Conformance Testing for OSEK/VDX Operating System based on Model Checking

By CHEN Jiang

A thesis submitted to
School of Information Science,
Japan Advanced Institute of Science and Technology,
in partial fulfillment of the requirements
for the degree of
Master of Information Science
Graduate Program in Information Science

Written under the direction of
Associate Professor Toshiaki Aoki

March, 2011

Conformance Testing for OSEK/VDX Operating System based on Model Checking

By CHEN Jiang (0910036)

A thesis submitted to
School of Information Science,
Japan Advanced Institute of Science and Technology,
in partial fulfillment of the requirements
for the degree of
Master of Information Science
Graduate Program in Information Science

Written under the direction of
Associate Professor Toshiaki Aoki

and approved by
Associate Professor Toshiaki Aoki
Professor Kokichi Futatsugi
Associate Professor Masato Suzuki

February, 2011 (Submitted)

Abstract

The OSEK/VDX operating system is a real-time operating system widely used in the automotive applications. Due to the standardized system services as specified in the specification, it can reduce the effort to port application software and the development cost. However, the specification itself does not prescribe any particular implementation language of the system services and leaves a certain amount of flexibility. Therefore, conformance testing is needed to test whether or not an OSEK implementation is correct with respect to the specification.

This thesis deals with an original approach to conformance testing for OSEK OS. The main goal is to develop a method to automatic generation of test cases by model checking technique. Because a formal model of OSEK OS called as design model of OSEK OS is available, we consider making full use of it as a test oracle in this method. Starting point is to extract conformance requirements from the specification. The extracted conformance requirements are described in a precise and unambiguous manner by using a formal specification language. Test model is constructed based on the formalized conformance requirements. By model checking the test model with the design model, on the one hand, if the extracted conformance requirements are not correct with respect to the design model, correction of them are needed according to the checking result, on the other hand, if no violation is detected, exhaustive state space searching will be conducted and then test cases can be automatically generated from the witnesses.

Based on the proposed approach, test model can be constructed in a systematic way. We can assure the correctness of the test purposes with respect to the design model of OSEK OS. Moreover, the generated test cases from test model can achieve the corresponding test purposes. Therefore, test quality can be significantly enhanced.

Acknowledgments

First and foremost, I would like to show my deepest gratitude to my supervisor, Assoc. Prof. Toshiaki Aoki for his constant guidance, advice, assistance and support during my whole Master's course.

I would like to express my sincere thanks to Prof. Kokichi Futatsugi for his encouragement and helpful comments.

Furthermore, I am grateful to thanks all members in both Aoki Laboratory and Futatsugi Laboratory for their kindly helps to me not only in the research but also in the daily life.

I would like to thank Zhang Min for giving me so much help in completing this work.

Finally, I would like to thank my parents for encouraging and supporting me during my whole study in Japan, and Anita Duan for her love, support, and understanding.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

The OSEK/VDX operating system (further called OSEK OS)[gro05] is a real time operating system widely used in the automotive applications. The OSEK OS specification has been provided by the OSEK/VDX group, a joint project of the automotive companies in Europe, which aimed at establishing an industry standard for an open architecture for distributed control units in vehicles.

To achieve high portability and re-usability of automotive application software, the OSEK OS specification defines a set of interfaces between the application software and the OS by system services. Due to the standardized system services, it can reduce the effort to port application software as well as save the development cost. On the other hand, the specification itself does not restrict any implementation language of the system services. Moreover, the specification leaves a certain amount of flexibility. It is the responsibility of OS developers to define implementation specific issues. As a result, it may raise the risk that an OSEK OS implementation does not comply with the OSEK OS specification. Therefore, it must be possible to ascertain that the implementations of OSEK OS really conform to the specification of OSEK OS. One way to do this is by testing these OSEK OS implementations. This activity is known as conformance testing.

In order to support checking the conformance of OSEK OS implementations to the OSEK OS specification, the OSEK/VDX group has founded a project named MODISTARC[gro99]. In this project, a standard conformance testing methodology has been developed. It serves as the basis for the development of the test specifications and test tools. However, we found that based on this methodology, it is difficult to assure the correctness of the extracted test purposes with respect to the OSEK OS specification without verifying them. Even though test purposes are correct, it is still difficult to make sure that test cases can achieve the corresponding test purposes without validating them. Furthermore, it is also difficult to obtain exhaustive test cases since test cases are designed manually. As a consequence, test quality might be impacted for these shortages.

On the other hand, in recent years, we have developed a formal model of OSEK OS. This formal model is called as the design model of OSEK OS. It was written in PROMELA[GMP04], which is an accept language of model checker SPIN[Hol04]. In the design model, it models the functionality of system services as specified in the OSEK specification so that it can manipulate such as task state or resource state by calling the system services. In other related researches, we have done a huge number of experiments by model checking the design model of OSEK OS. Thus we have got sufficient confidence that it conforms to the OSEK OS specification. We think that the design model of OSEK OS can be used not only in the design stage but also in the test stage.

1.2 Objective

The main goal of this thesis is to propose a new approach to conformance testing for OSEK OS. In the proposed approach, we will concentrate mainly on developing a method to automatic generation of test cases by using model checking technique. Based on the proposed approach, the correctness of test purposes can be assured. Moreover, the generated test cases can achieve the corresponding test purposes. Therefore, test quality can be significantly enhanced.

1.3 Structure of the Thesis

This thesis is organized as follows: Chapter 2 introduces the background of this work. Then in Chapter 3, the overview of proposed approach is presented and the process of test generation is defined. Chapter 4 introduces the details of conformance requirements and the formal test specification. Chapter 5 presents the test model which is a key role for generation of test cases. In chapter 6, the experimental results are shown. Finally, Chapter 7 concludes this work.

Chapter 2

Background

2.1 OSEK/VDX

2.1.1 OSEK/VDX Group

OSEK/VDX has been founded as a joint project of the German OSEK group and the French VDX group since 1994. The project aimed at establishing an industry standard for an open architecture for distributed control units in vehicles. Various parts were proposed for the standard: OS (the basic services of the real-time kernel), COM (the communication services), NM (the Network Management services). This thesis deals with the OS part. For more information about OSEK/VDX refer to [gro05].

2.1.2 OSEK OS Overview

The OSEK OS provides several service groups which are structured in terms of functionality.

- Task management
- Resource management
- Event management
- Interrupt management
- Alarms
- Intra processor message handling
- Error treatment

This section gives a brief overview of OSEK OS. Most attention is spent on task management and resource management, since this work concentrates mainly on these functions of OSEK OS.

Task management The OSEK OS provides two different types of tasks, namely basic tasks and extended tasks. There are two main differences between basic tasks and extended tasks. One difference is that basic tasks might be allowed to support the multiple requesting of task activation whereas extended tasks are not. Multiple requesting of task activation means that the OSEK OS receives and records parallel activation of a basic task already activated. Another difference is that extended tasks are allowed to wait for events whereas basic tasks are not. Basic tasks have three states, shown in figure 2.1.

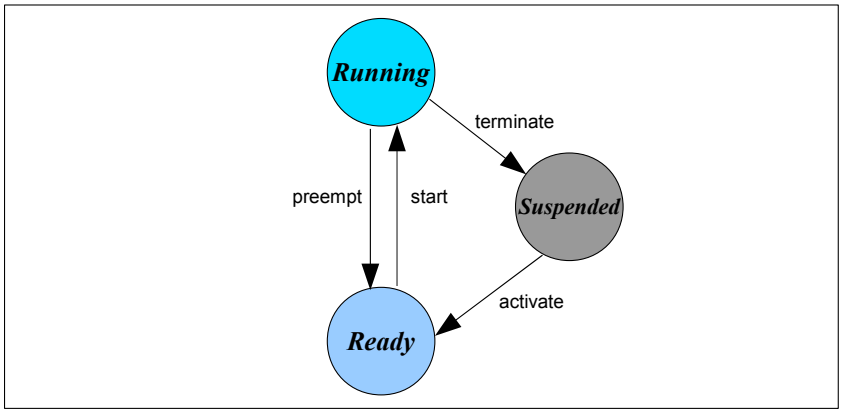

Figure 2.1: Basic task state model

The OSEK OS prescribes that all the OS objects must be defined statically during the system design. Tasks created as suspended state at the system generation time then wait for being activated. They become ready state after activation by calling some OS system services. At any point in time, only one task selected by the scheduler can be in the running state. The running task may terminate its execution and changed into suspended state or ready state by calling some OS system services. Comparing with basic tasks, extended tasks has the additional waiting state, shown in figure 2.2. The waiting state allows the processor to be released and to be reassigned to other task without the need to terminate the running extended task.

Conformance classes The OSEK OS specifies the concept of conformance class to describe different features of OS. The conformance classes define four kinds of the kernel to provide better scaling of the system so as to adapt to different requirements. They are determined by three main attributes:

- supporting of extended tasks or basic tasks only
- multiple requesting of activation of a basic task

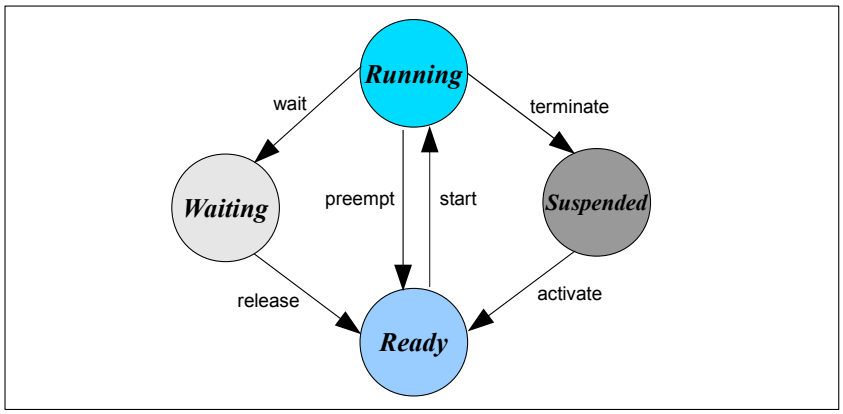

Figure 2.2: extended task state model

- the number of tasks per priority level

Scheduling policy The OSEK OS supports three types of scheduling policies which are described as follows:

- Full-preemptive scheduling: the currently running task can be rescheduled and changed into ready state as soon as a higher priority task has got ready.
- Non-preemptive scheduling: with non-preemptive scheduling, the currently running task can still be in the running state even though in the case of a higher task has got ready.
- Mixed-preemptive scheduling: tasks with full preemptive and tasks with non preemptive both exist in the same system. The scheduler is invoked depending on the policy of the currently running task which means if the task is non preemptive, non preemptive scheduling is performed and if the task uses full-preemptive scheduling policy, the running task can be rescheduled at any location.

Resource management The OSEK OS provides resource service to co-ordinate concurrent accesses of several tasks or interrupt service routines to access the critical section. The resource access protocol, namely priority ceiling protocol, is used to provide mutually exclusive access and prevent priority inversion and deadlock. According to this protocol, the priority ceiling is assigned to the resource statically at the system generation time. Its value is equal to the highest priority of all tasks (or ISR) accessing the resource. When a task gets a resource, its priority might be raised to the resource priority, so that other tasks which might share the same resource as the running task do not enter the running state, due to their lower or equal priority than the running task.

2.1.3 System Services

One goal of OSEK/VDX is to establish the standard to work against recurring expenses in development as well as incompatible interfaces defined by different manufactures. This is intended to be achieved by creating a set of abstract and application independent interfaces which do not rely on any specific hardware. To achieve high portability and re-usability of application software, the OSEK OS specification defines a set of interfaces between the application software and the OS by standardized system services. Thus the OSEK OS specification can be regarded as an interface specification defining system services to use within applications.

This section gives a brief introduction into some of the system services defined in the OSEK OS specification. In this work, we focus on five system services (*ActivateTask*, *TerminateTask*, *ChainTask*, *GetResource*, *ReleaseResource*) which belong to the service groups of task management and resource management.

- **ActivateTask**: The main function of this system service is to cause the activation of the target task. If the target task is in the suspended state, it will be transferred into the ready state. On the other hand, if the target task is not in the suspended state and supports multiple activation requests, the activation will be recorded. Furthermore, it will enforce a rescheduling in the case of the calling task is full-preemptive scheduling.

- **TerminateTask**: The main function of this system service is to cause the termination of the calling task. If the calling task does not support multiple activation requests, it will become suspended state. However, if the calling task supports multiple activation requests and multiple activation requests have been recorded, the calling task will transferred into ready state. Finally, once the system service is called successfully, it will enforce a rescheduling.

- **ChainTask**: This system service is the combination of *ActivateTask* and *TerminateTask*. The main function of it is to cause the termination of the calling task and the activation of the target task immediately. If the calling task is identical with the target task, this does not result in multiple activation requests. As far as the system service is called successfully, it will enforce a rescheduling.

- **GetResource**: This system services causes the occupation of target resource by the calling task.

- **ReleaseResource**:This system service causes the releasing of target resource by the calling task.

2.2 Conformance Testing

Software testing is an essential activity in any kind of software development process. By applying a set of experiments to a system implementation(implementation under test-

IUT), sufficient confidence of its correct function is gained.

There exists many strategies of software testing with multiple aims. In [Mye08] , two of the most prevalent strategies, namely black-box testing and white-box testing are defined. White-box testing, also referred to as structural testing, is to examine the internal structure of the program. Test data are derived from an examine of the program's logic. On the other hand, black-box testing, also called functional testing, in the sense that we do not concern about the internal behavior and structure of the program, instead, concentrate on checking correct functionality with respect to the specification. Test data are solely derived from the specifications.

In this thesis, we are interested in so-called conformance testing, which is a kind of black-box testing. The aim of conformance testing is to check that whether or not an implementation conforms to its specification.

2.2.1 MODISTARC

To support checking the conformance of OSEK OS implementations, OSEK/VDX group has founded another project named MODISTARC[gro99]. In the MODISTARC project, a standard conformance testing methodology has been developed. It serves as the basis for the development of the test specifications and test tools. The standard methodology defines a framework in which sets of tests, called test suites, should be defined and generated, including test suites for OS as well as COM and NM.

According to the standard conformance testing methodology, definition of the conformance testing for OSEK OS is a two-stage process:

- definition of the test purposes
- definition of the test cases

In the first stage, the test purposes are developed by analyzing the OSEK OS specification and extracting testable assertions. Testable assertions are, on the one hand observable actions (task switches, interrupts, etc.) performed by the OS, on the other hand the correctness of the return status of OS services. These assertions build the basis on which the test cases and the test suite are developed. The assembly of the test purposes makes up the test plan.

In the second stage, test cases which specify the sequence of the interactions between the test application and the implementation are defined to verify one or more test purposes. The assembly of the test cases makes up the test suite.

Problems of MODISTARC's Methodology In the official document of OS Test Plan provided by OSEK/VDX, the test purposes are listed in a table containing for each assertion:

- a sequence number used as a reference for test suite traceability,
- the description of the test purpose extracted from the specification,
- the variants of the specification to which the purpose applies,
- a reference to the paragraph in the specification allowing traceability to be provided against the specification.

Figure 2.3 shows an example of the test purposes in the OS Test Plan.

No.	Assertion	Page	Paragraph in spec.	Affected variants
6	A task in *suspended* state is not active. Task activation puts it to *ready* state.	17	4.2.1	All
7	A task in *waiting* state waits at least for one event. With the occurrence the task is set to *ready* state.	17	4.2.1	ECC1, ECC2
8	Pre-empted task is treated as the first task in the *ready* list of its priority.	17	4.2.1	All

Figure 2.3: Test purposes of MODISTARC

As mentioned in the previous section, the test purposes should consist of the observable actions performed by the OS and return status of OS services. The inconsistency between two different definitions of the test purposes in the same document do exist.

Moreover, according to the standard conformance testing methodology, test cases are generated by using the Classification Tree Method. Figure 2.4 shows a part of the test cases defined in the OS Test Plan. The test purposes do not exist in the test cases, which is inconsistent with the description of the test cases in the previous section.

Test case No.	Sched. policy Conf. class Status	Action	Expected Result
1	n, m, f B1, B2, E1, E2 e	Call `ActivateTask()` from task-level with invalid task ID (task does not exist)	Service returns E_OS_ID
2	n, m B1, B2, E1, E2 s, e	Call `ActivateTask()` from non-preemptive task on *suspended* basic task	No preemption of *running* task. Activated task becomes *ready*. Service returns E_OK

Figure 2.4: Test cases of MODISTARC

2.3 Model Checking

Model checking[BK+08, CGP, Cla97] is an automatic technique for formal verification of finite-state reactive systems. By modeling the reactive systems as finite-state automata

and formulating the properties in temporal logic, an efficient search procedure is used to determine automatically whether the model violates the property or not. If the state space exploration shows no property violations, then correctness with regard to the property is proved. When a property violation is detected, a counterexample will be generated to illustrate the property violation.

2.3.1 Testing with Model Checking

With the growing significance of computer systems within industry and wider society, techniques that assist in the production of reliable software are becoming increasingly important. On the one hand, testing still remains the most important method to verify the quality of software. One the other hand, formal verification start playing an important role to assure the correctness of software.

At first sight, formal verification and testing seem to be quite different things. On the one hand, formal verification is a static activity that involves analyzing system models, with the analysis completely covering all paths in a model. In contrast, testing is a dynamic activity that studies the real-world system itself, that is its implementation or source code, but often cover only a limited number of system paths.

However, during the past decade, a new consensus has developed in both academic and industrial areas, that is, using formal verification to support testing. The most important role for formal verification in testing is the automated generation of test cases, since manual testing takes a lot of effort and is error prone. Due to the ability to generate witnesses and counterexamples, model checking is the formal verification technology of choice.

Testing using model checking is a model-based testing technique. The model-based approach to software testing encompasses the creation of an abstract model, which is used to automatically create test cases. At the same time the model tells us the expected outcome, thus solving the test oracle problem. In [FWA09, HBB+09], the achievements made for testing using model checking have been well surveyed. To our knowledge, there still exists several obstacles in this area. For example, most work evolves around a set of small, well known example applications. In addition, in most work on model based testing, the existence of a suitable formal model is assumed. However, the creation and confirmation of the correctness of this model is one of the most difficult parts of the whole development process.

Chapter 3

Proposed Approach

3.1 Overview of Proposed Approach

In this work, we will propose a new approach to conformance testing for OSEK OS. Figure 3.1 shows the overview of the proposed approach. At first, test requirements will be extracted from the OSEK OS specification. By using a formal specification language, the extracted test requirements will be described in a precise and unambiguous manner. On the other hand, since a formal model of OSEK OS which is called as the design model of OSEK OS has been developed and assured to conform to the OSEK OS specification, we consider taking advantage of the existence of this formal model and use it as a test oracle. Therefore, we will translate the formal test specification into the test model. By model checking the test model with the design model of OSEK OS, exhaustive state space searching will be conducted and then test cases can be automatically generated from the witnesses. Then, we can use the generated test cases to test whether or not an implementation conforms to the OSEK OS specification.

3.2 Preliminaries

3.2.1 Model Checker SPIN

The model checking tool which we will use in this work is called SPIN[Hol04, Hol02]. To our knowledge, SPIN has become one of the most widely used model checkers in both academic and industrial areas during the past decade. This tool can be used to perform a very efficient verification of a system model against usual safety properties (like absence of deadlock) as well as complex liveness requirements expressed in linear temporal logic (LTL). The system model is specified in the verification language PROMELA (a Process Meta Language). PROMELA[GMP04] is a non-deterministic modeling language designed for describing systems composed of concurrent processes that communicate asynchronously. Syntax elements are borrowed from Dijkstra's guarded language, Hoare's CSP language and C programming language.

