figure 4.13 describes desirable behaviors that satisfy property (P7). Initially, two tasks are all in the suspended state. Next, task `T1` transfers to running state after `ActivateTask(T1)` and task `T2` is still suspended. Then, `T1` gets a resource. After that, `ChainTask(T1,T2)` is called. It is expected that an error is returned because this is an irregular behavior when `ChainTask(T1,T2)` is applied for the task occupying a resource. The least configuration of the OS model to check such behavior includes 2 tasks and one value for the task priorities. Therefore, the bounds with `TASK = {T1,T2}`, `PRI = {1}`, and `S` including `ActivateTask`, `GetResource`, and `ChainTask` are applied to the Event-B specification for checking the behaviors under consideration.

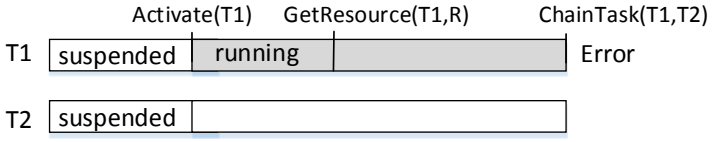


Figure 4.13: Scenario representing P7

Over activation of a task is also an irregular behavior of the OS. We expect that when an over activation happens, an error is returned. The least configuration for verification of property (P8) includes task `T1`, one value for the priority, and only `ActivateTask`. Table 4.4 shows bounds for checking the focused properties. Here, columns "Prp." lists the properties. As shown in Figure 4.5, variables `tasks`, `res`, `evt`, and `inr` define entities managed by the OS such as tasks, resources, events, and interrupt routines; variables `tpri`, `rpri`, and `ipri` define the priorities assigned to tasks, resources, and interrupt routines; and variable `tstate` defines the task state. Because of the space limitation, we show in column "$\widehat{D_S}$" restricted ranges of values for `tasks`, `tpri`, `res`, `rpri`, `evt`, `inr`, and `ipri` respectively. In column "$\widehat{\Sigma_S}$", we present the restricted set of OS services required for checking the corresponding properties, where DT, DR, DI, AT, CT, TT, GR, RR, WE, SE, SI, and RI stand for `DeclareTask`, `DeclareResource`, `DeclareISR`, `ActivateTask`, `ChainTask`, `TerminateTask`, `GetResource`, `ReleaseResource`, `WaitEvent`, `SetEvent`, `SetINTR`, and `ResetINTR`, respectively. Column "d" presents the maximum depth of the execution sequences from the Event-B specification. "-" indicates that no restriction is applied to the range.

As shown in Table 4.4, checking the different behaviors of the OS requires to use different bounds; therefore, when we extend ranges for checking specific properties, we need to perform the boundary check.

Table 4.4: Estimated Bounds

Prp.	$\widehat{D_S}$ for `tasks`, `tpri`, `res`, `rpri`, `evt`, `inr`, `ipri`	$\widehat{\Sigma_S}$	d
P1	{`T1,T2`}, {`1,2`}, {}, {}, {}, {},{}	DT,AT,TT	-
P2	{`T1,T2`}, {`1`}, {}, {}, {}, {},{}	DT,AT,CT,TT	-
P3	{`T1,T2`}, {`1`}, {},{}, {`Evt1`}, {}, {}	DT,AT,WE,SE	-
P4,5	{`T1,T2`}, {`1,3`}, {`Res1`},{`6`}, {}, {`Inr1,Inr2`}, {`4,7`}	DT,DR,DI,AT, GR,RR,SI,RI	-
P6	{`T1,T2,T3,T4`}, {`10`}, {}, {}, {}, {},{}	DT,AT	-
P7	{`T1,T2`}, {`1`}, {`Res1`}, {`2`},{}, {},{}	DT,DR,DI,AT, GR,RR,CT,TT	-
P8	{`T1`}, {`1`}, {}, {}, {},{},{}	DT,AT	-

4.6 Environment and Assertions

4.6.1 Variations of Environment

For the case of the OS, the scheduler decides on the basis of the task priority which is the next of the ready tasks to be transferred into the running state. Additionally, within the same ranges of values for the priorities, there are several patterns of assignment of the priorities for tasks, interrupt routines, and the ceiling priorities for resources. For example, if there are two tasks and the value domain for the task priorities is defined as [1..2], there are 4 patterns to assign the priority for the tasks. They are (1,1), (1,2), (2,1), and (2,2). The OS behaves differently with different patterns of assignment of the priorities for tasks, interruption routines, and the ceiling priorities for resources. Values for static priority of tasks and interruption routines, and for ceiling priority of resources must be assigned in the initialization to set the configuration for the operating system. We describe the configuration data in the environment. Each configuration data reflects an applied pattern; therefore, it produces a variation of the environment. To investigate all possible behaviors of the scheduler, we use all patterns to assign the priorities for tasks, interrupt routines, and the ceiling priorities for resources. Consequently, there are several variations of the environment for the operating system.

4.6.2 Generating Environments and Assertions

The environment of OSEK OS and assertions are generated from the specification following the approach presented in Chapter 3. In previous step of the workflow, we already generated the LTS of the specification. In this step, we generate the environment and the assertions and translated them into Promela. Figure 4.14(left) demonstrates an LTS of the specification of OSEK OS and Figure 4.14(right) demonstrates a combination model between the design and the environment with the assertions to be inserted.

To generate the environment, we use a mapping from the events in the LTS to the function calls in the environment. For example, event `AT(t1)` in the LTS is mapped to function call `_ActivateTask(task1.tid)` in the environment; also, event `TT(t1)` is mapped to function call `_TerminateTask(task1.tid)`. The states and transitions in the LTS are represented by labels and if-statements in the environment. To generate the assertions, we

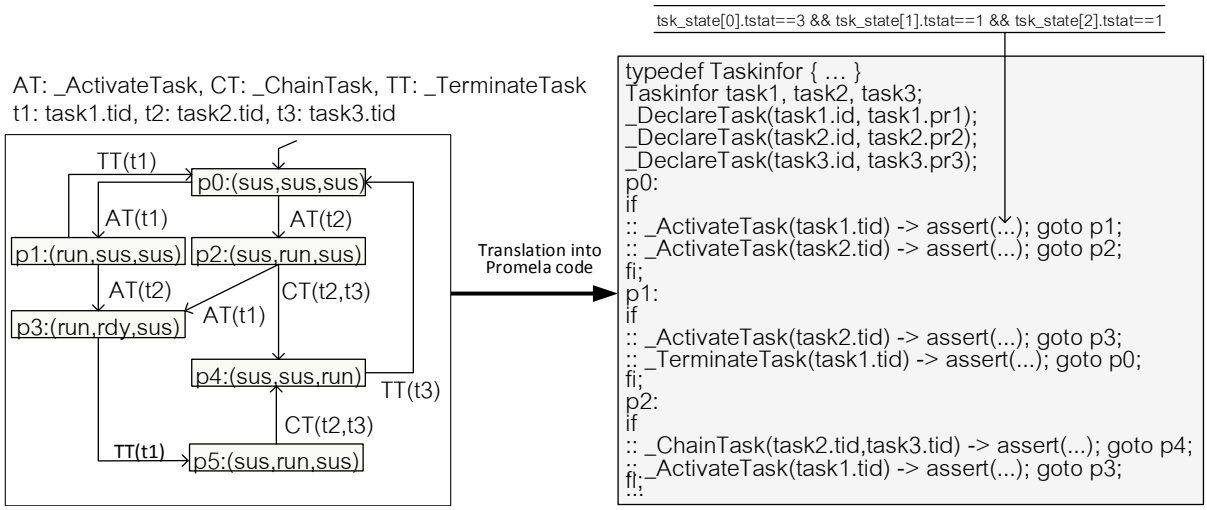


Figure 4.14: Generation of environments and assertions from LTS

use the mappings R and C from the variables, the values in the specification to those in the design. For example, from state $p1$ of the specification where $tstate(a) = run$, $tstate(b) = sus$, $tstate(c) = sus$ and with mappings $R(tstate(a)) = tsk\_state[0].tstat$, $R(tstate(b)) = tsk\_state[1].tstat$, $R(tstate(c)) = tsk\_state[2].tstat$, $C(sus) = 1$, $C(rdy) = 2$, $C(run) = 3$ the generator outputs an assertion $tsk\_state[0].tstat == 3\&\&tsk\_state[1].tstat == 1\&\&tsk\_state[2].tstat == 1$ for the corresponding state of the design.

4.7 Verification results

Our target system is an operating system compliant with OSEK/VDX standard. We focus on the verification of properties concerned with the correctness of scheduling. In section 4.5, we decided the proper bounds for the properties to be checked. This section presents experimental results. We use inputs including the Event-B specification, the bounds, and the OS design described in Promela and mappings between elements of the specification and those of the design.

All experiments were conducted on an Intel(R) Core(TM) i7 Processor at 2.67GHz running Linux. We performed experiments in two phases. In phase (I), we checked the OS design. No errors were returned in all cases of experiments. This result was considered due to the fact that the OS design had already been reviewed carefully by many researchers and engineers. Still, from this result, we can be sure that the OS design is correct with respect to its specification within input bounds. However, such successful experiment results do not show the effectiveness of the approach. We evaluate the effectiveness of the approach based on its bug-detecting ability. To show the bug-detecting ability of our approach, in phase (II), we intentionally added several bugs into the OS design and performed an experiment to check the modified OS design. Our purpose here is to make sure our approach can actually detect the bugs we added. In this paper, we focus on two kinds of typical bugs of the OS design: (i) the bugs that cause the condition enabling the OS services not to conform to the specification; and (ii) the bugs that cause the computational effects provided by the OS services to violate the specification. In practical environments, such kinds of bugs could be easily added into the design.

