Doctoral Dissertation

STUDY ON FACILITATING THE VARIATION OF SCHEDULERS IN MODEL CHECKING

TRAN NHAT HOA

Supervisor: Professor Toshiaki Aoki

School of Information Science
Japan Advanced Institute of Science and Technology

September, 2018

Abstract

Software applications play an important role in our lives. The failure of the applications may harm people or equipment. Therefore, the correctness of the software is important. In fact, an application may consist of multiple processes, which are developed based on programming languages and operating systems (OSs). Under the mechanisms provided by these languages and environments, the processes can run simultaneously to increase the scalability. The applications are called concurrent systems. In fact, these systems are error-prone; for example, deadlock, livelock, or violations of constraints may occur in them. Because the processes of a concurrent system can be executed in different orders, their behaviors are difficult to verify.

As an exhaustive and automatic technique, model checking explores every execution of a system and automatically find possible errors. In comparison with other techniques, such as testing and simulation, model checking is more suitable to verify the concurrent systems. To model checking a system, we need to specify its behaviors (usually in a modeling language); then travel all the states of the system (called the state space) represented by its model using a search algorithm to check the corresponding property.

With the increasing of the complexness of a concurrent system, there is a need to schedule the execution of the processes. There are several scheduling strategies applied by real systems. For instance, in OSEK OS for automotive devices, an application can have multiple tasks executed under the priority and mixed preemption strategy. In model checking, the behaviors of a scheduler associate with the algorithm that explores the state space. However, verifying a concurrent system with considering all possible executions (interleaving behaviors) is an over-approximation approach and can produce spurious counterexamples because the errors may occur outside the executions indicated by the scheduler. Therefore, to accurately verify the systems, we need to take the scheduler into account in the verification.

Current methods in model checking to deal with sequential/concurrent systems are difficult to apply to verify with scheduling policies because these methods consider a different kind of behaviors and can cause spurious counterexamples. To deal with the scheduling policies, existing approaches try to limit the executions of the systems by encoding both of the processes and the scheduler into a model using a modeling language (e.g. Promela). In this case, the scheduling policy needs to be specified from scratch. This approach is hard to model interesting schedulers, error-prone, and time-consuming. This means that an approach to easily and flexibly describe the scheduling policies is needed.

In reality, the OSs use different policies to control the executions of the processes. For example, Linux OS can support several policies for its tasks based on their priorities (e.g. *round-robin* and *first-in-first-out*). However, the existing approaches cannot deal with the variation of the schedulers because the policy is fixed in the model of a system. That means to ensure the accuracy of the concurrent systems, a study on facilitating the variation of schedulers in model checking is necessary and important.

To overcome the problems above, in this research, we propose a method to analyze and verify concurrent systems executed under different scheduling policies using model checking techniques. Our method contains three main parts: 1) a language for modeling the processes, 2) a domain-specific language (DSL) to describe the scheduling policies, and 3) an algorithm to search all of the states of the system.

The originality of this research is proposing a DSL to specify the scheduling policies used in model checking techniques. In this approach, our language aims to provide a high-level support for specifying different policies easily. All the information necessary to analyze the system are automatically generated. From the specification of the scheduling policy in the DSL, a search algorithm is realized to explore the state space. Following this approach, we implemented a tool named SSpinJa, which is extended from SpinJa, a model checker implemented in Java. Our experiments indicate that the method is practical; it is easy to describe different scheduling policies and accurately verify the behaviors of the systems. In addition, in this research, we apply model-based testing techniques to generate the tests to check the correspondence between the policy in our DSL and the real scheduler in an OS; it helps us to increase the confidence of the policy in the DSL and accurately verify the systems.

The impact of this research is that we can easily apply model checking techniques to verify the concurrent systems with the different scheduling policies. The state space to be searched is now limited because the scheduler is taken into account in the verification. Therefore, we can verify systems more accurately. In addition, with our method, we can reuse the specifications of the processes and the scheduling policy. It helps to decrease the time necessary for designing and developing a concurrent system.

Key words: concurrent systems, model checking, scheduler, domain-specific language, model-based testing

Acknowledgments

The author wishes to express his honest appreciation to his supervisor Professor Toshiaki Aoki of JAIST for his continual encouragement and kind guidance during this work. He provides a lot of advice with suggestions. Without his instructions, this research could not be completed. Besides, his kindness helps the author so much to enjoy the social life during the study time.

The author also wishes to express his thanks to Professor Kunihiko Hiraishi of JAIST for being his second supervisor with many helpful instructions for the study and writing this dissertation.

The author would like to thank his advisor Associate Professor Masato Suzuki of JAIST for the suggestions to complete the research.

The author is grateful to Dr. Yuki Chiba, Dr. Takashi Tomita, and Dr. Tatsuji Kawai. Their thoughtful guidance strongly contributes to the results of this work.

The author devotes his sincere appreciation to all of the members of Aoki Laboratory for their discussions and sharing the study time.

The author would like to acknowledge the Japanese Government for supporting the scholarship (MEXT) for financing the study. This is a big opportunity for the author to conduct the research in Japan.

Last but not least, the author would like to thank his parents for their consistent support. The author would like to express his greatest gratitude to his family members, Nguyen Quang Huong Tra, Tran Phuc Yen Thi, and Tran Phuc Giang Thi. Their infinite love is the biggest encouragement for the author to complete this research.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Software applications play an important role in our lives. The applications are used in many areas, such as medicine and aviation. Because any failure of the application may harm people or equipment, the correctness of the applications is very important. There is a critical problem named Concurrent Program Verification [23]. In fact, a program can consist of multiple processes, which are developed based on programming languages and OSs, such as Java, C#, Windows and Real-time OSs - RTOs. Under the mechanisms provided by these languages and environments, the processes can run simultaneously to increase the scalability. These programs (applications) are called concurrent systems [68]. In fact, these systems are error-prone; for instance, deadlock, livelock, or violations of constraints may occur in them. However, because these processes can run in the different ways, their behaviors are difficult to verify.

In our social life, many systems and devices require the correctness related to time. For instance, in an automotive system with the controllers for braking, when the brake is applied, the controller analyzes the situation of the system (car speed, environment condition, etc.) and activates the brakes within fractions of a second. That means these behaviors related to time and the time effects to the system. Usually, the systems must guarantee time constraints, such as no deadline violation occurs. These systems are called real-time systems [18]. Actually, with this kind of systems, the timing properties are needed to consider carefully.

In general, there are two ways to guarantee the safety and the reliability of a system. The first one aims to minimize the possible errors in the implementation stage using testing techniques [49] to increase the confidence of the accuracy of a system. The other focuses on verifying whether the system satisfies the requirements using analysis and formal approaches [16].

Testing techniques are usually applied to validating the implementation. The verification carries out pre-designed test cases following limited execution orders. Since testing considers only a subset of executions of a system, it cannot be complete. In addition, testing only shows the existing of the errors but cannot guard the error-free. With concurrent systems, unexpected errors are difficult to realize by testing because these errors are hard to reproduce. In addition, testing is only performed during or after implementing the system; therefore, the errors cannot be found before programming.

With the formal approaches, we can verify whether the corresponding properties hold with a system. As an exhaustive and automatic technique, model checking [9] will

explore every execution to check whether a system can meet a given specification (which is represented in temporal logic). Actually, model checking plays an important role to ensure the accuracy of the systems. The key idea is to use algorithms for checking all reachable states of a system systematically. If an error is found, it shows the correspond counterexample.

To model checking a system, we need to specify the behaviors of the processes of the system, usually using a modeling language, such as Promela[1] [40]. The system is then presented as a finite state graph. A model checking tool uses an algorithm to construct the state space by traveling all reachable states from the starting state. With each possible behavior, which is represented by a statement, it constructs the next state and checks whether this state already exists or not. All the reachable states are constructed as the same way until no more states to consider. Usually, the two algorithms: depth-first search (DFS) and breadth-first search (BFS) are used to perform this task. The data structure used by DFS to store the search steps is a stack while BFS uses a queue instead. The differences between these two algorithms are the number of states needed to be stored in the stack/queue at each step and their visited order. However, the main purpose of these algorithms is to explore all states of a system. With these states, the corresponding property (e.g. safety property) is checked to indicate whether the requirement is satisfied or not. There is another algorithm named nested-depth first search [47], which uses two searches, to handle the accepting cycles in a graph for checking liveness properties.

In order to find the existing errors faster or to find short counterexamples, directed model checking techniques [35] were studied to decrease the number of states based on abstractions [4, 51] or calculating the distance [34, 33, 87] to the possible error states for fast finding them. These techniques focus on the algorithms to travel the state space. The well-known algorithm A^* [45] is used for the exploring. The process has the best-evaluated value (lowest/highest) is chosen for the execution. This means guiding the exploration with a particular purpose. Nonetheless, all of the states still need to visit in the worst case.

Using a general-purpose language (GPL) is an ordinary way to implement or model a system (such as, using Java [7] for the implementation or using UML [12] for the modeling). For the verification, although some tools provide the graphical editor to easily model and specify the system, such as TIMES [3] and UPPAAL [52]. Nevertheless, using a modeling language still is a common way to describe the behaviors of a system. In fact, these GPLs can be applied to broad systems. However, the languages lack specialized features for any appropriate domain, such as communication protocols or scheduling policies. That means these features need to be encoded in the existing languages from scratch. This task is usually error-prone and time-consuming.

Actually, in some situations, there is a need to increase the characteristics of a language to handle a specific problem, such as scheduling domain or database manipulation. That fact motivates designing a special type of language (called domain-specific language (DSL) [38]). A DSL is limited expressive and focuses on a special domain with the main aim to create a normally small language to solve the problems in a special domain (e.g. Unix shell scripts [60]). In fact, a DSL is always simplified and productive. Since a DSL is usually small and easy to understand, it can help writing the code faster,

[1]Promela is used by a model checker named Spin [46] to verify the consistency of a distributed system. There are other languages, such as SMV [20], which is used by a symbolic model checking tool named NuSMV based on representing sets of states as logical formulas [57].

modifying it easier, and reducing bugs.

With the increasing of the complexness of a concurrent system, there is a need to schedule the execution of the processes. In fact, there are several scheduling policies used by the real OSs, such as Linux uses the *priority* strategy to select a process to run. It means that the processes are executed under the scheduling policies. In fact, there are methods to verify sequential software systems or concurrent systems, such as [21, 26, 88]. Nonetheless, the methods are difficult to adopt to deal with schedulers directly because the behaviors they consider are different from that of the systems with scheduling policies. Actually, if we consider all possible executions, the verification can produce spurious counterexamples because the errors may occur outside the executions indicated by the scheduler. We consider a system with two processes (`t1` and `t2`) depicted in Figure 1.1 as the motivating example. With the condition `a + b < 5`, process `t1` increases the shared variable `a` repeatedly, while process `t2` increases the variable `b`[2]. At the initial state, these two processes are run and the variables `a` and `b` are set to 2 for `a` and 0 for `b`.

```
int a, b;

proctype t1() {
 do
 :: d_step{ (a+b)<5 -> a++}
 :: d_step{ else -> break}
 od;
 assert (a >= b)
}

proctype t2() {
 do
 :: d_step{ (a+b)<5 -> b++}
 :: d_step{ else -> break}
 od;
 assert (a >= b)
}

init {
 a = 2; b = 0;
 run t1();
 run t2();
}
```

Figure 1.1: Motivating example

Figure 1.2 depicts the state space corresponding to this system. We can easily see that an error happens (violation of an assertion) if these two processes are executed in an interleaving manner. However, no error occurs if the system uses *round-robin* policy with the *time slice* being equal to 1 (one statement is executed at one time unit) as indicated in Figure 1.3. With this example, we can see that for the accurate verification, the scheduling strategies need to be taken into account.

Actually, real systems use many kinds of schedulers with different strategies. These policies are different from the '*textbook*' ones. For example, in OSEK OS [62], the tasks in an application are executed using their priority with mixed preemption; Linux OS supports different policies for its real-time and non-real-time processes. In fact, to perform scheduling strategies in model checking, the existing approaches use a modeling

[2]A `d_step` statement is executed as if it is one single statement.

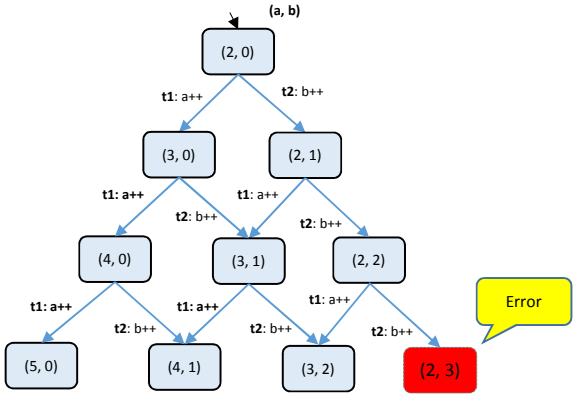


Figure 1.2: The state space

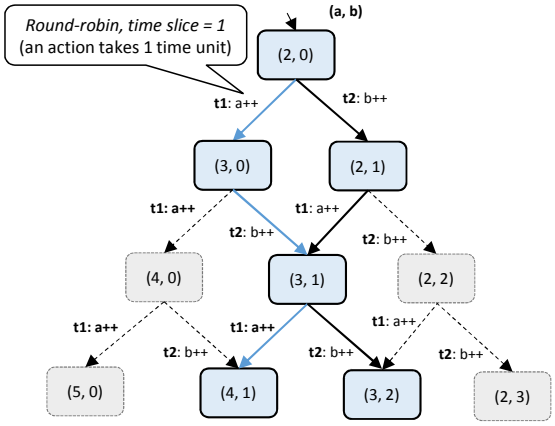


Figure 1.3: State space following the scheduling policy

language (e.g., Promela) to encode the scheduler and the processes into the *model* of a system [5, 56]. In this case, the scheduling strategy is fixed in the model. Actually, different policies cannot be handled with one *model* in the verification. If we want to adopt another policy, we must prepare another *model*. This task is both prone to errors and time-consuming. That means finding another way to easily and flexibly describe the scheduling strategies is necessary.

There are existing tools like UPPAAL and TIMES, which provide both of a graphical user interface and a language for modeling the system. However, UPPAAL only deals with real-time systems and it lacks the flexibility to describe the behaviors of the system. It is because the processes and the scheduler are encoded into the same model and we must define the information of a process using public variables. This tool also has a challenging in dealing with the variation of scheduling policies using timed automata. With TIMES, we can specify various attributes for processes (e.g. periodic and priority). However, this tool only supports limited policies, such as *rate-monotonic* and *deadline-monotonic*. Actually, TIMES cannot be extended to handle other policies.

There are some existing languages for describing the scheduling policies, such as Bossa [10] and Catapults [71]. However, these languages cannot be applied or reused for the verification because of their characteristics based on the target systems and the

techniques behind. Firstly, these languages deal with particular systems and rely on their techniques. Therefore, only limited policies are supported. This fact does not match with our purpose, which aims to support a variety of policies. Secondly, the techniques they used are for implementing the schedulers in a real system. In our case, the purpose is to verify the correctness of a system using model checking. The main point is that we need to consider all the possible behaviors based on the corresponding scheduling policy. However, these languages aim to handle the implementation; with this purpose, considering only one execution is enough. Another language is introduced in the work UML profile for MARTE [55] to model real-time system with timing properties for analyzing the time constraints. In fact, this language does not fit to our purpose because it can not handle the behaviors of the scheduler and the processes. However, we realize from these languages that using a DSL is an easy and flexible way to describe the scheduling strategies because it is a small language and aims at a special domain; it is also simple and productive. That motivates us to propose a DSL for the scheduling to model checking the systems.

In reality, the scheduler limits the executions of a system. In model checking, that fact relates to the search algorithm. To accurately verify systems, we need to consider the behaviors of the scheduler to explore the state space. However, the existing algorithms applied in model checking (e.g. DFS and BFS) do not consider the behaviors of the scheduler. That means another algorithm to deal with the scheduling strategies is needed.

In this research, we aim to verify concurrent systems run on OSs. The scheduler of the OS controls the execution of the processes. For the verification, we propose a language to describe the scheduling strategies. Actually, the correspondence between the scheduling policy in the DSL and the real scheduler in an OS affects the verification results. Thus, there is a need to check that the behaviors indicated by the policy are the same as the real one. In addition, the specification of a policy for a real system does not exist or is not clear to describe the behaviors of the scheduler. For instance, the specification of Linux real-time FIFO policy indicates that functions `sched_setscheduler` or `sched_setparam` increases the priority of a runnable `SCHED_FIFO` thread and may preempt the currently running thread with the same priority. That means there are two options for the implementation: preempt or not preempt the current thread. However, which option is implemented on each version of Linux is not described in the specification. In addition, the behaviors of the scheduler in a real OS can be observed only in executing the system. Therefore, we apply testing techniques to check the correspondence between the scheduling strategy (in the DSL) and the behaviors of the real scheduler of an OS.

There are approaches for conformance testing the systems to indicate whether the implementation follows the specification or the design of a system, such as [17], which assumes that the design is correct before checking the implementation; if the test fails, it means that the implementation does not follow the design. In our case, the purpose is different because we want to check that the behaviors indicated by the policy (in the DSL) are the same as the real behaviors of the scheduler of an OS (an implementation). However, we can expect the accuracy of the strategy and then use it to test the behaviors of the real scheduler. Our method for testing is as follows. We prepare the tests following the policy in the DSL. We then execute the tests to check whether the behaviors of the scheduler in a real system follows this policy or not. If the tests passed, we are confident about the specification. However, if any test fails, the test now may not correspond

to the execution because there is a case that the implementation only deals with an option of the specification (as explained before with Linux FIFO policy). Therefore, we cannot conclude anything. To prepare the tests, we apply model-based testing (MBT) techniques [6] to automatically and exhaustively generate the tests following the policy.

1.2 Research Problems and Objective

In this research, we aim at verifying concurrent applications (systems) which run on OSs using model checking techniques. We address the following problems: a) the scheduler of the OS controls the executions of the system, b) there is a variation of the strategies used by the OS, and c) existing approaches are difficult to handle this variation. The objective of this research is ***proposing a method to facilitate the variation of schedulers in model checking***.

To achieve this objective, our method needs to a) easily deal with different policies with small effort, b) flexibly change the scheduling strategies, and c) accurately verify the behaviors of the systems. Our ideas to deal with these problems are as follows. The details of our approach are described in Section 1.4.

- Firstly, to flexibly change the policy, we separate the scheduler from the behaviors of the processes. This approach is different from the existing ones [5, 56, 89], which encode the scheduler and the processes into the same model. With this approach, the behaviors of the processes and the policy can be reused.

- Secondly, to specify the behaviors of the processes, we use Promela as the based modeling language with introducing several API functions for supporting the communication between the processes and the scheduler in the system.

- Thirdly, to deal with various strategies, we propose a DSL to describe the policies with the attributes of the processes to handle the scheduling tasks. The DSL aims to provide high-level language to specify various policies easily. By focusing on a domain, using a DSL to specify the policies is easier than using an existing modeling language (e.g. Promela [40], which is used by Spin tool or CSP# [79], which is used by PAT model checker [54]).

- Fourthly, to verify systems following the scheduling, we introduce an algorithm to explore the state space. This algorithm is different from the existing one (e.g. DFS, BFS). We now deal with the behaviors of the scheduler in the verification. In our approach, all of the information necessary for the analysis and verification is generated automatically from the policy in the DSL. With the new search algorithm, we can verify systems accurately.

- Lastly, to increase the confidence of the policy in the DSL, we apply testing techniques to check the policy with its implementation in a real OS. Our approach is checking the correspondence of the policy with the behaviors of the scheduler in a real OS. That fact is different from the ordinary conformance testing, which aims to test the compliance of an implementation following the requirements of a specification [85]. It is easy to see that the more confidence in the policy, the more accurately verifying the system.

1.3 Research Contributions

Following the method proposed, we implemented a tool named SSpinJa, which is extended from SpinJa [29], an implementation of the core of Spin in Java. We conducted several experiments. The evaluation results demonstrate that our approach is practical; the tool can verify systems with scheduling policies easily, flexibly, and accurately.

Our method has the following advantages: a) the language for the strategies is simple, b) the behaviors of the system can be extended and captured easily for the analysis, c) the description of the strategies can be reused completely, d) the tool can verify and analyze the systems accurately, and e) our approach is practical.

We have the following contributions for this research.

- C_1: Proposing a DSL to describe scheduling policies used in model checking techniques;
- C_2: Proposing a model checking algorithm to verify systems under the scheduling policies;
- C_3: Proposing a method to generate the tests to check the correspondence between the scheduling policy in the DSL with the behaviors of the scheduler in a real OS;
- C_4: Implementing a tool for verifying systems under the scheduling policies.

- Contribution C_1 concerns a language for scheduling strategies. The originality of our research is proposing a DSL to describe the strategies used in model checking techniques for verifying the systems. This DSL is different from the existing ones, such as Bossa [10] and Catapults [71], which aim to implement the schedulers in a real system. In our approach, the language aims to provide a high-level support to specify various scheduling policies easily. For the verification, the processes that can be executed are considered using ordering methods; the behaviors of the scheduler and the processes with the scheduling tasks are indicated by handling the scheduling events (see Section 1.4 for more details).

- Contribution C_2 concerns the exploration of the state space. The scheduling policy is now taken into account in the search. We extend DFS algorithm by considering the behaviors of the scheduler. The reason is that the memory usage for recording the search steps used by DFS is usually less than other algorithms (e.g. BFS). To do this, we adopt the compilation approach to automatically generate all of the information necessary from the policy specified in the DSL beforehand to analyze the behaviors of the system. The information generated from the policy is used by the search algorithm. The behaviors of the scheduler are now considered in the verification.

- Contribution C_3 concerns to the accuracy of the policy. We aim to verify systems executed under the strategies. Actually, the correspondence between the policy in the DSL and the real behaviors of the scheduler affects the verification results. That means the accuracy of the strategy is also important. In our approach, we adopt testing techniques to check the policy in the DSL to enhance the certainty of the policy. To do that, we apply MBT techniques to automatically prepare the tests for testing the scheduling policy.

- Contribution C_4 concerns the applicability of our method. With our DSL, we can easily describe the policy to verify the system. That is necessary because real systems usually use schedulers to handle their executions. This approach is distinct from the existing ones that use a modeling language to specify the systems. Moreover, the method can help the developers to accurately verify the systems with different scheduling policies. With these advantages, we can reduce the time necessary for developing a concurrent system.

1.4 The Approach

To address the problems above, we propose a method for analyzing and verifying concurrent systems executed under different scheduling policies using model checking techniques. The approach is depicted in Figure 1.4.

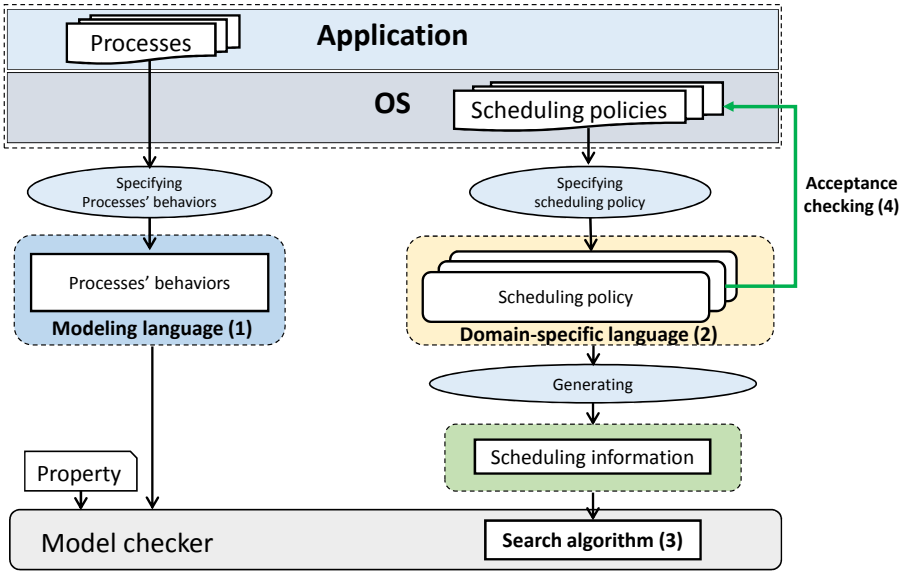


Figure 1.4: Research approach

1.4.1 Specifying the Behaviors of the Processes

To flexibly change the policy, we separate the scheduler from the behaviors of the processes. In our approach, the behaviors of the processes are described in a modeling language (1), which is based on Promela to describe the behaviors of concurrent processes. To deal with the strategy, we introduce API functions to handle the scheduling tasks with these processes. These functions are as follows.

1. executing a process with its attributes (such as deadline),
2. accessing the scheduler information[3], and

[3]to implement the algorithms like slack stealing [53].

3. performing the user-defined functions defined in *interface* part (*) (as depicted in Figure 1.5) to handle the actions of the processes with scheduling tasks.

1.4.2 Specifying Scheduling Policy

To facilitate various policies, we propose a DSL (2) to provide a language to describe different policies. The main role of the scheduler is to select a process for the execution, to manage the processes using their attributes (e.g. priority), change their execution statuses (e.g. blocks a process), and to manage the time.

In the DSL, we separate the attributes of the processes and the behaviors of the scheduler for flexibly changing them (as depicted in Figure 1.5). The attributes of the processes are defined in *process attribute* (a), and the behaviors of the scheduler are defined in *scheduler description* (b).