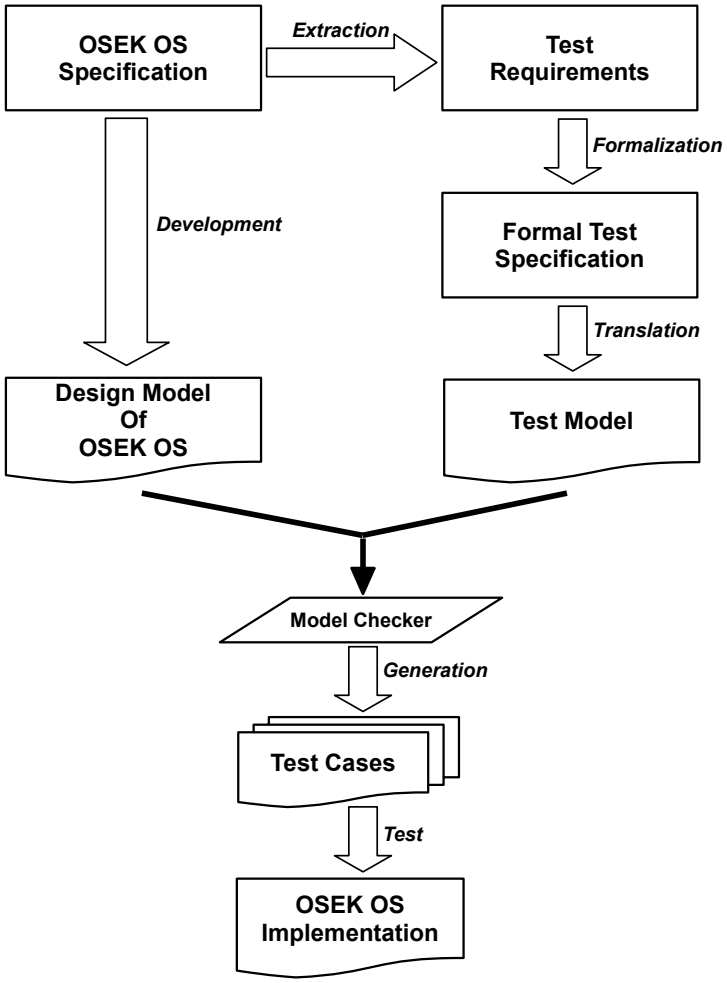

Figure 3.1: Overview of Proposed Approach

3.2.2 Z-Notation

Z-notation[Spi92, PTS96, II93] is a formal specification language which is based on the set theory and the first order predicate logic. In this section, we introduce some notations which are used in this thesis.

Schema Schema is one of the principal features of Z to specify the data of a system and the operations on the data. The form of schema is shown below:

ExampleA

$a : \mathbb{N}$
$b : \mathbb{N}$

$a \geq b$

The schema consists of two parts, the part above the central dividing line is known as the signature and the part below the line is known as the predicate. The signature introduces variables and assigns them a set theoretic type. The schema above named *ExampleA* introduces two variables a and b and states that a and b are two natural numbers. The predicate of a schema refers to the variables introduced in the schema or global variables in other schema and relates the values of these variable to each other. The predicate in the schema above asserts that a is greater than or equal to b.

The signature in a schema can introduce variables of any set theoretic type. They range from natural numbers up to complicated high-order functions. Whenever functions and relations are defined in a signature, their type is designated by the types of their domains and range, together with a symbol which gives the type of the function or relation.

Free type Free type is allowed to declare a set with a finite number of members. In the declaration the members are separated by the | symbol. For example,

$$Cars ::= Toyota \mid Honda \mid Nissan \mid Suzuki$$

defines the set of cars which has four members.

Schema inclusion The means by which schemas can be referred to by other schemas is known as schema inclusion. As an example consider the schema below:

ExampleB

$ExampleA$
$c : \mathbb{N}$
$d : \mathbb{N}$

$c \geq d$
$a + c \geq b + d$

In the schema *ExampleB*, it introduces two new fresh variables c and d of natural numbers, and states that c is greater than or equal to d in the predicate part. Since *ExampleA* is included in the signature part of *ExampleB*, *ExampleB* can refer to a and b from *ExampleA*. The effect of including one schema in the signatures of another schema is to form the union of their signatures and to conjoin their predicates.

Other notations Some other notations are also used in this thesis. They are summarized in table 3.1.

Notation	Explanation
$\neg\ P$	negation
$P\ \wedge\ Q$	conjunction
$P\ \vee\ Q$	disjunction
$P\ \Rightarrow\ Q$	implication
$P\ \Leftrightarrow\ Q$	equivalence
$\forall\ x:\ T\ \bullet\ P$	universal quantifier
$\exists\ x:\ T\ \bullet\ P$	existential quantifier
$\exists_1\ x:\ T\ \bullet\ P$	unique existence
$dom\ R$	domain
$X\ \rightarrow\ Y$	total function
$\mathbb{N}$	set of natural numbers
$\mathbb{N}_1$	set of positive natural numbers
$seq\ S$	set of the finite sequences of S
$\langle\rangle$	empty set
$head\ S$	first element of $seq\ S$

Table 3.1: Summary of other notations

3.3 Design Model of OSEK OS

In recent years, we have developed a formal model of OSEK OS in PROMELA named the design model of OSEK OS. The design model of OSEK OS consists of variables representing the OS objects (e.g. task), the finite state automata modeling the functionality of OS system services and controlling the ready queue. See the overview of the design model of OSEK OS in figure 3.2.

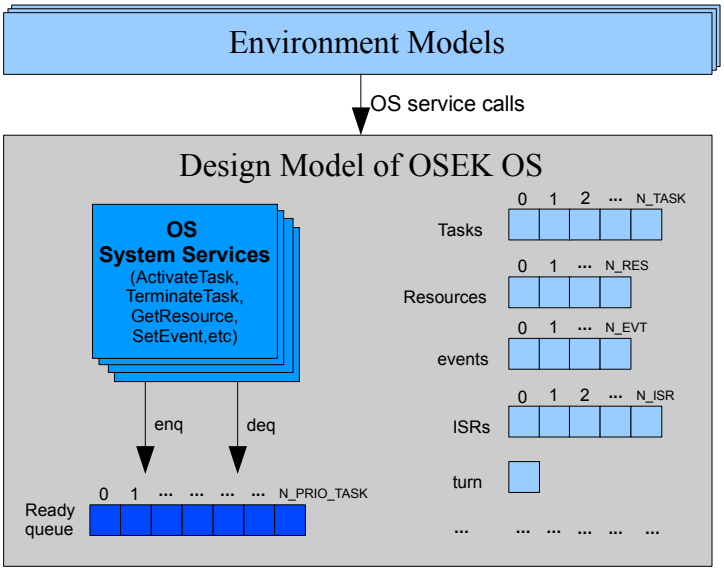

Figure 3.2: Overview of the design model of OSEK OS

Variables of the OS objects Task information including such as task identifier, task state and task priority, resource information as well as other OS objects information are stored in the arrays, respectively. The variable *turn* stores the task identifier of the currently running task. All task identifiers which are ready for execution are stored in the ready queue. The ready queue must be reordered according to task priorities after adding a new task or removing some task. The automata *enq* and *deq* provide mechanisms to control the ready queue.

OS system services Each OS system service is modeled by a finite state automaton representing its functionality as specified in the OSEK OS specification. The automata of system services are waiting in their initial states until they are invoked by outside stimuli. Then they can manipulate such as tasks states or other OS objects states as well as ready queue. As an example of OS system service, we introduce *ActivateTask(id)* modeled in the design model of OSEK OS. The automaton of *ActivateTask* system service is depicted in figure 3.3.

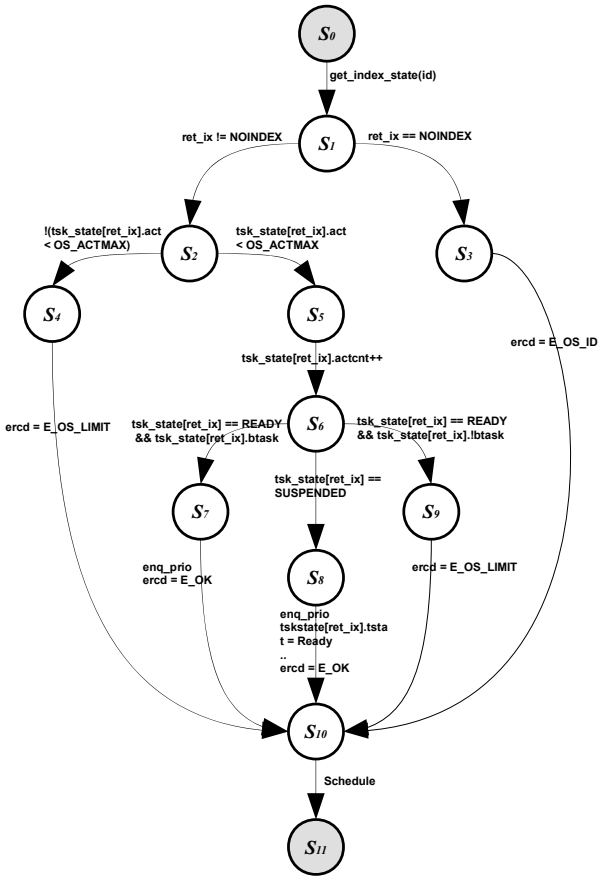

Figure 3.3: Automaton of ActivateTask system service

In [YA10], a method of modeling the outside environments has been proposed and a tool for generating all possible environments has been implemented. According to this method, huge amounts of experiments by model checking the design model of OSEK OS with environment models have been done. Therefore, we have sufficient confidence that the design model is correct and it conforms to the OSEK OS specification.

3.4 TGT: Test Case Generation Tool

Besides the design model of OSEK OS, we also developed a tool named test case generation tool, abbreviate to TGT. Figure 3.4 shows the mechanism of TGT for deriving sequences of called system services from the design model of OSEK OS automatically.

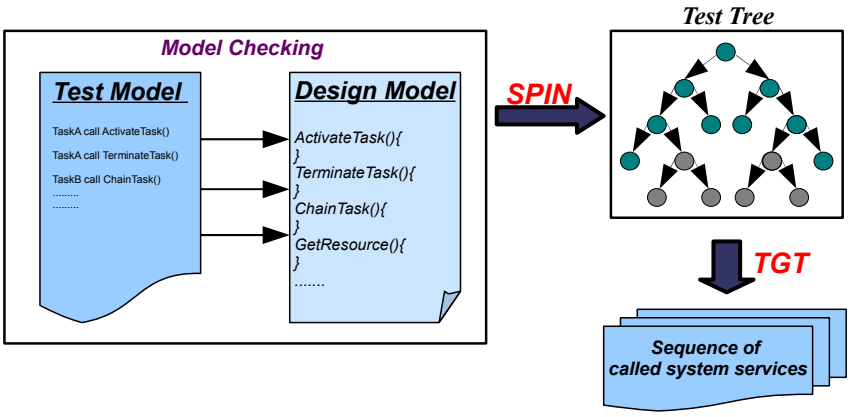

Figure 3.4: Mechanism of TGT

Because OS is a kind of reactive system, generally speaking, it always waits for external stimuli. Suppose that there exists a test model, that in this test model, some system services that are modeled in the design model of OSEK OS are going to be invoked. By model checking the test model with the design model of OSEK OS, some task states or other OS objects states might be changed according to the functionality of system services. Meanwhile, by using the exhaustive state search function of SPIN, all of possible reachable states and invoked system services can be accessed and checked. The witness of searching these states and system services can be recorded on the nodes and edges of the test tree in figure 3.4, respectively. As a consequence, sequence of called system services in this test tree can be derived by TGT automatically.

3.5 Test Generation Process

In the proposed approach, we concentrate mainly on the generation of test cases. Therefore, a test generation process is needed to be defined. Figure 3.5 shows the overall view

of test generation process.

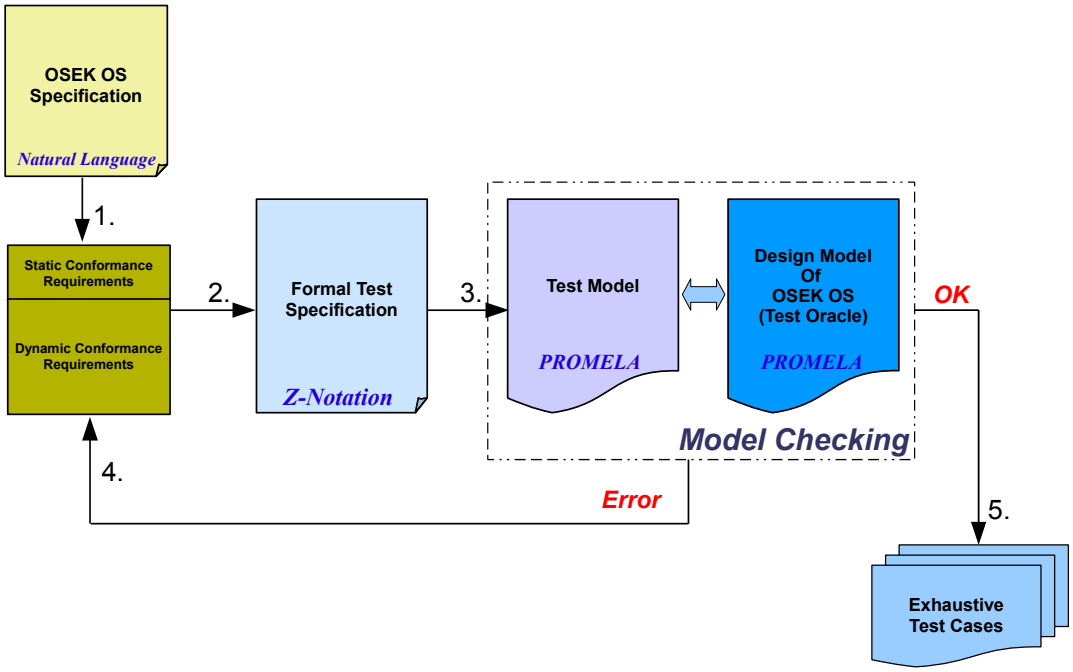

Figure 3.5: Test Generation Process

The whole process of test generation includes five steps as follows:

1. Extract the conformance requirements from the OSEK OS specification. In general, before testing a system implementation, the requirements of testing must be clarified. From conformance testing point of view, in order to test whether an implementation conforms to its specification, at first, the meaning of conformance should be clearly defined. According to the [Ray87, Tre92, Tre], conformance requirements make up of two kinds of requirements, namely static conformance requirements and dynamic conformance requirements. Thus a conforming implementation is one which satisfies both static conformance requirements and dynamic conformance requirements. In this work, we re-use these concepts and define the conformance requirements in the context of the OSEK OS specification.

2. Formalize the extracted conformance requirements. To remove ambiguous and give a precise description of the conformance requirements, we choose Z-notation as a formal specification language to formalized the extracted conformance requirements. The formalism is called as the formal test specification.

3. Construct test model based on the formal test specification. Since the design model of OSEK OS is available, we consider taking advantage of the existence of this model and use it as a test oracle for the generation of test cases to test OSEK

implementations. The test model will be implemented in PROMELA. Furthermore, to achieve automatic translation from the formal test specification to the test model, a translation algorithm is proposed.

4. Make correction of the conformance requirements. Since the OSEK OS specification does not state so-called conformance requirements explicitly, we have to extract the conformance requirements based on our knowledge of the specification. As a result, it is possible that the extracted conformance requirements are not correct with respect to the design model of OSEK OS. By model checking the test model with the design model of OSEK OS, we can get feedback from the checking result. If a violation is detected, SPIN will stop searching and report the violation place in the test model. Therefore, we can make correction of the corresponding conformance requirements according to the checking result.

5. Use TGT to generate test cases. If all the extracted conformance requirements are correct with respect to the design model of OSEK OS, by model checking the test model with the design model of OSEK OS, no violations will be reported, instead, the exhaustive state space searching will be conducted and the witnesses will be generated. Then we can use TGT to derive the exhaustive test cases from the witnesses.

Chapter 4

Formal Test Specification

4.1 Overview of Formal Test Specification

According to the test generation process, the starting point of conformance testing is to extract the conformance requirements, including the static conformance requirements and the dynamic conformance requirements from the OSEK OS specification. Since the OSEK OS specification does not state so-called conformance requirements explicitly, at first, we should give the definition of these two kinds of conformance requirements. According to the definitions, the conformance requirements are extracted from the OSEK OS specification.

To provide a precise and unambiguous description of the extracted conformance requirements, we choose Z-notation to formalize them. The formalism is called as the formal test specification. Another intended purpose of the formal test specification is to serve as a basis for constructing the test model. Figure 4.1 shows the overview of the formal test specification.

Formal test specification consists of four parts:

- Data Definition
- Data Declaration
- System Configuration Specification
- Test Purpose Specification

System configuration specification makes up of the formalized static conformance requirements extracted from the OSEK OS specification as well as the concrete system configuration. Concrete system configuration is a kind of test environment consisting of the OS objects and some other OSEK OS information which should satisfy the specific static conformance requirements. Test purpose specification consists of the formalized test purposes and some formal definitions of the generic OS objects and the system services that

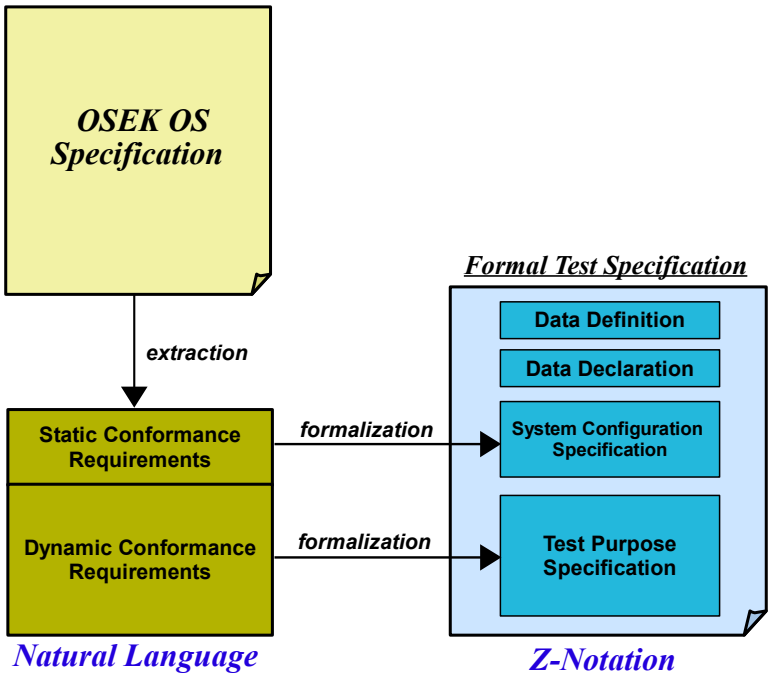

Figure 4.1: Overview of Formal Test Specification

are used in the formalized test purposes. Furthermore, in order to formalize the extracted requirements, the OS primitive objects such as task and resource and some other OSEK OS information should be formally defined and declared. They are specified in the data definition and data declaration parts. In next section, we introduce the details of the data definition and data declaration parts.