Table 4.5: Experiment Outputs

No.	Prp.	Bounded Ranges						LTS Generation			Model Checking		
		$\widehat{D_S}$				$\widehat{\Sigma_S}$	d	#State.	#Trans	Time(s)	Mem(Mb)	Time(s)	Result
		t,tp	r,rp	i,ip	e								
I.1	P1	2,2	0,0	0,0	0	3	-	4	10	1.0	129.2	3.5	√
I.2-1	P2	2,1	0,0	0,0	0	4	-	4	10	1.0	129.2	3.5	√
I.2-2	P2	9,3	0,0	0,0	0	4	-	512	13824	3.5	430.0	362.3	√
I.2-3	P2	5,5	0,0	2,2	0	6	-	128	1536	2.2	133.5	17.6	√
I.3	P3	2,1	0,0	0,0	1	4	-	10	27	1.0	129.2	4.2	√
I.4	P4	2,2	1,1	2,2	0	8	-	80	520	1.3	129.2	8.3	√
I.5-1	P5	2,2	1,1	2,2	0	8	-	80	520	1.3	129.2	8.3	√
I.5-2	P5	3,3	1,1	2,2	1	12	-	152	1036	2.0	132.5	14.8	√
I.5-3	P5	7,3	1,1	2,2	1	-	6	374	7063	2.5	227.6	105.7	√
I.5-4	P5	7,3	1,1	2,2	1	-	7	731	14046	3.6	285.4	360.0	√
I.5-5	P5	10,3	1,1	2,2	1	-	4	335	9476	10.0	413.8	75.4	√
I.6	P6	4,10	0,0	0,0	0	3	-	102	1220	2.1	132.0	15.0	√
I.7	P7	2,1	1,1	0,0	0	7	-	8	22	1.0	130.0	7.2	√
I.8	P8	1,1	0,0	0,0	0	2	-	2	2	1.0	129.2	3.5	√
II.1	P1	2,2	0,0	0,0	0	3	-	4	10	1.0	129.2	3.5	√
II.2	P2	2,1	0,0	0,0	0	4	-	4	10	1.0	129.2	3.5	√
II.3	P3	2,1	0,0	0,0	1	4	-	10	27	1.0	129.2	4.2	Fail
II.4	P4	2,2	1,1	2,2	0	8	-	80	520	1.3	129.2	6.0	Fail
II.5	P5	2,2	1,1	2,2	0	8	-	80	520	1.3	129.2	6.0	Fail
II.6	P6	4,10	0,0	0,0	0	3	-	102	1220	2.1	132.0	15.0	√
II.7	P7	2,1	1,1	0,0	0	7	-	8	22	1.0	129.2.0	5.2	Fail
II.8	P8	1,1	0,0	0,0	0	2	-	1	1	1.0	129.2	3.5	√

We present the results of experiments in Table 4.5, which are outputted by Spin. Here, the first column ("No.") represents experiment number. Cases I.1-I.8 belong to phase (I) and cases II.1-II.8 belong to phase (II). Column "Prp." refers to the properties of the OS. Column "Bounded Ranges" represents bounds used in distinct experiments as explained above. "t", "tp", "r", "rp", "i", "ip", and "e" present the size of ranges for `tasks`, `tpri`, `res`, `rpri`, `inr`, `ipri`, and `evt`, respectively. In column "E", we show the size of the restricted set of system services that are required for checking the corresponding properties. Column "D" presents the maximum depth of execution sequences from the Event-B specification. "-" indicates that no restriction is applied to the range. Column "LTS Generation" shows statistic of the execution sequence generator. Here, columns "#State", and "#Trans" present the number of distinct states and that of transitions appearing in the execution sequences; column "Time" presents the time taken (s) for the generation. Column "Model Checking" presents statistic of the model checker including total actual memory usage for distinct verification, the time taken (s), and the verification result in which $\surd$ indicates the successful result - no error returned, and "Fail" indicates the fail result - unsatisfied property.

Firstly, we use the least ranges to check the desirable behaviors for the enumerated properties. The verification outputs for such cases are presented in rows I.1, I.2-1, I.3, I.4, I.5-1, I.6, I.7, and I.8. Then, we extend ranges gradually so that the verification covers many more behaviors than those focused in previous steps as long as the machine capacity allows this. For example, rows I.2-2 and I.2-3 show the verification outputs with extended ranges in comparison with cases in rows I.2-1; and, row I.5-2 presents the case that cover many more behaviors than case in row I.5-1.

In the cases of I.5-3, I.5-4, and I.5-5, we want to check the interaction of multiple system services; therefore, we do not restrict the system services in these experiments, i.e. E is presented as "-" in the table. However, within the given V, the LTS of the specification is huge. We restrict the depth of the execution sequences of the specification to reduce the size of LTS. We give values for the maximum depth based on the estimation of the size of LTS that the machine capacity allows. For example, with currently used machines, we estimated that the machine capacity allows round 20000 transitions appearing in LTS. Firstly, we generate LTS with the given ranges for V and no limitation for the depth of the execution. Then, if LTS is huge, we try some values for the depth. For example, if we use 7 tasks, 1 resource, 2 interrupt routines, 1 event, and depth = 7, then "#Trans" = 14046, the verification succeeds because the size of LTS is less than that the machine capacity allows. However, when we use the same ranges for V and depth = 8, "#Trans" = 32599, the size of LTS is significantly large compared with the used machine capacity. This could easily cause the state explosion. Therefore, we set the depth to 7 in our experiments as shown in the case of I.5-4.

From the experiment results, we can see that the time taken and the total actual memory usage for the generation of the execution sequences from Event-B specification and the verification of the simulation relation are reasonable. For the model checking result of phase (I) shown in the table, no errors were returned in all cases of experiments. However, in phase (II), we added bugs into the OS design. Consequently, some errors are reported in the model checking results of the modified OS design. From the experiment results, we can see that bugs are detected in short time and with reasonable total actual memory usage for the model checking.

We added a bug to the condition expression for waking up the task waiting for an event

in function `SetEvent`. This bug is detected in case II.3. A counter-example is shown by Spin against (P3): `ActivateTask(t1)`; `ActivateTask(t1,t2)`;
`WaitEvent(t2, evt1)`; `GetResource(t1,r1)`;
`ActivateTask(t1,t3)`; `SetEvent(t3, t2, evt1)`:
text of failed assertion: $assert((tsk\_state[1].tstat == 2))$. Task `t2` in the waiting state is not released to the ready state, even though `evt1` has been set for `t2` by tasks `t3`. This violates the specification. Because the condition expression for waking up the task waiting for an event in the OS design includes bugs, `t2` does not satisfy this condition. Thus, `t2` is not waked up.

We added another bug in the computational statement of function `GetResource`. This bug makes the dynamic priority of tasks not change for any case when the tasks get the resources. This bug is detected in case II.4. A counter-example is shown by Spin against (P4): `ActivateTask(t1)`; `ActivateTask(t1,t2)`; `WaitEvent(t2,evt1)`;
`GetResource(t1,r1)`: text of failed assertion:
$assert(tsk\_state[0].dpriority == 6)$. The priority of task `t1` is not raised to the ceiling priority of resource `r1`, even though `t1` has got `r1` successfully. This is because the ceiling priority of resource `r1` is not assigned to the dynamic priority of task `t1` even though the static priority of `t1` is lower than the ceiling priority. This violates the specification.

Experiment results above show the ability of our approach to detect the typical bugs of the OS design such as the bugs in the guard conditions enabling the computations and those in computational statements of functions. Such bugs cause undesirable design behaviors of the OS. With the exhaustive verification within the bounds, our approach provides rapid bug detection of design behavior.

4.8 Additional experiments

Section 4.7 presented the statistics of model checking for the properties shown in Table 4.3. In fact, we additionally conducted several experiments to verify various behavioral aspects of the OSEK/VDX OS. Firstly, the OS scheduler decides on the basis of the task priority which is the next of the ready tasks to be transferred into the running state. As we mentioned, within the same ranges of values for the priorities, there are several patterns of assignment of the priorities for tasks, interrupt routines, and the ceiling priorities for resources. For example, if there are two tasks and the value domain for the task priorities is defined as [1..2], there are 4 patterns to assign the priority for the tasks. They are (1,1), (1,2), (2,1), and (2,2). The OS behaves differently with different patterns of assignment of the priorities for tasks, interruption routines, and the ceiling priorities for resources. To show this difference, we present experimental results of typical cases in cases No.1-14 of Table 4.5. These cases are to verify the task management independently with the other functionalies of the OS. We define the bounds to restrict the number of tasks and the range of values for the task priority. Our purpose is to show that our verification returns different statistics of model checking when we conduct two typical cases: the tasks have the same priority, e.g, case No.8, and the tasks have different priority, e.g, case No.9, with the same number of tasks, e.g., five tasks. This is because sequences of functions triggered in the former case is different from that in the latter case.