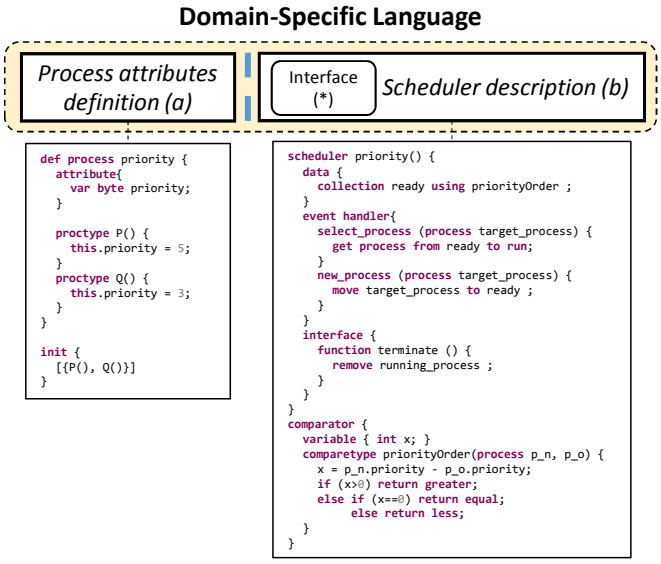


Figure 1.5: A scheduling strategy

Actually, to store the information of processes, the scheduler uses data structures (e.g. ready queue), which represent the execution statuses of the processes (such as ready or blocked). Following the order of the processes (e.g. their order in the queue), a process is selected to run. To represent that fact, in the DSL, we use collections for storing the processes. At a time, there may be many candidates for the execution (e.g. the processes with the same priority). In order to deal with that fact, we partially order [31] these processes in the collections for the selection. This order either is defined by a function, uses the `LIFO/FIFO` strategies, or combines these ordering methods. The ways for *selecting/adding* a process from/into a collection will follow this ordering. We also support the collection without any ordering method (that means these processes in this collection have the same order). An example for ordering the processes following their attributes is depicted in Figure 1.6. With this example, process `P` has the same *priority* as process `Q` does; therefore, when process `P` arrives at the system, it will have the same order as process `Q` in the queue. These two processes are also considered when the scheduler selects a process to run.

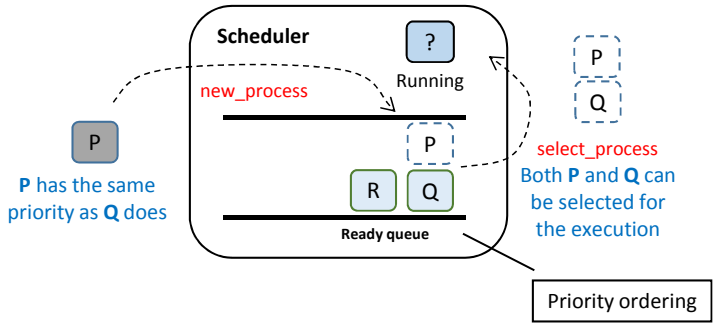


Figure 1.6: Ordering processes

We perform the scheduling tasks based on handling events (called *scheduling events*), which are specified in *scheduler description* (b). In this research, we indicate two types of scheduling events: *system scheduling events* and *process scheduling events*.

- The *system scheduling events* (with fixed names) are handled automatically by the scheduler. These events are `new_process`, `select_process`, and `clock` to specify the behaviors of the scheduler corresponding to three cases: 1) a new process arrives at the system, 2) the scheduler selects a process to run, and 3) a timer event occurs[4], respectively. Each event is handled by an event handler.

- The *process scheduling events* are raised by the process (e.g. terminates itself). These events are defined by users and can be declared in the *interface* part (*) (called interface functions) as indicated in Figure 1.5.

Figure 1.6 depicts an example for handling the events. When process `P` arrives at the system (`new_process`), it is put to the `ready` queue. The scheduler selects a process to run (`select_process`) from this queue (both of process `P` and process `Q` are considered). Several statements are introduced to perform the scheduling events. These statements are to select a process, change their statuses and attributes.

In this work, we consider individual behaviors of processes. Therefore, we use the discrete time to hand the time with behaviors of the system by considering an action of a process as taking one time unit. That means performing an action will consume 1 tick. To capture the time, we introduce the variables with *clock* type. Corresponding to each action of the processes, each clock variable is increased by 1 and the system will raise a timer event (`clock`). This event is handled by the scheduler. We use timed-Kripke structure [36] to model the system. The periodic behaviors are also considered using a loop. A loop is indicated when a visited state is reached. That determines a period. For instance, Figure 1.7 shows a system with a periodic process; this process needs 2 time units to perform its tasks in period 4. We now only consider the duration of 4 time units because the state of the system at time 4 is similar to that at time 0.

We prepare the information necessary to explore the states beforehand. The *scheduling information* is generated from the scheduling strategy in the DSL (as depicted in Figure 1.4). This *scheduling information* contains the functions for performing the scheduling tasks (i.e. handling the scheduling events) and the information to indicate

[4]We consider an action as taking one time unit and raising a timer event.

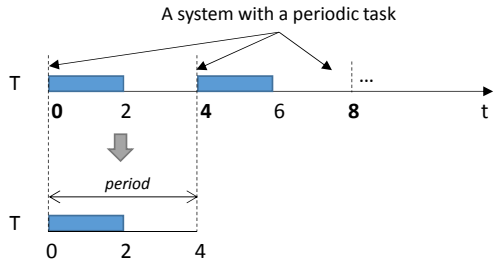


Figure 1.7: A system with a periodic task

the system state. A search algorithm (3) is realized (Figure 1.4) from the scheduling strategy to explore the state space following the policy in the DSL.

1.4.3 Verifying the Behaviors of a System

The behaviors of a system are determined by the behaviors of the scheduler and the behaviors of the processes. To model checking a system with the scheduler, we need to explore the state space. To do this, we generate all necessary information from the scheduling policy (in the DSL) to realize the search algorithm. The state space is determined according to the policy.

With the state space, we can verify whether the system can satisfy an invariant or a linear temporal logic (LTL) property, which is a kind of qualitative properties. In fact, linear temporal logic (LTL) or computation tree logic (CTL) are used for verifying reachability, liveness, and safety properties without any quantitative measurement. However, these properties cannot specify a time-bounded information. Actually, the behaviors of a real-time system are always related and bounded to time; when we consider the executions based on the management of the scheduler, the quantitative properties are also important. An example for the property is that *"the process needs to finish its task within 3 time units"*. This kind of property is proposed by Emerson [36] and represented as a real-time computation tree logic (RTCTL) formula to indicate a quantitative property of the time elapsed during the computation. For instance, the formula $\mathtt{AF}^{\leq 3}\mathtt{(p)}$ means that property `p` holds within 3 time units.

We note that the search algorithm now limits the state space following the scheduling strategy. Using the graph indicated by the state space, we can apply the algorithm introduced in [24] to check the property represented as a CTL formula. To handle the quantitative property, we adopt the algorithm proposed by Emerson et al. [36] to check the properties expressed in the form of RTCLT formula. For the implementation, we follow the method proposed in [18], which is based on [24], to label the graph realized from the search space. To do that, some elements are introduced in the DSL for capturing the behaviors of the system for the analysis. We then apply the existing algorithms (as mentioned above) to labeling the graph (indicated by the search algorithm) following a property defined in a form of a CTL/RTCTL formula.

1.4.4 Testing Scheduling Policies

We aim to verify concurrent systems executed under scheduling policies. The scheduler of an OS now controls the executions of the processes. To do that, we propose a DSL to describe the strategy. The correspondence between the scheduling policy in the DSL and the behaviors of the real scheduler affects the verification results. In fact, the description of the strategy for a real system does not exist or is not clear to describe the behaviors of the scheduler. The remaining problem now is ensuring that the scheduling strategy described in the DSL conforms with the implementation of a real scheduler. Actually, we can observe the behaviors of the scheduler only in executing the system. Because of this fact, we adopt testing techniques to check the correspondence between the specification in the DSL and an implementation of the scheduler in a real OS. Our method is checking whether the scheduling policy in the DSL is accepted by the behaviors of the real scheduler.

For preparing the tests, we still have a problem that there are multiple possible executions of a concurrent system. That lead to the fact that manually making the tests is time-consuming and prone to errors. To deal with this fact, we apply MBT techniques to overcome this problem. With MBT, we can automatically and exhaustively generate the tests for checking the policy with the behaviors of a real scheduler. We describe the overview of our approach for testing the scheduling policy below. The method is introduced in more details in Chapter 5.

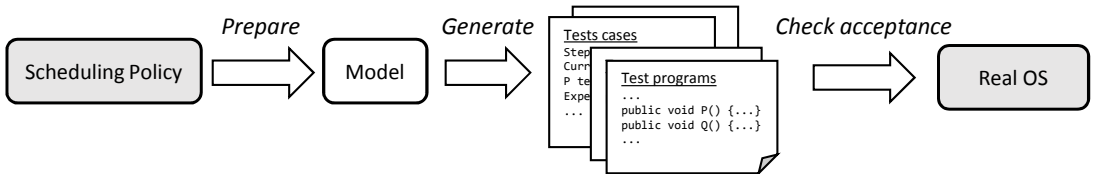


Figure 1.8: Testing approach

- Firstly, to apply the MBT techniques, we prepare the model of the system with the corresponding environment needed for performing the scheduling policy. Actually, the environment indicates a set of processes with their attributes. Note that we can only handle limited cases for testing a real system.
- Secondly, using the model, we can indicate the state space by searching. We then generate the tests (test cases and test programs) by mapping the behaviors of the system with the code generated.
- Lastly, we perform the tests to validate the scheduling policy with the behaviors of the scheduler in a real OS.

1.5 Thesis Structure

Following the approach introduced above, this thesis structured as follows.

- Chapter 1 (this chapter) devotes our motivation, the objective of the research, the overview of the approach, and the contributions.

- Chapter 2 presents the background of concurrent systems, model checking techniques, the properties to be verified, scheduling policies, and MBT techniques.
- Chapter 3 introduces the DSL for the scheduling policies with the design and the semantics of the language.
- Chapter 4 focuses on verifying the behaviors of the system. This chapter introduces the algorithm to explore the state space following the scheduling strategy and the method for analyzing the systems qualitatively and quantitatively by adopting the labeling algorithms indicated in [36, 18, 24].
- Chapter 5 presents our approach to check the description of the policy in the DSL with the real scheduler in an OS.
- Chapter 6 introduces the implementation of our approach.
- Chapter 7 shows the experiments for verifying system with the scheduler, analysis the behaviors of the system, and testing the scheduling strategies.
- In Chapter 8, we discuss the results of verifying the behaviors of the system under the scheduling strategies with our approach.
- Chapter 9 shows the related work.
- Lastly, Chapter 10 concludes our research and shows some future directions.

Chapter 2

Background

This chapter describes the background of this research. First, we consider the characteristics of a concurrent system and model checking techniques. Then, we introduce the properties, which need to be verified to ensure the correctness of the systems. Next, we introduce Promela language and the related model checking tools. The scheduling domain, the DSL, and MBT techniques are considered at the end of this chapter.

2.1 Concurrent Systems

A concurrent system has multiple computations where each of them can be performed without delay to wait for the others to complete [77]. That means in a concurrent system, the computations can perform concurrently with the time overlapping instead of sequentially (in which one completed before the next starts). The implementation of each computation can be taken with an OS process or using a set of threads in the same process [81]. We now use "*process*" to imply both of "*process*" and "*thread*". Actually, these processes can run in an interleaving manner on a processor with a single core using the time-slices. At a time, only one process can run. If the execution does not complete in the time allowed, this process is paused to make another one run. The execution of these processes is control by a scheduler.

The main difference between a concurrent system and a parallel system is that the executions of the processes in a parallel system occur at the same time (such as, on a multi-processor computer [77]), to increase the speed of the computation. That means the parallel execution cannot be performed on a single processor. In contrast, a concurrent system contains multiple processes that can share the time (lifetimes overlapping). Figure 2.1 depicts the executions of a system with two processes, which are executed at the same time (parallelly) (a) and alternately (concurrently) (b).

In this type of systems, the processes can communicate using shared memory or message passing methods [77]. To use the communication with the shared memory,

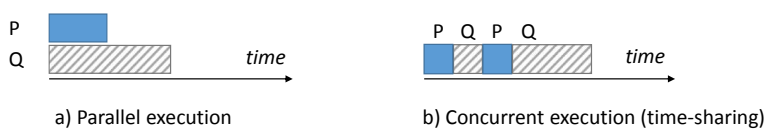


Figure 2.1: Parallel and concurrent executions

the implementation usually requires some kinds of control (such as mutexes) and the processes are allowed to update the contents of the memory. With message passing, the processes can exchange the information (i.e. messages) with asynchronous or synchronous "rendezvous" style.

The main advantage of concurrent execution is to increase the number of tasks to be finished in a period of time. However, because a concurrent system can have many different executions, it is difficult to be verified.

2.2 Real-Time Systems

A real-time system relies on not only the results of the execution but also the time when its computation is performed [78]. This kind of systems usually follow the time constraints, such as the system must complete its tasks within the limited amount of time. With this characteristic, each task/process of a real-time system has several attributes, such as initial offset (the time offset for releasing the task), best case execution time ($BCET$), worst-case execution time ($WCET$), the time between task releases ($PERIOD$) with minimum time between task release (MIN_PERIOD) and maximum time between task releases (MAX_PERIOD), deadline ($DEADLINE$), and priority ($PRIORITY$). These attributes satisfy the following constraints $BCET \leq WCET \leq DEADLINE \leq MIN\_PERIOD \leq MAX\_PERIOD$.

One of the essential objectives for the real-time systems is dealing with schedulability problem [18], which is described with the purpose that all processes will meet the deadlines. One of the methods to handle this problem is analysis. This technique is to determine the conditions on the design to indicate the feasibility. With this problem, the existing approaches use constraints solving [42], deal with fixed scheduling [41], or base on worst-case assumption [82]. Actually, each technique has its own limitations, for instance, with the worst-case assumption, the behaviors of the scheduler will not be taken into account in the analysis.

For non-real-time systems, the OS aims to provide an interface for the communication between the programs and the hardware while attempting to maximize average throughput, to minimize the waiting time, and/or to ensure the fairness to share the system resources. However, meeting the deadlines of tasks/processes is not a purpose since these systems do not consider the deadlines to make the scheduling decisions.

2.3 Model Checking

Testing [49] and simulation [70] are the two techniques for finding errors existing in a system.

- With testing, a set of inputs are used to check the corresponding outputs or the behaviors of a system respect to the specification. With real-time systems, both of input/output values and the time at which they are produced must be considered.
- With simulation, we execute possible actions on the model of a system (or a physical entity) to validate the system.

These two techniques are only good for finding existing errors. However, they usually cannot ensure that a system can satisfy the corresponding requirements. To address this problem, formal verification is applied.

Model checking [9] is a method to exhaustively and automatically verify the finite-state concurrent systems. The techniques use algorithms for searching all states of the system using its model to check whether the desired behaviors of the system can meet a given specification. For the specifications of a system, temporal logic [69] is used. The model of the system is traversed using the appropriate algorithms to check the properties. That means the models of a system are followed by algorithms to explore the states systematically. This is a technique to verify finite-state systems automatically.

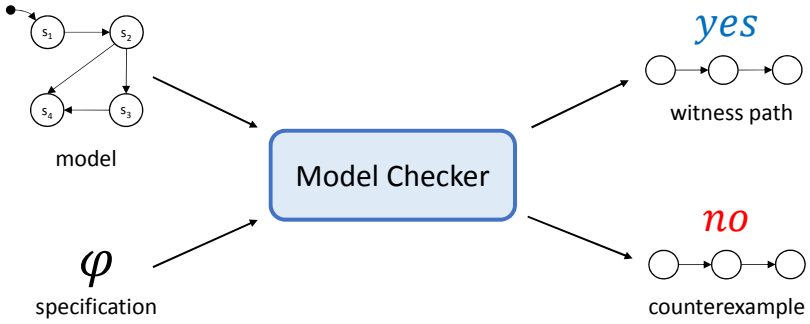


Figure 2.2: Model checking

We start with the basic definition for a concurrent system. Let AP be a set of atomic propositions, e.g. propositions variables and constants.

Definition 2.1 A Kripke structure M is a tuple $M = (S, I, T, L)$, where

- $S = \{s_1, s_2, ..., s_n\}$ is a finite set of states;
- $I \subseteq S$ is the set of initial states;
- $T \subseteq S \times S$ is the transition relation, where $\forall s \in S, \exists s' \in S$ such that $\langle s, s' \rangle \in T$;
- $L : S \rightarrow 2^{AP}$ is a labeling function, where for a state $s \in S$, the set $L(s)$ is made of the atomic propositions that hold in s.

We will refer to state set S of Kripke structure M with a dot notation $M.S$. Similarly, the initial state of M is referred as $M.I$, etc. Figure 2.3 represents a Kripke structure $M = (S, I, T, L)$, where

- $S = \{s_1, s_2, s_3, s_4\}$;
- $I = \{s_1\}$;
- $T = \{(s_1, s_2), (s_2, s_3), (s_2, s_4), (s_3, s_4), (s_4, s_3)\}$;
- $L = \{(s_1, \{p\}), (s_2, \{p\}), (s_3, \{q\}), (s_4, \{q, r\})\}$.

Timed-Kripke structures [36] improve the Kripke structures by labeling the transitions between the states with integer values. We have $(s, a, s') \in T$ if there is a transition from state s to state s' taking a time unit. A timed-Kripke structure $M = (S, I, T, L)$ defined as a Kripke structure except that $T \subseteq S \times \mathbb{N} \times S$. An example of a timed-Kripke structure is depicted in Figure 2.4, where

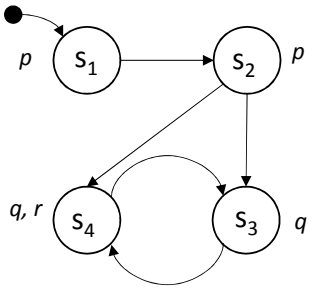


Figure 2.3: A Kripke structure

- $S = \{s_1, s_2, s_3, s_4\}$;
- $I = \{s_1\}$;
- $T = \{(s_1, 1, s_2), (s_2, 2, s_3), (s_2, 2, s_4), (s_3, 3, s_4), (s_4, 4, s_3)\}$;
- $L = \{(s_1, \{p\}), (s_2, \{p\}), (s_3, \{q\}), (s_4, \{q, r\})\}$.

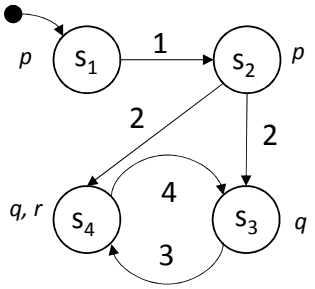


Figure 2.4: A timed-Kripke structure

Definition 2.2 A path π of a Kripke structure M is an infinite sequence $s_0, s_1, ...$ of states, where $\forall i, \langle s_i, s_{i+1} \rangle \in T$, $\pi^i = s_i, s_{i+1}, ...$ is the subpath of π starting from state s_i, and $\pi(i)$ denotes the i-th state s_i of path π. The state $\pi(0) \in I$ (of a path π) is called an initial state.

In Figure 2.3, M may produce a path $\pi = s_1, s_2, s_3, s_4, s_3, ...$

A model checker will explore all states s (of M) that are reachable from the initial states to determine whether the given property p holds or not. If the given property p holds in the state transition graph M, the trace in the graph M can be considered as a witness. If the given property p does not hold, the corresponding counterexample is shown.

Model checking has several advantages [23], such as the checking process is automatic and the counterexamples are valuable for finding the errors of the system. However, this technique can lead to the state explosion problem. It is because the number of states can be huge. In addition, because model checking uses the model for the verification, rather than the real one, this technique does not guarantee the accuracy of a real system.

2.4 Exploring the State Space

There are some algorithms to construct the state space (graph) with a given model of a system. Among them, depth-first search (DFS) and breadth-first search (BFS) are the common ones used by the model checkers for the exploration. The idea is that starting at the initial state and for each possible execution (next statement), we perform this statement and construct a new state. A directed edge between the current state and the new one is created. We repeat this step until there are no states to be considered.

Depth-First Search Algorithm (DFS). It is an algorithm to travel the graph data structure. The algorithm starts at the initial node of the graph then explores along the current branch as far as possible before trying another one. The algorithm is depicted in Algorithm 1 to visit every state in the set $M.S$, which can be reachable from the initial state $s_0 \in M.I$.

Algorithm 1 *Depth-first search algorithm*

```
1:  Input: s_0                                          ▷ initial state
2:  Output: SS                                          ▷ state space
3:  procedure START()
4:      Stack: ST = {}
5:      State space: SS = {}
6:      Push(ST, s_0)
7:      Add_state(SS, s_0)
8:      SEARCH()
9:  end procedure
10: procedure SEARCH()
11:     s = Top(ST)
12:     for ⟨s, s'⟩ ∈ M.T do
13:         if Contains(SS, s') == false then
14:             Push(ST, s')
15:             Add_state(SS, s')
16:             SEARCH()
17:         end if
18:     end for
19:     Pop(ST)
20: end procedure
```

We use two data structures: a state space $\mathcal{SS}$ and a stack $\mathcal{ST}$. To update the contents of the state space (a set of states), we use the following functions: $Add\_state(\mathcal{SS}, s)$ to add state s, and $Contains(\mathcal{SS}, s)$ to check whether element s exists in the state space. A stack is an ordered set of states with the corresponding operations: *Push* (to add a state), *Top* (to return the top element) and *Pop* (to remove an element) to record the searching. The search is performed starting from function START (line 3) to visit every reachable state from the starting state s_0. An example is represented in Figure 2.5; with the starting node a, the algorithm will visit the states in the following sequence: a, b, d, c, e, f.

Breadth-First Search Algorithm (BFS). This algorithm starts at the initial node of a graph to explore all nodes at the same level before trying the neighbors in the next level. The main difference between BFS and DFS is that BFS uses a queue

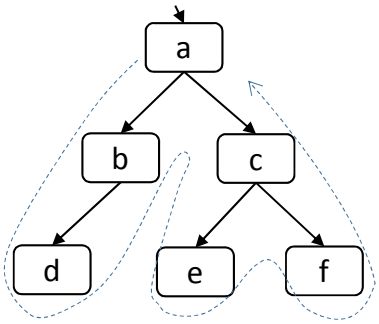


Figure 2.5: Depth-first search

to record the search steps instead of a stack. Two operations for the queue: ***EnQueue*** (to add a state) and ***DeQueue*** (to remove a state) are used for performing the search. The algorithm is shown in Algorithm 2. Figure 2.6 shows an example of this algorithm; starting at node a, the algorithm will visit the states in this sequence: a, b, c, d, e, f.

Algorithm 2 *Breath-first search algorithm*

```
1:  Input: s_0                                          ▷ initial state
2:  Output: SS                                          ▷ state space
3:  procedure START()
4:      Queue: Q = {}
5:      State space: SS = {}
6:      EnQueue(Q, s_0)
7:      Add_state(SS, s_0)
8:      SEARCH()
9:  end procedure
10: procedure SEARCH()
11:     s = DeQueue(Q)
12:     for ⟨s, s'⟩ ∈ A.T do
13:         if Contains(SS, s') == false then
14:             EnQueue(Q, s')
15:             Add_state(SS, s')
16:         end if
17:     end for
18:     SEARCH()
19: end procedure
```

2.5 Qualitative and Quantitative Properties

There are different ways to specify properties for verifying the behaviors of a system. In this research, we consider two types of properties:

1. qualitative properties, which are specified in temporal logic (LTL and CTL), and
2. quantitative properties, which are in discrete-time logic (RTCTL).

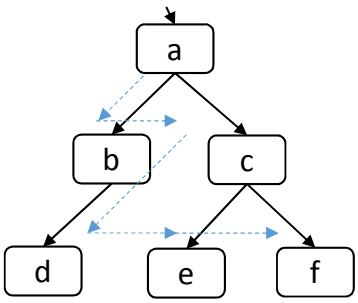


Figure 2.6: Breadth-first search

The formulae in LTL refers to time and can encode the future of paths. A well-formed formula, ϕ, is defined as follow:

$$
\begin{array}{rcll}
\phi & ::= & \top & \text{;top, or true} \\
 & | & \bot & \text{;bottom, or false} \\
 & | & p & \text{;p ranges over AP} \\
 & | & \neg\phi & \text{;negation} \\
 & | & \phi \wedge \phi & \text{;conjunction} \\
 & | & \phi \vee \phi & \text{;disjunction} \\
 & | & \phi \rightarrow \phi & \text{;implies} \\
 & | & X\phi & \text{;next time} \\
 & | & E\phi & \text{;eventually} \\
 & | & G\phi & \text{;always} \\
 & | & \phi U \phi & \text{;until}
\end{array}
$$

Let π be a path in Kripke structure M, ϕ be an LTL formula. The LTL formula ϕ is held by the path π iff $M, \pi \models \phi$, where

$$
\begin{array}{lll}
M, \pi \models \top & & M, \pi \not\models \bot \\
M, \pi \models p & \text{iff} & p \in L(\pi(0)) \\
M, \pi \models \neg\phi & \text{iff} & M, \pi \not\models \phi \\
M, \pi \models \phi_1 \wedge \phi_2 & \text{iff} & M, \pi \models \phi_1 \text{ and } M, \pi \models \phi_2 \\
M, \pi \models \phi_1 \vee \phi_2 & \text{iff} & M, \pi \models \phi_1 \text{ or } M, \pi \models \phi_2 \\
M, \pi \models \phi_1 \rightarrow \phi_2 & \text{iff} & M, \pi \not\models \phi_1 \text{ or } M, \pi \models \phi_2 \\
M, \pi \models G\phi & \text{iff} & \text{for all paths } \pi^i \text{ for all } i \geq 0, M, \pi^i \models \phi \\
M, \pi \models F\phi & \text{iff} & \text{for all paths } \pi^i \text{ for some } i \geq 0, M, \pi^i \models \phi \\
M, \pi \models X\phi & \text{iff} & M, \pi^1 \models \phi \\
M, \pi \models \phi_1 U \phi_2 & \text{iff} & \text{exists } i \geq 0 \text{ such that } M, \pi^i \models \phi_2 \text{ and for all } j < i, M, \pi^j \models \phi_1
\end{array}
$$

A CTL formula consists of a set of atomic propositions with operators `NOT`, `AND`, `AX`, `EX`, `AF`, `EF`, `AG`, `EG`, `AU`, and `EU`. The formula, ϕ, is defined as follows: $\phi ::= \top \mid \bot \mid p \mid \neg p \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \rightarrow \phi \mid AX\phi \mid EX\phi \mid AF\phi \mid EF\phi \mid AG\phi \mid EG\phi \mid A[\phi U \phi] \mid E[\phi U \phi]$.