4.2 Data Definition and Data Declaration

Firstly, the task object is specified by a schema as below:

TASK

$tskid : \mathbb{N}$
$tsktype : TaskType$
$schpol : SchPol$
$MAXMULTIACT : \mathbb{N}$
$tskstate : TaskState$
$tskpri : \mathbb{N}$
$multiactcnt : \mathbb{N}$
$tskdpri : \mathbb{N}$
$tskrscstate : RscState$
$tskrscid : \mathbb{N}$

The signature part of schema *TASK* introduces some variables which represent the corresponding attributes of a task object. The meanings of each variable are shown as follows:

- *tskid*: identifier of a task which is specified as a natural number.
- *tsktype*: type of a task. Its type is a free data type named *TaskType* defined as below. *TaskType* is a set containing exactly two values representing basic task and extended task.
  - $TaskType ::= Basic \mid Extended$
- *schpol*: scheduling policy of a task. Its type is a free data type named *SchPol* defined as below. *SchPol* is a set containing exactly two values representing non-preemptive and full-preemptive.
  - $SchPol ::= NON\_PREE \mid FULL\_PREE$
- *MAXMULTIACT*: maximal value of multiple activation requests of a task which is specified as a natural number assigned at the system generation time.
- *tskstate*: current state of a task. Its type is a free data type named *TaskState* defined as below. *TaskState* is a set containing four values.
  - $TaskState ::= Suspended \mid Ready \mid Running \mid Waiting$
- *tskpri*: priority of a task which is specified as a natural number assigned at the system generation time.
- *multiactcnt*: current value of multiple activation requests of a task which is specified as a natural number.
- *tskdpri*: priority of a task occupying one or more resources which is specified as a natural number.

- *tskrscstate*: whether or not a task is occupying some resources. Its type is a free data type named *RscState* defined as below. *RscState* is a set containing exactly two values.
  - $RscState ::= Free \mid Occupy$
- *tskrscid*: identifier of a resource occupied by a task which is specified as a natural number.

Secondly, the resource object is defined by a schema as below:

RESOURCE

$rscid : \mathbb{N}$
$rscstate : RscState$
$rscpri : \mathbb{N}$
$rsctskid : \mathbb{N}$

The signature part of schema *RESOURCE* introduces some variables which represent the corresponding attributes of a resource object. The meanings of each variable are shown as follows:

- *rscid*: identifier of a resource which is specified as a natural number.
- *rscstate*: current state of a resource. Its type is a free data type *RscState*.
- *rscpri*: ceiling priority of a resource which is specified as a natural number assigned at the system generation time.
- *rsctskid*: identifier of a task occupying this resource which is specified as a natural number.

Moreover, two other free data types named *CFClass* and *StatusType* are defined.

- CFClass is a set containing exactly four values representing the conformance classes.
  - $CFClass ::= BCC1 \mid BCC2 \mid ECC1 \mid ECC2$
- StatusType is a set representing the return values from the system services. Notice that, since some system services as specified in the OSEK OS specification do not return any value in the case of being called successfully. Therefore, *E_OS_NORET* is added for this purpose.

– $StatusType ::= E\_OK \mid E\_OS\_ACCESS \mid E\_OS\_CALLEVEL \mid E\_OS\_ID$
$\mid E\_OS\_LIMIT \mid E\_OS\_NOFUNC \mid E\_OS\_RESOURCE$
$\mid E\_OS\_STATE \mid E\_OS\_VALUE \mid E\_OS\_NORET$

Then, declaration of the OS Objects in an application on top of an OSEK OS is specified by a schema as below:

OSEK_OS_Objects

$$
\begin{array}{l}
TskNum : \mathbb{N} \\
RscNum : \mathbb{N} \\
Task : \mathbb{N}_1 \rightarrow TASK \\
Resource : \mathbb{N}_1 \rightarrow RESOURCE \\
cfclass : CFClass \\
MaxTskMultiCnt : \mathbb{N}_1 \\
tskempty : \mathbb{N} \\
rscempty : \mathbb{N} \\
\hline
\forall i, j : 1..TskNum \bullet i \neq j \Rightarrow (Task(i)).tskid \neq (Task(j)).tskid \\
\forall i, j : 1..RscNum \bullet i \neq j \Rightarrow (Resource(i)).rscid \neq (Resource(j)).rscid \\
\forall i : 1..TskNum \bullet (Task(i)).tskid \neq tskempty \\
\forall i : 1..RscNum \bullet (Resource(i)).rscid \neq rscempty
\end{array}
$$

The signature part of schema *OSEK_OS_Objects* introduces some variables. The meanings of each variable are shown as follows:

- *TskNum*: number of tasks in an application on top of an OSEK OS which is specified as a positive natural number.
- *RscNum*: number of resources in an application on top of an OSEK OS which is specified as a positive natural number.
- *Task*: a set of tasks in an application on top of an OSEK OS. It is modeled by a function from the set $\mathbb{N}_1$ of positive natural numbers to *TASK*. The fact that task identifiers are different from each other is specified by the first predicate in the predicate part of this schema.
- *Resource*: a set of resources in an application on top of an OSEK OS. It is modeled by a function from the set $\mathbb{N}_1$ of positive natural numbers to *RESOURCE*. The fact that resource identifiers are different from each other is specified by the second predicate in the predicate part of this schema.
- *cfclass*: conformance class of an OSEK OS. Its type is a free data type *CFClass*.
- *MaxTskMultiCnt*: maximal value of multiple activation requests of tasks in an application on top of an OSEK OS which is specified as a positive natural number.

- *tskempty*: a special task identifier. The fact that no task identifier in the application is same as this one is specified by the third predicate in the predicate part of this schema.
- *rscempty*: a special resource identifier. The fact that no resource identifier in the application is same as this one is specified by the last predicate in the predicate part of this schema.

4.3 System Configuration Specification

In this section, we present the details of system configuration specification. The system configuration specification consists of two parts. One part are the formalized static conformance requirements extracted from the OSEK OS specification. The other part is the concrete system configuration. The concrete system configuration is a kind of test environment including the concrete values of some OSEK information and the instantiated OS objects that should satisfy the static conformance requirements.

4.3.1 Definition of Static Conformance Requirements

It is the common case that most specifications leave open a lot of options, which may or may not be implemented in a specific implementation. An implementer selects a set of options for implementation and is required for listing them in some document. Generally, it may exist some restrictions on the selection of options. Furthermore, each option may allow the different minimum capabilities of an implementation. Such kind of restrictions and corresponding minimum capabilities should be defined as the static conformance requirements of a specification. In other words, static conformance requirements define requirements on the minimum capabilities of a specific option that an implementation is to provide, and on the combination and consistency of different options.

With respect to the OSEK OS specification, in order to achieve high scalability of applications executed on top of an OSEK OS, four conformance classes (BCC1, BCC2, ECC1, ECC2) are defined so as to meet different requirements concerning functionality and capability of the OSEK OS. On the one hand, the minimum requirements determined by several attributes for each conformance class are different with each other. On the other hand, the OSEK OS specification prescribes that it is mandatory to implement the complete conformance classes. It means that if an OSEK OS implementation conforms to the specification, it should satisfy the minimum requirements of each conformance class.

Therefore, the definition of static conformance requirements in the context of the OSEK OS specification is given as:

Static conformance requirements define the requirements on the minimum capabilities of each specific conformance class as specified in the OSEK OS specification.

4.3.2 Formalization of Static Conformance Requirements

Once we have given the definition of static conformance requirements, the next step is to extract them from the OSEK OS specification and use Z-notation to formalize them.

According to the OSEK OS specification, the minimum capabilities of each specific conformance class is determined by several attributes. They are summarized in table 4.1.

Conformance Class	Task Type	Multi Act.	Tasks / priority
BCC1	Basic	No	1
BCC2	Basic	Yes(Basic)	Many
ECC1	Basic + Extended	No	1
ECC2	Basic + Extended	Yes(Basic)	Many

Table 4.1: OSEK OS conformance classes summary

The formalism of static conformance requirements is a schema named *Stc_Cfm_Req* in Appendix A. In this section, we introduce the details about how to formalize the minimum capabilities of each specific conformance class.

For the case of **BCC1**, the OSEK OS only supports the basic task, and limits to one activation request per task and task per priority, while all tasks have different priorities.

Two predicates are applied to formalize the requirements of BCC1 as follows:

1. $\forall\, i : 1...TskNum \bullet 0 < Task(i).MAXMULTIACT \leqslant MaxTskMultiCnt$
2. $cfclass = BCC1 \Rightarrow (MaxTskMultiCnt = 1) \land (\forall\, i : 1...TskNum \bullet (Task(i)).tsktype = Basic) \land (\forall\, i, j : 1...TskNum \bullet i \neq j \Rightarrow (Task(i)).tskpri \neq (Task(j)).tskpri)$

For the case of **BCC2**, the OSEK OS only supports the basic task, and allows the multiple requesting of task activation and more than one task per priority.

Two predicates are applied to formalize the requirements of BCC2 as follows:

1. $\forall\, i : 1...TskNum \bullet 0 < Task(i).MAXMULTIACT \leqslant MaxTskMultiCnt$
2. $cfclass = BCC2 \Rightarrow (MaxTskMultiCnt \geqslant 1) \land (\forall\, i : 1...TskNum \bullet (Task(i)).tsktype = Basic)$

For the case of **ECC1**, the OSEK OS supports the basic task as well as the extended tasks, and limits to one activation request per task and task per priority, while all tasks have different priorities.

Three predicates are applied to formalize the requirements of ECC1 as follows:

1. $\forall\, i : 1...TskNum \bullet (Task(i)).tsktype = Extended \Rightarrow (Task(i)).MAXMULTIACT = 1$
2. $\forall\, i : 1...TskNum \bullet 0 < Task(i).MAXMULTIACT \leqslant MaxTskMultiCnt$
3. $cfclass = ECC1 \Rightarrow (MaxTskMultiCnt = 1) \wedge (\forall\, i : 1...TskNum \bullet (Task(i)).tsktype = Basic \vee (Task(i)).tsktype = Extended))$
   $\wedge\, (\forall\, i, j : 1...TskNum \bullet i \neq j \Rightarrow (Task(i)).tskpri \neq (Task(j)).tskpri)$

For the case of **ECC2**, the OSEK OS supports the basic task as well as the extended tasks, and multiple requesting of task activation and more than one task per priority are allowed for basic tasks.

Three predicates are applied to formalize the requirements of ECC2 as follows:

1. $\forall\, i : 1...TskNum \bullet (Task(i)).tsktype = Extended \Rightarrow (Task(i)).MAXMULTIACT = 1$
2. $\forall\, i : 1...TskNum \bullet 0 < Task(i).MAXMULTIACT \leqslant MaxTskMultiCnt$
3. $cfclass = ECC2 \Rightarrow (MaxTskMultiCnt \geqslant 1) \wedge (\forall\, i : 1...TskNum \bullet (Task(i)).tsktype = Basic \vee (Task(i)).tsktype = Extended))$

4.3.3 Concrete System Configuration

Since the OS is a kind of reactive system, to test such kind of system, in general, it is needed to build some test applications that the boundaries of them are clearly defined. With regards to the OSEK OS, for example, we need to clear defined the number of tasks, and other objects used to invoke the system services. Furthermore, the concrete initial values assigned to the corresponding attributes of OS objects are needed to be decided.

In this work, concrete system configuration includes the concrete values as follows:

1. Conformance class
2. Maximal value of multiple activation requests of tasks
3. Number of tasks
4. Number of resources
5. Initial values of OS objects
   - task: identifier, type, the scheduling policy, maximal value of multiple activation requests, priority

- resource: identifier, priority

Moreover, they should satisfy the corresponding static conformance requirements. We show an example of concrete system configuration as follows. In order to keep the consistency, we use the schema to specify the concrete system configuration.

Concrete_Sys_Cfg

OSEK_OS_Objects
Stc_Cfm_Req

$cfclass = BCC1$
$MaxTskMultiCnt = 1$
$TskNum = 1$
$RscNum = 1$
$(Task(1)).tskid = 1$
$(Task(1)).tsktype = Basic$
$(Task(1)).schpol = FULL\_PREE$
$(Task(1)).MAXMULTIACT = 1$
$(Task(1)).tskpri = 2$
$(Resource(1)).rscid = 1$
$(Resource(1)).rscpri = 2$

However, for this example, if the task type is changed to extended, it will not be a concrete system configuration because the BCC1 only support basic tasks.

As a matter of fact, the number of concrete system configuration is unbounded, in the formal test specification, we just give an example of it.

4.4 Test Purpose Specification

Once the static conformance requirements have been extracted and formalized, the next step is to extract dynamic conformance requirements and formalize them by Z-notation. In this section we present the details of test purpose specification.

4.4.1 Definition of Dynamic Conformance Requirements

Like the case of the static conformance requirements, the so-called dynamic conformance requirements are also not stated explicitly in the OSEK OS specification. With regards to the reactive systems, dynamic conformance requirements define the requirements on the observable behavior of implementations in the interaction with their environments. With respect to the OSEK OS, it interacts with outside environments by the system services defined in the specification.

Therefore, the definition of dynamic conformance requirements in the context of the OSEK OS specification is given as:

Dynamic conformance requirements define the requirements on the observable behavior of OSEK implementations in the interaction with their environments by the system services as specified in the OSEK OS specification.

Thus we focus on the system services and extract dynamic conformance requirements from them. In this work, we introduce the concept of test purpose and make it relate to the dynamic conformance requirements.

4.4.2 Definition of Test Purpose

In this section, the definition of test purpose is given and the relationship between test purpose and the dynamic conformance requirements is shown by a case study. After that, we introduce the details of formalization of test purposes in Z-notation. Moreover, in order to realize the automatic translation from test purposes to the test model, the syntax of test purpose has been described by BNF (Backus Normal Form) in the Appendix B.

The definition of test purpose is given as:

Test purpose is a set of dynamic conformance requirements, focusing on the pre-state, pre-condition, post-state and return value of a specific system service as specified in the OSEK OS specification.

To better understanding of the definition of test purpose, we study an example. The description of the system service *ActivateTask* in the OSEK OS specification is shown in figure 4.2.

Because it does not state any dynamic conformance requirements explicitly in this system service, at first, we need to extract them based on our knowledge of the specification. Here, some of them are listed as follows:

1. If the called task is in the suspended state, it should be transferred into the ready state.
2. If the scheduling policy of the calling task is full-preemptive, rescheduling should happen.
3. If the system service is called successfully, the return status should be E_OK.
4. etc,.

13.2.3.1 ActivateTask

Syntax:	StatusType ActivateTask (TaskType <TaskID>)
Parameter (In):	
TaskID	Task reference
Parameter (Out):	none
Description:	The task <TaskID> is transferred from the *suspended* state into the *ready* state[14]. The operating system ensures that the task code is being executed from the first statement.
Particularities:	The service may be called from interrupt level and from task level (see Figure 12-1).
	Rescheduling after the call to *ActivateTask* depends on the place it is called from (ISR, non preemptable task, preemptable task).
	If E_OS_LIMIT is returned the activation is ignored.
	When an extended task is transferred from suspended state into ready state all its events are cleared.
Status:	
Standard:	• No error, E_OK
	• Too many task activations of <TaskID>, E_OS_LIMIT
Extended:	• Task <TaskID> is invalid, E_OS_ID
Conformance:	BCC1, BCC2, ECC1, ECC2

[14] ActivateTask will not immediately change the state of the task in case of multiple activation requests. If the task is not suspended, the activation will only be recorded and performed later.

Figure 4.2: Description of ActivateTask in the OSEK OS specification

Secondly, an example of generic test case designed based on the dynamic conformance requirements list above is shown in figure 4.3.

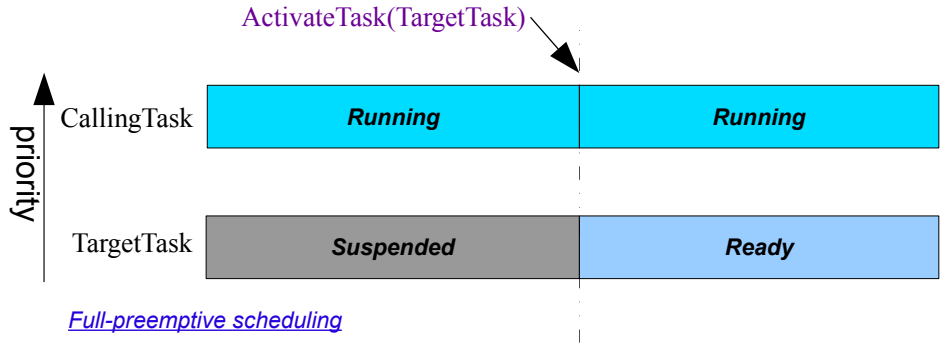

Figure 4.3: an example of generic test case

Two generic tasks are referred in this example of test case. The task which is in the

running state and ready to call the system service is regarded as the *CallingTask*. The task which will be activated by calling the system service is regarded as the *TargetTask*. Suppose that the scheduling policy of *CallingTask* is full-preemptive, and the priority of *CallingTask* is higher than the priority of *TargetTask*. *TargetTask* is in the suspended state before the system service is called. If *CallingTask* calls *ActivateTask*, state of *TargetTask* should be transferred from suspended state into ready state according to the first dynamic conformance requirement. Due to the scheduling policy of *CallingTask*, rescheduling should happen according to the second dynamic conformance requirement. Since the priority of *CallingTask* is higher than the priority of *TargetTask*, both of them are still in the states which are equal to the state before rescheduling. Finally, the system service should return *E_OK* because of the succeeding in calling the system service according to the third dynamic conformance requirement.

According to the definition of test purpose, it focuses on the pre-state, pre-condition, post-state and return value of a specific system service. Therefore, a test purpose derived from the generic test case above is shown as follows:

- pre-state: *CallingTask* is in the running state. *TargetTask* is in the suspended state.
- pre-condition: The scheduling policy of *CallingTask* is full-preemptive. The priority of *CallingTask* is higher than the priority of *TargetTask*.
- system service: *CallingTask* calls *ActivateTask(TargetTask)*.
- post-state: *CallingTask* is in the running state. *TargetTask* is in the ready state.
- return value: *E_OK*

According to this example, we can conclude the process of developing a test purpose as follows:

1. Identify and extract the dynamic conformance requirements from the system service.
2. Design a generic test case based on some dynamic conformance requirements.
3. Derive the test purpose from the corresponding generic test case.

4.4.3 Formalization of Test Purpose

To formalize the test purposes, we need to formally define the generic OS objects and the system services which are used in the test purposes.