Secondly, scalability depends on the machine power. We present statistics of LTS Generation and model checking in cases No.15-20, cases No.21-27 to demonstrate the

scalability. In cases No.15-20, we checked the task management independently with the other functionalies of the OS. The remaining cases verify the combination of functionalities of the OS.

In the table, two cases of our verification, No.18 and No.21, have not completed due to out of memory condition. When we extend the bounds, the size of LTS becomes larger. The total number of invocations increases according to the total number of enabled events appearing the LTS, which is indicated in column "#Trans". Using our machine (memory capacity: 8 gigabytes), Spin can use around 430 megabytes for total memory usage, in which around 230 megabytes is used to store states. We could estimate the total memory required to store states based on the size of LTS as follows:

- Total number of invocations: $|\Sigma_{D \cdot E}| = |\widehat{\Sigma_S}|$, this is because the mapping from enabled events to invocations is one-to-one mapping
- Total number of states: $|\widehat{Q_{D \cdot E}}| = |\Sigma_{D \cdot E}| * c1$, where $c1$ is total number of reachable states returned after applying each invocation.
- Total memory usage to store distinct states: $Mem = |\widehat{Q_{D \cdot E}}| * c2$, where $c2$ is the size of state vector.

We estimated the total memory required to store states in cases No.18 and No.21 as shown in Table 4.7.

Estimated data indicates that the total memory required to store states in cases No.18 and No.21 is over the total memory Spin can use. Therefore, the verification has not completed in these cases. However, we could restrict the depth of execution of specification as shown in cases No.19,20 to make the verification succeed when we still use the same ranges as No.18. Also, we could check the combination of all functionalities and avoid the state explosion by use the reasonable configurations as shown in cases No.22,23. In addition, if we could not check the combination of all functionalities, we at least can check critical cases as follows:

- The combination between the task management and the interrupt handling (case No.24)
- The combination between the task management and the resource management (case No.25)
- The combination between the task management, the resource management, and the event management (case No.26)
- The combination between the task management, the resource management, and the interrupt handling (case No.27).

In cases No.24-27, we used the restricted sets of events to check important groups of functionalities and avoid the state explosion. In the table, column "gf" presents the size of the restricted sets of events to be checked.

Table 4.6: Additional Experiment Outputs

No.	Bounded Ranges							LTS Generation			Model Checking		
	$\widehat{D_S}$					gf	d	#State.	#Trans	Time(s)	Mem(Mb)	Time(s)	Result
	tasks	pri	res	evt	inr								
1	1	1	0	0	0	-	-	2	2	1	128	3	√
2	2	1	0	0	0	-	-	4	10	1	128	3	√
3	2	2	0	0	0	-	-	4	10	1	128	3	√
4	3	1	0	0	0	-	-	8	36	1	128	3	√
5	3	3	0	0	0	-	-	8	36	1	128	3	√
6	4	1	0	0	0	-	-	16	112	1	129	4	√
7	4	4	0	0	0	-	-	16	112	1	129	4	√
8	5	1	0	0	0	-	-	32	320	1	133	5	√
9	5	5	0	0	0	-	-	32	320	1	130	4	√
10	6	1	0	0	0	-	-	64	864	1	164	10	√
11	6	6	0	0	0	-	-	64	864	1	132	10	√
12	7	1	0	0	0	-	-	128	2240	1	380	26	√
13	7	7	0	0	0	-	-	128	2240	1	324	26	√
14	8	8	0	0	0	-	-	256	5632	2	382	99	√
15	9	3	0	0	0	-	-	512	13824	3	430	362	√
16	9	3	0	0	0	-	4	130	3258	3	326	262	√
17	9	3	0	0	0	-	5	265	6912	3	364	282	√
18	10	3	0	0	0	-	-	1024	92160	22			*not completed*
19	10	3	0	0	0	-	5	385	10824	52	426	390	√
20	10	3	0	0	0	-	6	385	22430	85	486	590	√
21	8	6	1	1	2	-	-	1052	93600	26			*not completed*
22	3	6	1	1	2	-	-	152	1036	2	132	14	√
23	6	6	1	1	2	-	-	1216	18368	35	420	390	√
24	5	7	0	0	2	5	-	128	1536	2	133	17	√
25	9	1	2	0	0	5	-	1024	25600	25	450	510	√
26	8	1	2	1	0	8	-	1216	25639	28	450	520	√
27	7	6	1	0	2	7	-	1022	18926	15	375	310	√

Table 4.7: Estimation of the total memory required to store states

No.	$\lvert\Sigma_{D \cdot E}\rvert$	$c1$	$\lvert\widehat{Q_{D \cdot E}}$	$c2(Mb)$	$Mem(Mb)$
18	92160	10	921600	0.0008	737,28
21	93600	10	936000	0.0008	748,880

4.9 Summary

In this chapter, we presented a case study in which we applied the proposed framework to verify a real system - an operating system design compliant with the OSEK/VDX standard. To apply the framework, the users need to provide inputs including a formal specification of the OS in Event-B, the bounds, the mappings, and the design in Promela. There already exists an informal specification of the OS - the OSEK/VDX OS specification. This is an international standard for the automotive OS. We cannot change it but faithfully formalize it in Event-B. We presented the guidelines to obtain a likely equivalent formal specification with the informal one. The mappings relates elements in the Event-B specification to those in the Promela design. They could be described in the tables. The design is described as a collection of inline functions in Promela. As for simple systems, e.g., the target systems used in the case studies chapter 3, the bounds could be easily decided. As for the OSEK/VDX operating system, which has multiple groups of functionalities, the bounds need to be defined depending on the properties to be checked. In this chapter, we presented the guidelines to determine the proper bounds for the verification of the properties to be checked. By using the proper bounds, we could verify various aspects of the OS including (i) individual functionality and combination of functionalities, (ii) full preemptive scheduling with different priorities for the tasks and the interruptions, and (iii) regular and irregular behaviors of the OS. We presented the results of several experiments to show that the framework is able to verify the important properties and to detect the typical bugs of the OS.

Chapter 5

Discussion

Assumptions

Our framework is to check the conformance between the design and the specification of the reactive systems based on the simulation relation between them. In particular, we check whether the design simulates the specification. In the framework, we assume that (a.1) the functions in the design are deterministic even though the specification may describe non-deterministic behaviors. This is realistic because deterministic behaviors are required for many systems like operating systems. In addition, we assume that (a.2) the environment invokes functions in the design in a non-deterministic manner in each state. This is also realistic because the environment could trigger any function of several applicable functions in a certain moment. For example, when the customers of a vending machine have inserted money in excess of some item cost, the customers can select the item to be bought or insert more money. In the framework, the LTS of the specification represents events that may be applicable in each state. The environment is generated from the LTS; it triggers functions of the design that correspond to the enabled events. Therefore, we could check that for each event of the specification is enabled, a corresponding function of the design is triggered. For example, when event `AT(t1)` in the specification is enabled then the corresponding function in the design, i.e. `ActivateTask(t1)`, is also triggered. With the assumptions, when a function in the design is invoked, there is a unique successor state with respect to the specification. In this way, we could check whether the effect of each function interpreted in the reachable state of the design is the same as that of corresponding function in the specification via the mappings. This shows that the design follows the specification. Thus, with the two assumptions, checking whether the design simulates the specification is appropriate to show the conformance between the specification and the design.

Simulation relation

The simulation relation verified in our framework is similar to the refinement of [31] and the simulation of [34][43]; however, in our definition, a one-step transition in M_1 may correspond to an n-step transition in M_2. This is appropriate for a simulation from a specification to its design where the latter implements the former.

Considering bi-simulation in [34], a bi-simulation $\mathcal{S}$ between M_1 and M_2 requires that

M_1 simulates M_2 by $\mathcal{S}$ and M_2 simulates M_1 by the reverse of $\mathcal{S}$. In this framework, we check one direction of simulation, that is checking whether the design simulates the specification. As analyzed above, checking whether the design simulates the specification is appropriate to show whether the design follows the specification. Accordingly, two kinds of typical bugs of the design could be quickly detected if they exist in the design. They are bugs included in conditional expressions under which functions are triggered, and bugs included in computation of functions. Such kinds of bugs could be easily added into the design of practical systems. For example, a bug was intentionally included in the condition expression for waking up the task waiting for an event in function `SetEvent` of the OSEK OS design. Even though the function is still called according to the specification, it does nothing because the conditional expression is not true. This bug was detected in case II.3 of table 4.5. Again, some bugs can be easily included in the computational statement of functions in the design. For example, a bug in the computational statement of function `GetResource` makes the dynamic priority of tasks not change for any case when the tasks get the resources. However, according to the specification, the ceiling priority of resource must be assigned to the dynamic priority of the task if the priority of the task is lower than the ceiling priority. In this situation, the effect of this function in the design is not as same as that of corresponding function in the specification via mappings. This bug was detected in case II.4 of table 4.5. Detecting bugs in the design is an important objective of our framework.

One may say that the design can do something more than the specification. We consider that this may happen in two situations: (i) the design contains non-deterministic behaviors that are out of the specification; and (ii) the design contains redundant functions which are not described in the specification. Considering another direction of simulation relation, that is verifying whether the specification simulates the design. Checking whether the specification simulates the design may be appropriate to prevent the design from doing something more than the specification. Nevertheless, situation (i) is out of our scope due to assumption (a.1); situation (ii) may happen because the designers may add functions out of the specification by mistake. In our framework, the redundant functions, if existing in the design, are never triggered since the environment just calls functions driven by the specification. Since they are never triggered in our framework, they could not be checked. This is a limitation of the framework where only one direction of simulation relation is taken into account.