The operators `NOT` and `AND` are straightforward as their meanings. The remains operators refer to a path as a sequence of states (`A` : All, `E` : Exists, `X` : Next, `F` : Finally, `G` : Globally, `U` : Until). The semantics are indicated as follows.

- $s_0 \models p$ iff $p \in L(s_0)$
- $s_0 \models \neg p$ iff $s_0 \not\models p$
- $s_0 \models \phi_1 \wedge \phi_2$ iff $s_0 \models \phi_1$ and $s_0 \models \phi_2$
- $s_0 \models \phi_1 \vee \phi_2$ iff $s_0 \models \phi_1$ or $s_0 \models \phi_2$
- $s_0 \models \phi_1 \rightarrow \phi_2$ iff $s_0 \not\models \phi_1$ or $s_0 \models \phi_2$
- $s_0 \models \mathtt{AX}\phi$ iff for all paths $\sigma = s_0, s_1, ..., s_1 \models \phi$
- $s_0 \models \mathtt{EX}\phi$ iff for some path $\sigma = s_0, s_1, ..., s_1 \models \phi$
- $s_0 \models \mathtt{AG}\phi$ iff for all paths $\sigma = s_0, s_1, ..., \forall i,\ s_i \models \phi$
- $s_0 \models \mathtt{EG}\phi$ iff for some path $\sigma = s_0, s_1, ..., \forall i,\ s_i \models \phi$
- $s_0 \models \mathtt{AF}\phi$ iff for all paths $\sigma = s_0, s_1, ..., \exists i,\ s_i \models \phi$
- $s_0 \models \mathtt{EF}\phi$ iff for some path $\sigma = s_0, s_1, ..., \exists i,\ s_i \models \phi$
- $s_0 \models \mathtt{A}(\phi\ \mathtt{U}\ \psi)$ iff for all paths $\sigma = s_0..., \exists i \geq 0$, such that $s_i \models \psi$ and $\forall j, 0 \leq j < i, s_j \models \phi$
- $s_0 \models \mathtt{E}(\phi\ \mathtt{U}\ \psi)$ iff for some path $\sigma = s_0..., \exists i, i \geq 0$, such that $s_i \models \psi$ and $\forall j, 0 \leq j < i, s_j \models \phi$

The minimal set of operators is $\{\mathtt{EG}, \mathtt{EU}, \mathtt{EX}\}$. That is because the other modalities can be declared as abbreviations, such as $\mathtt{AF}\phi$ abbreviates $\mathtt{A(true\ U}\ \phi\mathtt{)}$, $\mathtt{EF}\phi$ abbreviates $\mathtt{E(true\ U}\ \phi\mathtt{)}$, $\mathtt{AG}\phi$ abbreviates $\neg\mathtt{EF}\neg\phi$, $\mathtt{EG}\phi$ abbreviates $\neg\mathtt{AF}\neg\phi$, and $\mathtt{AX}\phi$ abbreviates $\neg\mathtt{EX}\neg\phi$.

An LTL/CTL formula can specify the order of the behaviors of a system. That means LTL/CTL can specify the ordering of events/actions. The properties in an LTL/CTL formula can be used for verifying reachability, liveness and safety properties, but they cannot specify when these events occur. In particular, real-time systems are always bounded on the response time. To handle the quantitative measurement, CTL has been extended. Below we describe an extension proposed by Emerson [36] that consider the discrete time to handle quantitative assertions called RTCTL. Each action of the system consumes 1 tick (corresponding to one time unit to be finished). The formula in this logic represents the bounded time. For instance, the RTCTL formula $\mathtt{EF}^{\leq 3}\phi$ states that property ϕ eventually holds within the bounded time, namely, 3 time units. The meaning of the operations is as follows.

- $s_0 \models \mathtt{AF}^{\leq n}\ \psi$ iff for all paths $\sigma = s_0..., \exists i, 0 \leq i \leq n$, such that $s_i \models \psi$
- $s_0 \models \mathtt{EF}^{\leq n}\ \psi$ iff for some path $\sigma = s_0..., \exists i, 0 \leq i \leq n$, such that $s_i \models \psi$
- $s_0 \models \mathtt{AG}^{\leq n}\ \psi$ iff for all paths $\sigma = s_0..., \forall i, 0 \leq i \leq n$, such that $s_i \models \psi$
- $s_0 \models \mathtt{EG}^{\leq n}\ \psi$ iff for some path $\sigma = s_0..., \forall i, 0 \leq i \leq n$, such that $s_i \models \psi$
- $s_0 \models \mathtt{A}(\phi\ \mathtt{U}^{\leq n}\ \psi)$ iff for all paths $\sigma = s_0..., \exists i, 0 \leq i \leq n$, such that $s_i \models \psi$ and $\forall j, 0 \leq j < i, s_j \models \phi$

- $s_0 \models \mathtt{E}(\phi\ \mathtt{U}^{\leq n}\ \psi)$ iff for some path $\sigma = s_0...$, $\exists i, 0 \leq i \leq n$, such that $s_i \models \psi$ and $\forall j, 0 \leq j < i, s_j \models \phi$.

In these formulas, n specifies a time-bound corresponding to the maximum number of permitted transitions along a path. We note that $\mathtt{AF}^{\leq n}$ and $\mathtt{EF}^{\leq n}$ are special cases of $\mathtt{AU}^{\leq n}$ and $\mathtt{EU}^{\leq n}$. The other operations can be specified using these operations above.

2.6 Model Checking Algorithms

We can use model checking to check the accuracy of the systems relative to their specifications. A system is presented as a graph (Kripke structure). The specification is declared in propositional temporal logic. The algorithms will then indicate whether the structure models the formula or not.

This section explains the algorithm introduced in [24] to check whether formula f_0 is true or not in the finite structure $M = (S, R, L)$, where S is a finite set of states, $I \subseteq S \times S$ is the transition relation, and $L : S \rightarrow 2^{AP}$ is a labeling function as described in a Kripke structure. The specification is written as a CTL formula. To deal with quantitative property, we consider the algorithm proposed by Emerson et al. [36] to check the properties expressed in the form of RTCLT formula. The algorithms include two steps: 1) building the state graph and 2) labeling the graph following the corresponding formula to check the property.

In the first step, we construct the state graph with the starting state. We then execute the possible statement and examine the changing of the system state. This is the way to construct a new state from an existing state. A directed edge is determined to connect the current state with the new one. We repeat this step until there is no more state.

In the second step, we start labeling the state graph. We note that during constructing the graph, the labels of the current node are initialized. The summary of the algorithm is as follows[1]:

- Firstly, the formula is rewritten in a prefix form, e.g. `f = A` $(\phi$ `U` $\psi)$ will be rewritten as `f = AU` $\phi\ \psi$. The lengths of each sub-formula are determined (e.g. the lengths of `AU`, ϕ, and ψ are 3, 2, and 1, respectively).

- Secondly, the labeling process will be started from the last sub-formula to the first sub-formula of the formula f with the starting state s_0.

  - The first step labels the states with all sub-formulas of f with length 1. Indeed, the formula of length 1 is an atomic proposition and the state has already labeled in the creation of the graph.
  - The second step labels the states with sub-formula of length 2 based on the results in step 1.
  - The next steps are performed the same way.

  Each state of the system will be labeled with a set of sub-formulas with the length less than or equal to i after completing the ith step. We use $label(s)$ to denote this set of state s. After completing all the sub-formulas, the algorithm terminates

[1]The reader can refer to [24] for more information.

at the step $n = length(f)$, now the entire formula f is checked. For all states s, $M, s \models f$ iff $f \in label(s)$ for all subformulas f of f_0.

The primitives below are used to manipulate formulas and access the labels corresponding to states:

- $arg1(f)$ and $arg2(f)$ return the arguments of a temporal operator (e.g. suppose f = `AU` f_1 f_2, we have $arg1(f) = f_1$ and $arg2(f) = f_2$).
- $label(s, f)$ returns a boolean value to indicate whether state s is labeled with formula f or not.
- $add\_label(s, f)$ adds formula f to the set of labels of state s.

Function $label\_graph(f)$ (as depicted in Figure 2.7) handles the formula f (i.e. atomic, or has the form of the operators: `NOT`, `AND`, `AX`, `EX`, `AF`, `EF`, `AG`, `EG`, `AU`, and `EU`) to check the satisfaction of this formula. We consider only `AU` since the other operators are similar. The algorithm use a DFS for exploring the state graph. A stack ST is used and function $stacked(s)$ returns whether state s is on the stack. In this algorithm, a bit array $marked[1 : nstates]$ ($nstates$ is the number of states) indicates the states have been visited.

```
procedure label_graph(f)
begin
    ...
    //AU operator
    begin
        ST := {};
        for all s in S do
            marked(s) := false;
        for all s in S do
            if not marked(s) then
                au( f, s, b)
    end
    ...
end
```

Figure 2.7: CTL model checking algorithm

The procedure $au(f, s, b)$ (as depicted in Figure 2.8) performs recursively the search to check formula f = `AU` f_1 f_2 start with state s. The boolean parameter b is used to indicate the return value of this procedure. That means it will be set to $true$ iff $s \models f$. This procedure checks that state s is already marked or not. If it is marked, we return the result following the label f of s. In the other case, we need to get the results of checking the sub-formulas of f at state s. We consider the case that f_2 is false, this thing is handled using a DFS with the stack ST for checking the sub-formula f_1 of f.

An example for the labeling steps for checking the corresponding formula is depicted in Figure 2.9. With this example, we check the property expressed by formula `AU(true)(t == 0)` meaning that `t == 0` holds at some state in all execution paths of the system. The first step is to build the state space and initialize the labels with the atomic propositions. The second step labels the formula `AU` starting with the state S_0.

```
procedure au(f, s, b)
begin
    if marked(s) then
    if labeled(s, f) then
        begin
            b := true; return
        end
    else
        begin
            b := false; return
        end

    marked(s) := true;
    if labeled(s, arg2(f)) then
        begin
            add_label(s, f);
            b := true; return
        end
    else
        if not labeled(s, arg1(f)) then
            begin b := false; return end
        push(s, ST);
        for all s1 in successors(s) do
        begin
            au(f, s1, b1);
            if not b1 then
                begin pop(ST); b = false; return end
        end;
        pop(ST); add_label(s, f);
        b := true; return
end
```

Figure 2.8: Algorithm for checking formula AU

We use a DFS to travel through the state space (i.e. state graph) to label each state in the case that its children states have not labeled.

To deal with quantitative property, Emerson et al. [36] proposed an algorithm for checking properties in the form of RTCLT. Figure 2.10 shows the algorithm for checking an RTCTL formula with the bound time k. For each sub-formula, the algorithm uses the corresponding procedure to check the property represented by formula f.

In this algorithm, we can see the procedures `AU_check` and `EU_check` indicate an input as the depth to search on the state space (denoted by `min(k, |S|)`). For the recursive function calls, the bound of time `k` can be handled by setting the limit of the depth in the DFS (the superscript indicated in the RTCTL formula, such as 3 in $\mathtt{AU}^{\leqslant 3}$, is used to set the depth. This value indicates the maximum number of steps which can be taken when labeling the graph).

There is a guideline with examples for the implementation of these algorithms in C language, the reader can refer to [18] for more detail.

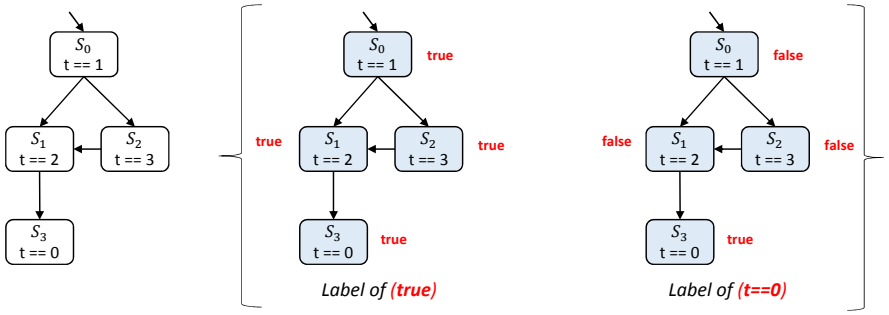


Figure 2.9: An example for labeling the state graph

2.7 Promela (Process or Protocol Meta Language)

Promela is a language introduced by Holzmann [40] for modeling the behaviors of the processes in a concurrent distributed system. The specification of a system in Promela defines the processes, which can be created dynamically, to represent the concurrent tasks in a distributed system. This language provides primitives for the communication between the processes using shared variables or message channels with synchronous or asynchronous (rendezvous or buffered). However, Promela is not a programming language and it only supports limited data structures (without pointers) and does not have functions with return values.

An example for modeling a simple *producer* and *consumer*[2] is depicted in Figure 2.11. With this example, we use `mtype` to define symbolic values. A global variable named `turn` is declared to indicate the execution turn for the *producer* and the *consumer*, which are defined as two processes named `P` and `C` using keyword `proctype`. With the prefix `active`, these processes are automatically instantiated. Each process uses statement "`do ... od`" to perform a loop with keyword "`::`" for each option of the loop. To indicate the guard of each option, condition `(turn == PRO)` and `(turn == CON)` are used. We note that, if no guard is true the process blocks otherwise we can have non-determinism choices. Using the keyword "`->`" and "`;`" are equivalent to indicate the sequence of the corresponding statements.

[2]This example is taken from [46].

```
/* Input: A structure M = (S, R, L) and an RTCTL formula f */
/* Output: There is a state s in S such that s satisfies. */
for i := 1 to length(f) do
begin
    for subformula p of f of length i do
    begin
        switch(p)
            case atomic: ; /* Nothing to do */;
            case (q and r):
                for each s in S do
                    if (q in L(s)) and (r in L(s)) then
                        add_label(s, (q and r));
            case (not q):
                for each s in S do
                    if (q not in L(s)) then add_label(s, (not q));
            case (EX q):
                for each s in S do
                    if q in L(t) for some successor t of s then
                        add_label(s, (EX q));
            case A(q U<=k r):
                AU_check (q, r, min(k, |S|), A(q U r));
            case A(q U r):
                AU_check (q, r, |S|, A(q U r));
            case E(q U<=k r):
                EU_check (q, r, min(k, |S|), E(q U r));
            case E(q U r):
                EU_check (q, r, |S|, E(q U r));
        end;
    end;
end;

if f in L(s) for some s in S then return true
else return false;
```

Figure 2.10: RTCTL model checking algorithm

2.8 Spin Tool

Spin [46] model checker is to verify the consistency of distribute systems. The input of this tool is a Promela program. The tool is to verify the behaviors of a system using its model to check the correctness by performing the simulations randomly or iteratively. Spin tool generates a program (in C language) to verify the system exhaustively. The verification can be taken to prove the accuracy of a system with finding non-progress cycles. It supports verifying LTL formulas using never-claims. With Spin model checker, each model of the system specified by the Promela program can be verified under the assumptions of its environment. When the accuracy of the model of a system is established, it can be used for verifying all the corresponding models. However, the optimized code in C of this tool is difficult to extend. Moreover, it is difficult to reuse the existing algorithms in Spin for other tools.

With the example in Figure 2.11, the initial value for `turn` is `PRO`, the guard `(turn == PRO)` is `true`, thus the *producer* process will execute the next two statements to prints a string and sets the variable `turn` to `CON`. This makes the *producer* process blocked and enables the *consumer* process. These two processes will execute in

```
mtype = { PRO, CON };
mtype turn = PRO;

active proctype P(){
   do
      :: (turn == PRO) -> printf("Producer\n"); turn = CON
   od
}

active proctype C(){
   do
      :: (turn == CON) -> printf("Consumer\n"); turn = PRO
   od
}
```

Figure 2.11: Producer and consumer example

alternating order. With the simulation, Spin produces the following result.

```
> spin producerconsumer.pml | more
Producer
    Consumer
Producer
    Consumer
Producer
...
```

However, if we extend the example with more processes for each type (*producer* and *consumer*) as indicated below:

```
    active [2] proctype P {...}
    active [2] proctype C {...}
```

The alternation is no more guaranteed and Spin may produce the following result in the simulation:

```
> spin producerconsumer.pml | more
    Producer
                Consumer
            Consumer
    Producer
Producer
            Consumer
...
```

The reason is that both two processes `P` can perform statement `printf("Producer")` when the guard holds before the variable `turn` is assigned with `CON`. That fact can be detected by the Spin tool.

2.9 SpinJa Tool

With the optimized C code for the main goal of the speed, the Spin tool is difficult to understand and extend. Besides, the algorithms implemented in this tool are also hard to reuse in other tools.

Following the same approach as Spin, SpinJa [29] is developed in Java using the object-oriented design principle with the aim to extend easily while being competitive in

memory usage and runtime time. This tool generates a program (named *PanModel.java*) from a Promela model to verify the system.

SpinJa is designed following the framework proposed by Mark Kattenbelt et al. [50] to flexibly develop an explicit-state model checker. This also helps to increase the maintainability and reusability of the system. Figure 2.12 show the architecture of SpinJa.

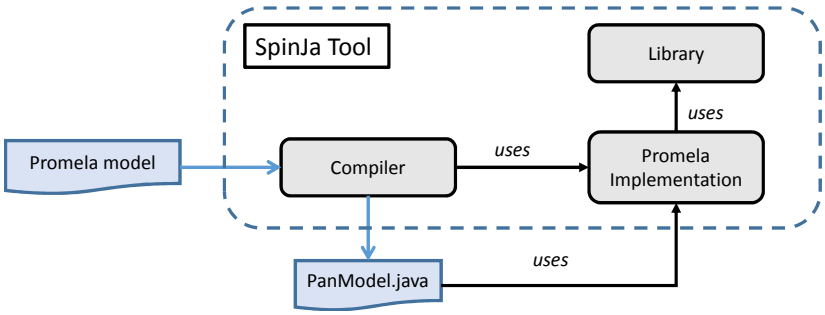


Figure 2.12: The architecture of SpinJa

Some techniques are supported and implemented in this tool, such as bitstate hashing, hash compaction modes, and partial order reduction. We can use this tool to verify the behaviors of a system (e.g. checking the absence of deadlocks, violation of assertions, or liveness properties, etc.) with Promela models using the (nested) DFS or BFS.

2.10 Scheduling Domain

Scheduling is a method that is used to assign resources (such as processor time, bandwidth and memory) to the various processes, data flows, and applications that need them to complete the works. The main purpose of scheduling is to balance the load, ensure the equal distribution of resources on a system according to the set of corresponding rules. This means that a system is able to do its jobs, serve the requests, and guarantee the quality of its services to achieve one or more of many goals such as maximizing the number of tasks per a unit of time (throughput), decreasing the time to finish the tasks with the fairness, or ensuring the processes can meet deadlines in real-time systems. In general, the goals often conflict with each other; therefore, the scheduling strategy will be implemented to keep a suitable compromise depending on the needs of the system.

Process execution states. In an OS, which runs on a central processing unit (CPU), the role of a scheduler is to select a process at a certain time to run. The main task of a scheduler is changing the execution statuses of the processes (e.g. running, ready, blocked, and terminated as depicted in Figure 2.13). The algorithm used by the scheduler is called the scheduling algorithm. Each algorithm represents a policy.

Scheduling events. The main issue related to scheduling is when to make scheduling decisions. In fact, there are a variety of situations in which scheduling is needed. We consider the occurrence of each situation as a scheduling event. The common events are as follows.

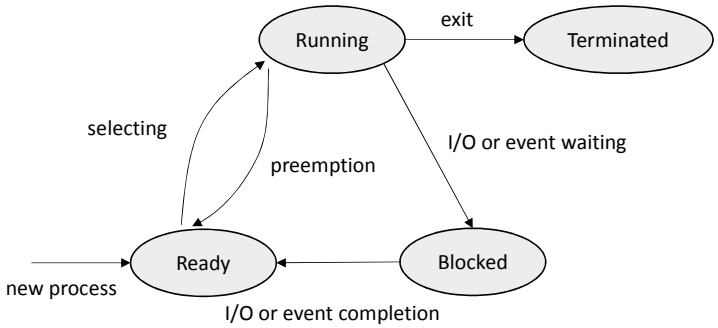


Figure 2.13: State transitions for a process

- Firstly, when a process arrives at the system, the scheduler must decide whether to run this process or not.
- Secondly, when the currently running process blocks waiting for the resource, another process has to be selected.
- Thirdly, when the current process exits, a scheduling decision is made to choose another process for the execution.
- Fourthly, the scheduler must decide which process to run when an interrupt occurs.

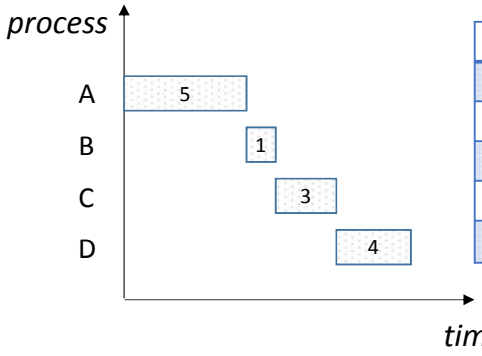
Two types of policies regarding the clock (timer) scheduling event. A non-preemptive algorithm and a preemptive one. The process in the non-preemptive algorithm can run until blocking or it releases the CPU. In contrast, with the preemptive scheduling, the current process can run for an amount of time (time slice). If this process is still running at the end of the time slice, it is suspended and the scheduler selects another process (if it is available).

Process attributes. To perform the scheduling tasks, an OS (such as Linux) usually keep a table (called *process table*) to store the information of its processes. This data structure holds the information of each process (e.g. identifier, priority, environment variables, etc.) called process attributes. The scheduler uses these attributes for performing the scheduling tasks.

In general, the scheduler manages one or several queues for storing the processes which have the same state (such as ready or waiting). Whenever an event occurs (e.g. a process finishes, a new process is released, etc.), the scheduler will search on the queue(s) to find the next process to run based on the attributes.

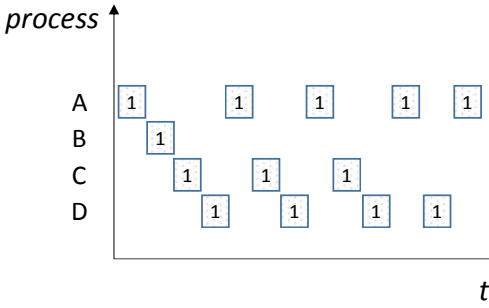
Typical scheduling policies. In theory, there are several policies. Some typical ones are described below.

- First-in, first-out (FIFO): A non-preemptive policy (referred as first-come, first-severed) with the most straightforward as the name suggests. An example of FIFO scheduling policy for a system with 4 processes is depicted in Figure 2.14.
- Round-robin (RR): A preemptive policy. Each task/process in the system is given a time slice to use the resources following FIFO policy. An example of this policy is shown in Figure 2.15.



Process	Service time	Turnaround time	Waiting time
A	5	5	0
B	1	6	5
C	3	9	6
D	4	13	9
Average		8.25	5

Figure 2.14: FIFO scheduling policy



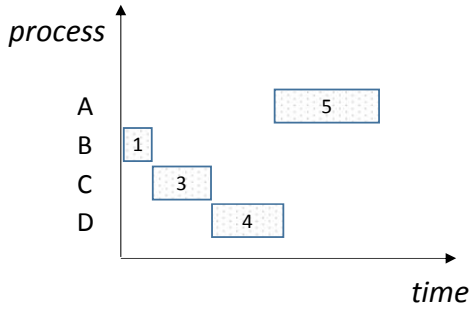
Process	Service time	Turnaround time	Waiting time
A	5	13	8
B	1	2	1
C	3	9	6
D	4	12	8
Average		9	5.75

Figure 2.15: RR scheduling policy

- Shortest job first (SJF): A non-preemptive policy. The process which needs the least amount of time is selected to run. An example of SJF scheduling policy is depicted in Figure 2.16.
- Priority: A non-preemptive policy. Processes are assigned priorities for controlling their executions (depending on their priorities). However, with this policy, the more important process always preempts the least important one. Therefore, it can cause the starvation. An example of priority policy is shown in Figure 2.17.

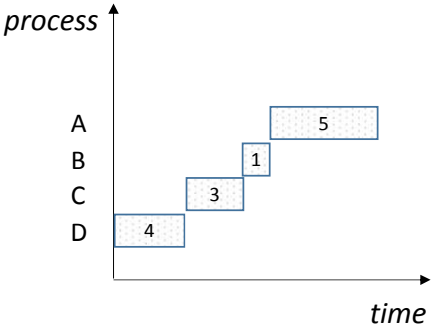
Scheduling policies in OSs. There is a variation of policies implemented in OSs.

- Windows NT OS uses a multilevel feedback queue strategy with different priority levels. Depending on the execution (running time or waiting time), a process can dynamically increase or decrease its priority. The RR policy is used for the high priority processes while the FIFO policy is applied for the low priority ones.
- In Linux, the policies are based on the priorities of the processes. For real-time processes, the FIFO and RR scheduling policies are applied. Linux uses two types of queue named *active* and *expired*. For selecting a process, the scheduler gets the highest priority process by going through the queue of all ready processes (active



Process	Service time	Turnaround time	Waiting time
A	5	13	8
B	1	1	0
C	3	4	1
D	4	8	4
Average		6.5	3.25

Figure 2.16: SJF scheduling policy



Process	Priority	Service time	Turnaround time	Waiting time
A	4	5	13	8
B	3	1	8	7
C	2	3	7	4
D	1	4	4	0
Average			8	4.75

Figure 2.17: Priority policy

processes). After its running time, the current process will be put into the expired queue. These two queues are swapped when the active one is empty.