At first, the generic OS objects used in the test purposes are specified by a schema named *Test_Purpose_Objects* in the Appendix A. The meanings of each variable and formalization of them are shown as follows:

- EmptyTask: a special task that its task identifier is not equal to any task in the system, which is specified as:
  - $EmptyTask : TASK$
  - $EmptyTask.tskid = tskempty$
- CallingTask: the running task if there exists running task in the system, otherwise it is the *EmptyTask*, which is specified as:
  - $CallingTask : TASK$
  - $\exists_1\ i : 1..TskNum \bullet (Task(i).tskstate) = Running \Rightarrow CallingTask = (Task(i))$
  - $\neg\ (\exists\ i : 1..TskNum \bullet (Task(i).tskstate) = Running \Rightarrow CallingTask = EmptyTask)$
- TargetTask: Any task except the running task if there exists the running task in the system, otherwise it is the *EmptyTask*.
  - $TargetTask : TASK$
  - $\forall\ i : 1..TskNum \bullet (Task(i).tskstate) \neq Running \wedge CallingTask \neq EmptyTask \Rightarrow TargetTask = (Task(i))$
  - $CallingTask = EmptyTask \Rightarrow TargetTask = EmptyTask$
- ReadyQueue: ready queue of tasks in the system.
  - $ReadyQueue : seq\ TASK$
- NextRunTask: first task in the ready queue if the ready queue is not empty, otherwise, it is the *EmptyTask*.
  - $NextRunTask : TASK$
  - $ReadyQueue \neq \langle\rangle \Rightarrow (\exists_1\, i : 1..TskNum \bullet headReadQueue = (Task(i)) \Rightarrow NextRunTask = Task(i))$
  - $ReadyQueu = \langle\rangle \Rightarrow NextRunTask = EmptyTask$
- TargetRsc: any resource in the system.
  - $TargetRsc : RESOURCE$
  - $\forall\, i : 1..RscNum \bullet TargetRsc = (Resource(i))$

Secondly, the system services which are used in test purpose are specified by a schema named *Test_Purpose_SysCalls*. The formalization of them are shown as follows:

- ActivateTask: the system call *ActivateTask*, which is specified as a function type.
  - $ActivateTask : TASK \times TASK \rightarrow StatusType$

  - $CallingTask.tskstate = Running \Leftrightarrow (CallingTask, TargetTask) \in dom\ ActivateTask$
  - $CallingTask.tskstate = Running \Leftrightarrow (CallingTask, CallingTask) \in dom\ ActivateTask$

- TerminateTask: the system call *TerminateTask*, which is specified as a function type.
  - $TerminateTask : TASK \rightarrow StatusType$
  - $CallingTask.tskstate = Running \Leftrightarrow (CallingTask) \in dom\ TerminateTask$

- ChainTask: the system call *ChainTask*, which is specified as a function type.
  - $ChainTask : TASK \times TASK \rightarrow StatusType$
  - $CallingTask.tskstate = Running \Leftrightarrow (CallingTask, TargetTask) \in dom\ ChainTask$
  - $CallingTask.tskstate = Running \Leftrightarrow (CallingTask, CallingTask) \in dom\ ChainTask$

- GetResource: the system call *GetResource*, which is specified as a function type.
  - $GetResource : TASK \times RESOURCE \rightarrow StatusType$
  - $CallingTask.tskstate = Running \Leftrightarrow (CallingTask, TargetRsc) \in dom\ GetResource$

- ReleaseResource: the system call *ReleaseResource*, which is specified as a function type.
  - $ReleaseResource : TASK \times RESOURCE \rightarrow StatusType$
  - $CallingTask.tskstate = Running \Leftrightarrow (CallingTask, TargetRsc) \in dom\ ReleaseResource$

Once the generic OS objects and the system services are formally defined, the test purposes can be formalized. The test purpose in the case study is formalized as depicted in figure 4.4.

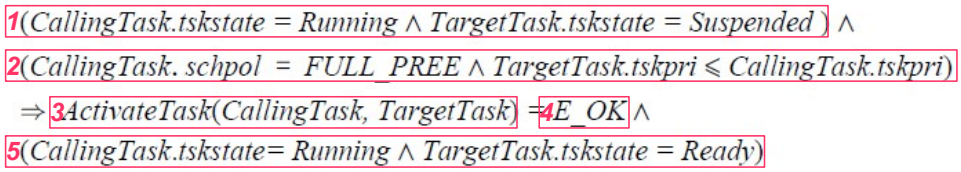

Figure 4.4: an example of formalized test purpose

The formalized test purpose consists of five parts:

- Pre-state: $CallingTask.tskstate = Running \ \wedge \ TargetTask.tskstate = Suspended$

- Pre-condition: $CallingTask.schpol = FULL\_PREE \ \wedge \ TargetTask.tskpri \leqslant CallingTask.tskpri$
- Action: $ActivateTask(CallingTask, TargetTask)$
- ReturnValue: E_OK
- Post-state: $CallingTask.tskstate = Running \ \wedge \ TargetTask.tskstate = Ready$

Chapter 5

Construction of Test Model

5.1 Overview of Test Model

According to the test generation process, test model should be constructed based on the formal test specification. Since the design model of OSEK OS is available, we consider taking advantage of the existence of this model and use it as a test oracle for the generation of test cases to test OSEK implementations. Moreover, another intended purpose for constructing the test model is to assure the correctness of the extracted conformance requirements with respect to the design model of OSEK OS. Because the OSEK OS specification does not state so-called conformance requirements explicitly, we have extracted conformance requirements based on our knowledge of the specification. As a consequence, it is possible that the extracted conformance requirements are not correct with respect to the design model of OSEK OS. By model checking the test model with the design model of OSEK OS, we can get feedback from the checking result. If a violation is detected, the model checker SPIN will report the violation place in the test model, which can help us to find out the incorrect conformance requirements. See the overview of the test model depicted in figure 5.1

Test model consists of three parts:

- Data definition part
- OS objects part
- Verification part

As shown in figure 5.1, each part in the test model has the relationship with the corresponding parts in the formal test specification. To achieve automatic translation from the formal test specification to the test model, a translation algorithm has been proposed. In next sections, we introduce the details of each part by concentrating on showing the relationships between the test model and the formal test specification.

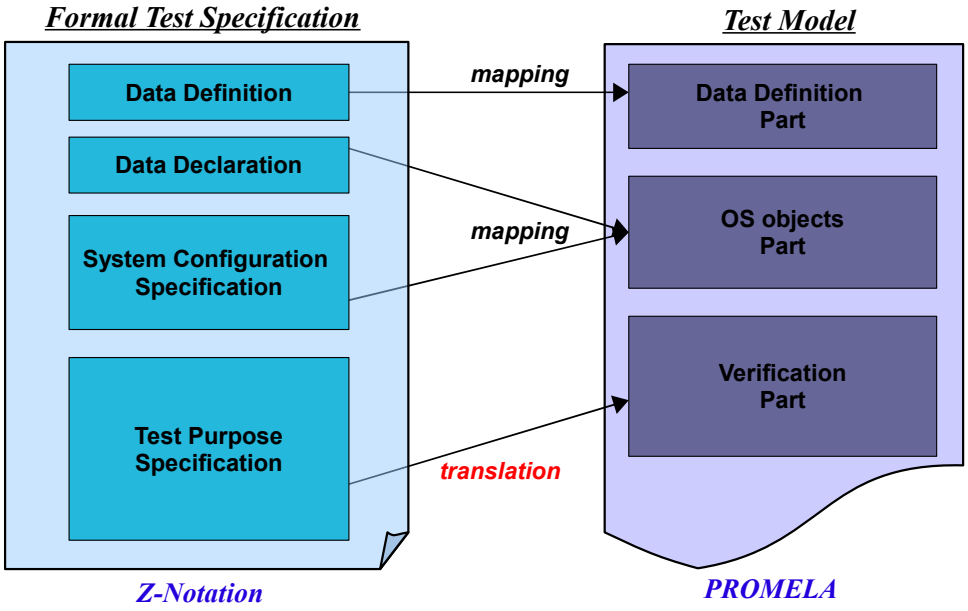

Figure 5.1: Overview of Test Model

5.2 Data Definition Part

The data definition part consists of the definitions of data types used in the test model. On the one hand, we use macro definitions to define the data types which has been specified as free data types in the formal test specification. For example, in the data definition part of the formal test specification, a free data type named *TaskType* is specified as:

$$TaskType ::= Basic \mid Extended$$

In the test model, the corresponding part in the data definition part is defined as:

```
#define     TaskType     bit
#define     Basic        1
#define     Extended     0
```

On the other hand, we use structure types to define the data types which are specified as schema types in the data definition part of the formal test specification. For instance, in the data definition part of the formal test specification, a schema named *TASK* is specified as:

TASK

$tskid : \mathbb{N}$
$tsktype : TaskType$
$schpol : SchPol$
$MAXMULTIACT : \mathbb{N}$
$tskstate : TaskState$
$tskpri : \mathbb{N}$
$multiactcnt : \mathbb{N}$
$tskdpri : \mathbb{N}$
$tskrscstate : RscState$
$tskrscid : \mathbb{N}$

In the data definition part of the test model, the corresponding structure type named *TASK* is defined as:

```
typedef TASK{
            byte tskid;
            TaskType tsktype;
            SchPol schpol;
            byte MAXMULTIACT;
            TaskState tskstate;
            byte tskpri;
            byte multiactcnt;
            byte tskdpri;
            RscState tskrscstate;
            byte tskrscid;}
```

The data type $\mathbb{N}$ in the schema *TASK* corresponds to the data type *byte* in the structure type *TASK*. Other data types in the schema are mapped to the corresponding fields of the structure type in a straightforward way.

5.3 OS Objects Part

The second part of test model is OS objects part. It consists of three parts, with each part has the relationship with the corresponding part in the formal test specification as shown in figure 5.1.

The first part is the declaration of the data objects used in the test model. Its corresponding part in the formal test specification is the signature part of the schema *OSEK_OS_Objects*. Notice that in the schema *OSEK_OS_Objects*, the concrete values of *TskNum* and *RscNum* are not given. However, in the test model, in order to declare the OS objects *Task* and *Resource*, it is necessary to determine the concrete values referred from the concrete system configuration.

The second part is the instantiation of the OS objects and other OS information which are used in the test model. Its corresponding parts in the formal test specification are the concrete system configuration and the predicate part of the schema *OSEK_OS_Objects*. To check some basic properties which have been specified in the predicate part of the schema *OSEK_OS_Objects*, the predicates are implemented in PROMELA. For example, one predicate in the schema *OSEK_OS_Objects* says that the task identifiers are different from each other:

$$\forall\, i, j : 1..TskNum \bullet i \neq j \Rightarrow (Task(i)).tskid \neq (Task(j)).tskid$$

The corresponding PROMELA descriptions are implemented as:

```
i = 1;
do
:: i < TskNum ->
          j = i + 1;
          do
          :: j < TskNum ->
               assert(Task[i].tskid != Task[j].tskid);
               j++;
          :: else -> break
          od;
          i++;
:: else -> break
od;
```

The third part is the implementation of the formalized static conformance requirements in the system configuration specification. Since the OS objects in the concrete system configuration should satisfy the requirements of its corresponding conformance class. By implementing the static conformance requirements, we can check whether or not a concrete system configuration is correct. For example, in the schema *Stc_Cfm_Req*, the requirement of conformance class BCC2 is specified as:

$$cfclass = BCC2 \Rightarrow (MaxTskMultiCnt \geqslant 1) \wedge (\forall\, i : 1...TskNum \bullet (Task(i)).tsktype = Basic)$$

The corresponding PROMELA descriptions are implemented as:

```
cfclass == BCC2 ->
   assert(MaxTskMultiCnt >= 1);
   i = 1;
   do
   :: i < TskNum ->
      assert(Task[i].tsktype == Basic);
      i++
   :: else -> break
   od;
```

5.4 Verification Part

Verification part is the main part of the test model. Its corresponding part in the formal test specification is the test purpose specification as shown in figure 5.1. As the definition of test purpose in section 4.4.2, test purpose is a set of dynamic conformance requirements. Therefore, one intended purpose of the verification part is to provide the mechanism to test whether or not the extracted dynamic conformance requirements are correct with respect to the design model of OSEK OS. On the other hand, if all the extracted conformance requirements are correct with respect to the design model of OSEK OS, by model checking the test model with the design model of OSEK OS, the exhaustive state space searching will be conducted and the witnesses will be generated. Therefore, another intended purpose of the verification part is to provide the mechanism to search the state space as exhaustive as possible. In this section, we present the details of how to construct the verification part so as to fulfill these two intentions.

At first, the definition of correctness of test purpose with respect to the design model of OSEK OS is given as:

By model checking the test model with the design model of OSEK OS, if the pre-state part and pre-condition part of a formalized test purpose are satisfied, after calling the system service specified in the action part of this test purpose, the expectant results specified in the post-state part and returnvalue part of the test purpose should be same as the corresponding actual results in the design model.

In the early stage of this work, we focus on assuring the correctness of test purposes. As a result, we have to implement the test models based on the formalized test purposes, respectively.

At that time, the verification part only consists of one implemented test purpose. The relationship between a formalized test purpose and its PROMELA implementation in the verification part is shown in figure 5.2.

To test whether or not a formalized test purpose is correct with respect to the design model, by model checking the test model with the design model of OSEK OS, the expectant result specified in the post-state part and returnvalue part of the formalized test purpose are checked against the corresponding actual result from the design model of OSEK OS. If the results are not same, SPIN should report a violation. The process are described as follows:

1. Instantiate the OS objects which can satisfy the pre-state and the pre-condition of the formalized test purpose. For the test purpose in the figure 5.2, two tasks named task1 and task2 can be instantiated, which the scheduling policy of task1 is full-preemptive and the priority of task1 is higher than the priority of task2.

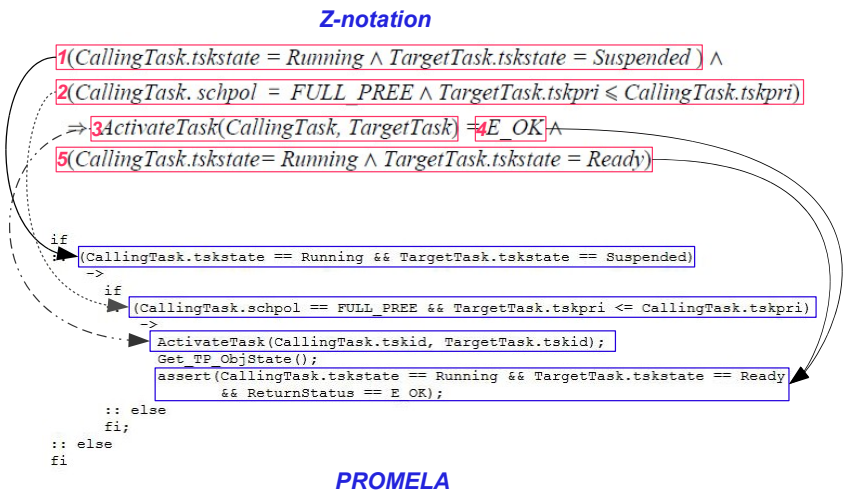

Figure 5.2: Relationship between a formalized test purpose and its implementation in the verification part

2. Set the OS objects to the pre-state of the test purpose. For example, let the state of task1 transfer into the running state by calling the system service.

3. Set the OS objects to the generic OS objects. For example, let task1 and task2 be the CallingTask and TargetTask, respectively.

4. Get the post-states of the OS objects from the design model after calling the system service.

5. Use assertion to check the consistency between the two results. For example, if the post-state of task1 is in the running state and the post-state of task2 is in the ready state, return value is E_OK, no violation will be reported. Otherwise, SPIN will report a violation to us.

On the one hand, such kind of test model based on only one test purpose is simple and the incorrect test purpose can be found out in an easy way. On the other hand, some drawbacks are also obvious. One drawback is that we must not only set the OS objects to the generic OS objects but also set them to the pre-state manually. Another drawback is that by model checking the test models, even if the test purpose is correct, the generated test case only consists of several transitions, which means that model checking are not used in an effective way. Thus, we need to further improve the verification part. The ideas of improvement are as follows:

- Translate all formalized test purposes into the verification part.

- After some test purpose is checked, by using a function, the generic OS objects can be reset, which may satisfy the pre-state and pre-condition of other test purpose.

According to the ideas above, the pseudo-code of improved verification part is shown as below:

```
do
:: Set_TP_Objs();
           if
           :: Block 1
           :: Block 2
           :: Block 3
           .
           .
           .
           :: Block n
           :: else
           fi;
od
```

The formalized test purposes that share the same pre-state and action are translated into the same block. The functionality of *Set_TP_Objs* is to set the OS objects to the generic OS objects, such as *CallingTask*, *TargetTask* and so on.

5.5 Translation Algorithm

To achieve the automatic translation from the formalized test purposes in the formal test specification to the verification part in the test model, we proposed a translation algorithm. In this section, we show the details of this translation algorithm and use an example to explain the mechanism of it.

Consider the test purpose specification as a set of formalized test purposes, which is denoted as:

$$TPS = \{TP_1, TP_2, ..., TP_n\}$$

According to the syntax of test purpose shown in Appendix B, we consider each formalized test purpose TP_i in the TPS as a set containing the Pre-State, Pre-Condition, Action, Post-State, ReturnValue, which is denoted as:

$$TP_i = \{Pre\text{-}State, Pre\text{-}Condition, Action, Post\text{-}State, ReturnValue\}$$

On the other hand, the verification part is considered as a set of blocks, which is denoted as:

$$VP = \{BLK_1, BLK_2, ..., BLK_m\}$$

Consider each block as a set that contains Bexp1, Aexp1 and segments, which is denoted as:

$$BLK_i = \{Bexp1, Aexp1, SEG_1, SEG_2, ..., SEG_k\}$$

Bexp1 corresponds to the Pre-state of test purpose. The corresponding part of Aexp1 in the test purpose is the Action.

Each segment in the block is considered as a set consisting of Bexp2, Bexp3, Bexp4, which is denoted as:

$$SEG_i = \{Bexp2, Bexp3, Bexp4\}$$

Bexp2, Bexp3, Bexp4 correspond to the Pre-Condition, Post-State and ReturnValue of the test purpose. The syntax of block, segment, Aexp1, Bexp1, Bexp2, Bexp3, Bexp4 can be referred from Appendix C (The syntax of verification part).

We use an example to demonstrate the translation process according to the translation algorithm.

Example

Suppose that there are three formalized test purposes in the test purpose specification as follows:

1. $(CallingTask.tskstate = Running \wedge TargetTask.tskstate = Suspended) \wedge$
   $(CallingTask.schpol = NON\_PREE)$
   $\Rightarrow ActivateTask(CallingTask, TargetTask) = E\_OK \wedge$
   $(CallingTask.tskstate = Running \wedge TargetTask.tskstate = Ready)$

2. $(CallingTask.tskstate = Running \wedge TargetTask.tskstate = Suspended) \wedge$
   $(CallingTask.schpol = FULL\_PREE \wedge TargetTask.tskpri \leqslant CallingTask.tskdpri)$
   $\Rightarrow ActivateTask(CallingTask, TargetTask) = E\_OK \wedge$
   $(CallingTask.tskstate = Running \wedge TargetTask.tskstate = Ready)$

3. $(CallingTask.tskstate = Running) \wedge$
   $(CallingTask.tskrscstate = Free \wedge CallingTask.multiactcnt \leqslant 1)$
   $\Rightarrow TerminateTask(CallingTask) = E\_OS\_NORET \wedge$
   $(CallingTask.tskstate = Suspended)$

Firstly, we translate the first formalized test purpose into the verification part. Each part of this formalized test purpose can be denoted as follows:

- $tp_1.Pre\text{-}State : CallingTask.tskstate = Running \wedge TargetTask.tskstate = Suspended$
- $tp_1.Pre\text{-}Condition : CallingTask.schpol = NON\_PREE$

Algorithm 1 Translation Algorithm

```
tps : TPS;
tps = {tp_1, tp_2, ..., tp_n};
VP_Blocks = {};
tp_i : TP;
blk : BLK;
seg : SEG;
function TransRule(tp : TP, blk : BLK, seg : SEG) {
        MapOp(tp);
        MapPar(tp);
        blk.Bexp1 = tp.Pre-State;
        blk.Aexp1 = tp.Action;
        seg.Bexp2 = tp.Pre-Condition;
        seg.Bexp3 = tp.Post-State;
        seg.(Bexp4.ReturnValue) = tp.ReturnValue;
    }
function TRANS() {
        i = 1;
        while(i ⩽ n) {
            TransRule(tp_i, blk, seg);
            if(In_VP_Blocks(blk.Bexp1, blk.Aexp1) == true) {
                blknum = Search_Blk(blk.Bexp1, blk.Aexp1);
                Add_Seg(blknum, blk.Aexp1, seg.Bexp2, seg.Bexp3, seg.Bexp4);
            }
            else {
                Add_VP_Blocks(blk.Bexp1, blk.Aexp1);
                blknum = Add_Blk(blk.Bexp1);
                Add_Seg(blknum, blk.Aexp1, seg.Bexp2, seg.Bexp3, seg.Bexp4);
            }
        }
    }
```

- $tp_1.Action$: $ActivateTask(CallingTask,\ TargetTask)$
- $tp_1.Post\text{-}State$: $CallingTask.tskstate = Running\ \wedge\ TargetTask.tskstate\ =\ Ready$
- $tp_1.ReturnValue$: E_OK

By applying the function *TransRule*, we can get:

- $blk.Bexp1$: $CallingTask.tskstate == Running\ \&\&\ TargetTask.tskstate\ ==\ Suspended$
- $blk.Aexp1$: $ActivateTask(CallingTask.tskid,\ TargetTask.tskid)$
- $seg.Bexp2$: $CallingTask.schpol == NON\_PREE$
- $seg.Bexp3$: $CallingTask.tskstate == Running\ \&\&\ TargetTask.tskstate\ ==\ Ready$
- $seg.Bexp4$: $ReturnStatus == E\_OK$

Notice that in the function *TransRule*, there are two other functions named *MapOp* and *MapPar*. By applying *MapOp*, the operations defined in the syntax of test purpose are changed to the operations that defined in the syntax of verification part. For example, '=' is changed to '==', and '∧' is changed to '&&', respectively. On the other hand, by applying *MapPar*, the parameters defined in the syntax of test purpose are changed to the parameters defined in the syntax of verification part. For the example above, '*CallingTask*' and '*TargetTask*' in the $tp_1.Action$ are changed to the '*CallingTask.tskid*' and '*TargetTask.tskid*' in the $blk.Aexp1$.