Verifying whether the specification simulates the design may be appropriate to show there is no extra behavior added into the design with respect to the specification. We planed to consider this in the future. On the other hand, the opposite direction, checking whether the design simulates the specification, is sufficient to show every behavior in the specification are actually followed by the design; it is also sufficient to detect typical bugs if they exist in the design.

Bounds

We introduce a formal definition of the bounds for verifying the simulation relation of the design and its formal specification. These bounds can be applied generally to any design and its formal specification as long as the formal models of the inputs are defined as LTSs. In section 3.5, we present the interpretation of the bounds in a concrete model, i.e.,

Event-B model. Data elements including variables, constants and parameters in Event-B model can obtain values in infinite domains such as NAT or INT. Therefore, an LTS of Event-B model may be infinite LTS in which the state space and the set of actions may be infinite sets. Applying the model checking technique requires a finite representation of the target system. Firstly, the bounds are defined to restrict the range of every data element including variables, constants and parameters. We can obtain a finite LTS associated to the Event-B specification within the bounds. Basically, the restriction of every data element produces a finite representation of the target system; this makes possible to apply the model checking technique. This is shown in some case studies such as vending machines, elevators, controlling cars on a bridge. On the other hand, as for complex systems like OSEK/VDX operating systems, which include several data elements and several functionalities, the size of the LTS may be so large that it could easily cause the state explosion when we apply model checking, event though the LTS is finite. To avoid the state explosion, our idea is to lead the verification to focus on partial behaviors of target systems. We additionally define restriction of service functions of the target system and the depth of the execution as well. With these restrictions, we can check each functionality of OSEK/VDX OS independently from the other functionalities, e.g., the task management is checked independently from the resource management, the event management, and the interrupt handling. We also can check each small groups of functionalities instead of all at one, e.g., checking the combination of task management, resource management and event management. Each of these groups represents some essential behaviors of the target system. We could distribute partial behaviors in variations of the environment. In our idea, partial behaviors are decided according to the properties and the bugs of the target systems to be checked. For example, one important property of the OS is that "An extended task in the waiting state must be released to the ready state if at least one event for which the task is waiting has occurred". In order to check this property, we need use two tasks, one event, and three service function including `ActivateTask`, `WaitEvent`, and `SetEvent`. We found that one could avoid the state explosion if we use reasonable ranges for data elements and service functions. Even though we cannot show the conformance of the design and the specification in the infinite scope but if an error is returned within the bounds, we can show it really exists in the design.

Mappings

In the framework, we use three kinds of mappings to relate elements in the Event-B specification to those in the Promela Design: Variables $\rightarrow$ Variables, Values $\rightarrow$ Values, and Events $\rightarrow$ Function calls.

In Event-B, variables have either basic types, set types, or types defined by relations and functions. In Promela, variables have either basic types or complex data structure types like arrays and record types. We consider possible mappings for variables in Event-B and Promela as follows:

- Mappings relate variables having basic types in Event-B to those in Promela, e.g., `cred` $\mapsto$ `credit`
- Mappings relate variables having set types in Event-B to arrays in Promela, e.g., `stock` $\mapsto$ `stock`

- Mappings relate variables, which are elements of a set in Event-B ($stock = \{a, b, c\}$), to variables, which are elements of an array in Promela ($byte\ stock[3]$), e.g., `a` $\mapsto$ `stock[0]`

- Mappings relate variables defined by relations ($tstate \in TASK \times STATE$) or functions ($tstate \in TASK \rightarrow STATE$) in Event-B to variables being elements of a record type in Promela ($typedef\ TCB\{byte\ tstat; ...\};\ TCB\ tsk\_state[N\_TASK];$): e.g., `tstate(a)`$\mapsto$`tsk_state[0].tstat`

We found that we dealt with all mappings for variables as mentioned above in the case studies; and, it is not difficult to define those mappings. In the case studies, we just enumerate all variables appearing in the specification and the design and then relate them to each other.

There are several variations to define mappings from values in the Event-B specification to those in the Promela design even though every range of values has been bounded. For example, we consider the range of values for `tasks`. We define a carrier set namely `TASK` as a finite domain for `task`. `TASK` consists of identifiers of tasks defined in Event-B, e.g. `TASK = {a,b,c}`. In the Promela design, identifiers of tasks may obtain values in range `[0..255]` of *byte*. Therefore, `a` may be mapped to any value in `[0..255]`. Similarly, any value in `[0..255]` is possible for `b` and `c` as long as values mapped to `a`, `b`, and `c` are not identical. Consequently, there are $255 * 254 * 253$ variations to define mappings for values in `TASK` in Event-B to `[0..255]` in Promela. A question may arise here, i.e., whether we need to take all variations into account in the verification. Firstly, we consider variations for the task priority, as explained in section 4.8, if there are two tasks and the value domain for the task priorities is defined as [1..2], there are 4 variations to assign the priority for the tasks. They are (1,1), (1,2), (2,1), and (2,2). The OS behaves differently with different variations of assignment of the priorities for tasks, interruption routines, and the ceiling priorities for resources. To make sure all possible behaviors to be checked, all variations should be taken into account in the verification. Then we consider variations for the task identifier. We assume that the behaviors of target system are independent from what variation is used for the task identifier in the verification. Therefore, we can choose one of variations to use in the verification. Thus, we consider that if the behaviors of the target system depend on what variation is used, all variations should be used so that the verification could cover all possible behaviors of the target system. Otherwise, we can choose one of variations to use in the verification. However, if the range of values is large, e.g., hundreds of elements, it takes the users significant time to enumerate all variations. Regarding to mappings for values, another question here is that in an actual implementation of vending machines, the customers can press two buttons, left and right, to select the same drink; how to define mappings for values in the specification and the design in this example. We suppose that `b` is a value in the specification that represents a drink to be selected by the customers; and, `left` and `right` are two values in the design that represent the same drink to be selected. We need to relate `b` to `left` and `right`. Generally, we may define one-to-one and many-to-one mappings. For this example, we can define two one-to-one mappings: one mapping is relating `b` to `left`; and another is relating `b` to `right`.

To relate events in the Event-B specification to function calls, we enumerate all enabled events appearing in the LTS generated from the specification; we also enumerate all function calls which can trigger functions in the design. Then we define mappings to

relate the events to the corresponding function calls. Mappings may be one-to-one when one event in the specification is realized by one function in the design like mappings defined in case studies of vending machines (e.g., `restock(a)`↦`restock(M1)`), operating systems (e.g., `GR(b,res1)`↦`_GetResource(task2._tid,res1._rid)`); or, many-to-one when multiple events in the specification are realized by a function in the design. Many-to-one mapping is defined when the specification include non-deterministic actions. We regard an event with a non-deterministic action as multiple events with deterministic actions. For example, event e in Event-B with a non-deterministic action has the general form 'e : any u where $g(x, u)$ then $x' :\in [a1(u), a2(u)]$'. We suppose that event e is realized by function F in the design. We regard e as two events with deterministic actions: '$e1$: any u where $g(x, u)$ then $x' := a1(u)$' and '$e2$: any u where $g(x, u)$ then $x' := a2(u)$'. It seems that two events $e1$ and $e2$ in the specification are realized by one function F in the design. Therefore, $f(e1) = F$ and $f(e2) = F$. We consider that it is not difficult to define this kind of mappings.

Coverage

The coverage of this verification is evaluated by how much of the specification is satisfied by the design. In our experiments, the design is checked against the LTS which are generated from the specification within input bounds. There are two viewpoints to evaluate how much of specification is represented in the LTS. They are structure and behavior. Structure refers to a set of entities concerned with the configuration. Behavior refers to system services. Therefore, we divided the coverage criteria into two types: structure coverage means how large of the configuration is used in the verification; and behavior coverage refers to not only functions called but also the order of function calls. For structure coverage, we determine the bounds of the execution sequences based on the properties of interest. Specifically, we define the bounds at least as large as to cover the configuration appearing in the scenario corresponding to the property. For these bounds, we were able to check important properties of the OS within a reasonable time and memory space. To get more reliability in the verification, we need to extend the bounds as large as possible depending on the machine capacity.

For behavior coverage, we have checked each functionality such as task management, resource management, event handling independently of others, including both of regular sequences and irregular sequences. Even though we cannot check all the functionalities at once due to the state explosion we still need to check at least the combination of functionalities such as the combination of resource management and event handling, and combination of event handling and interrupt processing. Checking this combination is important because it is known that bugs often come from the interaction of different functionalities. In order to check multiple functionalities at once while avoiding the state explosion, we need to make the bounds of configuration as small as possible. We consider that it is important to have a good balance between the ranges of structures and behaviors based on the properties to be checked.

It is important to check that the intended scenarios are actually contained in the execution sequences generated from the Event-B specification. This can be easily checked by traversing the execution sequences accordingly with the scenarios. By checking this, we can raise the reliability of the Event-B specification as a generalization of the scenarios.

We can also ensure that the design model is checked at least with these scenarios.