- OSEK/VDX is a standard for the architecture of the units in vehicles. For the scheduling, non-preemptive and full-preemptive policies with static priority assignment are adopted to control the processes. Particularly, the scheduler manages a queue to determine the running process.

2.11 Domain-Specific Language (DSL)

The purpose. In some situations, there is a necessity to increase the characteristics of a language to deal with a specific problem. A DSL is always designed for a appropriate domain [59] (e.g. SQL [27] for database manipulation). Figure 2.11 shows an example of a query command in SQL. Some DSLs with their domains are given in Table 2.1.

The main aim of a DSL is to create a normally small language to hand the problems in a special domain and is not for the others issues outside of it. That is different from a general-purpose language (GPL) (such as, C++), which are usually for multiple domains and lacks special features for an appropriate domain. Table 2.2 gives a comparison between GPLs and DSLs.

```
SELECT name
FROM  Guest
INNER JOIN Booking ON Booking.guestID = Guest.guestID
GROUP BY city;
```

Figure 2.18: SQL sample code

Table 2.1: DSL examples

DSL	**Domain**
SQL [27]	Database
HTML [43]	Web layout
VHDL [19]	Hardware design
Bossa [10]	Scheduling
Catapults [71]	Scheduling

Types of DSLs. Considering the language syntax and the implementation, there are two types of DSLs [38] as follows.

- An internal DSL is a language embedded into a host GPL using its syntax. The existing languages, e.g. Java or Ruby, are used to describe and implement the DSL. An example for internal DSL code with traditional implementation (the APIs implemented in the existing language) is shown below.

```
include Coffee;
CoffeeCup ord = new Latte(VENTI, MILK);
CoffeeCup cup = ord.set();
```

- An external DSL is represented separately with the language it is working with. This language may have its own syntax, or inspire the syntax of another one (such as XML). The main difference between a GPL and an external DSL is that when the source of the DSL is compiled, it is generally not necessary to output an executable program; it can be in the form of an intermediate representation. An example for external DSL approach with its own syntax is shown below.

```
include Coffee
ord = latte venti, no milk
ask order.set
```

Advantages and disadvantages of DSLs. DSL aims to improve the level of abstraction for the syntax and the semantics of the language. The main purpose is to make the corresponding solutions closer to the problems. Based on this fact, DSLs are popular for the main reasons as follows.

- Creating a DSL allows to solve and focus on a particular domain can help to express the problems clearly in comparison with using an existing GPL.
- A DSL makes it easy to understand the problem and the solution without using complicated codes. Therefore, it can improve the productivity and also make it easier to communicate with domain experts by expressing the ideas with solutions in a common text.

Table 2.2: GPLs and DSLs

	GPLs	**DSLs**
Community size	large	small
Size of domain	large	small
Size of language	large	small
Designed by	committee	experts
Completed	always	often not
Changing	impossible	feasible

- Using a DSL can narrow down the parts of programming, and make them easier to understand, easier to learn with their limited scope. Thus, it helps writing the code faster, modifying it easier, and reducing bugs. The languages allow non-programmers to focus on the important parts of their business.

However, DSLs have some disadvantages based on their characteristics. First, it is difficult to balance between the structure of the language for a particular domain and for the general-purpose one. Second, it is also hard to maintain the scope of the language.

DSLs development. The work-flow for developing a DSL is depicted in Figure 2.19. It contains several steps including domain analysis, designing the language, implementation, and code generation.

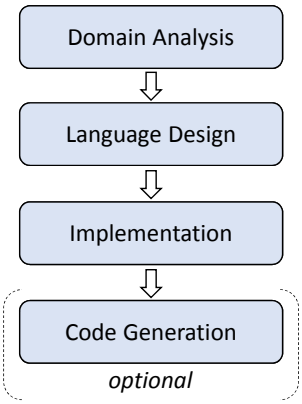


Figure 2.19: DSL work-flow

To design a DSL, because the language is usually abstract and expressive, the domain needs to be aware and the notations of the language should express the semantics of the domain implicitly. It helps the users can use the language statements to describe their purposes, such as making a relation between the syntax and the objects of the domain.

The code generation step is optional based on the purpose of the DSL. For instance, with an external DSL, the script can be converted to an intermediate representation (e.g, program data), or the script in an internal DSL embedded to the host language can run directly and it is not necessary to make any generation.

Code generation. This is optional to develop a DSL. This step starts with the DSL script and parses it. At this step, a syntax tree is determined. The tree is used to present

the semantic model, which indicates the meaning of the script in the corresponding domain. The semantic model is then used to map to the suitable target code (generating the code). Actually, we can directly parse the script into the target code. However, using the semantic model allows separating the two steps: parsing and generation. This separation makes these two steps can be updated easily (when necessary). Figure 2.20 shows the processing of a DSL from the script to the code generated.

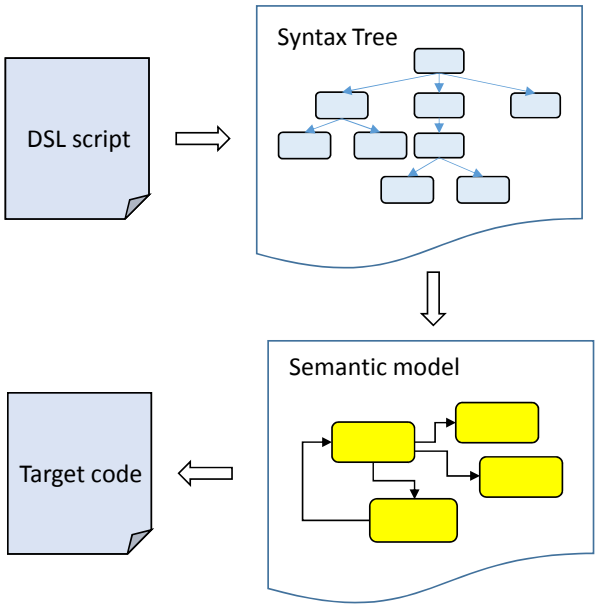


Figure 2.20: Processing of a DSL

There are two main styles for generating the code: *transformer-based* and *template-based* [38] as follows.

- In the *transformer-based* case, we map the semantic model with the corresponding codes in the target one. With this approach, we need to build a program with the semantic model as its input, then export the code in the target environment as its output (as indicated in Figure 2.21). This approach is likely generating each line of code one by one from the DSL script.

- For the *template-based* generation, we must start with designing the sample output code (a template). After that, the semantic model is used to put the corresponding code into this template. That means we structure the code to be generated by inserting the necessary code realized from the DSL script. With this approach, a special component called *template processor* is needed to put the corresponding code into the template (as depicted in Figure 2.22).

2.12 Xtext Framework

Xtext [13] is a framework to develop programming languages and DSLs with the supports for defining the language grammar and providing a full infrastructure including a

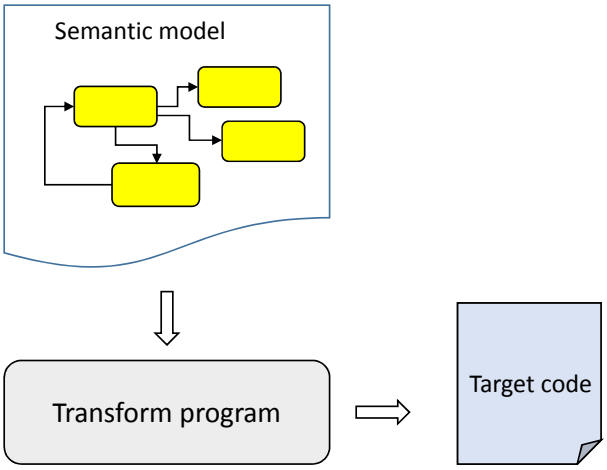


Figure 2.21: Transformer-based generation approach

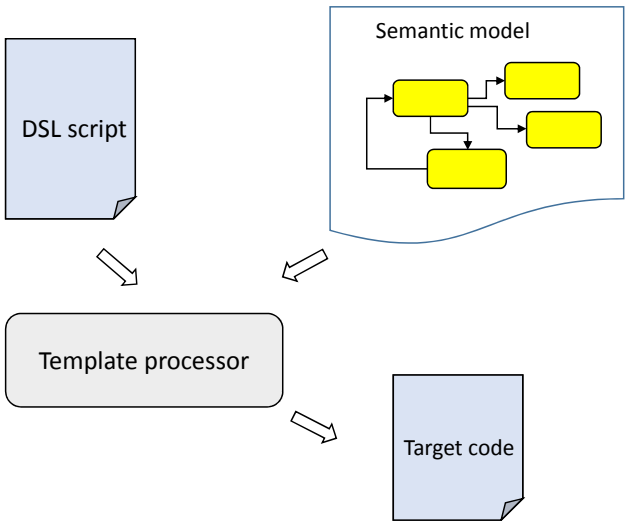


Figure 2.22: Template-based generation approach

parser, a linker, a compiler, and an environment for editing. In addition, Xtext supports the model of abstract syntax tree (AST) corresponding to the DSL script and an IDE environment based on Eclipse tool. The languages and the environment are highly configurable with easily replacing the components. Xtext can facilitate the development of DSLs and support mapping the DSL concepts to the artifacts needed to be generated.

The language grammar is written in Xtext using the EBNF syntax. This framework will generate an ANTLR parser. During the parsing, the AST is generated to construct the corresponding language model. The grammar definition language of Xtext is not just for the parser. Besides, many IDE features provided by Xtext adapt to the language automatically, such as code completion, syntax coloring, or validation, etc.

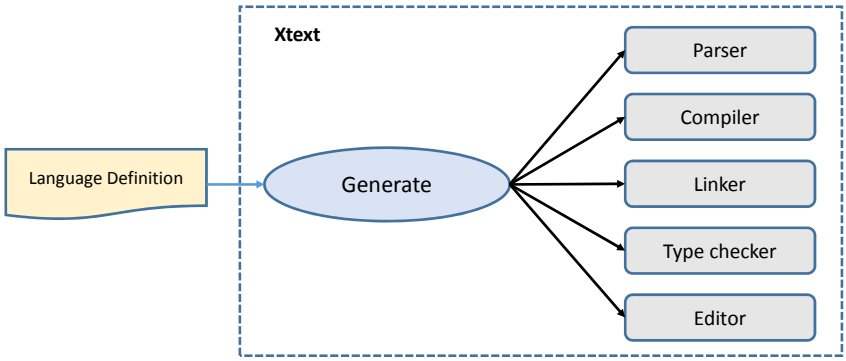


Figure 2.23: The support of Xtext framework

2.13 Model-Based Testing (MBT)

Testing techniques. Testing is used to validate software systems and greatly supports to check the quality of software systems. Nonetheless, with the pre-designed tests (test cases, test programs) following limited execution orders, using this technique is difficult to find unexpected errors. As a fact that there can be multiple executions for a concurrent system, making the tests manually is time-consuming and prone to errors. That means it is hard to apply a manual approach to test different and complex variants of software systems. That motivates a systematic approach towards software testing.

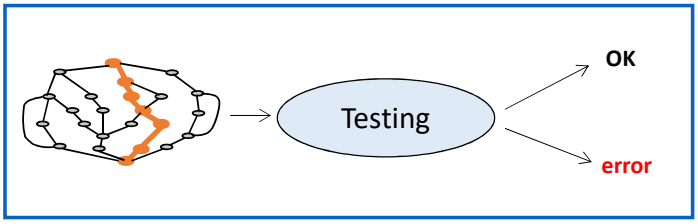


Figure 2.24: Testing technique

Model-based testing. MBT [6] is a technique aims at systematic the generation, execution, and evaluating the results for the design and performing the tests. MBT technique is usually considered as black-box testing [11]. In particular, the behaviors of a system are described by a corresponding model. Using this model, the test suites are derived. The process of MBT is depicted in Figure 2.25.

A model used in MBT is usually abstract and represent the behaviors of the system partially. The tests (test cases, test programs, or test data) get from that model have the same abstraction level as the model. In general, these tests are realized in a test suite cannot execute directly against a system under test (SUT) because of the different abstraction. A test suite can be obtained from the abstract tests by mapping to the concrete ones that are suitable to be executed. Nonetheless, in some cases, the models can have enough information to prepare the test suites directly. In other cases, to make a test suite, the relation between the components and the specific statements in the software is needed. This mapping is used for the test generation.

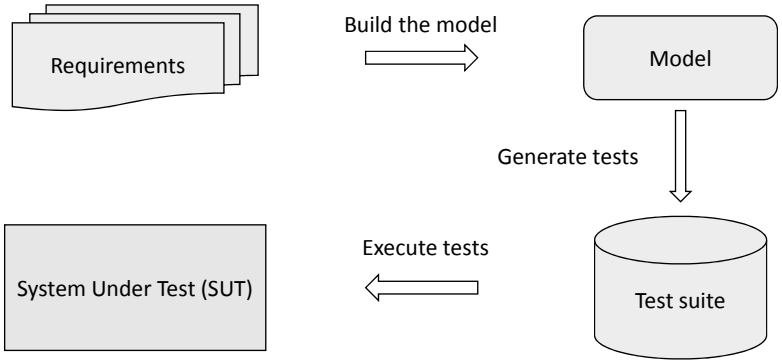


Figure 2.25: MBT process

Test generation using MBT. Some languages for modeling systems, such as UML, SysML, Z, Event-B, or Alloy are used for generating the tests [61] using several techniques [72]. The effectiveness of MBT is derived from the automation it offers based on the desire behaviors represented by the model. We can derive the test mechanically if the model can be read by the machine. In fact, the model is converted to a transition graph to indicate the states of the SUT. The graph is then searched for finding the corresponding paths [58] to realize the tests (as indicated in Figure 2.26). However, because the number of the states is usually huge, the number of paths can be large. To generate the appropriate tests, we need a suitable criteria [1].

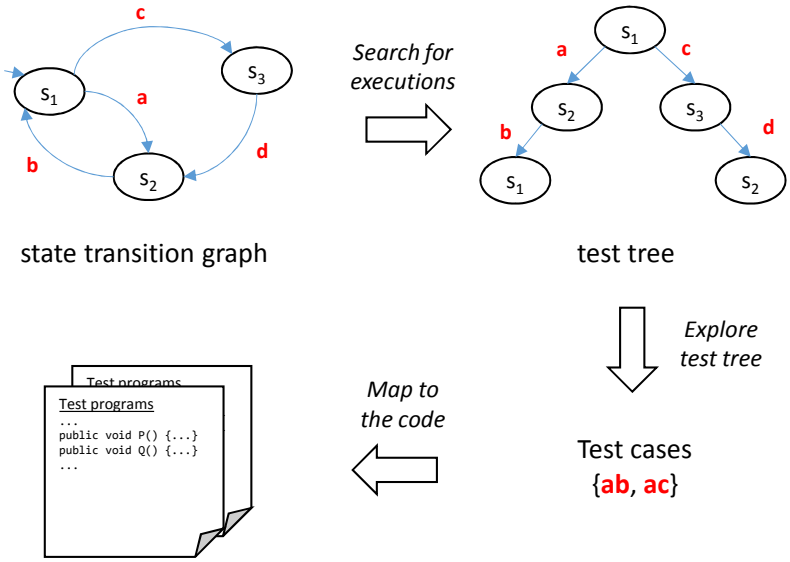


Figure 2.26: Test generation approach

The tests can be generated using model checking techniques [39]. In fact, for verifying a system, a model and a property are used as an input of the model checker. Then, using an exhaustive search, the model checker discovers the corresponding witnesses or counterexamples. A path is called a witness when the property holds. If the property is violated [14], this path is called a counterexample. Each path is used as a test case for generating the tests.

There are methods and tools, such as SAL [44] and STG [22] used for the test generation. SAL uses a specification with Boolean *trap variables* representing the goals to generate the tests. The idea behind that is to construct input sequences that lead the SUT exhibit the expected behaviors (*test goals*). These goals are derived from the requirements or the input domain. STG use symbolic generation techniques [73] to handle state space explosion problem. This technique uses the specification of a SUT, and compute a program to detect the non-conformance behaviors.

2.14 Summary and Discussion

This section discusses our approach with the relation to the background to achieve the objective indicated in Chapter 1.

- Firstly, we aim to deal with systems executed under scheduling strategies. As indicated at the beginning of this chapter, a concurrent can have multiple processes executed in different ways. Therefore, it is hard to verify this kind of systems. Because model checking can exhaustively and automatically find possible errors by exploring every running order of the system, we can apply this technique to verify the concurrent systems.

- Secondly, to deal with the scheduling policy, we introduce a DSL to describe the behaviors of the scheduler used in model checking techniques.

  - a DSL focuses on a specific domain, therefore, it is easy to use in comparison with using a GPL or an existing modeling language, and
  - the existing languages in the scheduling domain (e.g. Bossa [10] and Catapults [71]) cannot be applied or reused because of their target systems and the techniques behind (i.e. for the schedulers implementation).

  Here, we design our language with its own syntax as an external DSL to handle the scheduling tasks. These tasks correspond to the behaviors of the scheduler, especially, for selecting a process to run (see Chapter 3 for more details). To apply model checking techniques, we consider the processes can be selected to run by partially ordering these processes in the system.

- Thirdly, to describe the processes of a system, we use Promela as the base modeling language. It is because this language can specify the concurrent behaviors of a set processes in a system. In reality, a process can perform the scheduling tasks (e.g. terminates itself). Thus, we provide a mechanism for defining the interaction between the processes and the scheduler. This is done through the interface specified in the DSL.

- Fourthly, to explore the system states, we propose a search algorithm with considering the behaviors of the scheduler. That is the difference from the existing algorithm (e.g. DFS and BFS). To do this, all the information necessary for the policy are prepared beforehand. The algorithm is introduced in Chapter 4.

- Fifthly, the scheduler in a system manages the time. Because we consider individual behaviors, we use the discrete time as an action taking one time unit to

deal with the time related to the behaviors of the system. This approach allows to handle the quantitative property by adopting Emerson approach [36] and using the existing algorithms [18, 24] to check the properties expressed in the form of a CTL/RTCTL formula. This is done by labeling the graph constructed by the search algorithm following the scheduling strategy (see Chapter 4).

- Sixthly, to increase the confidence of the scheduling strategy specified in the DSL, we adopt testing techniques to check the correspondence between the policy in the DSL and the implementation of the scheduler in a real system. It is different from the conformance testing approach to check whether the implementation conforms to the design or the corresponding specification. To deal with the testing, we apply MBT techniques to generate the tests automatically and exhaustively. Our method is described in Chapter 5.

- Lastly, for the implementation of the DSL, we use Xtext; this framework supports defining the language and a full infrastructure to implement a DSL. In this research, we adopt the approach as used by Spin and SpinJa tools to prepare the necessary information beforehand. We extend SpinJa model checker for the based tool in our approach. That is because this tool follows the object-oriented design principle and it is easy to extend. The details of the implementation are presented in Chapter 6.

Chapter 3

Domain-Specific Language for Scheduling Policies

As we can see, real systems use schedulers to control the executions of the processes. Therefore, in order to accurately verifying the behaviors of the system, we need to take the schedulers into account in the verification. In fact, there is a variation of policies used by the OSs (e.g. Linux uses both of FIFO and RR for real-time policies). Using existing modeling languages, such as Promela, is difficult to handle various types of strategies because the scheduling policy is fixed in the model of a system and it is hard to model interesting policies. In this research, we proposed a DSL to specify the scheduling policies and deal with the variation of schedulers. This chapter gives the design of the language with its grammar and the semantics.

3.1 Overview of the Language Design

In this work, to flexibly change the policy, we separated the scheduler from the processes. This approach is different from that of combining the processes and scheduler into the same model. It helps us to reuse the scheduling policies and the behaviors of the processes. That means we can use the scheduling policy to verify multiple systems, and with a description of the behaviors of the processes, we can adopt different policies. That fact can increase the reusability of the specification.

To facilitate the scheduling strategies, we propose a DSL for specifying the scheduling tasks. The top-level structure of our DSL is depicted in Figure 3.1. Actually, the scheduler uses the attributes of the processes in the systems to perform its behaviors. In order to flexibly change the configuration of the system (i.e. the set of processes and their attributes), we also separate the attributes of the processes (`<ProcDSL>`) and the behaviors of the scheduler (`<SchDSL>`).

The configuration of a system with the set of processes and their initial attributes are determined in the definition of the attributes of the processes. That can be done by defining a) the attributes of the processes (`<ProcAttr>`), b) the process types (i.e. the processes with the same behaviors) with the initial values for the attributes (`<Process>`), and c) the initialization of the processes (`<ProcInit>`). We note that the configuration part (`<ProcConf>`) is for defining the periodic and sporadic processes.

The scheduler handles the scheduling events to perform the scheduling tasks. In our DSL, the *system scheduling events* are defined in the *event handlers* (`<HandlerDef>`) of the scheduler description (`<SchDSL>`). To allow the communication between the

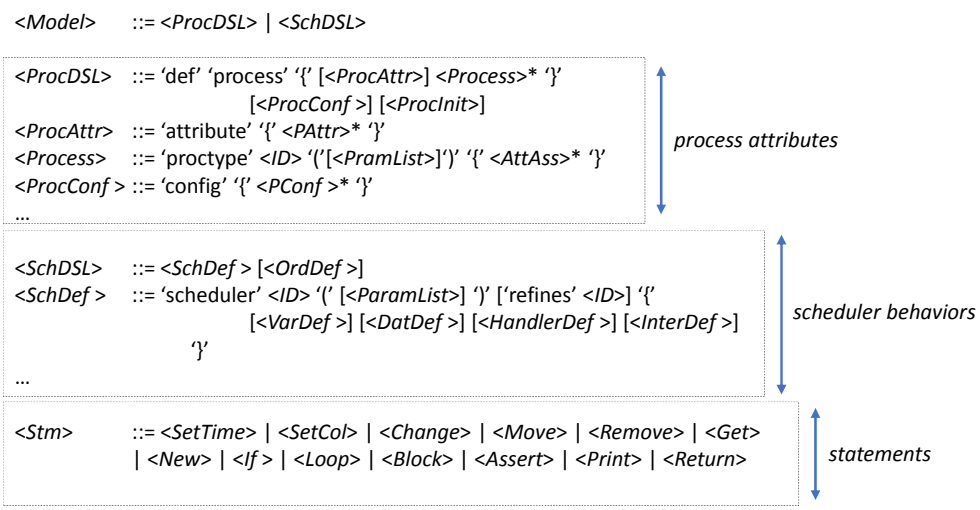


Figure 3.1: The structure of the DSL

processes and the scheduler, such as a process can increase its priority, we provide an interface to define the scheduling tasks performed by the process. That is defined in the *interface functions* part (`<InterDef>`).

The main task of a scheduler is changing the running statuses of the processes in a system. To represent these statuses, we use collections, which are defined in the *data* part (`<DatDef>`). We can define the order of the processes by declaring a function in the *ordering* part (`<OrdDef>`).

In this chapter, we use a system with the *priority* policy as an introductory example, which is depicted in Figure 3.2. Our DSL contains two types of the specification: the attributes of the processes and the behaviors of the scheduler as described before. With this example, the behaviors of the processes, the attributes of the processes, and the behaviors of the scheduler are specified in a) *process program*, b) *process attribute*, and c) *scheduler description*, respectively.

3.2 Language for the Behaviors of the Processes

The processes in the system are specified in the *process program* using the modeling language, which is based on Promela to specify the concurrent processes in the system. To deal with the communication between the processes and the scheduler for performing the scheduling tasks, we introduce several API functions as follows.

- Function `sch_exec` is to execute a process with its attributes (e.g. statement `sch_exec(P())` is for executing process `P` without initial values for its attributes).
- Function `sch_get` is to access the information managed by the scheduler (e.g. statement `sch_get(v, P, priority)` is to assign the value of `priority` of process `P` to variable `v`).
- Functions `sch_api` and `sch_api_self` are to execute the interface functions (e.g. statement `sch_api_self(terminate)` calls interface function `terminate`, which is defined in the *scheduler description*, to terminate the currently running process).

```
int a, b;

proctype P() {
next:
 if
  :: (a+b) < 100000 -> a++; goto next;
  :: else -> sch_api_self(terminate)
 fi;
}

proctype Q() {
next:
 if
  :: (a+b) < 100000 -> b++; goto next;
  :: else -> sch_api_self(terminate)
 fi;
}

init {
 sch_exec(P());
 sch_exec(Q());
}
```

a) Process program

```
def process Priority {
 attribute{
  var byte priority;
 }
 proctype P() {
  this.priority = 5;
 }
 proctype Q() {
  this.priority = 3;
 }
}
init {
 [{P(), Q()}]
}
```

b) Process attribute

```
scheduler Priority () {
 data {
  collection ready using priorityOrder;
 }

 event handler {
  select_process (process p) {
   get process from ready to run;
  }

  new_process (process target) {
   move target to ready;
   if (!running_process.isNull()) {
    if (target.priority > running_process.priority) {
     move running_process to ready;
    }
   }
  }
 }

 interface {
  function terminate(process target) {
   remove target;
  }
 }
}

comparator {
 variable { int x; }
 comparetype priorityOrder(process p_n, p_o) {
  x = p_n.priority - p_o.priority;
  if (x>0) return greater;
  else if (x==0) return equal;
       else return less;
 }
}
```

c) Scheduler description

Figure 3.2: An introductory example

Figure 3.2.a indicates a system with two processes (`P` and `Q`) with shared variables `a` and `b`. If the condition `a + b < 100000` is satisfied, process `P` increases variable `a`, while process `Q` increases variable `b`. Statement `sch_api_self` calls function `terminate` (Figure 3.2.c) to terminate the currently running process. Statements `sch_exec` used in `init` part determine that both of these two processes are executed at the initial time.