Since *VP_Blocks* is empty, *In_VP_Blocks* should return *false*. Then, $blk.Bexp1$ and $blk.Aexp1$ are added into the set *VP_Blocks* by applying *Add_VP_Blocks*.

After applying *Add_Blk* and *Add_Seg*, a new block with a new segment is translated into the verification part as depicted in figure 5.3.

```
do
:: Set_TP_Objs();
       if
       :: (CallingTask.tskstate == Running && TargetTask.tskstate == Suspended)   Block 1
          ->
             if
             :: (CallingTask.schpol == NON_PREE)                                  Segment 1
                ->
                   ActivateTask(CallingTask.tskid, TargetTask.tskid);
                   Get_TP_ObjState();
                   assert(CallingTask.tskstate == Running && TargetTask.tskstate == Ready
                          && ReturnStatus == E_OK);
             :: else
             fi;
       :: else
       fi;
od
```

Figure 5.3: Verification Part after first step translation

Then, the second formalized test purpose is translated into the verification part. Because

the Pre-State and the Action of this test purpose is same as those of the first one. It means that no block is needed to be added. After searching the block number, a new segment is added into the block identified by *blknum*. The PROMELA description of verification part after translating the second formalized test purpose is shown in figure 5.4.

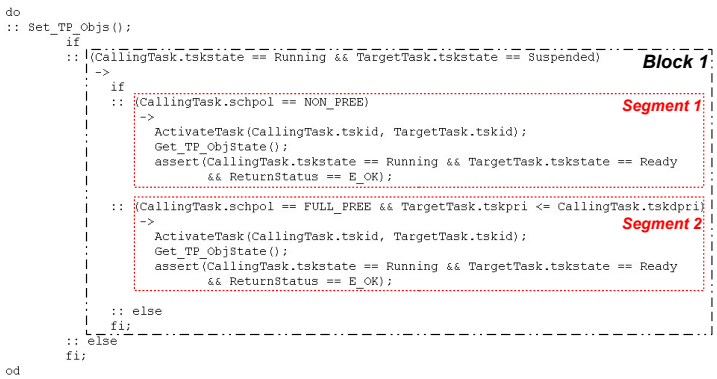

Figure 5.4: Verification Part after second step translation

At last, the last formalized test purpose is translated into the verification part. Although *VP_Blocks* is not empty, the Pre-State and Action of the last test purpose is different from those of the first test purpose. It means that a new block with a new segment is added into the verification part. The PROMELA description of verification part after translating the last formalized test purpose is depicted in figure 5.5.

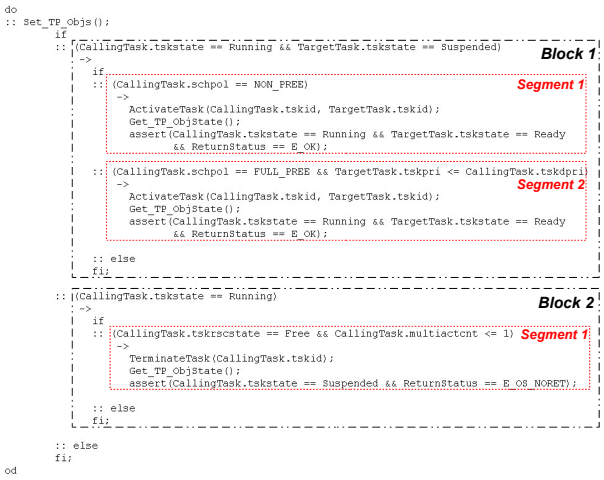

Figure 5.5: Verification Part after third step translation

Chapter 6

Experiment

According to the test generation process as shown in figure 3.5, the conformance requirements are extracted from the OSEK OS specification based on our knowledge of the specification. Therefore, it is possible that the extracted conformance requirements are not correct with respect to the design model of OSEK OS. By model checking the test model with the design model of OSEK OS, we can get feedback from the checking result. On the one hand, if the extracted conformance requirements are not correct, violations will be detected. Then, we can make correction of the corresponding conformance requirements by analysis of the checking result. On the other hand, if the extracted conformance requirements are correct with respect to the design model, the exhaustive state space searching of the test model will be conducted and the witnesses will be generated. Then, we can use TGT (Test Generation Tool) to derive the test cases from the witnesses.

To evaluate the proposed approach, test purposes have been developed from the system services in the scope of task management and resource management. Furthermore, test cases generated from the test model have been compared with the test cases defined in the MODISTARC.

6.1 Preparation for Experiment

The first task for preparing the experiments is to add some functions into the design model of OSEK OS, such as non-preemptive scheduling policy and nested resource occupation and so on. The aim is to let design model support all the functions specified in the test purposes.

The details of supporting the non-preemptive scheduling policy are as follows:

- Add a variable named *schpol* into the TCB(Task control block) in the design model. It identifies the scheduling policy of a task.
- When declaring a task, its scheduling policy is assigned to *schpol*.

- As an example, we show the automaton of refined ActivateTask in figure 6.1. Notice that inside the red rectangle, if a task is full-preemptive scheduling and the return status is E_OK, Schedule will be called, otherwise, do nothing.

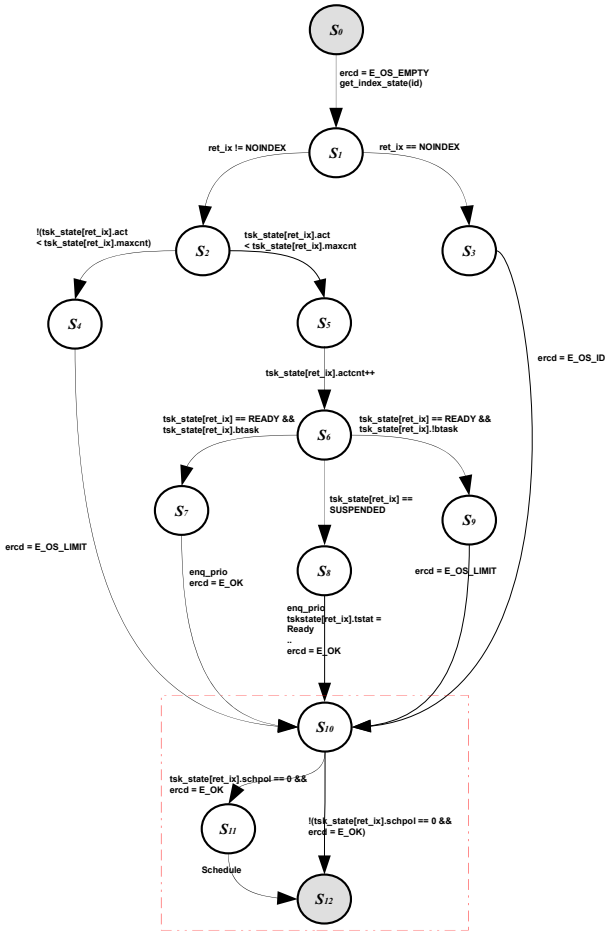

Figure 6.1: Automaton of refined ActivateTask system service

To support nested resource occupation, each task in the system is equipped with a stack. The stack is used to store the resource identifiers occupied by the corresponding task. In the case of getting a resource, the corresponding resource identifier will be pushed into the stack. If a task releases a resource, the corresponding resource identifier will be pop out from the stack.

The second task for preparing the experiments is to improve the TGT. The aim is to let the generated test cases consist of not only the called system services but also the

states of tasks and the corresponding return value. Figure 6.2 shows an example of test case generated by the improved TGT. Furthermore, test purposes can be traced from the generated test cases.

```
TestCase:
TASK_1: RUNNING
TASK_2: SUSPENDED
                        ActivateTask(2)
                        E_OK
                        TestPurpose: 102
TASK_1: RUNNING
TASK_2: READY
                        ActivateTask(2)
                        E_OS_LIMIT
                        TestPurpose: 105
TASK_1: RUNNING
TASK_2: READY
```

Figure 6.2: Test case generated by improved TGT

6.2 Development of Test Purposes

In the section 4.4, we have given the definition of test purpose and define the process of developing a test purpose from the system service.

Therefore, we need to develop the test purposes from the system service by following the process and then formalize them. After that, the test purposes will be translated into the test model and checked whether or not they are correct or not. If violations are detected, the test purposes will be corrected based on the checking results.

At first, we give several examples that show how to develop test purposes from the system services and make correction if a violation is detected.

Example 1

In this example, the test purpose was developed from the system service *ActivateTask*. The description of *ActivateTask* is shown in figure 4.2.

Some conformance requirements extracted from the system service are shown as follows:

1. If the called task is in the suspended state, it should be transferred into the ready state.

2. If the scheduling policy of the calling task is non-preemptive, no rescheduling should happen.

3. If the system service is called successfully, the return status should be E_OK.

Then, an example of generic test case designed based on the conformance requirements list above is shown in figure 6.3.

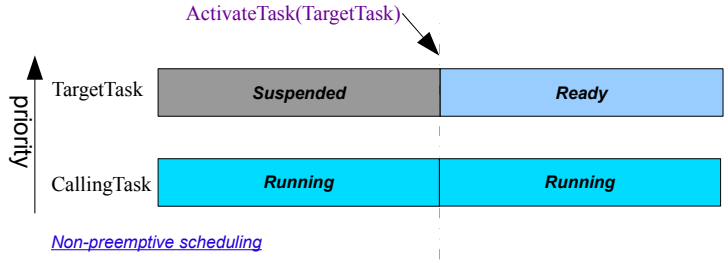

Figure 6.3: Generic test case for example 1

Test purpose derived from the generic test case above is as follows:

- pre-state: *CallingTask* is in the running state. *TargetTask* is in the suspended state.
- pre-condition: The scheduling policy of *CallingTask* is non-preemptive. The priority of *CallingTask* is lower than that of *TargetTask*
- system service: CallingTask calls *ActivateTask(TargetTask)*
- post-state: *CallingTask* is in the running state. *TargetTask* is in the ready state.
- return value: E_OK

The next step is to formalize this test purpose. The formalized test purpose is shown as below:

$(CallingTask.tskstate = Running \wedge TargetTask.tskstate = Suspended)$
$(CallingTask.schpol = NON\_PREE)$
$\Rightarrow ActivateTask(CallingTask, TargetTask) = E\_OK \wedge$
$(CallingTask.tskstate = Running \wedge TargetTask.tskstate = Ready)$

To check this test purpose, the OS objects in the test model can be instantiated as follows:

```
Task[1].tskid  = 1;
Task[1].schpol = NON_PREE;
Task[1].tskpri = 1;

Task[2].tskid  = 2;
Task[2].schpol = FULL_PREE;
Task[2].tskpri = 2;
```

By model checking the test model with the design model, no violation was reported and the test purpose is exactly covered by the generated test cases. It means that the test purpose is correct with the design model of OSEK OS.

Example 2

For the second example, we developed the test purpose from the system service *ChainTask*. The description of *ChainTask* can be referred from [gro05].

Some conformance requirements extracted from the system service *ChainTask* are shown as follows:

1. If the calling task does not have multiple activation requests and does not occupy any resource, it should be transferred into the suspended state.
2. If the called task is in the suspended state, it should be transferred into the ready state.
3. If the system service is called successfully, rescheduling should happen and no return value should be returned.

Then, an example of generic test case designed based on the conformance requirements list above is shown in figure 6.4.

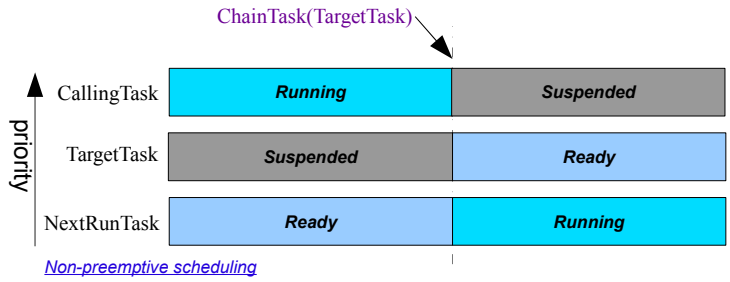

Figure 6.4: Generic test case for example 2

In the generic test case above, the task which is the first task in the ready queue is regarded as the *NextRunTask*. Suppose that the scheduling policy of *NextRunTask* is non-preemptive, and its priority is lower than other two tasks in the system.

Test purpose derived from the generic test above is as follows:

- pre-state: *CallingTask* is in the running state. *TargetTask* is in the suspended state.

- pre-condition: *CallingTask* does not have multiple activation requests and does not occupy any resource. The scheduling policy of *NextRunTask* is non-preemptive. The priority of *NextRunTask* is lower than those of *CallingTask* and *TargetTask*. The priority of *TargetTask* is lower than that of *CallingTask*.
- system service: CallingTask calls *ChainTask(TargetTask)*
- post-state: *CallingTask* is in the suspended state. *TargetTask* is in the ready state.
- return value: no return value is returned

The formalized test purpose is shown as below:

$(CallingTask.tskstate = Running \wedge TargetTask.tskstate = Suspended)$
$(CallingTask.tskrscstate = Free \wedge CallingTask.multiactcnt \leqslant 1 \wedge$
$NextRunTask.tskpri < TargetTask.tskpri \wedge NextRunTask.schpol = NON\_PREE)$
$\Rightarrow ChainTask(CallingTask, TargetTask) = E\_OS\_NORET \wedge$
$(CallingTask.tskstate = Suspended \wedge TargetTask.tskstate = Ready)$

To check this test purpose, the OS objects in the test model can be instantiated as follows:

```
Task[1].tskid  = 1;
Task[1].schpol = FULL_PREE;
Task[1].tskpri = 3;

Task[2].tskid  = 2;
Task[2].schpol = FULL_PREE;
Task[2].tskpri = 2;

Task[3].tskid  = 3;
Task[3].schpol = NON_PREE;
Task[3].tskpri = 1;
```

By model checking the test model with the design model, a violation was detected and SPIN outputted the checking result as follows:

```
pan: assertion violated (((CallingTask.tskstate==1)&&(TargetTask.tskstate==2))
                         &&(ReturnStatus==9)) (at depth 7040)
```

According to the checking result, this test purpose is not correct with respect to the design model of OSEK. After analysis of the system service in the specification and the design model more carefully, we found the reason. Our original understanding is that the rescheduling of *ChainTask* is ahead of activating the target task. Thus *NextRunTask*

will be transferred into the running state as soon as *CallingTask* is terminated, after that even though *TargetTask* is activated and the priority of *TargetTask* is higher than that of *NextRunTask*, due to the non-preemptive scheduling policy of *NextRunTask*, *TargetTask* should still in the ready state. However, in the design model, the rescheduling is after activating the target task. This means that the state of target task should be transferred into the running state due to the higher priority. Therefore, we corrected the formalized test purpose as follows:

$(CallingTask.tskstate = Running \wedge TargetTask.tskstate = Suspended)$
$(CallingTask.tskrscstate = Free \wedge CallingTask.multiactcnt \leqslant 1 \wedge$
$NextRunTask.tskpri < TargetTask.tskpri)$
$\Rightarrow ChainTask(CallingTask, TargetTask) = E\_OS\_NORET \wedge$
$(CallingTask.tskstate = Suspended \wedge TargetTask.tskstate = Running)$

By model checking the test model with the design model, no violation was detected and the test purpose was exactly covered by the generated test cases. It means that the test purpose is correct with the design model of OSEK OS.

Example 3

For the third example, the test purpose was developed from the system service *GetResource*. The description of *GetResource* can be referred from [gro05].

The conformance requirements extracted from the system service *GetResource* are shown as follows:

1. If the target resource is not occupied by any task and the priority of calling task is not higher than the ceiling priority of the target resource, the target resource will be occupied by the calling task.

2. If the system service is called successfully, the return status should be E_OK.

Then, an example of generic test case designed based on the conformance requirements list above is shown in figure 6.5.

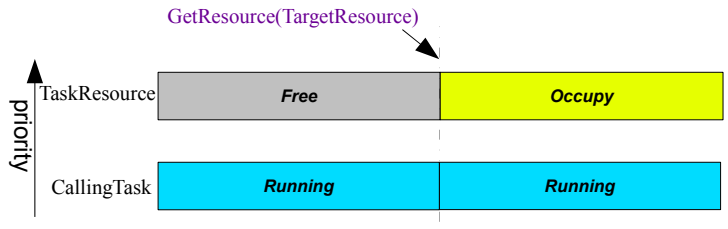

Figure 6.5: Generic test case for example 3

Test purpose derived from the generic test above is shown as follows:

- pre-state: *CallingTask* is in the running state. *TargetResource* is free.
- pre-condition: The priority of *CallingTask* is not higher than the ceiling priority of *TargetResource*
- system service: CallingTask calls *GetResource(TargetResource)*
- post-state: *CallingTask* is in the running state. *TargetResource* is occupied.
- return value: E_OK

The formalized test purpose is shown as below:

$(CallingTask.tskstate = Running \wedge TargetRsc.rscstate = Free)$
$(CallingTask.tskpri \leqslant TargetRsc.rscpri)$
$\Rightarrow GetResource(CallingTask, TargetRsc) = E\_OK \wedge$
$(CallingTask.tskstate = Running \wedge TargetRsc.rscstate = Occupy)$

To check this test purpose, the OS objects in the test model can be instantiated as follows:

```
Task[1].tskid  = 1;
Task[1].schpol = FULL_PREE;
Task[1].tskpri = 1;

Resource[1].rscid  = 1;
Resource[1].rscpri = 2;
```

By model checking the test model with the design model, no violation was reported and the test purpose was exactly covered by the generated test cases. It means that the test purpose is correct with the design model of OSEK OS.