Generation of Environment

The behaviors of the target systems depend on sequences of function calls from their environments. The size of the environment depends on the number of invocations and relations between them. The number of invocations equals to the number of transitions in the LTS generated from the specification. Considering the statistics shown in Tables 3.5, 4.5, and 4.6, the number of transitions (#Trans) may reach thousands or more. For the comprehensive verification, we need to use the environments that cover all possible sequences of invocations, this means, all possible relations between invocations must be included. It is hard to manually describe the environment with hundreds of invocations or more. Accordingly, an advantage of our framework is that it is able to systematically generate all possible sequences of invocations from the execution sequences of the specification in Event-B. This is essential to generate the environments for the comprehensiveness of verification with respect to the specification. Another advantage of the framework is to produce reliable environments that there is no contradiction between sequences of invocations.

Target of Framework

Our framework was applied to verify the designs of practical systems. The framework directly checks the designs against their formal specifications. Although we show the experiments, when our framework is applied to the automotive operating system, the vending machine, controlling cars on a bridge, and elevator controller, it is not limited to these applications. Characteristic of the framework is as follows:

- The simulation relation is defined based on LTSs.
- The states are interpreted as value assignments.
- The design is described as a collection of functions which update the value assignments.
- The environment is described as a collection of invocations.

Reactive systems are event-driven systems; it is convenient to describe each event as a function. Therefore, this style of models is adopted not only for target systems used in our case studies but also other reactive systems. If a system is described according to this style, it can be verified by straightforwardly applying our framework.

In our case studies, Promela is used as a specification language to describe the design and the environment; however, our framework can be applied for the designs described in not only Promela but also other languages as long as they can deal with a collection of functions for the design and sequences of invocations for the environment.

Languages for Specification and Design

In the framework, we use different languages to describe the specification and the design. In particular, Event-B is adopted for the specification and Promela is adopted for the design. From case studies, we found Event-B is indeed convenient to describe the specification. We can easily define abstract data types using carrier sets (e.g., $PRODUCT$ in case of vending machines) and define effect of operations using set operators (e.g., set minus used to describe effect of moving an item from a container). In addition, Event-B provides tools to show the consistency of the specification before using the specification to check the design. Hence, we could check the design against a reliable specification. This would drastically improve the reliability of model checking results because the specification is reliable. We introduced the bounds for the verification. Such bounds could be easily implemented in Event-B model. Also, Promela is convenient to describe the design. It provides implementable data structures, e.g., arrays, record types, and various control structures, e.g., loops, selection. We could straightforwardly describe design decisions by using these control structures based on complex data structures. Due to this convenience, low cost and low effort are required for describing the specification and the design. This also improves the quality of the specification and the design.

Verification of OSEK/VDX OS Design

We applied the framework to verify the design of OSEK OS. We aimed at some essential behaviors of OSEK OS including effect of each function, combination of functions, and scheduling policy. We aimed at both of regular and irregular behaviors of OSEK OS. We described all essential behaviors, that we intended to verify, in Event-B. We found that it is convenient to describe these behaviors in Event-B. We strictly followed the procedure to formalize the behaviors of OSEK OS in Event-B. Therefore, we believe in the quality of the specification of OSEK OS before using it to verify the design. To avoid the state explosion, we used proper bounds so that the verification focused on partial behaviors of OSEK OS. The bounds are determined according to the properties of interest. The mappings were easily defined in this case study. Consequently, essential behaviors of OSEK OS were really verified against intended behaviors appearing in the specification. All bugs, which were intensionally added into the design of OSEK OS, were detected quickly. After verifying the design of OSEK OS by applying our framework, we get high confidence in quality of the OSEK OS design. In addition, we also considered that there may be some kinds of bugs that could not be detected by our framework. For example, boundary check could not be checked for large number of tasks (e.g., hundreds, thousands of tasks). However, this kind of bugs is easy to review. Separately, the correctness of scheduling is an important property of OSEK OS. It is difficult to review; however, it could be checked by the framework.

Practicality of Framework

We performed some case studies to apply the framework to verify reactive systems including vending machine, elevator, a system for controlling cars on a bridge, and OSEK OS.

We consider the scalability based on outputs of our experiments. Using our machine (memory capacity: 8 gigabytes), Spin can use around 430 megabytes for total memory usage, in which around 230 megabytes is used to store states. The outputs of the experiments show that the framework can handle around 25,000 invocations and store 400,000 states. This scalability is reasonable with OSEK OS because we could have a good balance between structure and behavior perspectives of OSEK OS in the verification. OSEK OS provides several groups of functions and its design describes several complex data structures. These easily cause the state explosion; however, we could properly distribute the partial behaviors to be checked in variations of the environment by using bounds decided according to the properties of interest. Thus, we consider that this framework could be applied to verify real systems that are similar to OSEK OS.

In the framework, the bounds are introduced to restrict the behaviors to be checked. Even though the verification focuses on a finite scope, the intended properties could not be missed if we follow the procedure to decide the proper bounds. In particular, the essential behaviors satisfying the intended properties are included in the behavior scenarios, which must be studied to determine the bounds. Then, we could check that these scenarios are actually contained in the execution sequences generated from the Event-B specification by traversing the execution sequences accordingly with the scenarios. This makes sure that the essential behaviors are really verified. Accordingly, the typical bugs, which could be included in the conditional expression and the computation of functions, could be detected by the framework. We consider that the framework can be applied to detect such kinds of bugs in real systems.

In order to apply the framework to verify the practical systems, the users must provide inputs: the formal specifications in Event-B, the bounds, the mappings, and the design in Promela. It is not avoidable to manually produce the first two inputs; however, we presented guidelines to facilitate these tasks. As for the third tasks, based on the case study of a real system, we consider that the users could easily give the mappings and describe them in the form of tables. As for the last tasks, it is possible to generate the design from the specification. Some existing tools support this task; for example, [37] supports translating from Event-B to Promela. Then, we add details of design decisions into the target model. Consequently, it is possible for the users to provide the necessary inputs for applying the framework to check the designs of the reactive systems.

One may have a question that whether this framework is appropriate for every reactive system. In our case studies, the target systems range from simple (e.g., vending machines with a few of service functions, or controlling cars on a bridge with only basic data types) to complex (e.g., OSEK/VDX OS with several groups of service functions and complex data structures). The framework could be straightforwardly applied to verify these systems. However, with regard to the system for controlling cars on a bridge, which contains only basic data types, one can easily describe both the specification and the design in Event-B. In this case, one can check the consistency between the specification and the design in Event-B based the refinement technique provided by Event-B. As for the other systems, which contain complex data structures and highly optimized behaviors, describing the design in Event-B returns a lot of proof obligations, whereas it is convenient to describe the design of such systems in imperative languages like Promela. Therefore, we consider that our framework is appropriate to apply to the reactive systems that contain complex data structures and highly optimized behaviors.

Chapter 6

Related works

Verification of reactive systems

There are some research directions for verification of the reactive systems. We present here typical researches in this topic. [50] proposes an approach for modeling and verification of reactive systems using Rebeca (Reactive Object Language). Rebeca is an actor based language, which is precisely defined with a formal foundation; however, it could be easily used by software engineers. Furthermore, Rebeca facilitates applying compositional verification techniques by a modular design approach. [50] also provides a tool to translate the Rebeca models into target languages of existing model checkers, e.g., SMV and Promela. This makes it possible to applying model checking to actor-based models. The properties to be checked are described in LTL-X, that is LTL without the next operator.

[17] proposes a natural integration of information flow properties into linear-time temporal logics. This is to verify the information flow properties with constraints such as when and under which conditions a variable has to be kept secret. A case study is presented to demonstrate the motivation. The target system is used in the case study is a conference management system. A typical property to be checked is "All intermediate decisions of the program committee are never revealed to the author until the notification". The approach presented here is adding a new modal operator (hide), that expresses the requirement that the observable behavior of a system is independent of the choice of a secret, into the linear-time temporal logics.

[35] proposes an approach to verify the liveness properties of the reactive systems using the Event-B method. By considering the limitation of the Event-B method, this work focuses on applying the language TLA+ to verify the liveness properties on Event-B models. TLA+ is an extension of the temporal logic introduced in [41]. The contribution of this work is divided into two parts. The first part extends the expressivity and the semantics of the Event-B model to be able to transform the Event-B model to a TLA+ module. The second part is to verify the liveness properties of both finite systems and infinite systems, which are described in the extended Event-B. For the finite model, the TLC model checker[55] is used. For the infinite model, the refinement technique of Event-B is applied to preserve the liveness properties through refinement steps.

[2] applies the Reactive Systems Development Support (RSDS) method to specify and verify the reactive systems. A RSDS is verified using the B theorem-prover. B however requires user interaction and is not capable of proving temporal properties easily, this work extends RSDS by integrating model checking so that temporal properties can be

verified. The model checker used is the Symbolic Model Verifier (SMV).

The above researches aim at the verification of the reactive systems against the temporal logic properties. The target systems are originally described or translated into languages of the verifiers. Our motivation is the gap between the specification and the design of the reactive systems. For this gap, we use appropriate specification languages for each of them, separately. We adopt Event-B for the specification and Promela for the design. Our verification framework is an integration between Event-B and model checking technique when Promela/Spin is used in the framework. However, this integration is different from the integrations presented in [35] and [2]. We originally describe the design, model of the system to be checked, in Promela, whereas both [35] and [2] describe the system model in Event-B. The specification is described in Event-B to ensure the internal consistency. The verification conditions are generated from the Event-B specification and encoded in assertions which can be checked by Promela/Spin.