3.3 Language for Scheduling Policies

Our DSL provide two types of specifications for the policy: one for the attributes of the processes and the other for the behaviors of the scheduler as `<ProcDSL>` and `<SchDSL>` of the grammar[1].

$$\langle Model \rangle ::= \langle ProcDSL \rangle \mid \langle SchDSL \rangle$$

3.3.1 Attributes of the Processes

The attributes of the processes are used to carry out the scheduling tasks (especially for selecting a process to run). The attributes of the processes (`<ProcDSL>`) includes:

1. the definition of the attributes (`<ProcAttr>`) (e.g. in the `attribute` part shown in Figure 3.2.b, only attribute `priority` is defined),

2. the values for the attributes of each process at initial time (`<Process>`),

[1]See the Appendix A for the detailed language grammar.

3. the declaration of the periodic/sporadic processes (<ProcConf>).

   - each periodic process (<PeriodicP>) is determined using a period, and
   - a sporadic process (<SporadicP>) is defined using the duration that the process becomes ready and a number indicating the maximum instances of the process.

   An example to define these processes is depicted in Figure 3.3. With this example, four processes with the same period (20) are defined. However, these processes have different priority, deadline and initial offset. We handle the execution of these processes by generating the corresponding variables and manage them using the clock event (see Section 3.3.2 for more details), and

```
def process experiment1{
 attribute {
  clock c ;
  var byte priority ;
  var byte deadline ;
 }
 proctype P(byte priority = 0; byte deadline = 10){
  this.priority = priority;
  this.deadline = deadline;
 }
}

config {
 periodic process P(2,16) offset = 6 period = 20 ;
 periodic process P(3,11) offset = 9 period = 20 ;
 periodic process P(2,8) offset = 11 period = 20 ;
 periodic process P(5,20) offset = 10 period = 20 ;
}
```

Figure 3.3: Defining periodic processes

4. the execution of the processes at the starting time (<ProcInit>) (defined in the init part of the *process attribute*).

The grammar for the attributes of the processes is depicted in Figure 3.4.

In the introductory example, the value for the attribute (priority) of each process is determined by a template (proctype). With this example, the priority of P is greater than that of Q. The execution order of these processes is indicated in init part using a partially ordered set ([{P(), Q()}]). Note that the init part in the *process program* determines the existence of the processes using sch_exec statement(s) as depicted in Figure 3.2.a. But their execution order is defined in the init part of the *process attribute*. This order affects the selection of the scheduler. With this example, only process P can be selected (because this process has higher *priority*) although both of these two processes are run at the initial time.

3.3.2 Scheduling Policy

The behaviors of the scheduler defining as <SchDSL> of the grammar[2] (as depicted in Figure 3.5) are specified in the *scheduler description*, which consists of the definition of

[2]The <Verify> part is for the analysis of behaviors of the system (see Chapter 4 for more details).

⟨*ProcDSL*⟩ ::= ‘`def` ‘`process` ‘`{` [⟨*ProcAttr*⟩] ⟨*Process*⟩* ‘`}` [⟨*ProcConf*⟩] [⟨*ProcInit*⟩]
⟨*ProcAttr*⟩ ::= ‘`attribute` ‘`{` ⟨*PAttr*⟩* ‘`}`
⟨*PAttr*⟩ ::= [‘`var`|‘`val`] ⟨*Type*⟩⟨*ID*⟩ (‘`,`⟨*ID*⟩)* [‘`=`⟨*Value*⟩]‘`;`
⟨*Type*⟩ ::= ‘`int` | ‘`byte` | ‘`clock`
⟨*Process*⟩ ::= ‘`proctype` ⟨*ID*⟩ ‘`(`[⟨*PramList*⟩]‘`)` ‘`{` ⟨*AttAss*⟩* ‘`}`
⟨*PramList*⟩ ::= ⟨*PramAss*⟩ (‘`;` ⟨*PramAss*⟩)*
⟨*PramAss*⟩ ::= ⟨*Type*⟩ ⟨*ID*⟩ (‘`,` ⟨*ID*⟩)* ‘`=` ⟨*Value*⟩
⟨*AttAss*⟩ ::= [‘`this` ‘`.`] ⟨*ID*⟩ ‘`=` (⟨*Value*⟩ | ⟨*ID*⟩) ‘`;`
⟨*ProcConf*⟩ ::= ‘`config` ‘`{` ⟨*PConf*⟩* ‘`}`
⟨*PConf*⟩ ::= ⟨*SporadicP*⟩ | ⟨*PeriodicP*⟩
⟨*SporadicP*⟩ ::= ‘`sporadic` ‘`process` ⟨*Proc*⟩ ‘`in` ‘`(` ⟨*INT*⟩ ‘`,` ⟨*INT*⟩ ‘`)` [‘`limited` ⟨*INT*⟩] ‘`;`
⟨*PeriodicP*⟩ ::= ‘`periodic` ‘`process` ⟨*Proc*⟩ ‘`offset` ‘`=` ⟨*INT*⟩ ‘`period` ‘`=` ⟨*INT*⟩ [‘`limited` ⟨*INT*⟩] ‘`;`
⟨*Proc*⟩ ::= ⟨*ID*⟩ ‘`(` [⟨*Value*⟩ (‘`,` ⟨*Value*⟩)*] ‘`)`
⟨*ProcInit*⟩ ::= ‘`init` ‘`{` ‘`[` ⟨*PSet*⟩ (‘`,` ⟨*PSet*⟩)* ‘`]` ‘`}`‘`;`
⟨*PSet*⟩ ::= ‘`{` ⟨*Proc*⟩ (‘`,` ⟨*Proc*⟩)* ‘`}`

Figure 3.4: The grammar for the attributes of the processes

the scheduler (`<SchDef>`) and the definition of the methods for ordering the processes (`<OrdDef>`).

The scheduler definition (`<SchDef>`) contains[3]:

1. the variables used by the scheduler (`<VarDef>`),
2. the data used by the scheduler (`<DatDef>`) containing the definition of the collections (`<ColDef>`) to store the processes,
3. the handlers for the events[4] (`<HandlerDef>`), and
4. the interface functions (`<InterDef>`).

In the introductory example, there is no variable (as depicted in Figure 3.2.c). Only a collection `ready` is used to store the processes following the ordering defined by function `priorityOrder`. Here, we provide a mechanism to order the processes in a collection by defining the comparison functions (`<CompDef>`), which are used to compare each two processes in a collection. The return value (‘`greater`’, ‘`less`’ or ‘`equal`’) of the function (using statement `<Return>`) indicates that the process will be placed in front of (‘`greater`’), behind (‘`less`’) the other, or will have the same order (‘`equal`’) as the other (as depicted in Figure 3.6). Using this function, the processes in the collection are ordered.

To perform the scheduling policies, we handle the scheduling events.

- Three fixed *system scheduling events*: `new_process`, `select_process`, and `clock` as indicated before with the corresponding handlers, which are specified as `<EventDef>`, are defined in `<HandlerDef>`.

[3]The `<Generate>` part is for the test generation (see Chapter 5 for more details).
[4]The events `pre_take`, `post_take` and `action` are used for the test generation (see Chapter 5).

⟨*SchDSL*⟩ ::= ⟨*SchDef*⟩ [⟨*OrdDef*⟩] [⟨*Verify*⟩]
⟨*SchDef*⟩ ::= '`scheduler`' ⟨*ID*⟩ '`(`' [⟨*ParamList*⟩] '`)`' ['`refines`' ⟨*ID*⟩] '`{`' [⟨*Generate*⟩] [⟨*VarDef*⟩] [⟨*DatDef*⟩] [⟨*HandlerDef*⟩] [⟨*InterDef*⟩] '`}`'
⟨*VarDef*⟩ ::= '`variable`' '`{`' ⟨*VDec*⟩* '`}`'
⟨*VDec*⟩ ::= [⟨*IfDef*⟩] (⟨*VBlockDef*⟩ | ⟨*VOneDef*⟩)
⟨*IfDef*⟩ ::= '`#`' '`ifdef`' '`(`' ⟨*Expr*⟩ '`)`'
⟨*VBlockDef*⟩ ::= '`{`' ⟨*VOneDef*⟩* '`}`'
⟨*VOneDef*⟩ ::= ⟨*Type*⟩ ⟨*ID*⟩ ('`,`' ⟨*ID*⟩)* ['`=`' ⟨*Value*⟩] '`;`'
⟨*DatDef*⟩ ::= '`data`' '`{`' ⟨*DDef*⟩* '`}`'
⟨*DDef*⟩ ::= [⟨*IfDef*⟩] '`data`' (⟨*DBlockDef*⟩ | ⟨*DOneDef*⟩)
⟨*DBlockDef*⟩ ::= '`{`' ⟨*DOneDef*⟩* '`}`'
⟨*DOneDef*⟩ ::= ⟨*VOneDef*⟩ | ⟨*ColDef*⟩
⟨*ColDef*⟩ ::= ['`refines`'] '`collection`' ⟨*ID*⟩ ['`using`' ⟨*ID*⟩ ('`,`' ⟨*ID*⟩)*] ['`with`' ⟨*OrdType*⟩] '`;`'
⟨*OrdType*⟩ ::= '`lifo`' | '`fifo`'
⟨*HandlerDef*⟩ ::= '`event`' '`handler`' '`{`' ⟨*EventDef*⟩* '`}`'
⟨*EventDef*⟩ ::= ⟨*Event*⟩ '`(`' [⟨*ID*⟩] '`)`' '`{`' ⟨*IfDefStm*⟩* '`}`'
⟨*IfDefStm*⟩ ::= [⟨*IfDef*⟩] ⟨*Stm*⟩
⟨*Event*⟩ ::= '`select_process`' | '`new_process`' | '`clock`' | '`pre_take`' | '`post_take`' | '`action`'
⟨*InterDef*⟩ ::= '`interface`' '`{`' ⟨*InterFunc*⟩* '`}`'
⟨*InterFunc*⟩ ::= '`function`' ⟨*ID*⟩ '`(`' [⟨*IParamList*⟩] '`)`' '`{`' ⟨*Stm*⟩* '`}`'
⟨*IParamList*⟩ ::= ⟨*IParamDec*⟩ ('`,`' ⟨*IParamDec*⟩)*
⟨*IParamDec*⟩ ::= ⟨*Type*⟩ ⟨*ID*⟩
⟨*OrdDef*⟩ ::= '`comparator`' '`{`' [⟨*CVarDef*⟩] ⟨*CompDef*⟩* '`}`'
⟨*CVarDef*⟩ ::= '`variable`' '`{`' ⟨*VOneDef*⟩* '`}`'
⟨*CompDef*⟩ ::= '`comparetype`' ⟨*ID*⟩ '`(`' '`process`' ⟨*ID*⟩ '`,`' ⟨*ID*⟩ '`)`' '`{`' ⟨*Stm*⟩* '`}`'

Figure 3.5: The grammar for scheduling policies

```
comparator {
 variable { int x; }
 comparetype priorityOrder(process p_n, p_o) {
  x = p_n.priority - p_o.priority;
  if (x>0) return greater;
  else if (x==0) return equal;
       else return less;
 }
}
```

Figure 3.6: Defining comparison function

- The events raised from the process (*process scheduling events*) using interface functions (`<InterFunc>`) can be declared in the `interface` part (`<InterDef>`).

In Figure 3.7, the scheduler handles two *system scheduling events* and provides a *process scheduling events* with function named `terminate`.

Several statements (defined as `<Stm>` of the grammar) are introduced to describe the behaviors of the scheduler (as depicted in Figure 3.8).

The functionalities of these statements are as follows.

```
...
event handler {
   select_process (process p) {
    get process from ready to run;
   }

   new_process (process target) {
    move target to ready;
    if (!running_process.isNull()) {
     if (target.priority > running_process.priority) {
      move running_process to ready;
     }
    }
   }
  }

  interface {
   function terminate(process target) {
    remove target;
   }
  }
...
```

Figure 3.7: Handling scheduling events

⟨*Stm*⟩ ::= ⟨*SetTime*⟩ | ⟨*SetCol*⟩ | ⟨*Change*⟩ | ⟨*Move*⟩ | ⟨*Remove*⟩ | ⟨*Get*⟩ | ⟨*New*⟩ | ⟨*If*⟩ | ⟨*Loop*⟩ | ⟨*Block*⟩ | ⟨*Assert*⟩ | ⟨*Print*⟩ | ⟨*Return*⟩ | ⟨*Gen*⟩ |⟨*GenLn*⟩

⟨*SetTime*⟩ ::= '`time_slice`' '`=`' ⟨*Expr*⟩ '`;`'

⟨*SetCol*⟩ ::= '`return_set`' '`=`' ⟨*ID*⟩ '`;`'

⟨*Change*⟩ ::= ⟨*ChgUnOp*⟩ | ⟨*ChgExpr*⟩

⟨*ChgUnOp*⟩ ::= ⟨*QualName*⟩ ('`++`' | '`--`') '`;`'

⟨*ChgExpr*⟩ ::= ⟨*QualName*⟩ '`=`' ⟨*Expr*⟩ '`;`'

⟨*QualName*⟩ ::= ⟨*ID*⟩ ['`.`'⟨*ID*⟩]

⟨*Move*⟩ ::= '`move`' ⟨*ID*⟩ to ⟨*ID*⟩ '`;`'

⟨*Remove*⟩ ::= '`remove`' ⟨*ID*⟩ '`;`'

⟨*Get*⟩ ::= '`get`' '`process`' '`from`' ⟨*ID*⟩ '`to`' '`run`' '`;`'

⟨*New*⟩ ::= '`new`' ⟨*Proc*⟩ ['`,`' ⟨*INT*⟩] '`;`'

⟨*If*⟩ ::= '`if`' '`(`' ⟨*Expr*⟩ '`)`' ⟨*Stm*⟩ ['`else`' ⟨*Stm*⟩]

⟨*Loop*⟩ ::= '`for`' '`each`' '`process`' ⟨*ID*⟩ '`in`' ⟨*ID*⟩ ⟨*Stm*⟩

⟨*Block*⟩ ::= '`{`' ⟨*Stm*⟩* '`}`'

⟨*Assert*⟩ ::= '`assert`' ⟨*Expr*⟩ '`;`'

⟨*Print*⟩ ::= '`print`' ⟨*Expr*⟩ '`;`'

⟨*Return*⟩ ::= '`return`' ⟨*OrderType*⟩ '`;`'

⟨*OrderType*⟩ ::= '`greater`' | '`less`' | '`equal`'

⟨*Gen*⟩ ::= '`gen`' [⟨*ID*⟩ '`,`'] ⟨*Expr*⟩ '`;`'

⟨*GenLn*⟩ ::= '`genln`' [⟨*ID*⟩ '`,`'] ⟨*Expr*⟩ '`;`'

Figure 3.8: The grammar for statements

- The values of the variables and the attributes of the processes can be changed using the following statements:
  - `<SetTime>` to indicate the time slice for the current process,
  - `<SetCol>` to determine the collection that contains the process after its execution time (i.e. time slice) and
  - `<Change>` to change a variable or an attribute of a process.

  For instance, statement "`running_process.priority = 1;`" sets the *priority* of

the currently running process to 1.

- We can update the running status of a process using statements `<Move>`, `<Remove>`, and `<Get>` by changing the collection containing this process. As depicted in Figures 3.7, statement "`move running_process to ready;`" is to move the current process to the `ready` collection (the current process is preempted).

- Statement `<New>` is for executing a process, such as "`new P();`" is for executing an instance of process `P`.

- The language also supports branch statement (`<If>`) and loop over a collection statement (`<Loop>`).

  For instance, statement "`if(target.priority>running_process.priority){..}`" (as depicted in Figures 3.2) checks *priority* of process `target` with the current one (`running_process`) to preempt the current process.

- Statements `<Assert>` and `<Print>` are used for tracking the behaviors of the system. For instance, statement "`print "x = " + Sys(x) ;`" print the value of variable `x` defined in the *process program*.

In Figure 3.7, when process `target` arrives, if it has higher priority is than the currently running process, the `running_process` is preempted using `<Move>` statement to put to `ready` collection. This makes the scheduler select another process. To do this, the scheduler gets a process from this collection. This behavior is specified in the `select_process` event handler using statement `<Get>`. The processes selected is determined by the ordering method used by the collection. Function `terminate` in the `interface` part is to terminate a process using statement `<Remove>`.

3.4 Language Semantics

We now give formal definitions for a system with the scheduler. Let $\mathcal{PID}$ be a set of process identifiers, $\mathcal{X}$ be a set of variables and $\mathcal{V}$ be a set of values. We use $\perp$ to denote the non-determined value.

Definition 3.1 (Process state): A process state is a tuple $\langle \sigma_g, \sigma_l \rangle$, where $\sigma_g : \mathcal{X} \to \mathcal{V}$ and $\sigma_l : \mathcal{PID} \to (\mathcal{X} \to \mathcal{V})$.

- σ_g is the mapping from the set of the global variables to the set of values and

- σ_l is the mapping from the set of process identifiers to the mappings from the local variables of a process to the set of values.

Here, the variables are defined in the *process program*. For instance, in Figure 3.2.a, `a` and `b` are the global variables; there is no local variable. At the initial time, we have a process state $\langle \{(a, 0), (b, 0)\}, \{\} \rangle$. We use $\mathcal{S}_{proc}$ to represent the set of process states.

Let $\mathcal{L}_{proc} = \mathit{normal} \cup \{\mathit{get}\} \cup \mathit{scheduling}$ be a set of labels represent the behaviors of the processes, where:

- *normal* is a set of normal actions described in Promela. These actions are to change the process state (e.g. in Figure 3.2.a, `a++` represents a normal action).

- *get* is an action to access the information of the scheduler. This action correspond to function `sch_get` (e.g. statement `sch_get(v, P, priority)` is to assign the value of `priority` of process `P` to variable `v`).

- *scheduling* = {*api*, *exec*} is a set of scheduling actions performed by the currently running process using the API functions.

  – The *api* action corresponds to the `sch_api`, and `sch_api_self` functions called by the current process (e.g. in the example, `sch_api_self(terminate)` represents an *api* scheduling action).

  – The *exec* action is to execute a process by calling function `sch_exec` (e.g. in the example, statement `sch_exec(P())` represent an *exec* action).

  These actions are handled by the event handlers and the interface functions specified in the *scheduler description*. In the example, function `sch_api_self(terminate)` is handled by the interface function `terminate`.

Definition 3.2 (Process): A process is a tuple $\langle S_p, L_p, T_p, s_0 \rangle$, where $S_p \subseteq \mathcal{S}_{proc}$ is a set of process states, $L_p \subseteq \mathcal{L}_{proc}$ is a set of labels, $T_p \subseteq S_p \times L_p \times S_p$ is a set of transitions, and $s_0 \in S_p$ is the initial state.

In this definition, L_p represents the set of actions of the processes (e.g., `a++` is an action of the processes). These actions change the states of the processes to represent the transition relation $T_p \subseteq S_p \times L_p \times S_p$.

We use $\Sigma_{proc} = [S_1, ..., S_i, ..., S_m]$ to denote a sequence of process states in a system, where $S_i \in \mathcal{S}_{proc}$ is a state of ith process and m is the number of the processes in the system. This sequence is used to determined the state of the system.

Definition 3.3 (Process collection): A process collection is a tuple $\langle Pid, \succ, \sim \rangle$, where $Pid \subseteq \mathcal{PID}$, $\succ$ and $\sim$ are binary relations defined as follow:

- $\succ \subseteq \mathcal{PID} \times \mathcal{PID}$ is an irreflexive, antisymmetric, transitive binary relation and

- $\sim \subseteq \mathcal{PID} \times \mathcal{PID}$ is an reflexive, symmetric, transitive binary relation.

For instance, $C = \langle Pid, \succ, \sim \rangle$ where $Pid = \{P_1, P_2, P_3\}$, $\succ = \{(P_1, P_3), (P_2, P_3)\}$, and $\sim = \{(P_1, P_2)\}$ determines a collection with 3 processes: process P_1 and process P_2 have the same order, and they are placed in front of process P_3.

We use the collections to denote the running statuses of the processes (such as *ready*, *blocked*). In the example, we define only one collection named `ready`, which uses the ordering method defined by function `priorityOrder` in the *comparator* part of the *scheduler description*. This function compares two processes using their `priority` values. The return value of this function determines the order of these two processes (see Section 3.3.2 for more details). Based on that fact, the processes in this collection are ordered. In this definition, the binary relations ($\succ$ and $\sim$) are globally given. We use $\mathcal{COL}$ to denote a set of collections.

Definition 3.4 (Scheduler state): A scheduler state is a tuple $\langle \sigma_{sch}, (C_1, ..., C_k), P_r \rangle$, where

- $\sigma_{sch} : \mathcal{X}_s \cup \mathcal{X}_c \cup \{run, tslice, rcol\} \rightarrow \mathcal{V} \cup \{\perp\}$ is a mapping from the set of normal variables ($\mathcal{X}_s \subset \mathcal{X}$), the set of clock variables ($\mathcal{X}_c \subset \mathcal{X}$) used by the scheduling strategy, and the set of variables $\{run, tslice, rcol\} \subseteq \mathcal{X}$ to the set of values, where

  - *run* indicates the current process,
  - *tslice* is the time slice to run of the current process, and
  - *rcol* is the collection that stores the process after finishing its execution time.

- $C_i \in \mathcal{COL}, i \in \mathbb{N}$ is a collection;
- $P_r \subseteq \mathcal{PID} \cup \{\perp\}$ is a set of process identifiers represent the set of processes that can be run.

The set of variables ($\mathcal{X}_s$ and $\mathcal{X}_c$) used in the scheduling strategy is defined in the *scheduler description*. The variables: *run*, *tslice*, and *rcol* are pre-defined. In some states, these variable is non-determined ($\perp$). For instance, $run = \perp$ means that there is no running process. In this situation, the scheduler will select a process to run by performing a *select* action (see the description of the behaviors of the scheduler below).

The set of collections $\{C_i\}, i = \overline{1..k}$ (with the number of collections being fixed) are also defined in the *scheduler description*. P_r is pre-defined to represent the set of processes which can be run (determined by the scheduler). This set of processes is used for indicating the possible states leading from the current state (by performing a process action) in the search of the state space (see Chapter 4 for more details).

In the example (as depicted in Figure 3.2.c), only one collection (`ready`) is defined; there is no variable. The *time slice* (with the collection containing the process after its execution time) is not defined (i.e. using statement `<SetTime>` and `<SetCol>`). One of the scheduler state of this system is $\langle\{run, tslice, rcol\}, (ready), P_r\rangle$, where a) $run = P$, b) $tslice = \perp$, c) $rcol = \perp$, d) $ready = \langle Pid, \succ, \sim\rangle$ with $Pid = \{P, Q\}$, $\succ = \{(P, Q)\}$, and $\sim = \{\}$, and e) $P_r = \{P\}$.

Definition 3.5 (System state): A system state is a tuple $\Sigma = \langle\Sigma_{proc}, \Sigma_{sch}\rangle$, where $\Sigma_{proc} = [S_1, ..., S_i, ..., S_m]$ is the sequence of the process states and $\Sigma_{sch} = \langle\sigma_{sch}, (C_1, ..., C_k), P_r\rangle$ is a scheduler state.

The system state is derived from the set of processes and the scheduler. Therefore, to represent a system state we use the states of the processes (a sequence of process states) and the state of the scheduler. For the convenience of writing, we use both $\langle\Sigma_{proc}, \Sigma_{sch}\rangle$ and $\langle[S_1, ..., S_m], \Sigma_{sch}\rangle$ to denote the system state. We indicate $\mathcal{S}_{sys}$ as the set of system states.

Let $\mathcal{L}_{sch} = \{select, clock, new, inter\}$ be a set of labels representing the behaviors (actions) of the scheduler, where:

- *select*, *clock* and *new* are the actions corresponding to the events `select_process`, `clock` and `new_process` defined in the *scheduler description*, respectively, and
- *inter* is an action corresponding to the event raised by an interface function called by the current process.

Definition 3.6 (System): A system is a tuple $\langle\Sigma_s, L_s, T_s, \Sigma_0\rangle$, where $\Sigma_s \subseteq \mathcal{S}_{sys}$ is a set of system states, $L_s \subseteq (\mathcal{L}_{proc} \cup \mathcal{L}_{sch})$ is a set of labels, $T_s \subseteq \Sigma_s \times L_s \times \Sigma_s$ is a set of transitions, and $\Sigma_0 \in \Sigma_s$ is the initial state.

Note that L_s represents the set of behaviors/actions of the system including a) the behaviors of the processes ($\mathcal{L}_{proc}$) and b) the behaviors of the scheduler ($\mathcal{L}_{sch}$). For instance, `sch_api_self(terminate)` is an action of a process, the behavior of the scheduler indicated by the interface function `terminate` called by the function `sch_api_self`

is an action of the scheduler. In this definition, T_s represents the transition relation between the system states. That relation is defined by both of the statements in Promela for the behaviors of the processes and the statements in the DSL for the behaviors of the scheduler.

We define the following functions to indicate the collection and to select processes to run.

- Function $getCol : \mathcal{PID} \rightarrow \mathcal{COL}$ to determine the collection containing the process. In the example, at the initial time, function $getCol(P)$ returns $ready$.
- Function $max : \mathcal{COL} \rightarrow 2^{\mathcal{PID}}$ is to select processes from a collection to run. We have $max(\langle Pid, \succ, \sim \rangle) = P$, where P is the smallest set satisfying for any $y \in Pid$ there exists $x \in P$ such that $x \succsim y$ with $\succsim = \succ \cup \sim$.

  For example, suppose that a system has a collection named $ready$ contains 3 processes: P_1, P_2, and P_3 with their priorities being set to 2, 2, and 1, respectively (the greater value means the higher priority). If this collection use *priority* ordering method, we have $ready = \langle Pid, \succ, \sim \rangle$ with $Pid = \{P_1, P_2, P_3\}$, $\succ = \{(P_1, P_3), (P_2, P_3)\}$, and $\sim = \{(P_1, P_2)\}$. Function $max(ready)$ returns $\{P_1, P_2\}$. That means the set of processes selected is $\{P_1, P_2\}$.