However, when we tried to instantiate another task like that:

```
Task[2].tskid  = 2;
Task[2].schpol = FULL_PREE;
Task[2].tskpri = 3;
```

The priority of Task[2] is higher than that of Task[1]. By model checking the test model with the design model again, a violation was detected and SPIN outputted the checking result as follows:

```
pan: assertion violated (((CallingTask.tskstate==2)&&(TargetTask.tskstate==3))
                         &&(ReturnStatus==0)) (at depth 4441)
```

From the checking result, we found that the violation is not caused by the new added test purpose which related to the *GetResource*. After analysis of the generated test cases, we found the reason of violation was related to another test purpose as below:

$(CallingTask.tskstate = Running \wedge TargetTask.tskstate = Suspended)$
$(CallingTask.schpol = FULL\_PREE \wedge TargetTask.tskpri > CallingTask.tskpri)$
$\Rightarrow ActivateTask(CallingTask, TargetTask) = E\_OK \wedge$
$(CallingTask.tskstate = Ready \wedge TargetTask.tskstate = Running)$

In this test purpose, it says that if the priority of *TargetTask* is higher than that of *CallingTask* and the scheduling policy of *CallingTask* is full-preemptive, then *CallingTask* will be transferred into the ready state while *TargetTask* will be transferred into the running state. Nevertheless, what will happen if *CallingTask* occupying a resource that the ceiling priority of this resource is not lower than the priority of *TargetTask*. For this example, suppose that Task[1] is occupying the Resource[1], according to the ceiling protocol, the priority of Task[1] should be raised to the ceiling priority of the Resource[1]. Therefore, after calling *ActivateTask*, *CallingTask* should be still in the running state while *TargetTask* will be transferred into the ready state because the current priority of Task[1] is equal to the priority of Task[2]. Thus we made correction of this test purpose as follows:

$(CallingTask.tskstate = Running \wedge TargetTask.tskstate = Suspended)$
$(CallingTask.schpol = FULL\_PREE \wedge TargetTask.tskpri > CallingTask.tskdpri)$
$\Rightarrow ActivateTask(CallingTask, TargetTask) = E\_OK \wedge$
$(CallingTask.tskstate = Ready \wedge TargetTask.tskstate = Running)$

In the corrected test purpose above, *CallingTask.tskdpri* means the current priority of *CallingTask*. If *CallingTask* does not occupy any resource, its value should be equal to the initial priority, otherwise, it should be equal to the highest ceiling priority of the resource among which are occupied by *CallingTask*. On the other hand, to check the corrected test purpose, the priority of Task[2] was changed to 3, which is higher than the ceiling priority of Resource[1], so that the pre-condition can be satisfied in the case of Task[1] occupies Resource[1].

By model checking the test model with the design model, no violation was reported and the test purpose was exactly covered by the generated test cases. It means that the test purpose is correct with the design model of OSEK OS.

In this work, we have developed the test purposes from the system services in the scope of task management and resource management, including *ActivateTask, TerminateTask, ChainTask, GetResource, ReleaseResource*. The total numbers of developed test purposes for each system service are summarized in table 6.1. The formalization of them can be referred from Appendix A.

System Service	Num. of. TP
ActivateTask	7
TerminateTask	4
ChainTask	21
GetResource	3
ReleaseResource	7

Table 6.1: Numbers of Formalized Test Purposes

To get sufficient confidence of the correctness of the formalized test purposes summarized in the table above, we need to check them in a more systematic way. On the one hand, it is almost impossible to check the formalized test purposes under all the variants of concrete system configuration. For example, suppose that the test environment consists of three tasks without resource, there exists more than 2000 variants. On the other hand, we still can choose some variants within some bounded range to check the formalized test purposes. It needs us to make some selection strategies to choose the variants.

At first, task patterns in the OSEK OS can be summarized as shown in table 6.2.

P.N \ KEY	TaskType	MultiAct	SchPol
Ptn01	Basic	×	Full
Ptn02	Basic	×	Non
Ptn03	Basic	√	Full
Ptn04	Basic	√	Non
Ptn05	Extended	×	Full
Ptn06	Extended	×	Non

Table 6.2: Task Patterns of OSEK OS

As shown in the table above, there are six patterns of task in the OSEK OS. Since the main difference between basic tasks and extended tasks is that extended tasks have waiting state so as to be allowed to wait for events, in the scope of task management and resource management, the functions of two tasks are same. Therefore, basic tasks are selected. On the other hand, multiple activation requests includes the situation of limiting to one activation request, thus the task patterns that support multiple activation requests are selected. Then, we make the selection strategies as follows:

1. Selected task patterns: Ptn03 and Ptn04 (listed in table 6.2)

2. Priority relationship: priority of non-preemptive task is lower than that of full-preemptive task.

3. Add resource into the concrete system configuration which can cover the test purposes most.

The results of checking the formalized test purpose can be referred from Appendix D. Here, we explain the meaning of pattern name in each result table.

For the case of *Tsk:3 Rsc:0*, the meaning of pattern name, for example *Ptn*03_*Ptn*04_*Ptn*03 (GT_LT_EQ) is as follows:

- Ptn03_Ptn04_Ptn03: Task1 is the task of Ptn03, Task2 is the task of Ptn04, Task3 is the task of Ptn03.
- GT: priority of Task1 is higher than that of Task2.
- LT: priority of Task2 is lower than that of Task3.
- EQ: priority of Task1 is equal to that of Task3.

For the case of *Tsk:3 Rsc:1*, the meaning of pattern name, for example *Ptn*03_*Ptn*04_*Ptn*03_ $GT\_LT\_GT(GT\_LT\_EQ)$ is as follows:

- Ptn03_Ptn04_Ptn03_GT_LT_GT: Task1 is the task of Ptn03, Task2 is the task of Ptn04, Task3 is the task of Ptn03. Priority of Task1 is higher than that of Task2. Priority of Task2 is lower than that of Task3. Priority of Task1 is higher than that of Task3
- GT: priority of Task1 is higher than the ceiling priority of Resource 1.
- LT: priority of Task2 is lower than the ceiling priority of Resource 1.
- EQ: priority of Task3 is equal to the ceiling priority of Resource 1.

For the case of *Tsk:3 Rsc:2*, the meaning of pattern name, for example *Ptn*03_*Ptn*04_*Ptn*03_ $LT\_LT\_LT(LT\_LT\_GT)(LT\_LT\_EQ\_LT)$ is as follows:

- Ptn03_Ptn04_Ptn03_LT_LT_LT(LT_LT_GT): Task1 is the task of Ptn03, Task2 is the task of Ptn04, Task3 is the task of Ptn03. Priority of Task1 is lower than that of Task2. Priority of Task2 is lower than that of Task3. Priority of Task1 is lower than that of Task3. Priority of Task1 is lower than the ceiling priority of Resource 1. Priority of Task2 is lower than the ceiling priority of Resource 1. Priority of Task3 is higher than the ceiling priority of Resource 1.
- LT: Priority of Task1 is lower than the ceiling priority of Resource 2.
- LT: Priority of Task2 is lower than the ceiling priority of Resource 2.

- EQ: Priority of Task3 is equal to the ceiling priority of Resource 2.
- LT: Ceiling priority of Resource 1 is lower than the ceiling priority of Resource 2.

From the result table of *Tsk:3 Rsc:0*, 44 variants are selected in the case of task number is 3 and no resource according to the selection strategies. The variants that in the blue color can cover the test purposes most. Therefore, we add the resource into these concrete system configurations.

From the result table of *Tsk:3 Rsc:1*, 7 variants which can cover all the possibilities of the priority relationship among the tasks and resources are selected for each variants selected from the result table *Tsk:3 Rsc:0*. The variants that in the blue color can cover the test purposes most. Therefore, we add another resource into this concrete system configuration.

We have checked all the formalized test purposes under 94 variants. All the formalized test purposes have been covered and checked. Furthermore, no violation was detected, so that we have gotten sufficient confidence of the correctness of the test purposes.

6.3 Comparison with MODISTARC

To evaluate the proposed approach, we tried to compare the test cases generated from test model with the test cases defined in the MODISTARC. In order to compare them, we need to give some definitions beforehand.

Firstly, system state is defined as a set of tasks states. For example, if there are three tasks in a system, which are identified by task1, task2, and task3. Suppose that task1 is in the running state, task2 is in the ready state and task3 is in the suspended state. System state is denoted like (Run, Rdy, Sus), where *Run* represents that task1 is in the running state, *Rdy* represents that task2 is in the ready state, *Sus* represents that task3 is in the suspended state. Then, pre-state is defined as the system state before calling a system service, while post-state is defined as the system state after calling a system service.

Based on the definitions above, we can consider the system transition as a set containing pre-state, system service, post-state which can be denoted as:

$$ST = \{pre\text{-}state, systemservice, post\text{-}state\}$$

Then, test case is considered as a set of system transitions, which is denoted as:

$$TC = \{ST_1, ST_2, ..., ST_n\}$$

For the example of test case shown in the figure 6.2, it can be denoted as follows:

$$ST_1 = \{(Run, Sus), ActivateTask(Task2), (Run, Rdy)\}$$
$$ST_2 = \{(Run, Rdy), ActivateTask(Task2), (Run, Rdy)\}$$
$$TC = \{ST_1, ST_2\}$$

We choose a test case defined in the MODISTARC as shown in figure 6.6. According to the definition above, this test case can be considered as a set of system transitions, which are denoted as follows:

$$ST_1 = \{(Run, Sus, Sus), ActivateTask(Task3), (Rdy, Sus, Run)\}$$
$$ST_2 = \{(Rdy, Sus, Run), ActivateTask(Task2), (Rdy, Rdy, Run)\}$$
$$ST_3 = \{(Rdy, Rdy, Run), TerminateTask(Task3), (Rdy, Run, Sus)\}$$
$$ST_4 = \{(Rdy, Run, Sus), TerminateTask(Task2), (Run, Sus, Sus)\}$$
$$ST_5 = \{(Run, Sus, Sus), TerminateTask(Task1), (Sus, Sus, Sus)\}$$
$$TC_{modi} = \{ST_1, ST_2, ST_3, ST_4, ST_5\}$$

Tasks:

Task1		
	type:	basic
	autostart:	yes
	priority:	1
	max. activations:	1
	preemptive:	full
Task2		
	type:	basic
	autostart:	no
	priority:	2
	max. activations:	1
	preemptive:	full
Task3		
	type:	basic
	autostart:	no
	priority:	3
	max. activations:	1
	preemptive:	full

Running task	Called OS service	Return status	Test case
Task1	`ActivateTask(Task3)`	`E_OK`	3
Task3	`ActivateTask(Task2)`	`E_OK`	4
Task3	`TerminateTask()`		
Task2	`TerminateTask()`		
Task1	`TerminateTask()`		

Figure 6.6: Test Case defined in the MODISTARC

On the other hand, in order to compare the test cases generated from the test model with the test case above, the OS objects in the test model are needed to be instantiated as follows:

```
Task[1].tskid  = 1;
Task[1].schpol = FULL_PREE;
Task[1].tskpri = 1;

Task[2].tskid  = 2;
Task[2].schpol = FULL_PREE;
Task[2].tskpri = 2;

Task[3].tskid  = 3;
Task[3].schpol = FULL_PREE;
Task[3].tskpri = 3;
```

Furthermore, we have constructed the test model based on the formalized test purposes which are implied by the test case in figure 6.6. Three formalized test purposes are selected as follows:

1. $(CallingTask.tskstate = Running \wedge TargetTask.tskstate = Suspended)$
   $(CallingTask.schpol = FULL\_PREE \wedge TargetTask.tskpri \leqslant CallingTask.tskdpri)$
   $\Rightarrow ActivateTask(CallingTask, TargetTask) = E\_OK \wedge$
   $(CallingTask.tskstate = Running \wedge TargetTask.tskstate = Ready)$

2. $(CallingTask.tskstate = Running \wedge TargetTask.tskstate = Suspended)$
   $(CallingTask.schpol = FULL\_PREE \wedge TargetTask.tskpri > CallingTask.tskdpri)$
   $\Rightarrow ActivateTask(CallingTask, TargetTask) = E\_OK \wedge$
   $(CallingTask.tskstate = Ready \wedge TargetTask.tskstate = Running)$

3. $(CallingTask.tskstate = Running)$
   $(CallingTask.tskrscstate = Free \wedge CallingTask.multiactcnt \leqslant 1)$
   $\Rightarrow TerminateTask(CallingTask) = E\_OS\_NORET \wedge$
   $(CallingTask.tskstate = Suspended)$

By model checking the test model with the design model of OSEK OS, no violation was detected, instead, the exhaustive state space searching was conducted and the witnesses were generated. Then, test case was derived from the witnesses by the improved TGT.

After comparing the system states in the generated test case with the system states in the TC_{modi}, we conclude that TC_{modi} is a subset of the test case generated from the test model.

6.4 Evaluation

By developing the test purpose from the system services, we showed that the proposed approach provides an effective way to assure the correctness of test purposes with respect

to the design model of OSEK OS. The error message outputted by SPIN can help us to trace the incorrect test purpose in the test model. Furthermore, the actual result from the design model provides useful debug information for correction of the test purpose. Thus we make full use of the design model of OSEK OS to assure the correctness of test purposes. On the other hand, since it is impossible to check the test purposes under infinite variants, in other words, we can only assure the correctness of test purposes in the range of selected variants.

With regards to the generated test case from the test model, each system transition in a test case can achieve a corresponding test purpose. This promises the correctness of test cases. Moreover, by comparing with the test cases in the MODISTARC, we showed that test case generated from test model includes the test case defined in the MODISTARC if the developed test purposes can cover the conformance requirement implied by the test case define in the MODISTARC. However, since the test cases are derived from the witnesses, it results in generating considerable identical test cases. If a test case is a prefix of another test case, it is not necessary to select the shorter one.

Chapter 7

Conclusion

7.1 Summary

In this work, an original approach for automatic generation of test cases from the test model using model checking has been proposed. The main features of the proposed approach are summarized as follows:

- Conformance requirements can be extracted from the OSEK OS specification, since we have given the meaning of conformance and defined the conformance requirements in the context of OSEK OS.
- Conformance requirements can be described in a precise and unambiguous manner. Static conformance requirements and the test purposes developed from the system service in the scope of task management and resource management have been formalized in the Z-notation.
- Test model can be constructed in a systematic way. Test model has been constructed based on the formal test specification and a translation algorithm have been proposed to achieve the automatic translation from the formalized test purposes into the verification part in the test model.
- Test cases can be automatic generated from the test model. By model checking the test model with the design model, test cases can be derived from the witnesses of checking result by TGT. The generated test cases can be assured to achieve the test purposes.

7.2 Discussion

In the course of this work, a few problems have been discussed. One is the test oracle problem. In general, test oracle is used to generate expected results for each test input. However, in this work, the design model of OSEK OS has been used as a test oracle in an ad hoc way. We compare the results in the design model with the expectant result in

the test purpose to determine whether the test purpose is correct or not. The intention is that, comparing with the test purposes which have been developed based on our knowledge of the specification, the design model is a much more reliable formalism that promises the consistency with the specification. Even though it is rarely possible to prove the functional equivalence between design model and the specification. Another problem is the exhaustiveness of test cases. On the one hand, we have generated test cases from the test model by using the exhaustive state space searching function of model checking tool. It is reasonable to say that the generated test cases are exhaustive. However, on the other hand, such kind of exhaustiveness should be formal defined. Furthermore, the exhaustiveness problem can be discussed from different point of view. From the theoretics point of view, exhaustive space can be formally defined based on the system states and transitions. From practice point of view, the full conformance requirements can be used to define the exhaustive space.

7.3 Future Works

This work can be extended in several directions as follows:

- Develop test purposes from other functionality of OSEK OS, such as event management and interrupt management.
- Develop a formal framework of conformance testing for OSEK OS. In this thesis, we have given the definitions of conformance requirements, test purpose and the correctness of test purpose in natural language. As for the future work, formal definitions of them can be given and formal prove can be conducted to show the correctness of test purpose.
- Implement a tool to translate formal test specification into the test model. We have given a translation algorithm to translate the formalized test purposes into the verification part in the test model. The whole part of translation can be achieved by implementing a tool.
- Develop a technique to the problem of test case selection. Test cases derived from the witnesses result in identical test cases. If a test case is a prefix of another test case, it is not necessary to select the shorter one.

Appendix A

Formal Test Specification

A.1 Data Definition and Data Declaration

$CFClass ::= BCC1 \mid BCC2 \mid ECC1 \mid ECC2$

$StatusType ::= E\_OK \mid E\_OS\_ACCESS \mid E\_OS\_CALLEVEL \mid E\_OS\_ID$
$\mid E\_OS\_LIMIT \mid E\_OS\_NOFUNC \mid E\_OS\_RESOURCE$
$\mid E\_OS\_STATE \mid E\_OS\_VALUE \mid E\_OS\_NORET$

$TaskType ::= Basic \mid Extended$

$TaskState ::= Suspended \mid Ready \mid Running \mid Waiting$

$SchPol ::= NON\_PREE \mid FULL\_PREE$

$RscState ::= Free \mid Occupy$

TASK

$tskid : \mathbb{N}$
$tsktype : TaskType$
$schpol : SchPol$
$MAXMULTIACT : \mathbb{N}$
$tskstate : TaskState$
$tskpri : \mathbb{N}$
$multiactcnt : \mathbb{N}$
$tskdpri : \mathbb{N}$
$tskrscstate : RscState$
$tskrscid : \mathbb{N}$

RESOURCE

$rscid : \mathbb{N}$
$rscstate : RscState$
$rscpri : \mathbb{N}$
$rsctskid : \mathbb{N}$

OSEK_OS_Objects

$TskNum : \mathbb{N}$
$RscNum : \mathbb{N}$
$Task : \mathbb{N}_1 \rightarrow TASK$
$Resource : \mathbb{N}_1 \rightarrow RESOURCE$
$cfclass : CFClass$
$MaxTskMultiCnt : \mathbb{N}_1$
$tskempty : \mathbb{N}$
$rscempty : \mathbb{N}$

$\forall i, j : 1..TskNum \bullet i \neq j \Rightarrow (Task(i)).tskid \neq (Task(j)).tskid$
$\forall i, j : 1..RscNum \bullet i \neq j \Rightarrow (Resource(i)).rscid \neq (Resource(j)).rscid$
$\forall i : 1..TskNum \bullet (Task(i)).tskid \neq tskempty$
$\forall i : 1..RscNum \bullet (Resource(i)).rscid \neq rscempty$

A.2 System Configuration Specification

Stc_Cfm_Req

OSEK_OS_Objects

$\forall\, i : 1..TskNum \bullet (Task(i)).tsktype = Extended \Rightarrow (Task(i)).MAXMULTIACT = 1$
$\forall\, i : 1..TskNum \bullet 0 < (Task(i)).MAXMULTIACT \leqslant MaxTskMultiCnt$

$cfclass = BCC1 \Rightarrow (MaxTskMultiCnt = 1) \wedge$
$(\forall\, i : 1..TskNum \bullet (Task(i)).tsktype = Basic) \wedge$
$(\forall\, i, j : 1..TskNum \bullet i \neq j \Rightarrow (Task(i)).tskpri \neq (Task(j)).tskpri)$

$cfclass = BCC2 \Rightarrow (MaxTskMultiCnt \geqslant 1) \wedge$
$(\forall\, i : 1..TskNum \bullet (Task(i)).tsktype = Basic)$

$cfclass = ECC1 \Rightarrow (MaxTskMultiCnt = 1) \wedge$
$(\forall\, i : 1..TskNum \bullet ((Task(i)).tsktype = Basic \vee (Task(i)).tsktype = Extended)) \wedge$
$(\forall\, i, j : 1..TskNum \bullet i \neq j \Rightarrow (Task(i)).tskpri \neq (Task(j)).tskpri)$