Verification of operating systems

[10] and [24] present case studies on checking the operating systems compliant with OSEK/VDX. In [10], the authors describe the properties of interest in temporal logic formulas and describe the design in Promela. In [24], the authors express the properties in terms of the first-order logic and model the OS as CSP process. In these works, the limited configurations are used in the experiments to apply the model checking; however, how to estimate appropriate configurations for the verification is not explained. By using Event-B, we easily specify the properties and the external behaviors of the OS and ensure the quality of the specification before using it to check the design. In addition, we present a way to determine the appropriate bounds for the verification of desirable properties.

[26] and [28] present approaches to verify the OS kernels based on theorem proving. Theorem proving can be used to verify the infinite systems; however, it generally requires a lot of interactive proofs. In our framework, we use model checking combining with tools of Event-B. Although, ranges are bounded due to the limitation of model checking; however, we are able to improve quality of the properties checked and get completely automatic verification. Therefore, we have a high degree of confidence in the verification results.

Verification of systems based on simulation relation

FDR [46] is a refinement checker for the process algebra CSP. Inputs of FDR are the specifications and the implementations written in the same language, that is CSP. FDR has been applied in some verification frameworks as presented [8, 47, 15, 48]. These frameworks require to produce models of the target systems (both high-level and low-level descriptions) in CSP. Our framework, different specification languages are used for the high-level and the low-level descriptions. This allows to choose the appropriate languages to facilitates describing the specification and the design.

The simulation of two LTSs was introduced in [34] in which the key point is to find a relation between state spaces of two LTSs such that the simulation rule holds. In our approach, the relation between state spaces of two LTSs is given based on mappings between the variables and the values of two models. The verification conditions are generated for

such relation. In the last step of the framework, Spin checks whether the verification conditions hold. If so, we could show that the design simulates the specification by the given relation.

Generation of LTS from Event-B model

[29] presents the ProB tool which supports interactively animating B models. Using ProB, users can visualize the current state and the enabled operations in each state. Users also can set an upper limit on the number of ways that the same operation can be executed. However, ProB requires some interactions with the users. In our work, we firstly set finite ranges for types; for complex systems like operating systems, we may restrict the functionalites and the depth of the execution sequences in the Event-B specification. Then, we explore all possible sequences of state transitions within defined ranges. Our work does not support visualizing the LTS; however, the generation of the LTS is completely automatic.

[6] defines the semantic of Event-B model as labeled transition systems to reason about behavioral aspects of specifications in Event-B. We define the semantic of Event-B model in order to check the simulation relation by using Spin. Therefore, several notions used in our definition are different from those in [6]. In addition, we precisely define finite ranges of variable values in Event-B specification as bounds of our verification; then, we generate all possible behaviors from Event-B specification within defined ranges. We use tools of Event-B to ensure the internal consistency of the specification before using it to generate the properties to be checked.

Construction of environments of the open system

[52] specifies assumptions about environment behaviors to form a model of the environment. The approach is implemented in the Bandera environment generator (BEG) which automatically generates the environment scripts. The environment scripts are used to reason about properties of several nontrivial concurrent Java programs. This approach has been applied to web applications [42] and commercial softwares [53].

[54] verifies the OSEK OS by constructing a general model of the environment from scratch: the environment model includes a class diagram and state diagrams of objects in the environment. These diagrams are composed to generate the environment scripts.

In our work, the environment is generated from the Event-B specification. Hence, by construction, it is comprehensive with respect to the specification. The environment is used to exercise the design and check the given relation between variables of the Promela design and variables of the Event-B specification in every reachable state. This guarantees that the design conforms to the specification. Also, the correctness of the specification is guaranteed by tools of Event-B; the quality of the environment is improved.

Combination of Event-B model and model checking

For combination of Event-B and model checking, tools such as ProB[29] and Eboc[32] work as model checkers for Event-B. In these approaches, the target models are described in

Event-B. ProB and Eboc directly check the target models against the internal consistency. We use tools of Event-B to ensure the internal consistency. We use our own tool to generate the LTS of the Event-B specification.

In another approach, [37] translates Event-B model into Promela model and use Spin to check the model. We have not directly translated Event-B code into Promela but use the LTS of the Event-B specification to generate environments and assertions which are translated into Promela. Then, we use Spin to check the simulation relation between the design model and the specification by confirming the assertions.

Generation of C code from Event-B model

[33] presents a translation tool that automatically generates efficient target programming language code (C, C++, Java and C#) from Event-B formal specification. This work uses software model checking tool according to different target language like BLAST [9] for 'C' language. The software model checking tools are complement of the analysis conducted on Event-B specification by making it possible the verification of property preservation at the code level. Our tool also translates Event-B code to C++ code; however, our intention is different from that of [33]. [33] translate Event-B model to C code to obtain an implementation. Our intention is to generate the LTS of the specification. For generation of the LTS, we must execute the Event-B specification within some bounds. Our tool translates the specification from Event-B to C++ to execute the specification in C++ and generate the LTS.

From informal specification to formal specification

There are three steps in a general process of formalizing an informal specification: (1) Analyze the informal specification; (2) Interpretation and (3) Validation [7]. These steps match the step A, D1 and D2 in our framework, respectively. The steps B and C are added in our framework to support for validating the formal specification against the informal specification. The list of features can be given automatically thanks to some natural processing techniques [25]. When translating the informal descriptions of the constraint to the logical predicates, we can refer to some guidelines for translating from informal requirements to temporal logic formulae proposed by [20].

[16] applied B method to specify the main functionalities of FreeRTOS, a mini and specific RTOS has been implemented in C and assembly language. This work represented the first step of a formal modeling using B method. This model provides a functional specification of operations related to task management and message queues. It is the result of analysis of the documentation of the system as well as of the source code of its implementation. Checking the presented model with the document of the system is not discussed. Compared to this work, our work targets a faithful formalization of the OSEK/VDX OS specification. We present some ways to support for validating the formal specification against the original specification.

[9] introduced integrating specification techniques proposed by contributions from WIFT, 1998. These techniques only integrate Object Oriented models and formal models. Similarly, [21] focuses on integrating the structure analysis diagrams and formal models using Z. The common fundamental approach of the existing contributions is to define rules

for mapping the graphical constructs to the constructs in the formal modeling notations. Their objective is to increase the complement of formal and informal specification. Our work focuses on using Event-B to develop a formal model of RTOS from a specification that is written completely in natural language. Our objective is the likely equivalence between the formal specification and the original specification.

[27] proposed an approach to establish a traceability from informal requirements to formal refinements. In this approach, they used WRSPM reference model to provide structure to both of informal requirements and formal model for the purpose of achieving the traceability. This approach involves taking a requirement and identify phenomena and artifacts of the environment and system for that requirement. The identified phenomena and artifacts are then modeled and traceability information is produced. In our approach, we establish direct trace links from features singled out from the informal specification to the correspondent elements in the formal specification.

[14] formally specified a conventional paradigm of operating system. He used Z and Object-Z as formal specification languages in this work. Compared to our work, this work makes the formal specification from scratch and our work formalized an existing informal specification.

Chapter 7

Conclusion and Future Directions

Conclusion

We proposed an approach to verify designs against their formal specifications which are described in different specification languages respectively. This is a new combination between Event-B and Promela/Spin. A primary achievement of the approach is to make it possible to describe the specification and the design in appropriate languages for a verification of the design. Formal specification languages are intended to facilitate describing the specifications. Promela is intended to analyze the designs. Our approach follows these intentions faithfully. In fact, as mentioned in Chapter 2, it is natural for reactive systems like operating systems to describe the designs in the imperative specification languages. On the other hand, describing their detailed properties in temporal logic is generally hard. It is easy to imagine that the temporal logic formulas representing the specification shown in the case study become very complex and prone to mistakes. Instead of the temporal logic, we provide a way to represent the specification in formal specification language Event-B and check the design against it with the Spin model checker. Event-B is appropriate to represent the specification because it has rich notions such as sets and relations. By using appropriate specification languages for the specification and the design, we could deal with the gap between them. We also deal with the difference of languages used to describe two model by checking a simulation relation between them based on the semantics of the LTSs. In addition, Event-B allows us to ensure the consistency and the correctness of the specification by its verification facilities such as discharging proof obligations and refinement. That is, we can check the design against such consistent and correct specification. This would drastically improve the reliability of model checking results because the specification is reliable.

We conducted some case studies in which our framework is applied to verify the practical systems. The target systems used in our case studies range from simple systems, e.g., vending machines, to complex systems, e.g., automotive operating systems. Especially, we applied the framework to verify a real OS design which complies with the OSEK/VDX standard. The framework is straightforwardly applied to check the designs of the target systems with respect to their specifications in Event-B. The bounds can be effectively applied in the Event-B specifications. In addition, it is feasible to generate the LTS from the Event-B specifications. This is a source to generate exhaustive sequences of function calls for verification of the design. The results of the experiments demonstrate that this approach can be practically applied in verification of important properties and detection

of typical bugs of the target systems. This exhibits an ability to deal with the specifications and the designs which are described in different specification languages. Therefore, we can choose appropriate specification languages to describe the specification and the design for the purpose of verifying the design.

Future directions

There are some directions for our future works.