The semantics of the primitive functions defined by the statements in the DSL is described by the transition relations between the system states as follows.

- **Change the value of a variable**

  $\langle \Sigma_p, \langle \sigma_s, (C_1, ..., C_k), P_r \rangle \rangle \xrightarrow{v=\langle exp \rangle} \langle \Sigma_p, \langle \sigma_s[[\![exp]\!]/v], (C_1, ..., C_k), P_r \rangle \rangle$, where v is a variable used by the scheduler, $\sigma_s[[\![exp]\!]/v]$ means replacing the value of v by the value of $[\![exp]\!]$ obtained by evaluating expression exp.

  This function corresponds to the statements `<Change>`, `<SetTime>` and `<SetCol>` of the DSL to change the values of the variables used in the strategy.

  $\langle SetTime \rangle$ ::= '`time_slice`' '`=`' $\langle Expr \rangle$ '`;`'
  $\langle SetCol \rangle$ ::= '`return_set`' '`=`' $\langle ID \rangle$ '`;`'
  $\langle Change \rangle$ ::= $\langle ChgUnOp \rangle \mid \langle ChgExpr \rangle$
  $\langle ChgUnOp \rangle$ ::= $\langle QualName \rangle$ ('`++`' | '`--`') '`;`'
  $\langle ChgExpr \rangle$ ::= $\langle QualName \rangle$ '`=`' $\langle Expr \rangle$ '`;`'

  For instance, statement "`time_slice = 1;`" will change the state of the system as follows.

  $$\langle \Sigma_p, \langle \sigma_s, (C_1, ..., C_k), P_r \rangle \rangle \xrightarrow{\text{time\_slice} = 1} \langle \Sigma_p, \langle \sigma_s[1/tslice], (C_1, ..., C_k), P_r \rangle \rangle$$

- **Remove a process**

  $\langle \Sigma_p, \langle \sigma_s, (C_1, ..., C_k), P_r \rangle \rangle \xrightarrow{rem(p)} \langle \Sigma'_p, \langle \sigma_s, (C'_1, ..., C'_k), P'_r \rangle \rangle$ with $\Sigma_p = [S_1, ..., S_i, ..., S_m]$, S_i is the state of process p, where $\Sigma'_p = [S_1, ..., S_{i-1}, S_{i+1}, ..., S_m]$ and $C'_i = \langle P_i - \{p\}, \succ_i, \sim_i \rangle$ with $C_i = \langle P_i, \succ_i, \sim_i \rangle$, $i \in \mathbb{N}$ and $P'_r = P_r - \{p\}$ ($P - \{p\}$ means removing p from P).

  The function is defined by the statement `<Remove>` of the grammar to remove a process from the system.

$$\langle Remove\rangle ::= \text{`}\texttt{remove}\text{'}\ \langle ID\rangle\ \text{`}\texttt{;}\text{'}$$

For instance, suppose that a collection $C_1 = \langle Pid, \succ, \sim\rangle$ with $Pid = \{P, Q\}$, $\succ = \{\}$, and $\sim = \{(P, Q)\}$. Statement "`remove P;`" will procedure the following result.

$$\langle \Sigma_p, \langle \sigma_s, (C_1, ..., C_k), P_r\rangle\rangle \xrightarrow{\text{remove P;}} \langle \Sigma'_p, \langle \sigma_s, (C'_1, ..., C_k), P_r\rangle\rangle \text{ with}$$
$\Sigma_p = [S_1, ..., S_i, ..., S_m]$, where $\Sigma'_p = [S_1, ..., S_{i-1}, S_{i+1}, ..., S_m]$, S_i is the state of process P, and $C'_1 = \langle\{Q\}, \succ, \sim\rangle$.

- **Move a process to a collection**
  - If p is a new process[5] then

    $\langle \Sigma_p, \langle \sigma_s, (C_1, ..., C_i, ..., C_k), P_r\rangle\rangle \xrightarrow{mov(p,C_i)} \langle \Sigma_p, \langle \sigma_s, (C_1, ..., C'_i, ..., C_k), P_r\rangle\rangle$, where $C'_i = \langle P_i \cup \{p\}, \succ_i, \sim_i\rangle$ with $C_i = \langle P_i, \succ_i, \sim_i\rangle$, $i \in \mathbb{N}$.
  - If p is the current process then

    $\langle \Sigma_p, \langle \sigma_s, (C_1, ..., C_i, ..., C_k), P_r\rangle\rangle \xrightarrow{mov(p,C_i)} \langle \sigma_s[\bot\ /run], (C_1, ..., C'_i, ..., C_k), P_r\rangle\rangle$, where $C'_i = \langle P_i \cup \{p\}, \succ_i, \sim_i\rangle$ with $C_i = \langle P_i, \succ_i, \sim_i\rangle$, $i \in \mathbb{N}$.
  - If p belongs to collection C_i then

    $\langle \Sigma_p, \langle \sigma_s, (C_1, ..., C_i, ..., C_j, ..., C_k), P_r\rangle\rangle \xrightarrow{mov(p,C_j)} \langle \Sigma_p, \langle \sigma_s, (C_1, ..., C'_i, ..., C'_j, ..., C_k), P_r\rangle\rangle$, where $getCol(p) = C_i$, with $C_i = \langle P_i, \succ_i, \sim_i\rangle$, $i \in \mathbb{N}$, $C'_i = \langle P_i - \{p\}, \succ_i, \sim_i\rangle$, and $C'_j = \langle P_j \cup \{p\}, \succ_j, \sim_j\rangle$ with $C_j = \langle P_j, \succ_j, \sim_j\rangle$, $j \in \mathbb{N}$.

  The function is defined by the statement `<Move>` of the grammar to change the execution status of the corresponding process.

$$\langle Move\rangle ::= \text{`}\texttt{move}\text{'}\ \langle ID\rangle\ \text{to}\ \langle ID\rangle\ \text{`}\texttt{;}\text{'}$$

  For instance, suppose that we have a collection `ready`: $C_1 = \langle Pid_1, \succ_1, \sim_1\rangle$ with $Pid_1 = \{P, Q\}$, $\succ_1 = \{\}$, and $\sim_1 = \{(P, Q)\}$; a collection `blocked`: $C_2 = \langle Pid_2, \succ_2, \sim_2\rangle$ with $Pid_2 = \{\}$, $\succ_2 = \{P, Q\}$, and $\sim_2 = \{\}$.

  Statement "`move P to block;`" will procedure the following result.

$$\langle \Sigma_p, \langle \sigma_s, (C_1, C_2, ..., C_k), P_r\rangle\rangle \xrightarrow{\text{move P to block;}} \langle \Sigma_p, \langle \sigma_s, (C'_1, C'_2, ..., C_k), P_r\rangle\rangle, \text{ where}$$
$C'_1 = \langle\{Q\}, \succ_1, \sim_1\rangle$, and $C'_2 = \langle\{P\}, \succ_2, \sim_2\rangle$.

- **Select processes from a collection to run**

  $\langle \Sigma_p, \langle \sigma_s, (C_1, ..., C_k), P_r\rangle\rangle \xrightarrow{select(C_i)} \langle \Sigma_p, \langle \sigma_s, (C_1, ..., C_k), P'_r\rangle\rangle$, where $C_i = \langle P_i, \succ_i, \sim_i\rangle$, $i \in \mathbb{N}$, and $P'_r = max(\langle P_i, \succ_i, \sim_i\rangle)$.

  The function is defined by the statement `<Get>` of the grammar.

$$\langle Get\rangle ::= \text{`}\texttt{get}\text{'}\ \text{`}\texttt{process}\text{'}\ \text{`}\texttt{from}\text{'}\ \langle ID\rangle\ \text{`}\texttt{to}\text{'}\ \text{`}\texttt{run}\text{'}\ \text{`}\texttt{;}\text{'}$$

  For instance, suppose that a collection `ready`: $C_1 = \langle Pid_1, \succ_1, \sim_1\rangle$ with $Pid_1 = \{P, Q\}$, $\succ_1 = \{\}$, and $\sim_1 = \{(P, Q)\}$. Statement "`get process from ready to run;`" will procedure the following result.

[5] a new process does not belong to any collection

$$\langle \Sigma_p, \langle \sigma_s, (C_1, ..., C_k), P_r \rangle \rangle \xrightarrow{\text{get process from ready to run;}} \langle \Sigma_p, \langle \sigma_s, (C_1, ..., C_k), P'_r \rangle \rangle, \text{ where } P'_r = \{P, Q\}.$$

The statement `<Move>`, `<Remove>` and `<Get>` are used to update the running status of a process by changing the collection contains this process.

To define the semantics of non-primitive functions for the branch statement (`<If>`) and loop over a collection statement (`<Loop>`) of the DSL, we consider a group of statements as a single statement (called a block statement) and use the transitive closure for representing the transition relation.

- **Branch statement.** A branch statement is represented as $if(bexp, st_1, st_2)$, where $bexp$ is a boolean expression, st_1 and st_2 are block statements. If $[\![bexp]\!] = true$ then $\Sigma \xrightarrow{if(bexp,st_1,st_2)} \Sigma_1$, where $\Sigma(\xrightarrow{st_1})^+\Sigma_1$ else $\Sigma \xrightarrow{if(bexp,st_1,st_2)} \Sigma_2$, where $\Sigma(\xrightarrow{st_2})^+\Sigma_2$.

  The relation $(\xrightarrow{s})^+$ indicate the closure of $\xrightarrow{s}$ (i.e., the rule $\xrightarrow{s}$ is applied once or more times depend on the set of statements in the block statement s until it can not proceed any more).

  For instance, suppose that a collection `ready`: $C_1 = \langle Pid_1, \succ_1, \sim_1 \rangle$ with $Pid_1 = \{Q\}$, $\succ_1 = \{(P, Q)\}$, and $\sim_1 = \{\}$. This collection uses priority ordering method. The current process is `P`. The following statement

  ```
  if (P.priority > Q.priority)
      move P to ready;
  ```

  will procedure the following result.

$$\langle \Sigma_p, \langle \sigma_s, (C_1, ..., C_k), P_r \rangle \rangle \xrightarrow{\text{if (P.priority > Q.priority) ...;}} \langle \Sigma_p, \langle \sigma_s[\perp/run], (C'_1, ..., C_k), P_r \rangle \rangle, \text{ where } C'_1 = \langle \{(P, Q)\}, \succ_1, \sim_1 \rangle.$$

- **Loop over a collection.** A loop statement can be represented as $for(p, [p_1, ..., p_k], st)$, where p is the loop variable, $[p_1, ..., p_k]$ is a sequence of process identifiers represents a total order of the process in a collection[6], and st is a statement[7]. We have $\Sigma \xrightarrow{for(p,[p_1,...,p_k],st)} \Sigma_k$, where $\Sigma(\xrightarrow{p=p_1;st})^+\Sigma_1$, ..., $\Sigma_{k-1}(\xrightarrow{p=p_k;st})^+\Sigma_k$ with "$p = p_i$" meaning assigning p_i for p and "$p = p_i; st$" representing a sequence of statements.

  For instance, suppose that a collection `ready`: $C_1 = \langle Pid_1, \succ_1, \sim_1 \rangle$ with $Pid_1 = \{P, Q\}$, $\succ_1 = \{(P, Q)\}$, and $\sim_1 = \{\}$ and collection `blocked`: $C_2 = \langle Pid_2, \succ_2, \sim_2 \rangle$ with $Pid_2 = \{\}$, $\succ_2 = \{\}$, and $\sim_2 = \{(P, Q)\}$.

  Statement "`for each process p in ready move p to block;`" will procedure the following result.

$$\langle \Sigma_p, \langle \sigma_s, (C_1, C_2), P_r \rangle \rangle \xrightarrow{\text{for each process p in ready ...;}} \langle \Sigma_p, \langle \sigma_s, (C'_1, C'_2), P_r \rangle \rangle, \text{ where } C'_1 = \langle \{\}, \succ_1, \sim_1 \rangle \text{ and } C'_2 = \langle \{(P, Q)\}, \succ_2, \sim_2 \rangle.$$

The behaviors of the scheduler are defined in the *scheduler description*. These be-

[6] in our approach, we use the identifiers of the processes to define the order

[7] a statement can be a block statement (a sequence of statements)

haviors are handled by the scheduler using the events corresponding to these actions: *new, select, clock*, and *api* as explained before. Each event is specified using the DSL statements defined by the primitive functions above. The semantic of these behaviors of the scheduler is as follows.

- **New event.** $\langle [S_1, ..., S_m, S_{m+1}], \Sigma_s \rangle (\xrightarrow{new})^+ \langle [S_1, ..., S_m, S_{m+1}], \Sigma'_s \rangle$

  This event is defined in `new_process` event handler, which is performed when a process arrives. It happens when the current process executes another process or the system initialize its processes at the starting time (i.e. using `sch_exec` statements).

  For instance, suppose that the system has a collection named `ready`: $C_1 = \langle Pid_1, \succ_1, \sim_1 \rangle$ with $Pid_1 = \{\}$, $\succ_1 = \{\}$, $\sim_1 = \{\}$. The event `new_process` is defined as follow.

  ```
  new_process (process target) {
      move target to ready ;
  }
  ```

  If the current process executes statement `sch_exec("P")` to execute the process `P`. The state of the system is changed as follows.

  $\langle [S_1, ..., S_m, S_{m+1}], \Sigma_s \rangle (\xrightarrow{\text{new}})^+ \langle [S_1, ..., S_m, S_{m+1}], \Sigma'_s \rangle$ with $\Sigma_s = \langle \sigma_s, (C_1), P_r \rangle$, where S_{m+1} is the state of process `P`, and $\Sigma'_s = \langle \sigma_s, (C'_1), P_r \rangle$ with $C'_1 = \langle Pid'_1, \succ_1, \sim_1 \rangle$ and $Pid'_1 = \{P\}$.

- **Select event.** $\langle \Sigma_p, \Sigma_s \rangle (\xrightarrow{select})^+ \langle \Sigma_p, \Sigma'_s \rangle$

  This event is defined in `select_process` event handler, which is performed when the scheduler selects a process to run.

  For instance, suppose that a collection `ready`: $C_1 = \langle Pid_1, \succ_1, \sim_1 \rangle$ with $Pid_1 = \{P, Q\}$, $\succ_1 = \{P, Q\}$, and $\sim_1 = \{\}$. The event `new_process` is handed as follows.

  ```
  select_process (process target) {
      get process from ready to run;
  }
  ```

  If there is no current process, the scheduler will perform the `select` action to select a process to run. The state of the system is changed as follows.

  $\langle \Sigma_p, \Sigma_s \rangle (\xrightarrow{\text{select}})^+ \langle \Sigma_p, \Sigma'_s \rangle$ with $\Sigma_s = \langle \sigma_s, (C_1), P_r \rangle$, where $\Sigma'_s = \langle \sigma_s, (C_1), P'_r \rangle$, where $P'_r = \{P\}$.

- **Inter event.** $\langle \Sigma_p, \Sigma_s \rangle (\xrightarrow{api})^+ \langle \Sigma'_p, \Sigma'_s \rangle$

  This event is defined by the *interface functions*, which is performed by the current process (executes an API to call an interface function). Note that the process state can be changed (e.g. the current process terminates itself) and a *clock* event happens after this action (see the description of a *sequence-action* below).

  For instance, if process `P` is currently running and terminates itself by executing statement `sch_api_self(terminate)`. Function `terminate` is defined in the *scheduler description* (as shown below). The state of the system is changed as follows.

```
function terminate(process target) {
    remove target;
}
```

$\langle \Sigma_p, \langle \sigma_s, (C_1), P_r \rangle \rangle (\xrightarrow{inter})^+ \langle \Sigma'_p, \langle \sigma_s[\perp/run], (C_1), P'_r \rangle \rangle$ with $\Sigma_p = [S_1, ..., S_i, ..., S_m]$, where $\Sigma'_p = [S_1, ..., S_{i-1}, S_{i+1}, ..., S_m]$ with S_i is the state of the current process, and $P'_r = P_r - \{P\}$.

- **Clock event.** $\langle \Sigma_p, \Sigma_s \rangle (\xrightarrow{clock})^+ \langle \Sigma_p, \Sigma'_s \rangle$

  This event is defined in `clock` event handler which is performed after each action of the current process. It is used for handling the timer event. Beside changing the scheduler state using the statements defined in the *clock* event handler, each clock variable used in the scheduling strategy is increased by 1 when this event is handled: $\sigma'(c) = \sigma(c) + 1$, where $c \in \mathcal{X}_c$ is a clock variable defined in the *scheduler description*. Note that the clock event still occurs when the current process has no enabled action.

  For instance, if we do not define any clock variable, no execution time is indicated (i.e. using statement `<SetTime>`), and the timer event in the *scheduler description* (i.e. `clock` event handler) is not define, the clock event will do nothing and the state of the system is not changed as follows.

$$\langle \Sigma_p, \Sigma_s \rangle (\xrightarrow{clock})^+ \langle \Sigma_p, \Sigma_s \rangle$$

At the initial state Σ_0, the scheduler selects a process to run by performing the *select* action. Following an action performed by the current process, a *sequence-action* happens as follows.

- Let $a \in normal \cup \{get\}$ be an action[8] of process P_i and $\langle S_i, a, S'_i \rangle \in T_{p_i}$ be a transition of this process, two corresponding actions happen in the following sequence:

  1. $\langle [S_1, ..., S_i, ..., S_m], \Sigma_{sch} \rangle \xrightarrow{a} \langle [S_1, ..., S'_i, ..., S_m], \Sigma_{sch} \rangle$
  2. $\langle [S_1, ..., S'_i, ..., S_m], \Sigma_{sch} \rangle (\xrightarrow{clock})^+ \langle [S_1, ..., S'_i, ..., S_m], \Sigma'_{sch} \rangle$.

  For instance, suppose a system has only process `P` that is currently running; this process uses a variable named `a`; the scheduler does not handle the clock event. This process performs statement "`a = 1;`" to change its state from S_1 to S'_1. The state of the system is changed as follows.

  1. $\langle [S_1], \langle \sigma_s, (C_1, ..., C_k), P_r \rangle \rangle \xrightarrow{\text{a = 1;}} \langle [S'_1], \langle \sigma_s, (C_1, ..., C_k), P_r \rangle \rangle$
  2. $\langle [S'_1], \langle \sigma_s, (C_1, ..., C_k), P_r \rangle \rangle (\xrightarrow{clock})^+ \langle [S'_1], \langle \sigma_s, (C_1, ..., C_k), P_r \rangle \rangle$

- If the current process performs an *exec* action, three actions happen in the following sequence:

[8] Here, *get* is considered as an action to change the value of a variable.

1. $\langle [S_1, ..., S_m], \Sigma_{sch} \rangle \xrightarrow{exec} \langle [S_1, ..., S_m, S_{m+1}], \Sigma_{sch} \rangle$
2. $\langle [S_1, ..., S_m, S_{m+1}], \Sigma_{sch} \rangle (\xrightarrow{new})^+ \langle [S_1, ..., S_m, S_{m+1}], \Sigma'_{sch} \rangle$
3. $\langle [S_1, ..., S_m, S_{m+1}], \Sigma'_{sch} \rangle (\xrightarrow{clock})^+ \langle [S_1, ..., S_m, S_{m+1}], \Sigma''_{sch} \rangle$.

For instance, suppose that the system has a collection named `ready`: $C_1 = \langle Pid_1, \succ_1, \sim_1 \rangle$ with $Pid_1 = \{\}$, $\succ_1 = \{(P, Q)\}$, $\sim_1 = \{\}$. The scheduler does not handle the clock event and the event `new_process` is defined as follows.

```
new_process (process target) {
    move target to ready ;
}
```

If the current process executes statement `sch_exec("P")` to execute the process `P` (an *exec* action). The state of the system is changed as follows.

1. $\langle [S_1, ..., S_m], \Sigma_{sch} \rangle \xrightarrow{exec} \langle [S_1, ..., S_m, S_{m+1}], \Sigma_{sch} \rangle$, where S_{m+1} is the state of the new process `P`,
2. $\langle [S_1, ..., S_m, S_{m+1}], \Sigma_s \rangle (\xrightarrow{\text{new}})^+ \langle [S_1, ..., S_m, S_{m+1}], \Sigma'_s \rangle$ with $\Sigma_s = \langle \sigma_s, (C_1), P_r \rangle$, where $\Sigma'_s = \langle \sigma_s, (C'_1), P_r \rangle$ with $C'_1 = \langle Pid'_1, \succ_1, \sim_1 \rangle$ and $Pid'_1 = \{P\}$,
3. $\langle [S_1, ..., S_m, S_{m+1}], \Sigma'_{sch} \rangle (\xrightarrow{clock})^+ \langle [S_1, ..., S_m, S_{m+1}], \Sigma'_{sch} \rangle$.

- If the current process performs an *api* action that raises an *inter* action taken by the scheduler, two actions happen in the following sequence:

1. $\langle \Sigma_{proc}, \Sigma_{sch} \rangle \xrightarrow{inter} \langle \Sigma'_{proc}, \Sigma'_{sch} \rangle$
2. $\langle \Sigma'_{proc}, \Sigma'_{sch} \rangle (\xrightarrow{clock})^+ \langle \Sigma'_{proc}, \Sigma''_{sch} \rangle$.

For instance, if the current process executes statement `sch_api_self(terminate)` (an *api* action) to call api function `terminate` defined in the *scheduler description* as indicated below to terminate itself (an *inter* action) and the scheduler does not handle the clock event. The state of the system is changed as follows.

```
function terminate(process target) {
    remove target;
}
```

1. $\langle \Sigma_p, \langle \sigma_s, (C_1), P_r \rangle \rangle (\xrightarrow{inter})^+ \langle \Sigma'_p, \langle \sigma_s[\perp/run], (C_1), P'_r \rangle \rangle$ with $\Sigma_p = [S_1, ..., S_i, ..., S_m]$, where $\Sigma'_p = [S_1, ..., S_{i-1}, S_{i+1}, ..., S_m]$ with S_i is the state of the current process.
2. $\langle \Sigma'_{proc}, \Sigma'_{sch} \rangle (\xrightarrow{clock})^+ \langle \Sigma'_{proc}, \Sigma'_{sch} \rangle$.

We note that if there is no currently running process after the occurrence of these *sequence-action*s above, the scheduler will perform the *select* action to select a process to run.

In the example (as indicated in Figure 3.2), when a process executes the statement `sch_api_self(terminate)` to terminate itself (an *api* action), the scheduler will handle this task by performing the function `terminate` (an *inter* action). Because the scheduler

does not use any clock variable[9] and the handler for the `clock` event is not defined; therefore, nothing happens for the *clock* action. After that, because the running process is not determined, the scheduler will select another process to run (a *select* action).

3.5 Summary

We have introduced the DSL for the scheduling strategy. Each specification contains the following two parts: one for the attributes of the processes and the other for the behaviors of the scheduler. The attributes of the processes are used for performing the scheduling tasks, which are handled by the corresponding events. Some fixed events are introduced. Moreover, we also support defining the events raised from the current process to carry out the scheduling tasks. The process selected to run defined by the ordering method used by the collection, which stores the processes in the system. The behaviors of the scheduler are defined using the statements in the DSL.

With this language, we provide a method to specify the scheduling strategies. Actually, the DSL can facilitate the scheduling strategy by changing the definition of the behaviors of the scheduler using the corresponding statements. In this chapter, we introduced the formal definitions to indicate the system with the scheduling policy. We also introduced the language semantics represented by the transition relation between the system states. In the next chapter, we will consider the algorithm based on the scheduling strategies to search the states for verifying the behaviors of the system.

[9]Each clock variable is automatically increased by 1 after the occurrence of a process action.

Chapter 4

Verifying and Analyzing Systems under Scheduling Policies

This chapter introduces an algorithm to explore the state space under the scheduling policy described in the DSL to verify the behaviors of the system. We then present an approach to generating the information from the scheduling strategy to perform the scheduling tasks. For analyzing the behaviors of the system, we adopt the algorithm proposed by Clarke, Emerson, and Sistla [24] to check properties expressed as a CTL formula and an algorithm introduced in [36] by Emerson et al. to handle RTCTL formula (see Chapter 2). In order to check the properties in a form of a CTL/RTCTL formula, we provide a support for taking the time into account with handling `clock` event in the scheduling policy.

4.1 Search Algorithm with Scheduling Policies

We use a system with two processes `t1` and `t2` as shown in Figure 4.1 to demonstrate the case that takes the scheduling into account to search the system states. With this example, the system uses the *priority* strategy and the priority of `t1` is greater than that of `t2`. Figure 4.2 shows the policy and the attributes of the processes in the DSL.

At the initial state, because process `t1` has higher priority than process `t2`, process `t1` is selected to run. The executions of these processes now are limited by the scheduler (as depicted in Figure 4.3): only an execution is determined (process `t1` is first selected). Note that in the case that the priority of process `t1` is less than that of `t2`, process `t2` is selected to run; thus, we have another execution.

Because the scheduling strategy affects to the running order of a system, the executions determined by the policy are different from that defined by the existing algorithms (e.g. DFS or BFS) to search the states. Therefore, to perform the scheduling policy, we need another algorithm to explore the state space. In this research, we propose an algorithm for constructing the state graph. The search algorithm is shown in Algorithm 3. It is extended from the DFS with the two main differences as follows.