$cfclass = ECC2 \Rightarrow (MaxTskMultiCnt \geqslant 1) \wedge$
$(\forall\, i : 1..TskNum \bullet ((Task(i)).tsktype = Basic \vee (Task(i)).tsktype = Extended))$

A.3 Test Purpose Specification

Test_Purpose_Objects

$OSEK\_OS\_Objects$
$CallingTask : TASK$
$TargetTask : TASK$
$NextRunTask : TASK$
$EmptyTask : TASK$
$ReadyQueue : \text{seq}\ TASK$
$TargetRsc : RESOURCE$

$EmptyTask.tskid = tskempty$

$\exists_1\, i : 1..TskNum \bullet (Task(i)).tskstate = Running \Rightarrow CallingTask = (Task(i))$
$\neg\, (\exists\, i : 1..TskNum \bullet (Task(i)).tskstate = Running) \Rightarrow CallingTask = EmptyTask$

$\forall\, i : 1..TskNum \bullet (Task(i)).tskstate \neq Running \land CallingTask \neq EmptyTask$
$\Rightarrow TargetTask = (Task(i))$
$CallingTask = EmptyTask \Rightarrow TargetTask = EmptyTask$

$ReadyQueue \neq \langle\rangle \Rightarrow (\exists_1\, i : 1..TskNum \bullet head\ ReadyQueue = (Task(i))$
$\Rightarrow NextRunTask = (Task(i)))$
$ReadyQueue = \langle\rangle \Rightarrow NextRunTask = EmptyTask$

$\forall\, i : 1..RscNum \bullet TargetRsc = (Resource(i))$

Test_Purpose_SysCalls

$Test\_Purpose\_Objects$
$ActivateTask : TASK \times TASK \rightarrow StatusType$
$TerminateTask : TASK \rightarrow StatusType$
$ChainTask : TASK \times TASK \rightarrow StatusType$
$GetResource : TASK \times RESOURCE \rightarrow StatusType$
$ReleaseResource : TASK \times RESOURCE \rightarrow StatusType$

$CallingTask.tskstate = Running \Leftrightarrow (CallingTask, TargetTask) \in \text{dom}\ ActivateTask$
$CallingTask.tskstate = Running \Leftrightarrow (CallingTask, CallingTask) \in \text{dom}\ ActivateTask$
$CallingTask.tskstate = Running \Leftrightarrow (CallingTask) \in \text{dom}\ TerminateTask$
$CallingTask.tskstate = Running \Leftrightarrow (CallingTask, TargetTask) \in \text{dom}\ ChainTask$
$CallingTask.tskstate = Running \Leftrightarrow (CallingTask, CallingTask) \in \text{dom}\ ChainTask$
$CallingTask.tskstate = Running \Leftrightarrow (CallingTask, TargetRsc) \in \text{dom}\ GetResource$
$CallingTask.tskstate = Running \Leftrightarrow (CallingTask, TargetRsc) \in \text{dom}\ ReleaseResource$

TP_for_ActivateTask

Test_Purpose_SysCalls

$$
\begin{array}{l}
(CallingTask.tskstate = Running \wedge TargetTask.tskstate = Suspended) \wedge \\
(CallingTask.schpol = NON\_PREE) \\
\Rightarrow ActivateTask(CallingTask, TargetTask) = E\_OK \wedge \\
(CallingTask.tskstate = Running \wedge TargetTask.tskstate = Ready)
\end{array}
$$

$$
\begin{array}{l}
(CallingTask.tskstate = Running \wedge TargetTask.tskstate = Suspended) \wedge \\
(CallingTask.schpol = FULL\_PREE \wedge TargetTask.tskpri \leqslant CallingTask.tskdpri) \\
\Rightarrow ActivateTask(CallingTask, TargetTask) = E\_OK \wedge \\
(CallingTask.tskstate = Running \wedge TargetTask.tskstate = Ready)
\end{array}
$$

$$
\begin{array}{l}
(CallingTask.tskstate = Running \wedge TargetTask.tskstate = Suspended) \wedge \\
(CallingTask.schpol = FULL\_PREE \wedge TargetTask.tskpri > CallingTask.tskdpri) \\
\Rightarrow ActivateTask(CallingTask, TargetTask) = E\_OK \wedge \\
(CallingTask.tskstate = Ready \wedge TargetTask.tskstate = Running)
\end{array}
$$

$$
\begin{array}{l}
(CallingTask.tskstate = Running \wedge TargetTask.tskstate = Ready) \wedge \\
(TargetTask.multiactcnt < TargetTask.MAXMULTIACT) \\
\Rightarrow ActivateTask(CallingTask, TargetTask) = E\_OK \wedge \\
(CallingTask.tskstate = Running \wedge TargetTask.tskstate = Ready)
\end{array}
$$

$$
\begin{array}{l}
(CallingTask.tskstate = Running \wedge TargetTask.tskstate = Ready) \wedge \\
(TargetTask.multiactcnt = TargetTask.MAXMULTIACT) \\
\Rightarrow ActivateTask(CallingTask, TargetTask) = E\_OS\_LIMIT \wedge \\
(CallingTask.tskstate = Running \wedge TargetTask.tskstate = Ready)
\end{array}
$$

$$
\begin{array}{l}
(CallingTask.tskstate = Running) \wedge \\
(CallingTask.multiactcnt < CallingTask.MAXMULTIACT) \\
\Rightarrow ActivateTask(CallingTask, CallingTask) = E\_OK \wedge \\
(CallingTask.tskstate = Running)
\end{array}
$$

$$
\begin{array}{l}
(CallingTask.tskstate = Running) \wedge \\
(CallingTask.multiactcnt = CallingTask.MAXMULTIACT) \\
\Rightarrow ActivateTask(CallingTask, CallingTask) = E\_OS\_LIMIT \wedge \\
(CallingTask.tskstate = Running)
\end{array}
$$

TP_for_TerminateTask

Test_Purpose_SysCalls

$$
\begin{aligned}
&(CallingTask.tskstate = Running) \wedge \\
&(CallingTask.tskrscstate = Occupy) \\
&\Rightarrow TerminateTask(CallingTask) = E\_OS\_RESOURCE \wedge \\
&(CallingTask.tskstate = Running)
\end{aligned}
$$

$$
\begin{aligned}
&(CallingTask.tskstate = Running) \wedge \\
&(CallingTask.tskrscstate = Free \wedge CallingTask.multiactcnt \leqslant 1) \\
&\Rightarrow TerminateTask(CallingTask) = E\_OS\_NORET \wedge \\
&(CallingTask.tskstate = Suspended)
\end{aligned}
$$

$$
\begin{aligned}
&(CallingTask.tskstate = Running) \wedge \\
&(CallingTask.tskrscstate = Free \wedge CallingTask.multiactcnt > 1 \wedge \\
&CallingTask.tskid \neq NextRunTask.tskid) \\
&\Rightarrow TerminateTask(CallingTask) = E\_OS\_NORET \wedge \\
&(CallingTask.tskstate = Ready)
\end{aligned}
$$

$$
\begin{aligned}
&(CallingTask.tskstate = Running) \wedge \\
&(CallingTask.tskrscstate = Free \wedge CallingTask.multiactcnt > 1 \wedge \\
&CallingTask.tskid = NextRunTask.tskid) \\
&\Rightarrow TerminateTask(CallingTask) = E\_OS\_NORET \wedge \\
&(CallingTask.tskstate = Running)
\end{aligned}
$$

*TP_for_ChainTask_Part*01

Test_Purpose_SysCalls

$(CallingTask.tskstate = Running \wedge TargetTask.tskstate = Suspended) \wedge$
$(CallingTask.tskrscstate = Free \wedge CallingTask.multiactcnt \leqslant 1 \wedge$
$NextRunTask.tskid = tskempty)$
$\Rightarrow ChainTask(CallingTask, TargetTask) = E\_OS\_NORET \wedge$
$(CallingTask.tskstate = Suspended \wedge TargetTask.tskstate = Running)$

$(CallingTask.tskstate = Running \wedge TargetTask.tskstate = Suspended) \wedge$
$(CallingTask.tskrscstate = Free \wedge CallingTask.multiactcnt \leqslant 1 \wedge$
$NextRunTask.tskid \neq tskempty \wedge NextRunTask.tskdpri \geqslant TargetTask.tskpri)$
$\Rightarrow ChainTask(CallingTask, TargetTask) = E\_OS\_NORET \wedge$
$(CallingTask.tskstate = Suspended \wedge TargetTask.tskstate = Ready)$

$(CallingTask.tskstate = Running \wedge TargetTask.tskstate = Suspended) \wedge$
$(CallingTask.tskrscstate = Free \wedge CallingTask.multiactcnt \leqslant 1 \wedge$
$NextRunTask.tskid \neq tskempty \wedge NextRunTask.tskdpri < TargetTask.tskpri)$
$\Rightarrow ChainTask(CallingTask, TargetTask) = E\_OS\_NORET \wedge$
$(CallingTask.tskstate = Suspended \wedge TargetTask.tskstate = Running)$

$(CallingTask.tskstate = Running \wedge TargetTask.tskstate = Suspended) \wedge$
$(CallingTask.tskrscstate = Free \wedge CallingTask.multiactcnt > 1 \wedge$
$CallingTask.tskid \neq NextRunTask.tskid \wedge NextRunTask.tskdpri \geqslant TargetTask.tskpri)$
$\Rightarrow ChainTask(CallingTask, TargetTask) = E\_OS\_NORET \wedge$
$(CallingTask.tskstate = Ready \wedge TargetTask.tskstate = Ready)$

$(CallingTask.tskstate = Running \wedge TargetTask.tskstate = Suspended) \wedge$
$(CallingTask.tskrscstate = Free \wedge CallingTask.multiactcnt > 1 \wedge$
$CallingTask.tskid \neq NextRunTask.tskid \wedge NextRunTask.tskdpri < TargetTask.tskpri)$
$\Rightarrow ChainTask(CallingTask, TargetTask) = E\_OS\_NORET \wedge$
$(CallingTask.tskstate = Ready \wedge TargetTask.tskstate = Running)$

$(CallingTask.tskstate = Running \wedge TargetTask.tskstate = Suspended) \wedge$
$(CallingTask.tskrscstate = Free \wedge CallingTask.multiactcnt > 1 \wedge$
$CallingTask.tskid = NextRunTask.tskid \wedge NextRunTask.tskdpri \geqslant TargetTask.tskpri)$
$\Rightarrow ChainTask(CallingTask, TargetTask) = E\_OS\_NORET \wedge$
$(CallingTask.tskstate = Running \wedge TargetTask.tskstate = Ready)$

$(CallingTask.tskstate = Running \wedge TargetTask.tskstate = Suspended) \wedge$
$(CallingTask.tskrscstate = Free \wedge CallingTask.multiactcnt > 1 \wedge$
$CallingTask.tskid = NextRunTask.tskid \wedge NextRunTask.tskdpri < TargetTask.tskpri)$
$\Rightarrow ChainTask(CallingTask, TargetTask) = E\_OS\_NORET \wedge$
$(CallingTask.tskstate = Ready \wedge TargetTask.tskstate = Running)$

TP_for_ChainTask_Part02

Test_Purpose_SysCalls

$(CallingTask.tskstate = Running \land TargetTask.tskstate = Ready) \land$
$(CallingTask.tskrscstate = Free \land$
$TargetTask.multiactcnt < TargetTask.MAXMULTIACT \land$
$CallingTask.multiactcnt \leqslant 1 \land NextRunTask.tskid \neq TargetTask.tskid)$
$\Rightarrow ChainTask(CallingTask, TargetTask) = E\_OS\_NORET \land$
$(CallingTask.tskstate = Suspended \land TargetTask.tskstate = Ready)$

$(CallingTask.tskstate = Running \land TargetTask.tskstate = Ready) \land$
$(CallingTask.tskrscstate = Free \land$
$TargetTask.multiactcnt < TargetTask.MAXMULTIACT \land$
$CallingTask.multiactcnt \leqslant 1 \land NextRunTask.tskid = TargetTask.tskid)$
$\Rightarrow ChainTask(CallingTask, TargetTask) = E\_OS\_NORET \land$
$(CallingTask.tskstate = Suspended \land TargetTask.tskstate = Running)$

$(CallingTask.tskstate = Running \land TargetTask.tskstate = Ready) \land$
$(CallingTask.tskrscstate = Free \land$
$TargetTask.multiactcnt < TargetTask.MAXMULTIACT \land$
$CallingTask.multiactcnt > 1 \land CallingTask.tskid \neq NextRunTask.tskid \land$
$NextRunTask.tskid \neq TargeTask.tskid)$
$\Rightarrow ChainTask(CallingTask, TargetTask) = E\_OS\_NORET \land$
$(CallingTask.tskstate = Ready \land TargetTask.tskstate = Ready)$

$(CallingTask.tskstate = Running \land TargetTask.tskstate = Ready) \land$
$(CallingTask.tskrscstate = Free \land$
$TargetTask.multiactcnt < TargetTask.MAXMULTIACT \land$
$CallingTask.multiactcnt > 1 \land CallingTask.tskid \neq NextRunTask.tskid \land$
$NextRunTask.tskid = TargeTask.tskid)$
$\Rightarrow ChainTask(CallingTask, TargetTask) = E\_OS\_NORET \land$
$(CallingTask.tskstate = Ready \land TargetTask.tskstate = Running)$

$(CallingTask.tskstate = Running \land TargetTask.tskstate = Ready) \land$
$(CallingTask.tskrscstate = Free \land$
$TargetTask.multiactcnt < TargetTask.MAXMULTIACT \land$
$CallingTask.multiactcnt > 1 \land CallingTask.tskid = NextRunTask.tskid)$
$\Rightarrow ChainTask(CallingTask, TargetTask) = E\_OS\_NORET \land$
$(CallingTask.tskstate = Running \land TargetTask.tskstate = Ready)$

*TP_for_ChainTask_Part*03

Test_Purpose_SysCalls

$$
\begin{array}{l}
(CallingTask.tskstate = Running) \wedge \\
(CallingTask.tskrscstate = Free \wedge \\
\ CallingTask.multiactcnt < CallingTask.MAXMULTIACT \wedge \\
\ NextRunTask.tskid = tskempty) \\
\Rightarrow ChainTask(CallingTask, CallingTask) = E\_OS\_NORET \wedge \\
(CallingTask.tskstate = Running)
\end{array}
$$

$$
\begin{array}{l}
(CallingTask.tskstate = Running) \wedge \\
(CallingTask.tskrscstate = Free \wedge \\
\ CallingTask.multiactcnt < CallingTask.MAXMULTIACT \wedge \\
\ NextRunTask.tskid \neq tskempty \wedge CallingTask.tskid \neq NextRunTask.tskid \wedge \\
\ CallingTask.tskdpri \leqslant NextRunTask.tskpri) \\
\Rightarrow ChainTask(CallingTask, CallingTask) = E\_OS\_NORET \wedge \\
(CallingTask.tskstate = Ready)
\end{array}
$$

$$
\begin{array}{l}
(CallingTask.tskstate = Running) \wedge \\
(CallingTask.tskrscstate = Free \wedge \\
\ CallingTask.multiactcnt < CallingTask.MAXMULTIACT \wedge \\
\ NextRunTask.tskid \neq tskempty \wedge CallingTask.tskid \neq NextRunTask.tskid \wedge \\
\ CallingTask.tskdpri > NextRunTask.tskpri) \\
\Rightarrow ChainTask(CallingTask, CallingTask) = E\_OS\_NORET \wedge \\
(CallingTask.tskstate = Running)
\end{array}
$$

$$
\begin{array}{l}
(CallingTask.tskstate = Running) \wedge \\
(CallingTask.tskrscstate = Free \wedge \\
\ CallingTask.multiactcnt < CallingTask.MAXMULTIACT \wedge \\
\ NextRunTask.tskid \neq tskempty \wedge CallingTask.tskid = NextRunTask.tskid) \\
\Rightarrow ChainTask(CallingTask, CallingTask) = E\_OS\_NORET \wedge \\
(CallingTask.tskstate = Running)
\end{array}
$$

*TP_for_ChainTask_Part*04

Test_Purpose_SysCalls

$(CallingTask.tskstate = Running \wedge TargetTask.tskstate = Suspended) \wedge$
$(CallingTask.tskrscstate = Occupy)$
$\Rightarrow ChainTask(CallingTask, TargetTask) = E\_OS\_RESOURCE \wedge$
$(CallingTask.tskstate = Running \wedge TargetTask.tskstate = Suspended)$

$(CallingTask.tskstate = Running \wedge TargetTask.tskstate = Ready) \wedge$
$(CallingTask.tskrscstate = Occupy)$
$\Rightarrow ChainTask(CallingTask, TargetTask) = E\_OS\_RESOURCE \wedge$
$(CallingTask.tskstate = Running \wedge TargetTask.tskstate = Ready)$

$(CallingTask.tskstate = Running) \wedge$
$(CallingTask.tskrscstate = Occupy)$
$\Rightarrow ChainTask(CallingTask, CallingTask) = E\_OS\_RESOURCE \wedge$
$(CallingTask.tskstate = Running)$

$(CallingTask.tskstate = Running \wedge TargetTask.tskstate = Ready) \wedge$
$(CallingTask.tskrscstate = Free \wedge$
$TargetTask.multiactcnt = TargetTask.MAXMULTIACT)$
$\Rightarrow ChainTask(CallingTask, TargetTask) = E\_OS\_LIMIT \wedge$
$(CallingTask.tskstate = Running \wedge TargetTask.tskstate = Ready)$

$(CallingTask.tskstate = Running) \wedge$
$(CallingTask.tskrscstate = Free \wedge$
$CallingTask.multiactcnt = CallingTask.MAXMULTIACT)$
$\Rightarrow ChainTask(CallingTask, CallingTask) = E\_OS\_LIMIT \wedge$
$(CallingTask.tskstate = Running)$

TP_for_GetResource

Test_Purpose_SysCalls

$$
\begin{array}{l}
(CallingTask.tskstate = Running \land TargetRsc.rscstate = Occupy) \\
\Rightarrow GetResource(CallingTask, TargetRsc) = E\_OS\_ACCESS \land \\
(CallingTask.tskstate = Running \land TargetRsc.rscstate = Occupy) \\
\\
(CallingTask.tskstate = Running \land TargetRsc.rscstate = Free) \land \\
(CallingTask.tskpri \leqslant TargetRsc.rscpri) \\
\Rightarrow GetResource(CallingTask, TargetRsc) = E\_OK \land \\
(CallingTask.tskstate = Running \land TargetRsc.rscstate = Occupy) \\
\\
(CallingTask.tskstate = Running \land TargetRsc.rscstate = Free) \land \\
(CallingTask.tskpri > TargetRsc.rscpri) \\
\Rightarrow GetResource(CallingTask, TargetRsc) = E\_OS\_ACCESS \land \\
(CallingTask.tskstate = Running \land TargetRsc.rscstate = Free)
\end{array}
$$

TP_for_ReleaseResource

Δ *Test_Purpose_SysCalls*

$(CallingTask.tskstate = Running \land TargetRsc.rscstate = Free)$
$\Rightarrow ReleaseResource(CallingTask, TargetRsc) = E\_OS\_NOFUNC \land$
$(CallingTask.tskstate = Running \land TargetRsc.rscstate = Free)$

$(CallingTask.tskstate = Running \land TargetRsc.rscstate = Occupy) \land$
$(CallingTask.tskpri = TargetRsc.rscpri \land CallingTask.tskrscid = TargetRsc.rscid)$
$\Rightarrow ReleaseResource(CallingTask, TargetRsc) = E\_OK \land$
$(CallingTask.tskstate = Running \land TargetRsc.rscstate = Free)$