Detecting redundant behaviors in the design. The behaviors of the design are triggered by stimulus from the environment. In our framework, the environment is generated from the specification. Hence, by construction, it is comprehensive with respect to the specification. We are going to consider the behaviors of the design that are redundant compared with the specification. Such behaviors are never triggered by the environment; therefore, they could not be verified. An preliminary idea is to firstly construct an universal environment from scratch. Then, we could use this environment to exercise the behaviors of both specification and the design and check the correspondence of the results.

Extending of the framework for multiple languages of the specification and the design There is a possibility that our approach is applicable not only for Event-B and Promela but also the other specification languages. We plan to extend the verification framework to accept the additional choice of the specification languages. One of main tasks in the framework is to generate the LTS from the specification. Therefore, the framework could be extended to accept the different languages for the specification as long as it is feasible to generate the LTS from the specification described in those languages. In the framework, the design is described as a collection of functions which update the value assignments. The environment is described as a collection of invocations. Our framework can be extended to accept the designs described in not only Promela but also other languages as long as they can deal with a collection of functions for the design and sequences of invocations for the environment.

Generating test cases from the Event-B specification. Verification operates on formal models. This makes sure that the design satisfies the specification. Unlike verification, conformance testing is performed on the real implementation of the system by means of interactions between the implementation and test cases derived from the specification. Hence, the two techniques are complementary. In the proposed framework, we generated the LTS from the Event-B specification. This LTS contains all possible execution sequences described in the specification within the given bounds. Therefore, we could generate the test suite from the LTS. In this way, the test suite is exhaustive with respect to the specification within the given bounds.

Bibliography

[1] Jean-Raymond Abrial. *Modeling in Event-B: system and software engineering*. Cambridge Univ Press, 2010.

[2] Kalliopi Androutsopoulos. Specification and verification of reactive systems with rsds, 2004.

[3] Toshiaki Aoki. Model checking multi-task software on real-time operating systems. In *The 11th IEEE International Symposium on Object Oriented Real-Time Distributed Computing*, pages 551–555, 2008.

[4] RJR Back and Kaisa Sere. From modular systems to action systems. *Software-Concepts and Tools*, 17(1):26–39, 1996.

[5] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The MIT Press, 2008.

[6] Didier Bert, Marie-Laure Potet, and Nicolas Stouls. Genesyst: a tool to reason about behavioral aspects of B event specifications. application to security properties. 2010.

[7] Michel Bidoit, Claude Chevenier, Christine Pellen, and Jérôme Ryckbosch. An algebraic specification of the steam-boiler control system. In *Formal Methods for Industrial Applications, Specifying and Programming the Steam Boiler Control (the Book Grow out of a Dagstuhl Seminar, June 1995)*., pages 79–108, London, UK, UK, 1996. Springer-Verlag.

[8] Philippa J. Broadfoot and A. W. Roscoe. Tutorial on FDR and its applications. In *Proceedings of the 7th International SPIN Workshop*, pages 322–322, 2000.

[9] J.-M. Bruel, B. Cheng, S. Easterbrook, R. France, and B. Rumpe. Integrating formal and informal specification techniques. why? how? In *Industrial Strength Formal Specification Techniques, 1998. Proceedings. 2nd IEEE Workshop on*, pages 50–57, Oct 1998.

[10] Yunja Choi. Model checking trampoline os: a case study on safety analysis for automotive software. *Softw. Test., Verif. Reliab.*, 24(1):38–60, 2014.

[11] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Trans. Program. Lang. Syst.*, 16(5):1512–1542, September 1994.

[12] EdmundM. Clarke, William Klieber, Milos Novacek, and Paolo Zuliani. Model checking and the state explosion problem. In Bertrand Meyer and Martin Nordio, editors,

Tools for Practical Software Verification, volume 7682 of *Lecture Notes in Computer Science*, pages 1–30. 2012.

[13] CoEST. Traceability glossary, http://www.coest.org/index.php/traceability/.

[14] I. Craig. *Formal models of operating systems kernels*. Springer London, 2007.

[15] S. J. Creese and Joy N. Reed. Verifying end-to-end protocols using induction with csp/fdr. In *Proceedings of the 11 IPPS/SPDP'99 Workshops Held in Conjunction with the 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing*, pages 1243–1257, London, UK, UK, 1999. Springer-Verlag.

[16] David Déharbe, Stephenson Galvão, and Anamaria Martins Moreira. Formal methods: Foundations and applications. chapter Formalizing FreeRTOS: First Steps, pages 101–117. Springer-Verlag, Berlin, Heidelberg, 2009.

[17] Rayna Dimitrova, Bernd Finkbeiner, Máté Kovács, Markus N. Rabe, and Helmut Seidl. Model checking information flow in reactive systems. In *Proceedings of the 13th International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI'12, pages 169–185, Berlin, Heidelberg, 2012. Springer-Verlag.

[18] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in property specifications for finite-state verification. In *Proceedings of the 21st International Conference on Softw. Eng.*, pages 411–420, 1999.

[19] E. Allen Emerson. Handbook of theoretical computer science (vol. b). chapter Temporal and Modal Logic, pages 995–1072. MIT Press, Cambridge, MA, USA, 1990.

[20] A. Fantechi, S. Gnesi, G. Ristori, M. Carenini, M. Vanocchi, and P. Moreschini. Assisting requirement formalization by means of natural language translation. *Form. Methods Syst. Des.*, 4(3):243–263, May 1994.

[21] Robert B. France and Maria M. Larrondo-Petrie. A two-dimensional view of integrated formal and informal specification techniques. In *Proceedings of the 9th International Conference of Z Usres on The Z Formal Specification Notation*, ZUM '95, pages 434–448, London, UK, UK, 1995. Springer-Verlag.

[22] Jifeng He. Process simulation and refinement. *Formal Aspects of Computing*, 1(1):229–241, March 1989.

[23] Gerard J. Holzmann. *The SPIN Model Checker - primer and reference manual*. Addison-Wesley, 2004.

[24] Yanhong Huang, Yongxin Zhao, Longfei Zhu, Qin Li, Huibiao Zhu, and Jianqi Shi. Modeling and verifying the code-level osek/vdx operating system with csp. In *Theoretical Aspects of Software Engineering*, pages 142–149, Aug 2011.

[25] M. G. Ilieva and Olga Ormandjieva. Automatic transition of natural language software requirements specification into formal presentation. In *Proceedings of the 10th International Conference on Natural Language Processing and Information Systems*, NLDB'05, pages 392–397, Berlin, Heidelberg, 2005. Springer-Verlag.

[26] Tom In der Rieden and Steffen Knapp. An approach to the pervasive formal specification and verification of an automotive system. In *Proceedings of the 10th International Workshop on Formal Methods for Industrial Critical Systems*, pages 115–124, 2005.

[27] Michael Jastram, Stefan Hallerstede, Michael Leuschel, and Aryldo G. 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.

[28] Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an operating-system kernel. *Communications of the ACM*, 53(6):107–115, 2010.

[29] Michael Leuschel and Michael Butler. ProB: An automated analysis toolset for the B method. *International Journal on Software Tools for Technology Transfer*, 10(2):185–203, 2008.

[30] Seymour Lipschutz. *Schaum's outline of theory and problems of data structures*. Schaum's outline series: Schaum's outline series in computers. McGraw-Hill, 1986.

[31] Nancy Lynch and Frits Vaandrager. Forward and backward simulations: Untimed systems. *Inf. Comput.*, 121(2):214–233, September 1995.

[32] PauloJ. Matos, Bernd Fischer, and Joo Marques-Silva. A lazy unbounded model checker for event-b. In *Formal Methods and Softw. Eng.*, volume 5885. 485-503, 2009.

[33] Dominique Méry and Neeraj Kumar Singh. Automatic code generation from event-b models. In *Proceedings of the Second Symposium on Information and Communication Technology*, SoICT '11, pages 179–188, New York, NY, USA, 2011. ACM.

[34] Robin Milner. *Communication and concurrency*. PHI Series in computer science. Prentice Hall, 1989.

[35] Olfa Mosbahi, Leila Jemni Ben Ayed, and Mohamed Khalgui. A formal approach for the development of reactive systems. *Inf. Softw. Technol.*, 53(1):14–33, January 2011.

[36] Andreas Muller. Vdm the vienna development method, 2009.

[37] Thomas MULLER. Formal methods, model-cheking and rodin plugin development to link event-b and spin. 2009.

[38] V. Natarajan and Gerard J. Holzmann. Outline for an operational semantics of PROMELA. In *Proceedings of the 2nd International SPIN Workshop*, volume 32 of *DIMACS*, pages 133–152, 1997.

[39] Gerard ORegan. Z formal specification language. In *Mathematics in Computing*, pages 109–122. Springer London, 2013.

[40] OSEK/VDX Group. OSEK/VDX operating system specification 2.2.3, http://portal.osek-vdx.org/.

[41] Amir Pnueli. The temporal logic of programs. In *Foundations of Computer Science, 1977., 18th Annual Symposium on*, pages 46–57, Oct 1977.

[42] S.P. Rajan, O. Tkachuk, M. Prasad, I. Ghosh, N. Goel, and T. Uehara. Weave: Web applications validation environment. In *Software Engineering - Companion Volume, 2009. ICSE-Companion 2009. 31st International Conference on*, pages 101–111, May 2009.

[43] Steve Reeves and David Streader. Guarded operations, refinement and simulation. *Electron. Notes Theor. Comput. Sci.*, 259:177–191, 2009.