- Firstly, the scheduler determines a process that can be selected from a set of candidate processes.
- Secondly, both the behaviors of the scheduler and the behaviors of the processes are taken into account in the exploration.

```
int a, b;

proctype t1() {
 do
 :: d_step{ (a+b)<5 -> a++}
 :: d_step{ else -> sch_api_self(terminate) }
 od;
}

proctype t2() {
 do
 :: d_step{ (a+b)<5 -> b++}
 :: d_step{ else -> sch_api_self(terminate) }
 od;
}

init {
 a = 2; b = 0;
 run t1();
 run t2();
}
```

Figure 4.1: A process program

```
def process Priority {
 attribute{
  var byte priority;
 }
 proctype t1() {
  this.priority = 5;
 }
 proctype t2() {
  this.priority = 3;
 }
}
init {
 [{t1(), t2()}]
}
```

a) Process attribute

```
scheduler Priority () {
 data {
  collection ready using priorityOrder;
 }
 event handler {
  select_process (process p) {
   get process from ready to run;
  }

  new_process (process target) {
   move target to ready;
   if (!running_process.isNull()) {
    if (target.priority > running_process.priority) {
     move running_process to ready;
    }
   }
  }
 }
 interface {
  function terminate(process target) {
   remove target;
  }
 }
}
comparator {
 variable { int x; }
 comparetype priorityOrder(process p_n, p_o) {
  x = p_n.priority - p_o.priority;
  if (x>0) return greater;
  else if (x==0) return equal;
       else return less;
 }
}
```

b) Scheduler description

Figure 4.2: Priority policy

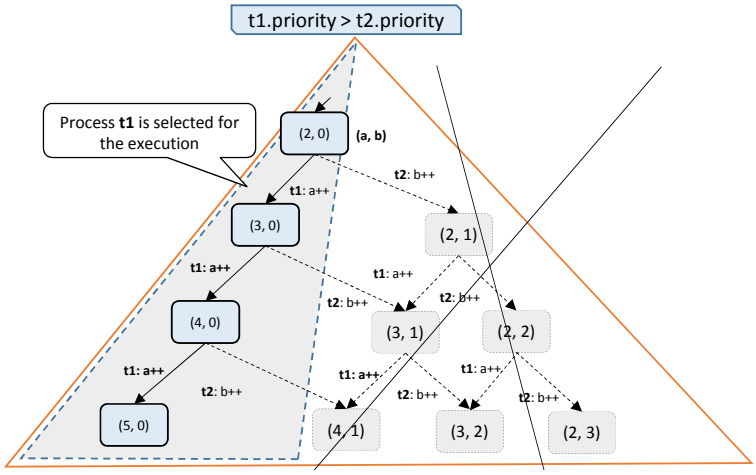


Figure 4.3: Exploring the states with scheduling policy

As we see in the previous chapter, the system state includes the scheduler state and the sequence states of the processes. In this algorithm, we also use a state space $\mathcal{SP}$ and a stack $\mathcal{ST}$. To update the contents of the state space, we use following functions: $Add\_state(\mathcal{SP}, \Sigma)$ to add state Σ as an element, and $Contains(\mathcal{SP}, \Sigma)$ to check whether element Σ has been visited or not. The stack is for storing the search steps with the corresponding operations *Push*, *Top*, and *Pop*.

This algorithm performs the search starting from function START (line 5) to visit the states of the system starting at state Σ_0. The scheduler state is initialized by function `SCH_INIT` (line 6). Function `SCH_SELECT` (line 15) corresponding to event *select_process* is to get the processes for the execution. This action is handled by the `select_process` event handler. Because the processes in the collection are ordered, a set of processes can be returned. If no process can be run, it returns an empty set (line 16). All of the actions[1] which can be performed are considered (line 25, 26). The state of the system is updated using function `SCH_TAKE` following an action a of the process (line 27): Σ_a = `SCH_TAKE`(a, Σ). This function represents the following behaviors:

- the behavior of the process defined by the transition $\langle S_i, a, S_i' \rangle \in T_p$, where S_i and S_i' are the states of the process and $a \in normal \cup \{get\}$ is an action of the process;
- the behavior of the process and the behavior of the scheduler corresponding to a scheduling action $a \in \{api, exec\}$ of the process (i.e. handling the scheduling events).

Function `SCH_CLOCK` (line 17, 28) is used to perform the behaviors related to time (defined by the *clock* event handler and the configuration of the processes).

To construct the state graph, the algorithm starts with the initial state of the system. Line 9 assigns the starting node of the graph. Function `NODE` (line 9, 18, 29) creates a new node (if it does not exist) corresponding to a system state. Following each search step, the edge between the current node and the new node is created. This is done by function `EDGE` (line 18, 29).

The functions `SCH_SELECT`, `SCH_TAKE` and `SCH_CLOCK` used in this algorithm to determine the transition relation $\langle \Sigma, a, \Sigma' \rangle \in T_s$ of the system. We do not show in this

[1]Each process can have many actions which can be performed non-deterministically.

Algorithm 3 Construct the state graph following the scheduling strategy

```
1: Stack: 𝒮𝒯 = ∅
2: State space: 𝒮𝒫 = ∅
3: Start node: StartNode = null
4:
5: procedure START
6:     Σ_0 = SCH_INIT()                          ▷ initializes system state
7:     Push(𝒮𝒯, Σ_0)                             ▷ pushes system state to stack
8:     Add_state(𝒮𝒫, Σ_0)                        ▷ adds a state to the state space
9:     StartNode = NODE(Σ_0)
10:    SEARCH
11: end procedure
12:
13: procedure SEARCH
14:    Σ = Top(𝒮𝒯)
15:    P = SCH_SELECT(Σ)
16:    if (P == ∅) then
17:        Σ_t = SCH_CLOCK(Σ)
18:        EDGE(NODE(Σ), NODE(Σ_t))
19:        if (Contains(𝒮𝒫, Σ_t) == false) then
20:            Push(𝒮𝒯, Σ_t)
21:            Add_state(𝒮𝒫, Σ_t)
22:            SEARCH
23:        end if
24:    else
25:        for (p ∈ P) do
26:            for (a ∈ p.L_p) do                ▷ a is an action of p
27:                Σ^a = SCH_TAKE(a, Σ)
28:                Σ_t^a = SCH_CLOCK (Σ^a)
29:                EDGE(NODE(Σ, NODE(Σ_t^a))
30:                if (Contains(𝒮𝒫, Σ_t^a) == false) then
31:                    Push(𝒮𝒯, Σ_t^a)
32:                    Add_state(𝒮𝒫, Σ_t^a)
33:                    SEARCH
34:                end if
35:            end for
36:        end for
37:    end if
38:    Pop(𝒮𝒯)
39: end procedure
```

algorithm the way to handle the errors. However, because the stack $\mathcal{ST}$ records the searching steps, if an error occurs (e.g. violating an invariance) when the system takes an action (i.e. using the function `SCH_TAKE`), we can display the counterexample by exporting the trail from this stack. In addition, with the algorithm proposed, we can perform the two searches as in the nested-depth first search to check the *liveness* properties under the scheduling policy.

Following this algorithm, the exploration of the state space in the example (Figure 4.1) with the priority policy shown in Figure 4.2 is depicted in Figure 4.4. The execution steps are as follows.

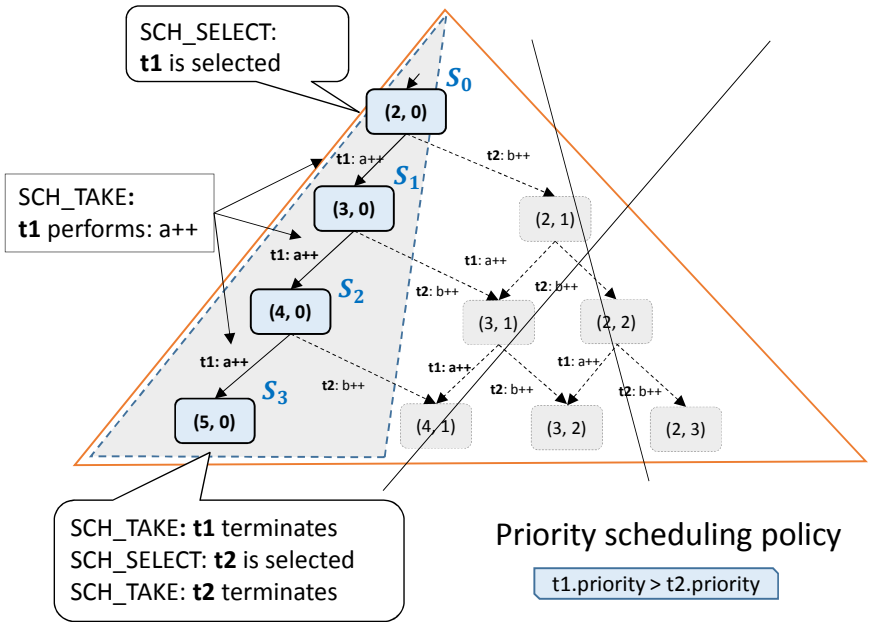


Figure 4.4: An example for exploring the state space

1. At the initial state, the values of the variables are initialized: `(a, b) = (2, 0)`, and we have two processes in the system (i.e. `t1` and `t2`).

2. The function `SCH_SELECT` is performed to select a process to run. As a result, a set of processes are returned. In fact, it contains only process `t1`. That is because process `t1` has higher priority than that of `t2`.

3. Now, the action `a++` of `t1` is performed to increase value of variable `a`. This is done by function `SCH_TAKE`.

4. Because we use the *priority* scheduling policy, the execution time is not considered. Therefore, the function `SCH_CLOCK` does nothing and a new node is created (corresponding to the state with `(a, b) = (3, 0)`).

5. Actually, process `t1` continues executing the statement `a++` while the condition `(a+b) < 5` satisfies. This fact will change the states of the system following this sequence: S_0, S_1, S_2, S_3.

6. At the state S_3, process `t1` terminates. Now the scheduler perform the `select` action (`SCH_SELECT`), only process `t2` is selected to run. However, this process also terminates because the condition `(a + b) < 5` does not satisfy. Now, the exploration is finished.

4.2 Generating Scheduling Information

In this research, the information necessary for performing the policy is generated from the specification of the scheduling strategy in the DSL. We note that the behaviors of the scheduler are realized from the scheduling strategy (i.e. *scheduling event*(s) defined in *event handler*(s) and *interface function*(s)). For instance, the selection of the scheduler is defined by `select_process` event handler; the scheduling event corresponds to the fact that current process terminates itself is defined by interface function `terminate` (as indicated in Figure 4.5).

```
...
 event handler {
  select_process (process p) {
   get process from ready to run;
  }

  new_process (process target) {
   move target to ready;
   if (!running_process.isNull()) {
    if (target.priority > running_process.priority) {
     move running_process to ready;
    }
   }
  }
 }
 interface {
  function terminate(process target) {
   remove target;
  }
 }
}
...
```

Figure 4.5: Scheduling events

The generation is depicted in Figure 4.6.

- The variables (1) are converted from the *scheduler data*.
- The template(s) (`proctype`) for the process(es) and the attributes of the processes are used to generate *process information* (2).
- The collections in *scheduler data* is used to generate *process collections* (3), which use the *ordering methods* declared in the *scheduler description*.
- The *process initialization* (i.e. `init` part of the *process attributes*) defines the initial function in (4).
- Other functions in (4) are also realized from the *interface functions*.
- The events are used to generate *event handlers* (5).

Using this information and the functions above, we can perform the search algorithm following the scheduling strategy.

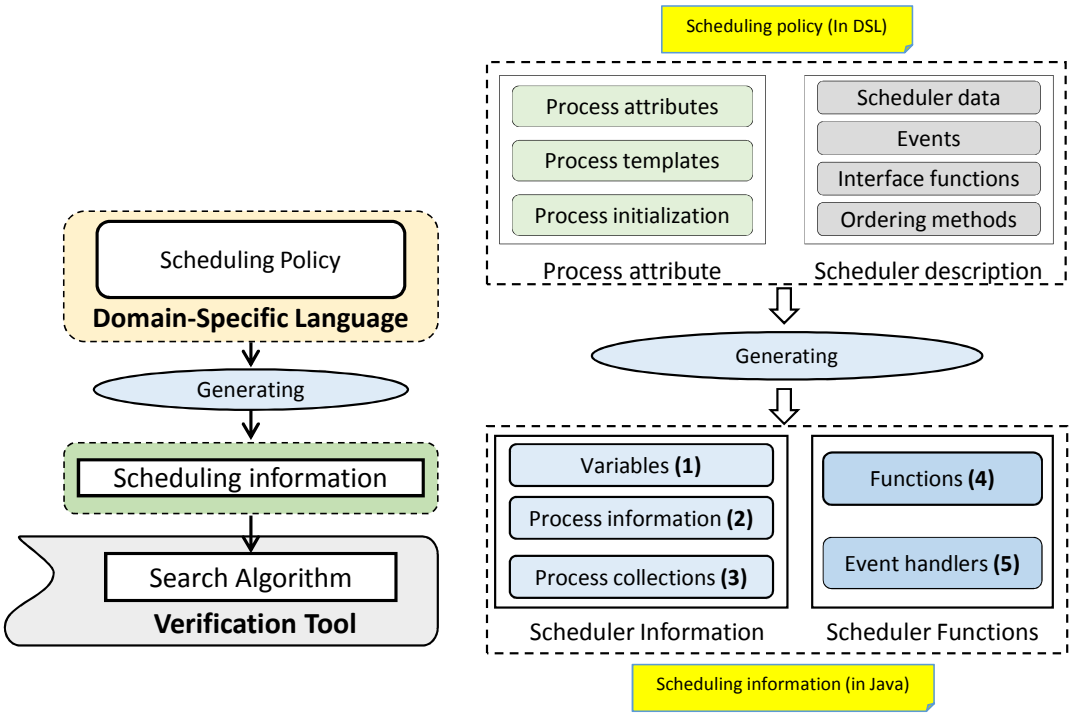


Figure 4.6: Scheduling information generation approach

4.3 Analyzing Systems with Scheduling Policies

We introduce an approach to analyze system under scheduling policy based on labeling the graph realized from the state space. We use a CTL/RTCTL formula to represent the qualitative/quantitative property. This approach is different from the current algorithms that construct the state space and evaluates the property on-the-fly.

In fact, with the algorithm to search the system states above we can verify the system qualitatively. However, it is still insufficient to capture the time related to the behaviors of the system. To deal with the time, we use timed-Kripke structure [36] for modeling the system with considering that an action consumes one tick.

- To represent the time, we use the clock variables to capture the time. In our research, we also consider the period of tasks and support the definitions of periodic behaviors in the DSL. An example for the definition is depicted in Figure 4.7. With this example, four periodic processes with the same period (20) are defined. However, these processes have different values for initial offset, priority, and deadline.

- Taking an action of a process will consume 1 tick and raise a timer event. This event is handled by the scheduler using the `clock` event handler. Figure 4.8 shows a simple example to check the deadline by handling the `clock` event.

For the analysis, we provide some language elements to support the definition of the property and observing the behaviors of the processes. The property is defined in the `verify` part of the *scheduler description* following the grammar shown in Figure 4.9.

The behaviors of the system can be observed using `checkpoint` statement. Corre-

```
def process experiment1{
 attribute {
  clock c ;
  var byte priority ;
  var byte deadline ;
 }
 proctype P(byte priority = 0; byte deadline = 10){
  this.priority = priority;
  this.deadline = deadline;
 }
}

config {
 periodic process P(2,16) offset = 6 period = 20 ;
 periodic process P(3,11) offset = 9 period = 20 ;
 periodic process P(2,8) offset = 11 period = 20 ;
 periodic process P(5,20) offset = 10 period = 20 ;
}
```

Figure 4.7: Dealing with time

```
scheduler FP(){
 data {
  collection ready using priorityOrder;
 }
 event handler {
  select_process (process p ) {
   get process from ready to run;
  }
  clock (){
   if (!running_process.isNull()) {
    assert running_process.c <= running_process.deadline ;
   }
  }
  new_process (process target) {
   move target to ready;
   if (!running_process.isNull()) {
    if (target.priority > running_process.priority) {
     move running_process to ready;
    }
   }
  }
 }
}
```

Figure 4.8: Handling the timer event

sponding to the execution of this statement, the state of the system is labeled with the corresponding value for the variable determined by this statement (this variable is automatically generated, see the explanation below). An example for using the language to define the analysis is depicted in Figure 4.10. With this example, the property indicates that for every execution, the value of `a` always greater than or equal to that of `b` within 2 time units (`AG<=2 (Sys(a) >= Sys(b))`); we use two `checkpoint` statements with the label `exec_t1` for checking the execution of process `t1`, and `end_t2` for checking the termination of process `t2`.

To analyze the behaviors of the system under scheduling policies, now the first step (constructing the state space) is changed to realize the state graph. In the second step, we also adopt the existing algorithms to label the state graph (as mentioned before). These steps are depicted in Figure 4.11.

⟨*Verify*⟩ ::= 'verify' '{' [⟨*CTL_AT*⟩] ⟨*CTL*⟩ '}'
⟨*CTL_AT*⟩ ::= '@' ⟨*Expr*⟩ ':'
⟨*CTL*⟩ ::= '(' ⟨*Expr*⟩ ')' | 'not' ⟨*CTL*⟩ | 'or' ⟨*CTL*⟩ ⟨*CTL*⟩ | 'implies' ⟨*CTL*⟩ ⟨*CTL*⟩ | 'AX' ⟨*CTL*⟩ | 'AF' ⟨*CTL*⟩ | 'AG' ⟨*CTL*⟩ | 'EX' ⟨*CTL*⟩ | 'EF' ⟨*CTL*⟩ | 'EG' ⟨*CTL*⟩ | 'AU' ⟨*CTL*⟩ ⟨*CTL*⟩ | 'EU' ⟨*CTL*⟩ ⟨*CTL*⟩

Figure 4.9: The grammar for the property

```
scheduler Priority () {
 data {
  collection ready using priorityOrder;
 }
 event handler {
  select_process (process p) {
   get process from ready to run;
   if (p.hasName("t1")) checkpoint exec_t1 ;
  }
  new_process (process target) {
   move target to ready;
   ...
  }
 }
 interface {
  function terminate(process target) {
   if (target.hasName("t2")) checkpoint end_t2;
   remove target;
  }
 }
}
comparator {
...
}
verify {
  AG <=2 (Sys(a) >= Sys(b))
}
```

Figure 4.10: Language elements for the analysis

In addition, corresponding to each *checkpoint* statement, each node of the graph is labeled with the numbers representing the values for the earliest time and the latest time (*min*, *max*) for the events perform the statement. These values are updated when the state is visited. If a loop is detected from the DFS stack, all events with the corresponding *checkpoint* statements will occur infinitely.

4.4 Summary

We have introduced an algorithm to construct the state graph based on the behaviors of the scheduler to model checking a system and verify its behaviors quantitative and qualitatively. With this algorithm, the behaviors of the scheduler are taken into account to search the system states. However, until now we expect that the scheduling strategy is specified correctly in the DSL. In the next chapter, we will validate the specification of the strategy in the DSL based on testing the policy with the implementation of the scheduler in a real OS to increase the confidence of specification of the scheduling policy.

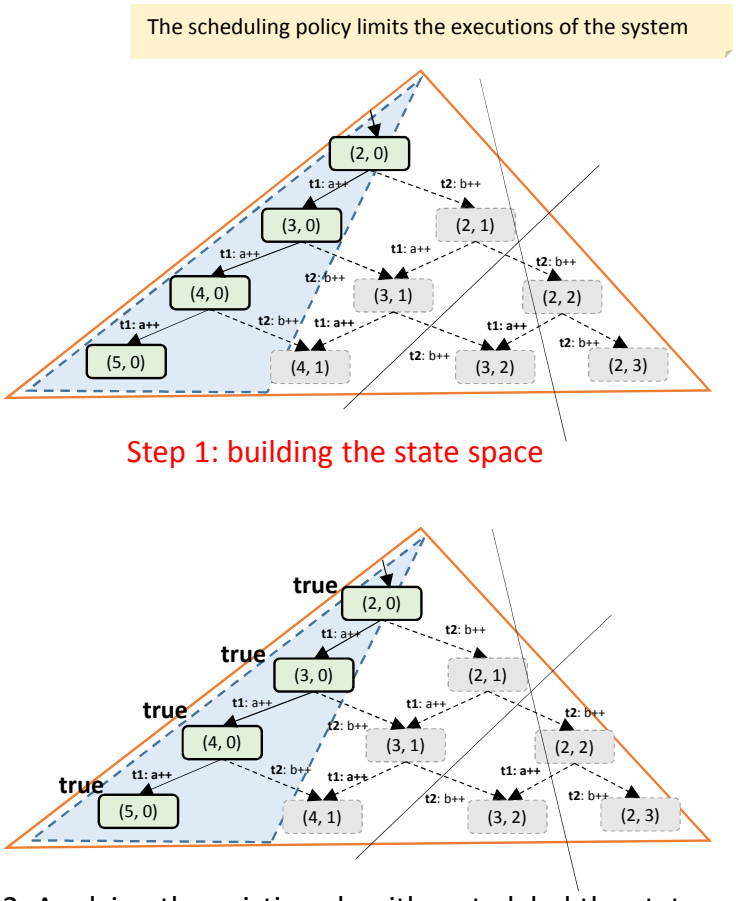


Figure 4.11: Labeling the state graph under the scheduling

Chapter 5

Testing Scheduling Policies

We aim to ensure the accuracy of a concurrent system executed under scheduling policies. To do that, we propose a DSL to describe the scheduling strategies. However, before doing any verification, we make an assumption that the strategy specified in the DSL is correct. Actually, the correspondence between the scheduling policy in the DSL and the real scheduler in the OS affects the verification results. The remaining problem now is ensuring that the specification of the scheduling strategy described in our DSL conforms with the real one in the OS. This chapter introduces an approach to address this problem using MBT techniques.

5.1 The Approach

As mentioned in Chapter 1, it is difficult to find the specification of the implementation of a scheduling policy in a real system. We mean that corresponding to an implementation, there is no specification or the specification is not clear to describe the behaviors of the scheduler. For instance, the specification of real-time FIFO scheduling policy of Linux OS indicates that if a call to the functions `sched_setscheduler`/`sched_setparam` to increase the priority of the running or runnable `SCHED_FIFO` thread, it *may preempt* the current thread with the same priority[1]. Therefore, the are two options for the implementation: the corresponding process preempts the current process or this running process isn't preempted. In fact, there are multiple versions of Linux OS and which option is implemented on each version of Linux is not described in the specification. In addition, the behaviors of the scheduler in a real OS can be observed only in executing the system. Therefore, we apply testing techniques to check the correspondence between the policy in the DSL and the implementation in a real system. That helps us to increase the confidence of the specification of the policy.

Our approach for the testing is to check that the behaviors indicated by the policy are the same as the real ones. The idea is to check the executions following the policy are accepted by the real scheduler. Figure 5.1 depicts an example of a system with 3 processes (`P`, `Q` and `R`) using *priority* scheduling policy. With this example, process `P` has the highest priority and process `R` has the lowest priority. The only action of these processes is terminating itself. We know that when the currently running process terminates itself, the scheduler will select the highest priority process to run. This fact is necessary to check. With this example, firstly, process `P` is selected to run because

[1] `http://man7.org/linux/man-pages/man7/sched.7.html` (accessed: April-2018)

it has the highest priority; this process terminates; then process `Q` is run; at the end, process `R` is selected and also terminates. We now have only one execution for this system. This execution is considered as a test case. We then check that the execution is accepted by the real scheduler.

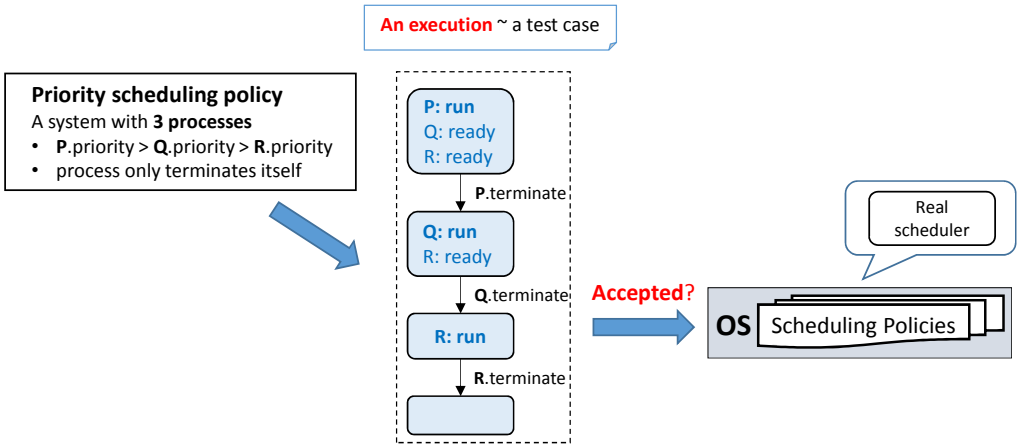


Figure 5.1: An example for testing

Corresponding to this execution, we can observe the running status of these processes in the real OS. This fact can be tested by checking the execution order of the processes. With the example in Figure 5.1, the code can be defined as follows.

```
int exec_order = 0 ;
void P(){
    assert (exec_order == 0);
    exec_order++ ;
}
void Q(){
    assert (exec_order == 1);
    exec_order++ ;
}
void R(){
    assert (exec_order == 2);
    exec_order++ ;
}
```

The problem now is that with a concurrent system, there are multiple executions of the processes. It leads to the fact that manually making the tests is error-prone and time-consuming. That means a systematic approach is necessary. To address this problem, we apply MBT techniques. The main reason is that MBT can automatically and exhaustively generate the tests for validating the systems.

Our approach for testing the scheduling strategy is depicted in Figure 5.2. It includes three main steps. First, we prepare the model of the system following the scheduling policy for the testing. Second, we apply MBT to generate the tests. Third, we conduct the testing to check the acceptance of the tests generated in the second step by the real scheduler.