$(CallingTask.tskstate = Running \land TargetRsc.rscstate = Occupy) \land$
$(CallingTask.tskpri < TargetRsc.rscpri \land CallingTask.tskrscid = TargetRsc.rscid \land$
$CallingTask.schpol = NON\_PREE)$
$\Rightarrow ReleaseResource(CallingTask, TargetRsc) = E\_OK \land$
$(CallingTask.tskstate = Running \land TargetRsc.rscstate = Free)$

$(CallingTask.tskstate = Running \land TargetRsc.rscstate = Occupy) \land$
$(CallingTask.tskpri < TargetRsc.rscpri \land CallingTask.tskrscid = TargetRsc.rscid \land$
$CallingTask.schpol = FULL\_PREE \land CallingTask.tskid = NextRunTask.tskid)$
$\Rightarrow ReleaseResource(CallingTask, TargetRsc) = E\_OK \land$
$(CallingTask.tskstate = Running \land TargetRsc.rscstate = Free)$

$(CallingTask.tskstate = Running \land TargetRsc.rscstate = Occupy) \land$
$(CallingTask.tskpri < TargetRsc.rscpri \land CallingTask.tskrscid = TargetRsc.rscid \land$
$CallingTask.schpol = FULL\_PREE \land CallingTask.tskid \neq NextRunTask.tskid)$
$\Rightarrow ReleaseResource(CallingTask, TargetRsc) = E\_OK \land$
$(CallingTask.tskstate = Ready \land TargetRsc.rscstate = Free)$

$(CallingTask.tskstate = Running \land TargetRsc.rscstate = Occupy) \land$
$(CallingTask.tskpri \leqslant TargetRsc.rscpri \land CallingTask.tskrscid \neq TargetRsc.rscid)$
$\Rightarrow ReleaseResource(CallingTask, TargetRsc) = E\_OS\_ACCESS \land$
$(CallingTask.tskstate = Running \land TargetRsc.rscstate = Occupy)$

$(CallingTask.tskstate = Running \land TargetRsc.rscstate = Occupy) \land$
$(CallingTask.tskpri > TargetRsc.rscpri)$
$\Rightarrow ReleaseResource(CallingTask, TargetRsc) = E\_OS\_ACCESS \land$
$(CallingTask.tskstate = Running \land TargetRsc.rscstate = Occupy)$

Appendix B

Syntax of Test Purpose

TestPurpose	::=	Pre-State Op_1 Pre-Condition Op_2 Action Op_3 ReturnValue Op_1 Post-State
Pre-State	::=	Exp1
Pre-Condition	::=	Exp2
Action	::=	ActName'(' Par ')'
Post-State	::=	Exp1
Par	::=	Task | Resource | Par ', ' Par
Exp1	::=	ObjName'.'ObjState Op_3 StateValue | Exp1 Op_1 Exp1
Exp2	::=	ObjName'.'ObjAttr Op_3 AttrValue
	|	ObjName'.'ObjAttr Op_4 ObjName'.'ObjAttr
	|	Exp2 Op_1 Exp2 | ε
ObjName	::=	*CallingTask* | *TargetTask* | *NextRunTask* | *TargetRsc*
ObjState	::=	*tskstate* | *rscstate*
StateValue	::=	*Suspended* | *Ready* | *Running* | *Waiting* | *Free* | *Occupy*
ObjAttr	::=	*tskid* | *tsktype* | *schpol* | *tskpri* | *tskdpri*
	|	*multiactcnt* | *MAXMULTIACT* | *tskrscstate* | *rscid*
AttrValue	::=	*Basic* | *Extended* | *NON_PREE* | *FULL_PREE* | *Free* | *Occupy* | 1

ActName	::=	*ActivateTask* | *TerminateTask*
	|	*ChainTask* | *GetResource* | *ReleaseResource*
Task	::=	*CallingTask* | *TargetTask* | *NextRunTask*
Resource	::=	*TargetRsc*
ReturnValue	::=	*E_OK* | *E_OS_ACCESS* | *E_OS_LIMIT*
	|	*E_OS_NOFUNC* | *E_OS_RESOURCE* | *E_OS_NORET*
$\mathbf{Op_1}$	::=	' $\wedge$ ' | ε
$\mathbf{Op_2}$	::=	' $\Rightarrow$ '
$\mathbf{Op_3}$	::=	' $=$ '
$\mathbf{Op_4}$	::=	' $\leq$ ' | ' $\geq$ ' | ' $>$ ' | ' $<$ ' | ' $\neq$ '

Appendix C

Syntax of Verification Part

VeriPart ::= *do*
::*Set_TP_Objs*();
if
:: Block
:: *else*
fi;
od;

Block ::= '(' Bexp1 ')' Op_2
if
:: Segment
:: *else*
fi;
| Block
:: Block

Segment ::= '(' Bexp2 ')' Op_2
Aexp1;
Get_TP_Objs();
assert'(' Bexp3 Op_1 Bexp4 ')';
| Segment
:: Segment

Bexp1 ::= ObjName'.'ObjState Op_3 StateValue | Bexp1 Op_1 Bexp1

Bexp2 ::= ObjName'.'ObjAttr Op_3 AttrValue
| ObjName'.'ObjAttr Op_4 ObjName'.'ObjAttr
| Bexp2 Op_1 Bexp2 | *true*

Aexp1 ::= ActName'(' Par ')'

Bexp3 ::= ObjName'.'ObjState Op_3 StateValue | Bexp3 Op_1 Bexp3

Bexp4 ::= *ReturnStatus* Op_3 ReturnValue

Par ::= Task.*tskid* | Resource.*rscid* | Par ', ' Par

ObjName ::= *CallingTask* | *TargetTask* | *NextRunTask* | *TargetRsc*

ObjState ::= *tskstate* | *rscstate*

StateValue ::= *Suspended* | *Ready* | *Running* | *Waiting* | *Free* | *Occupy*

ObjAttr ::= *tskid* | *tsktype* | *schpol* | *tskpri* | *tskdpri*
| *multiactcnt* | *MAXMULTIACT* | *tskrscstate* | *rscid*

AttrValue ::= *Basic* | *Extended* | *NON_PREE* | *FULL_PREE* | *Free* | *Occupy* | 1

ActName ::= *ActivateTask* | *TerminateTask*
| *ChainTask* | *GetResource* | *ReleaseResource*

Task ::= *CallingTask* | *TargetTask* | *NextRunTask*

Resource ::= *TargetRsc*

ReturnValue ::= *E_OK* | *E_OS_ACCESS* | *E_OS_LIMIT*
| *E_OS_NOFUNC* | *E_OS_RESOURCE* | *E_OS_NORET*

Op_1 ::= '&&' | ε

Op_2 ::= ' $\rightarrow$ '

Op_3 ::= ' == '

Op_4 ::= ' <= ' | ' >= ' | ' > ' | ' < ' | '! = '

Appendix D

Result Table

Tsk:3 Rsc: 0

No.	Pattern Name		TestCases	Not Covered Test Purposes				
				Activate Task	Terminate Task	Chain Task	Get Resource	Release Resource
1	Ptn03_Ptn03_Ptn03	EQ_EQ_EQ	8342	1,3	1	1,4,6,8,9,16, 19		
2		EQ_GT_GT	4899	1	1	1,6,9,16		
3		EQ_LT_LT	4101	1	1	1,5,9,12,16		
4		GT_EQ_GT	2124	1	1	1,5,9,12,16		
5		GT_GT_GT	1308	1	1,3	1,5,6,9,12,13 16,18		
6		GT_LT_EQ	3364	1	1	1,6,9,16		
7		GT_LT_GT	1246	1	1,3	1,5,6,9,12,13 16,18		
8		GT_LT_LT	1200	1	1,3	1,5,6,9,12,13 16,18		
9		LT_EQ_LT	3031	1	1	1,6,9,16		
10		LT_GT_EQ	2693	1	1	1,5,9,12,16		
11		LT_GT_GT	1134	1	1,3	1,5,6,9,12,13 16,18		
12		LT_GT_LT	1166	1	1,3	1,5,6,9,12,13 16,18		
13		LT_LT_LT	1104	1	1,3	1,5,6,9,12,13 16,18		
14	Ptn04_Ptn04_Ptn04	EQ_EQ_EQ	11152	2,3	1	1,4,6,8,9,16, 19		
15		EQ_GT_GT	4476	2,3	1	1,6,9,16		
16		EQ_LT_LT	4701	2,3	1	1,9,16		
17		GT_EQ_GT	6369	2,3	1	1,9,16		
18		GT_GT_GT	2338	2,3	1	1,9,16		
19		GT_LT_EQ	3779	2,3	1	1,6,9,16		
20		GT_LT_GT	1990	2,3	1	1,6,9,16		
21		GT_LT_LT	2233	2,3	1	1,9,16		
22		LT_EQ_LT	9510	2,3	1	1,6,9,16		
23		LT_GT_EQ	6372	2,3	1	1,9,16		
24		LT_GT_GT	2694	2,3	1	1,9,16		
25		LT_GT_LT	2752	2,3	1	1,9,16		
26		LT_LT_LT	2764	2,3	1	1,9,16		
27	Ptn04_Ptn03_Ptn03	LT_EQ_LT	9295	3	1	1,6,9,16		
28		LT_GT_LT	2259	Non	1	1,9,16		
29		LT_LT_LT	2193	Non	1	1,9,16		
30	Ptn04_Ptn04_Ptn03	EQ_LT_LT	3551	3	1	1,9,16		
31		GT_LT_LT	2280	3	1	1,9,16		
32		LT_LT_LT	2758	3	1	1,9,16		
33	Ptn03_Ptn04_Ptn03	GT_LT_EQ	3266	3	1	1,6,9,16		
34		GT_LT_GT	1582	Non	1	1,9,16		
35		GT_LT_LT	1678	Non	1	1,9,16		
36	Ptn03_Ptn03_Ptn04	EQ_GT_GT	7621	3	1	1,6,9,16		
37		GT_GT_GT	2154	Non	1	1,9,16		
38		LT_GT_GT	1771	Non	1	1,9,16		
39	Ptn03_Ptn04_Ptn04	GT_EQ_GT	2687	3	1	1,9,16		
40		GT_GT_GT	2318	3	1	1,9,16		
41		GT_LT_GT	2163	3	1	1,9,16		
42	Ptn04_Ptn03_Ptn04	LT_GT_EQ	5709	3	1	1,9,16		
43		LT_GT_GT	2495	3	1	1,9,16		
44		LT_GT_LT	2633	3	1	1,9,16		

Tsk:3 Rsc: 1(Ptn04_Ptn03_Ptn03_LT_GT_LT)

No.	Pattern Name		TestCases	Not Covered Test Purposes				
				Activate Task	Terminate Task	Chain Task	Get Resource	Release Resource
28.1	Ptn04_Ptn03_Ptn03_LT_GT_LT	LT_LT_LT	4663	Non	Non	Non	3	2,5,6,7
28.2		LT_EQ_LT	7242	Non	Non	Non	3	6,7
28.3		LT_GT_LT	4439	Non	Non	6	Non	2,5,6
28.4		LT_GT_EQ	4958	Non	Non	6	Non	4,5,6
28.5		LT_GT_GT	3779	Non	Non	6	Non	2,4,5,6,7
28.6		EQ_GT_GT	3585	Non	Non	6	Non	3,4,5,6,7
28.7		GT_GT_GT	3645	Non	1	1,6,9,16	1,2	2,3,4,5,6,7

Tsk:3 Rsc: 1(Ptn04_Ptn03_Ptn03_LT_LT_LT)

No.	Pattern Name		TestCases	Not Covered Test Purposes				
				Activate Task	Terminate Task	Chain Task	Get Resource	Release Resource
29.1	Ptn04_Ptn03_Ptn03_LT_LT_LT	LT_LT_LT	5051	Non	Non	Non	3	2,6,7
29.2		LT_LT_EQ	6499	Non	Non	Non	3	6,7
29.3		LT_LT_GT	4251	Non	Non	Non	Non	2,5,6
29.4		LT_EQ_GT	6292	Non	Non	Non	Non	4,5,6
29.5		LT_GT_GT	4026	Non	Non	5	Non	2,4,5,6,7
29.6		EQ_GT_GT	4400	Non	Non	Non	Non	3,4,5,6,7
29.7		GT_GT_GT	4682	Non	1	1,9,16	1,2	2,3,4,5,6,7

Tsk:3 Rsc: 1(Ptn03_Ptn04_Ptn03_GT_LT_GT)

No.	Pattern Name		TestCases	Not Covered Test Purposes				
				Activate Task	Terminate Task	Chain Task	Get Resource	Release Resource
34.1	Ptn03_Ptn04_Ptn03_GT_LT_GT	LT_LT_LT	3727	Non	Non	6	3	2,6,7
34.2		EQ_LT_LT	4337	Non	Non	6	3	6,7
34.3		GT_LT_LT	2776	Non	Non	6	Non	2,5,6,7
34.4		GT_LT_EQ	3248	Non	Non	6	Non	4,5,6
34.5		GT_LT_GT	2765	Non	Non	6	Non	2,4,5,6,7
34.6		GT_EQ_GT	2889	Non	Non	6	Non	3,4,5,6,7
34.7		GT_GT_GT	4183	Non	1	1,9,16	1,2	2,3,4,5,6,7

Tsk:3 Rsc: 1(Ptn03_Ptn04_Ptn03_GT_LT_LT)

No.	Pattern Name		TestCases	Not Covered Test Purposes				
				Activate Task	Terminate Task	Chain Task	Get Resource	Release Resource
35.1	Ptn03_Ptn04_Ptn03_GT_LT_LT	LT_LT_LT	3698	Non	Non	5	3	2,6,7
35.2		LT_LT_EQ	4331	Non	Non	5	3	6,7
35.3		LT_LT_GT	3088	Non	Non	5	Non	2,5,6
35.4		EQ_LT_GT	4245	Non	Non	5	Non	4,5,6
35.5		GT_LT_GT	2782	Non	Non	5	Non	2,4,5,6,7
35.6		GT_EQ_GT	3423	Non	Non	5	Non	3,4,5,6,7
35.7		GT_GT_GT	4327	Non	1	1,9,16	1,2	2,3,4,5,6,7

Tsk:3 Rsc: 1(Ptn03_Ptn03_Ptn04_GT_GT_GT)

No.	Pattern Name		TestCases	Not Covered Test Purposes				
				Activate Task	Terminate Task	Chain Task	Get Resource	Release Resource
37.1	Ptn03_Ptn03_Ptn04_GT_GT_GT	LT_LT_LT	1816	Non	3	5,6,12,13,18	3	2,3,6,7
37.2		EQ_LT_LT	2303	Non	3	5,6,12,13,18	3	3,6,7
37.3		GT_LT_LT	1550	Non	3	5,6,9,12,13	Non	2,3,5,6
37.4		GT_EQ_LT	1807	Non	3	5,6,12,13	Non	3,4,5,6
37.5		GT_GT_LT	3061	Non	Non	Non	Non	2,4,5,6,7
37.6		GT_GT_EQ	3100	Non	Non	Non	Non	3,4,5,6,7
37.7		GT_GT_GT	2623	Non	1	1,9,16	1,2	2,3,4,5,6,7

Tsk:3 Rsc: 1(Ptn03_Ptn03_Ptn04_LT_GT_GT)

No.	Pattern Name		TestCases	Not Covered Test Purposes				
				Activate Task	Terminate Task	Chain Task	Get Resource	Release Resource
38.1	Ptn03_Ptn03_Ptn04_LT_GT_GT	LT_LT_LT	4314	Non	Non	Non	3	2,6,7
38.2		LT_EQ_LT	5145	Non	Non	Non	3	6,7
38.3		LT_GT_LT	3393	Non	Non	12	Non	2,5,6
38.4		EQ_GT_LT	4312	Non	Non	12	Non	4,5,6
38.5		GT_GT_LT	2585	Non	Non	12	Non	2,4,5,6,7
38.6		GT_GT_EQ	2546	Non	Non	12	Non	3,4,5,6,7
38.7		GT_GT_GT	2414	Non	1	1,9,12,16	1,2	2,3,4,5,6,7

Tsk:3 Rsc: 2(Ptn04 Ptn03 Ptn03 LT LT LT(LT LT GT))

No.	Pattern Name		TestCases	Not Covered Test Purposes				
				Activate Task	Terminate Task	Chain Task	Get Resource	Release Resource
29.31	Ptn04_Ptn03_Ptn03_LT_LT_LT_LT_LT_GT	LT_LT_LT_LT	11533	Non	Non	5	Non	2
29.32		LT_LT_EQ_LT	13594	Non	Non	5	Non	Non
29.33		LT_LT_GT_EQ	12882	Non	Non	5	Non	2,5
29.34		LT_LT_GT_GT	12498	Non	Non	5	Non	2,5
29.35		LT_EQ_GT_GT	13537	Non	Non	5	Non	5
29.36		LT_GT_GT_GT	12928	Non	Non	5	Non	2,5
29.37		EQ_GT_GT_GT	12613	Non	Non	5	Non	5
29.38		GT_GT_GT_GT	11294	Non	Non	5	Non	2,5,6

Bibliography

[BK+08] C. Baier, J.P. Katoen, et al. Principles of model checking. 2008.

[CGP] E.M. Clarke, O. Grumberg, and D. Peled. Model Checking. 1999.

[Cla97] E. Clarke. Model checking. In *Foundations of Software Technology and Theoretical Computer Science*, pages 54–56. Springer, 1997.

[FWA09] G. Fraser, F. Wotawa, and P.E. Ammann. Testing with model checkers: a survey. *Software Testing, Verification & Reliability*, 19(3):215–261, 2009.

[GMP04] M.M. Gallardo, P. Merino, and E. Pimentel. A generalized semantics of promela for abstract model checking. *Formal Aspects of Computing*, 16(3):166–193, 2004.

[gro99] OSEK/VDX group. OSEK/VDX Conformance Testing Methodology Version 2.0. 1999.

[gro05] OSEK/VDX group. OSEK/VDX Operating System Version2.2.3. 2005. www.osek-vdx.org.

[HBB+09] R.M. Hierons, K. Bogdanov, J.P. Bowen, R. Cleaveland, J. Derrick, J. Dick, M. Gheorghe, M. Harman, K. Kapoor, P. Krause, et al. Using formal specifications to support testing. *ACM Computing Surveys (CSUR)*, 41(2):1–76, 2009.

[Hol02] G.J. Holzmann. The model checker SPIN. *Software Engineering, IEEE Transactions on*, 23(5):279–295, 2002.

[Hol04] G.J. Holzmann. *The SPIN model checker: Primer and reference manual*. Addison Wesley Publishing Company, 2004.

[II93] D.C. Ince and D. Ince. *Introduction to Discrete Mathematics, Formal System Specification, and Z*. Oxford University Press, 1993.

[Mye08] G.J. Myers. *The art of software testing*. Wiley-India, 2008.

[PTS96] B. Potter, D. Till, and J. Sinclair. *An introduction to formal specification and Z*. Prentice Hall PTR Upper Saddle River, NJ, USA, 1996.

[Ray87] D. Rayner. OSI conformance testing. *Computer Networks and ISDN Systems*, 14(1):79–98, 1987.

[Spi92] J.M. Spivey. The Z notation: a reference manual. 1992.

[Tre] J. Tretmans. An overview of osi conformance testing. *Samson, editor, Conformance Testen, in Handboek Telematica*, 2:4400.

[Tre92] G.J. Tretmans. A formal approach to conformance testing. 1992.

[YA10] K. Yatake and T. Aoki. Automatic Generation of Model Checking Scripts based on Environment Modeling. *Model Checking Software*, pages 58–75, 2010.