[44] Renesas. RX-OSEK850 (OSEK/VDX specification-compliant OS).

[45] RODIN and DEPLOY group. Event-B and the RODIN platform, http://www.event-b.org/.

[46] A. W. Roscoe. A classical mind. chapter Model-checking CSP, pages 353–378. Prentice Hall International (UK) Ltd., Hertfordshire, UK, 1994.

[47] A. W. Roscoe and Z. Wu. Verifying statemate statecharts using csp and fdr. In *Proceedings of the 8th International Conference on Formal Methods and Software Engineering*, ICFEM'06, pages 324–341, Berlin, Heidelberg, 2006. Springer-Verlag.

[48] A.W. Roscoe. Modelling and verifying key-exchange protocols using csp and fdr. In *Computer Security Foundations Workshop, 1995. Proceedings., Eighth IEEE*, pages 98–107, Jun 1995.

[49] Steve Schneider. *The B-Method an introduction.* Palgrave Macmillan, 2001.

[50] Marjan Sirjani, Ali Movaghar, Amin Shali, and Frank S. de Boer. Modeling and verification of reactive systems using rebeca. *Fundam. Inf.*, 63(4):385–410, June 2004.

[51] H.G. Tanner and G.J. Pappas. Abstractions of constrained linear systems. In *American Control Conference, 2003. Proceedings of the 2003*, volume 4, pages 3381–3386 vol.4, June 2003.

[52] O. Tkachuk, M.B. Dwyer, and C.S. Pasareanu. Automated environment generation for software model checking. In *Automated Software Engineering, 2003. Proceedings. 18th IEEE International Conference on*, pages 116–127, Oct 2003.

[53] Oksana Tkachuk and Sreeranga P. Rajan. Application of automated environment generation to commercial software. In *Proceedings of the 2006 International Symposium on Software Testing and Analysis*, ISSTA '06, pages 203–214, New York, NY, USA, 2006. ACM.

[54] Kenro Yatake and Toshiaki Aoki. Model checking of OSEK/VDX os design model based on environment modeling. In *Proceedings of the 9th International Colloquium on Theoretical Aspects of Computing (ICTAC '12)*, pages 183–197, 2012.

[55] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. Model checking tla+ specifications. In *Correct Hardware Design and Verification Methods*, pages 54–66. Springer-Verlag, 1999.

Publications

[1] Dieu Huong Vu, Yuki Chiba, Kenro Yatake and Toshiaki Aoki: A framework for Verifying the Conformance of Design to Its Formal Specifications. In: special Section on Formal Approach, IEICE Transaction (accepted on January 26, 2015)

[2] Dieu Huong Vu, Yuki Chiba, Kenro Yatake and Toshiaki Aoki: Checking the Conformance of a Promela Design to Its Formal Specification in Event-B. In: the preliminary proceeding of the third International Workshop on Formal Techniques for Safety-Critical Systems (FTSCS 2014), 188-203 (2014)

[3] Dieu Huong Vu, Toshiaki Aoki: Faithfully formalization of OSEK/VDX operating system specification. In: the third International Symposium on Information and Communication Technology, 23–24 (2012)

Appendix A

Expressions

A.1 Expressions and Boolean Expressions in Specification

The set of expressions in specification models is defined as $Exp ::= A|S$, where $A ::= c|x|\mathrm{N}|A_1{+}A_2|\ A_1{-}A_2|\ A_1{*}A_2|A_1/A_2$ and $S ::= \emptyset|\{A\}|S_1{\cup}S_2|S_1{\cap}S_2|\ S_1{\setminus}S_2|\overline{S}$. where "$A$" is arithmetic expression, "S" is set expression, "c" is constant, "x" is variable, and "N" is natural number.

Definition 14 *(Evaluation of expressions in Event-B). Let $[exp]_\sigma$ denote the value of an expression exp in state σ. $[exp]_\sigma$ is inductively determined as follows:* $[\mathrm{N}]_\sigma = N$, $[x]_\sigma = \sigma(x)$, $[c]_\sigma = c$, $[A_1 + A_2]_\sigma = [A_1]_\sigma + [A_2]_\sigma$, $[A_1 - A_2]_\sigma = [A_1]_\sigma - [A_2]_\sigma$, $[A_1 * A_2]_\sigma = [A_1]_\sigma * [A_2]_\sigma$, $[A_1/A_2]_\sigma = [A_1]_\sigma \ / \ [A_2]_\sigma$, $[\emptyset]_\sigma = \emptyset$, $[\{A\}]_\sigma = \{[A]_\sigma\}$, $[S_1 \cup S_2]_\sigma = [S_1]_\sigma \cup [S_2]_\sigma$, $[S_1 \cap S_2]_\sigma = [S_1]_\sigma \cap [S_2]_\sigma$, $[S_1 \setminus S_2]_\sigma = [S_1]_\sigma \setminus [S_2]_\sigma$, *and* $[\overline{S}]_\sigma = \overline{[S]_\sigma}$.

The boolean expression is defined as $BExp ::= \top|\bot|A_1{=}A_2|A_1{\geq}A_2|\ B_1{\wedge}B_2\ |B_1{\vee}B_2|\neg B|x \in S$. where "$B$" is boolean expression, "$\top$" is 'true', and "$\bot$" is 'false'.

Definition 15 *(Evaluation of boolean expressions). Let $[bexp]_\sigma$ denote the value of a boolean expression bexp in state σ. $[bexp]_\sigma$ is inductively determined as follows:*

$$[\top]_\sigma = tt,$$
$$[\bot]_\sigma = ff,$$

$$[A_1 = A_2]_\sigma = \begin{cases} tt, & \text{if } [A_1]_\sigma = [A_2]_\sigma \\ ff, & \text{otherwise,} \end{cases}$$

$$[A_1 \geq A_2]_\sigma = \begin{cases} tt, & \text{if } [A_1]_\sigma \geq [A_2]_\sigma \\ ff, & \text{otherwise,} \end{cases}$$

$$[B_1 \wedge B_2]_\sigma = \begin{cases} tt, & \text{if } [B_1]_\sigma = tt \text{ and } [B_2]_\sigma = tt \\ ff, & \text{otherwise,} \end{cases}$$

$$[B_1 \vee B_2]_\sigma = \begin{cases} tt, & \text{if } [B_1]_\sigma = tt \text{ or } [B_2]_\sigma = tt \\ ff, & \text{if } [B_1]_\sigma = ff \text{ and } [B_2]_\sigma = ff, \end{cases}$$

$$[\neg B]_\sigma = \begin{cases} tt, & \text{if } [B]_\sigma = ff \\ ff, & \text{if } [B]_\sigma = tt. \end{cases}$$

A.2 Expressions in Design

Expression in the design is defined as $PExp ::= c \mid x \mid p \mid PExp_1 + PExp_2 \mid PExp_1$-$PExp_2 \mid PExp_1 * PExp_2 \mid PExp_1 \,/\, PExp_2$. where '$c$" is constant, "$x$" is variable, and "$p$" is parameter.

Appendix B

Mappings used in verification of OS

Table B.1: Correspondences between values in Event-B and those in Promela

	D_S	D_D
task identifier	a	10
	b	11
	c	12
task priority	1	1
	2	2
task state	sus	1
	rdy	2
	run	2

Table B.2: Relating variables in S to those in D by using itermediate variables in E

Pattern No.	V_S	V_E	V_D
1	a	_task1	tsk_state[0]
	b	_task2	tsk_state[1]
	c	_task3	tsk_state[2]
2	a	_task1	tsk_state[1]
	b	_task2	tsk_state[2]
	c	_task3	tsk_state[0]

Table B.3: Correspondences between enabled events and function calls

Enabled events	**Function calls**
AT(a)	_ActivateTask($\_task1.\_tid$)
CT(a, b)	_ChainTask($\_task1.\_tid$, $\_task2.\_tid$)
AT(c)	_ActivateTask($\_task3.\_tid$)
AT(b)	_ActivateTask($\_task2.\_tid$)
CT(a, c)	_ChainTask($\_task1.\_tid$, $\_task3.\_tid$)
CT(b, c)	_ChainTask($\_task2.\_tid$, $\_task3.\_tid$)
TT(b)	_TerminateTask($\_task2.\_tid$)
TT(c)	_TerminateTask($\_task3.\_tid$)
TT(a)	_TerminateTask($\_task1.\_tid$)
GR($a, res1$)	_GetResource($task1.\_tid, res1.\_rid$)
WAITEVT($a, evt1$)	_WaitEvent($task1.\_tid, 1$)
SETINTR($inr1$)	_SetINTR($isr1.\_iid$)
SETINTR($inr2$)	_SetINTR($isr2.\_iid$)
RESETINTR($inr2$)	_ResetINTR($isr2.\_iid$)
GR($b, res1$)	_GetResource($task2.\_tid, res1.\_rid$)
RESETINTR($inr1$)	_ResetINTR($isr1.\_iid$)
RR($b, res1$)	_ReleaseResource($task2.\_tid, res1.\_rid$)
RR($a, res1$)	_ReleaseResource($task1.\_tid, res1.\_rid$)
SETEVT($b, a, evt1$)	_SetEvent($task2._tid, task1._tid, 1$)
GR($c, res1$)	_GetResource($task3.\_tid, res1.\_rid$)
RR($c, res1$)	_ReleaseResource($task3.\_tid, res1.\_rid$)