- In the first step, a model is used to represent the behaviors of the system. With a scheduling policy, the system contains a set of processes and a scheduler, which

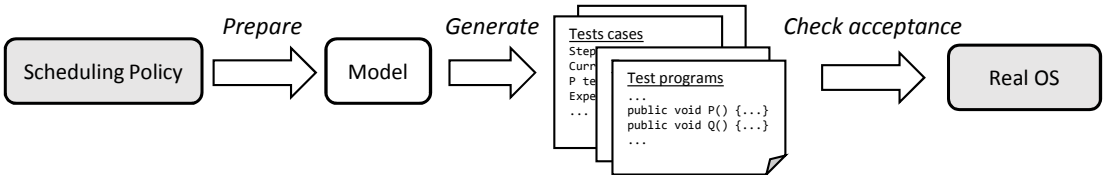


Figure 5.2: Testing approach

determines a configuration. We call the processes and their attributes as an environment. Actually, the environment is necessary for testing because the processes are used to perform the scheduling tasks.

In fact, manually preparing an appropriate environment is also time-consuming and easy to make errors. It is because there are several cases for the number of the processes and various values for their attributes. For instance, we may have multiple environments for the *priority* policy with the different number of processes and the values for their priorities (as indicated in Figure 5.3). Therefore, we adopt an automated approach to prepare the necessary environments. To do that, we introduce a language for the environments and generate environments automatically from the corresponding specification.

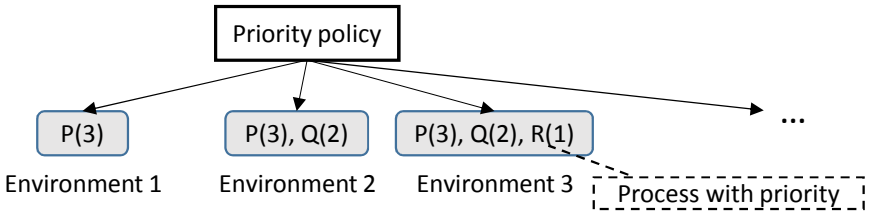


Figure 5.3: Multiple environments with a scheduling policy

- In the second step, we explore the model of the system (indicated by the set of processes and the scheduler) to determine the executions of the processes to generate the tests. From each execution, a test case is realized. The test is now constructed by mapping the behaviors of the system in the test case to the code generated. The approach is depicted in Figure 5.4.
- In the last step, we perform the tests to check whether these executions are accepted by the real scheduler in an OS.

5.2 Preparing Environments

In our work, the scheduling strategy is specified in the DSL. To indicate the model of the system, we need to prepare the set of processes with the corresponding configuration (i.e. process attributes) called an environment. The problem now is that how to prepare an appropriate environment for testing the scheduling strategy. To address this problem,

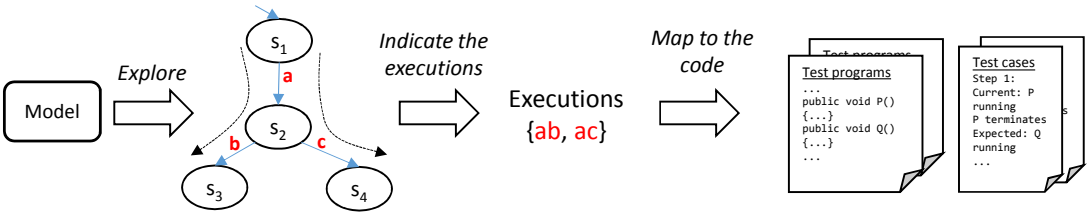


Figure 5.4: Generating the tests approach

we need to answer the following questions: a) How many processes are used? b) How to assign the attributes for the processes? and c) How to present the scheduling tasks (i.e. the behaviors of the scheduler)?

Actually, the number of processes with their attributes can be determined based on the purpose of the tests. For instance, to check the selection of the scheduler with *priority* policy we can use two processes with different values for the priorities of these processes. In fact, the values for the attributes can be limited, e.g. we can use different priorities in the range [0..1] for the set of processes. Moreover, we can determine the behaviors needed for the testing, e.g. the processes perform scheduling tasks: terminate, execute, etc.

Base on that fact, we propose a DSL for defining the environments. Here, our approach follows the class diagram as in the object-oriented design principle. That means we provide a method to define the attributes and the behaviors (i.e. the methods) of the processes (called *process class*). From the definition of the *process class*, we generate the *process program*, which specifies the behaviors of the processes, and the initial values for the attributes of the processes. The approach is depicted in Figure 5.5.

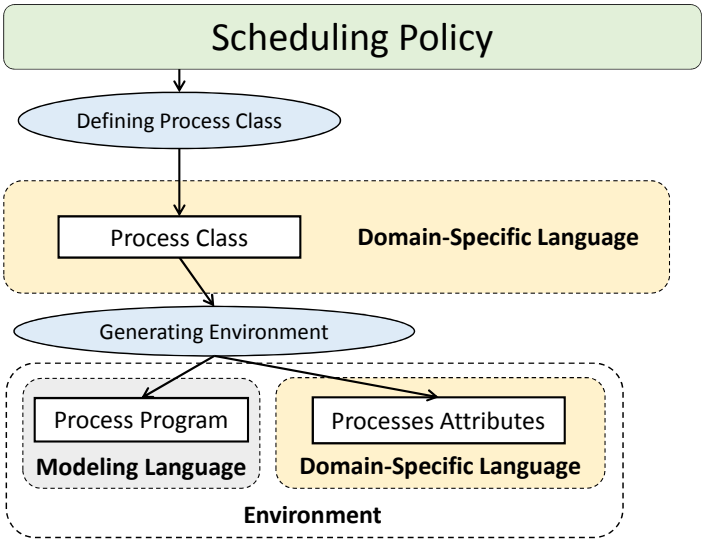


Figure 5.5: Preparing the environment approach

The grammar of the language for defining the *process class* is depicted in Figure 5.6. An example for the processes with the attributes and the methods in the DSL is depicted in Figure 5.7.

⟨*ProcClass*⟩ ::= '`process`' ⟨*ID*⟩ ['`refines`' ⟨*ID*⟩] '`{`' [⟨*DefAttr*⟩] ⟨*DefBehavior*⟩ '`}`' '`configuration`' '`{`' [⟨*ProcessConfig*⟩] ⟨*ProcessInit*⟩ '`}`'
⟨*DefAttr*⟩ ::= '`attribute`' '`{`' (⟨*AttDef*⟩)* [⟨*Constraints*⟩] '`}`'
⟨*AttDef*⟩ ::= ⟨*ID*⟩ '`:`' '`type`' '`=`' ⟨*Type*⟩ ['`,`' '`value`' '`=`' ⟨*ListDef*⟩] '`,`' '`default`' '`=`' ⟨*Value*⟩ '`;`'
⟨*ListDef*⟩ ::= '`[`' ⟨*List*⟩ ('`,`' ⟨*List*⟩) * '`]`'
⟨*List*⟩ ::= ⟨*Range*⟩ | ⟨*BOOL*⟩ | ⟨*ID*⟩
⟨*Range*⟩ ::= ⟨*INT*⟩ '`..`' ⟨*INT*⟩
⟨*Value*⟩ ::= ⟨*BOOL*⟩ | ⟨*INT*⟩
⟨*DefBehavior*⟩ ::= ⟨*ProcType*⟩ | ⟨*ProcBehav*⟩
⟨*ProcType*⟩ ::= (⟨*ProcessType*⟩)*
⟨*ProcessType*⟩ ::= '`proctype`' ⟨*ID*⟩ '`{`' [⟨*Constraints*⟩] (⟨*ProcBehav*⟩)* '`}`'
⟨*ProcBehav*⟩ ::= '`behavior`' '`{`' (⟨*PBehav*⟩)* '`}`'
⟨*PBehav*⟩ ::= ⟨*Constructor*⟩ | ⟨*Method*⟩
⟨*Constructor*⟩ ::= '`constructor`' '`:`' ⟨*ID*⟩ '`(`' [⟨*PramList*⟩]'`)`' '`;`'
⟨*Method*⟩ ::= '`method`' '`:`' ⟨*ID*⟩ (('`(`' '`)`' '`;`') | ('`(`' ⟨*PramList*⟩ '`)`' '`{`' (⟨*AssignPara*⟩)* [⟨*Constraints*⟩] '`}`'))
⟨*AssignPara*⟩ ::= ⟨*ID*⟩ '`:`' '`value`' '`=`' ⟨*ListDef*⟩ '`;`'
⟨*Constraints*⟩ ::= '`constraint`' '`{`' (⟨*Constr*⟩) * '`}`'
⟨*Constr*⟩ ::= ⟨*Or*⟩ '`;`'

Figure 5.6: The grammar for the process class

- Each attribute of the processes is defined by the list of values or a range with a lower and an upper bound. We can indicate the constraints for assigning the values to each attribute. Each constraint is specified using a boolean expression.

  In the example shown in Figure 5.7, we define the attributes for Linux processes including the *priority* (`priority`), *nice* value (`nice`), the *type* of each process to indicate whether it is a real-time process or not (`sch_type`), the `preempt` attribute of a process indicates whether this process can be preempted or not by a new process which arrives to the system. For the purpose of testing, the upper bound and lower bound for the values can be limited, with this example, we set the range of the `priority` values to `[0..2]`.

```
//static priority; 0: normal, 1..99 real time
priority: type = byte, value = [0..2], default = 0;
//dynamic priority for SCHED_OTHER -20..19
nice: type = int, value = [0..2], default = 0;
//SCHED_OTHER, SCHED_FIFO, SCHED_RR
schtype: type = byte, value = [0,1,2], default = 0 ;
preempt: type = byte, value = [0,1], default = 1 ;
```

  In addition, the values assigned to the attributes can follow the corresponding constraints. For instance, with a normal process, the priority must be set to 0; this fact is indicated by the constraint "`(sch_type == 0) => (priority == 0)`".

```
(schtype == 0) => (priority == 0) ; //normal process
(schtype > 0) => (nice == 0) ; //real-time process
(priority == 0) => (schtype == 0) ; //normal process
```

```
process linux {
 attribute {
  priority: type = byte, value = [0..2], default = 0; //static priority; 0: normal, 1..99 real time
  nice: type = int, value = [0..2], default = 0; //dynamic priority for SCHED_OTHER -20..19
  schtype: type = byte, value = [0,1,2], default = 0 ; //SCHED_OTHER, SCHED_FIFO, SCHED_RR
  preempt: type = byte, value = [0,1], default = 1 ;

  constraint {
   (schtype == 0) => (priority == 0) ;  //normal process
   (schtype > 0) => (nice == 0) ;
   (priority == 0) => (schtype == 0) ;
  }
 }

 behavior {
  constructor : P(byte priority = 0; byte nice = 0; byte schtype = 0) ;
  method : terminate () ;
  method : runP(byte priority; byte nice; byte schtype){
   nice: value = [0..1];
   priority: value = [0..2] ;
   schtype: value = [0..2];

   constraint {
    (schtype == 0) => (priority == 0) ;  //normal process
    (schtype > 0) => (nice == 0) ;
    (priority == 0) => (schtype == 0) ;
   }
  }
 }
}

configuration {
 init {
  [{P(?,?,?), P(?,?,?)}]
 }
}
```

Figure 5.7: An example for defining process class

- There are two types of methods can be declared to describe the behaviors of the processes in the *process class*. One is the constructor, which is used to determine the initial values for the attributes of the processes; the other is for expressing the behaviors of the processes. Here, we aim to check the scheduling strategy; therefore, the behaviors corresponding to the scheduling tasks are considered. We use the *scheduling event*(s) raised from the processes (*process scheduling event*s) to specify these behaviors. To do that, we define interface functions in the *scheduler description*. Then, we define the methods in *process class* to call these functions to carry out the scheduling tasks (such as terminating the current process or executing a process, etc.). Each method can have parameters, which are also assigned following the constraints determined by the corresponding boolean expressions.

  With this example, three methods are defined for the Linux processes, the constructor for initializing the processes (method named `P`) and two methods corresponding to the behaviors of the processes: terminates itself (`terminate`) and executes another process (`runP`). These methods are with the values of the attributes for the constructor (all the values are set to 0) and the constraints for the parameters of the method `runP`. We note that these methods are to testing the scheduling tasks based on these behaviors of the processes defined above.

  ```
  constructor : P(byte priority=0; byte nice=0; byte schtype=0) ;
  method : terminate () ;
  ```

```
method : runP(byte priority; byte nice; byte schtype){
    nice: value = [0..1];
    priority: value = [0..2] ;
    schtype: value = [0..2];

    constraint {
        (schtype == 0) => (priority == 0);//normal process
        (schtype > 0) => (nice == 0);//real-time process
        (priority == 0) => (schtype == 0);//normal process
    }
}
```

- The system's configuration is defined with a set of processes, which is indicated in the *configuration* part. In the example, for testing the selection, we use only two processes to define the configurations at the initial time.

```
configuration {
    init {
        [{P(?,?,?), P(?,?,?)}]
    }
}
```

We adopt a simple approach for generating the corresponding *process program* and the *process attributes* from the *process class* by assigning the values for the attributes and for the parameters of the methods with checking the satisfaction of the constraints. The *process program* and a *process attributes*[2] generated from the *process class* in the example above are shown in Figure 5.8 and Figure 5.9.

```
proctype P () {
 do
   :: sch_api_self(terminate) ;
   :: sch_api_self(runP, 0,0,0) ;
   :: sch_api_self(runP, 0,1,0) ;
   :: sch_api_self(runP, 1,0,1) ;
   :: sch_api_self(runP, 1,0,2) ;
   :: sch_api_self(runP, 2,0,1) ;
   :: sch_api_self(runP, 2,0,2) ;
   :: skip ;
 od
}

init {
 run P() ;
 run P() ;
}
```

Figure 5.8: The process program generated from the process class

5.3 Test Generation for Scheduling Policies

We now have the environments generated from the definition of the *process class*. Actually, each environment is used to realize the model of the system, which determines

[2]We may have a set of *process attributes* generated from a *process class* definition.

```
def process linux_0  {
 attribute {
  var byte priority = 0 ;
  var int nice = 0 ;
  var byte schtype = 0 ;
  var byte preempt = 1 ;
 }

 proctype P (byte priority=0; byte nice=0; byte schtype=0) {
  this.priority = priority ;
  this.nice = nice ;
  this.schtype = schtype ;
 }
}

init {
 [{P(0,0,0),P(0,0,0)}]
}
```

Figure 5.9: A description of the attributes of the processes

the behaviors of the system following the scheduling strategy. We then use this model to generate the tests (test cases and test programs) for testing the scheduling policy implemented in an OS.

Our idea for the test generation is based on searching the state space using the model of the system. From the state space, we can indicate the executions of the processes. Each execution represents a test case. To make the tests, we map the behaviors of the system with the code generated. Each test is constructed by combining the code generated following the behaviors of the system. The approach for our test generation is depicted in Figure 5.10. Our ideas are as follows.

- Firstly, we aim to define the codes generated corresponding the behaviors of the system for specifying the test generation. Based on the roles of a scheduler, we support to specify that the scheduler captures the behaviors of the processes in our DSL. This allows us to generate the tests following the behaviors of the scheduler.

- Secondly, we extend our DSL to specify the test generation (called *test specification*). The description of the test is used to generate 1) a *generate structure* and 2) a *generate function*. These two artifacts are used to generate the tests following the search on the state space.

  - The *generate structure* determines the structure of the tests, and
  - The *generate function* is for constructing the tests in the search.

- Third, with the *scheduling information* generated from the strategy in the DSL, we can search the system states. We note that the codes (for the tests) are determined in the search.

  - Each test can be realized (on-the-fly) from the trail that 1) leads to the violation of a property (expressed by an assertion statement in the *process program*) at a state of the system (an error happens) or 2) contains a state that has been visited.

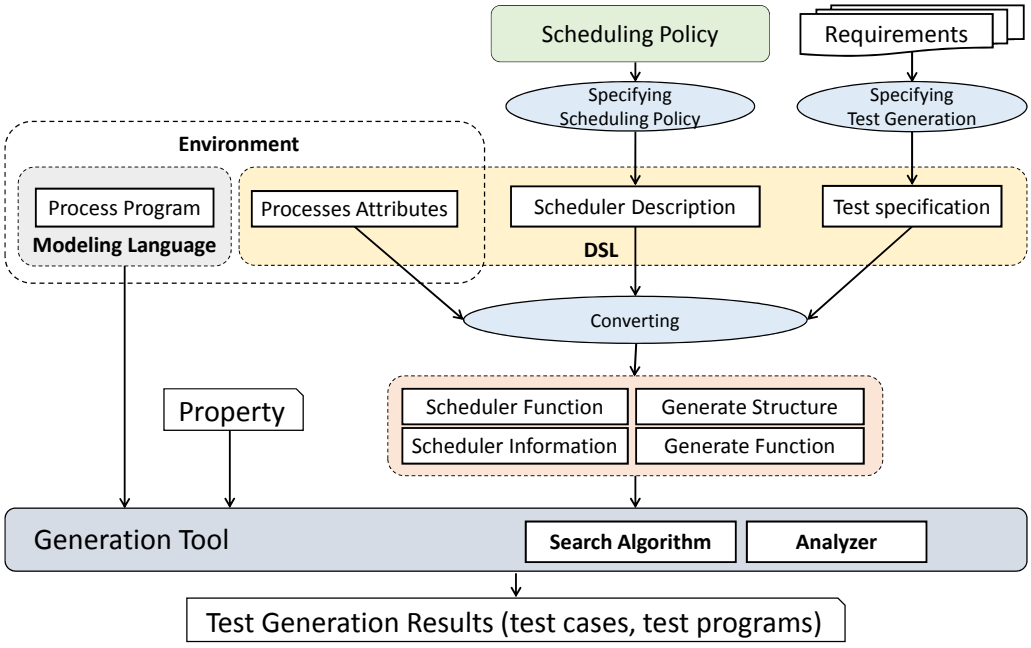


Figure 5.10: Test generation approach

– We can also generate the tests following a property (e.g. an execution of the system in which process `P` runs). To find the trails that satisfy a property (witnesses), we use a CTL/RTCTL formula to express the property and adopt the existing algorithms to label the state graph and find out the corresponding trails for generating the tests. To do this, we build an *analyzer* to determine the trails that satisfy the input property. From the corresponding trails, we can generate the tests.

5.3.1 Specifying Test Generation

We use `FIFO` scheduling policy to demonstrate the test generation as depicted in Figure 5.11. This example contains three files for describing the behaviors of the processes, the attributes of the processes and the behaviors of the scheduler with the description of test generation, which are specified in *process program* (a), *process attribute* (b) and *scheduler description with test specification* (c), respectively.

The description of a test generation contains the definitions of a) the *configuration* and b) the *component* of the tests. The grammar for the test generation is depicted in Figure 5.12.

The *configuration* specifies the option of the search for the test generation (see Section 5.3.3 for more details). The filename with its extension and the directory for putting the results of the generation can also be indicated in the *configuration* part. We support two types of the test, i.e. *test case* and *test program*.

- In general, each test case contains multiple steps, which indicate the values of the variables, the behavior of the system, and the expected values for these variables.
- A *test program* is a program for testing the components of a system (such as the

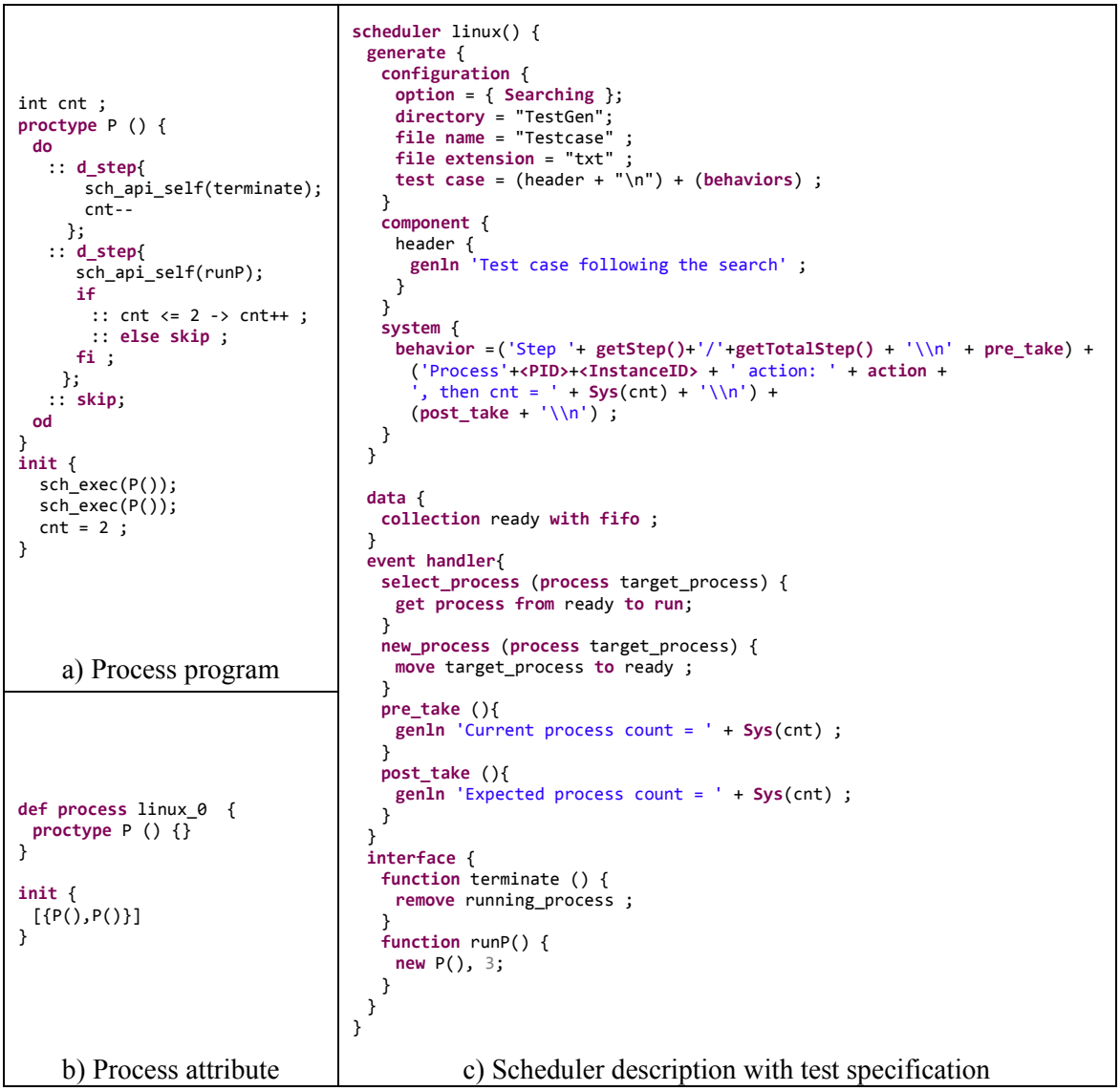


Figure 5.11: An example for the test generation

scheduler of an OS). The program usually has a structure (e.g. the header for the declaration, the *main* function for performing the program and the functions that express the behaviors of the processes).

We support defining the structure of a test with its *component*(s). Some special components including `init`, `processes`, `behaviors` and `error` are pre-defined.

- The `init` component is used for initializing the generation. We can use it to prepare the declaration of the test programs.
- The `processes` component corresponds to the set of processes.
- The `behaviors` component indicates the set of actions of a process.
- The `error` component points out the corresponding error (violation of a property indicating by an assertion in the *test program*) happening during the execution.

⟨*Generate*⟩ ::= '`generate`' '`{`' ⟨*GenConfig*⟩ ⟨*GenComp*⟩'`}`'

⟨*GenConfig*⟩ ::= '`configuration`' '`{`' [⟨*GenOption*⟩ '`;`'] [⟨*Dir*⟩ '`;`'] [⟨*FName*⟩ '`;`'] [⟨*FExt*⟩ '`;`'] '`test`' ('`program`' | '`case`' | '`data`') '`=`' ⟨*TestPart*⟩ '`}`'

⟨*GenOption*⟩ ::= '`option`' '`=`' '`{`' ⟨*GenOpt*⟩ ('`,`' ⟨*GenOpt*⟩)* '`}`'

⟨*GenOpt*⟩ ::= '`Searching`' | '`Error`' | '`Property`' | '`All`'

⟨*Dir*⟩ ::= '`directory`' '`=`' ⟨*STRING*⟩ '`;`'

⟨*FName*⟩ ::= '`file`' '`name`' '`=`' ⟨*STRING*⟩ '`;`'

⟨*FExt*⟩ ::= '`file`' '`extension`' '`=`' ⟨*STRING*⟩ '`;`'

⟨*TestPart*⟩ ::= ⟨*GenPart*⟩ ('`+`' ⟨*GenPart*⟩)*

⟨*GenPart*⟩ ::= '`(`' [⟨*STRING*⟩ '`+`'] (⟨*ID*⟩ | '`init`' | '`processes`' | '`behaviors`' | '`error`') ['`+`' ⟨*STRING*⟩] '`)`'

⟨*GenComp*⟩ ::= '`component`' '`{`' (⟨*Comp*⟩)* [⟨*InitGen*⟩] [⟨*ProcGen*⟩] '`}`';

⟨*Comp*⟩ ::= ⟨*ID*⟩ '`{`' (⟨*Gen*⟩ |⟨*GenLn*⟩)* '`}`'

⟨*InitGen*⟩ ::= '`init`' '`{`' ⟨*Template*⟩ '`}`'

⟨*ProcGen*⟩ ::= '`process`' '`{`' ⟨*Template*⟩ '`}`'

⟨*Template*⟩ ::= [⟨*SetTemplate*⟩] ⟨*Behavior*⟩

⟨*SetTemplate*⟩ ::= '`template`' '`=`' ⟨*Expr*⟩ '`;`'

⟨*Behavior*⟩ ::= '`behavior`' '`=`' ⟨*EventTemp*⟩ ('`+`' ⟨*EventTemp*⟩)* '`;`'

⟨*EventTemp*⟩ ::= '`(`' [⟨*Expr*⟩ '`+`'] ⟨*Event*⟩ ['`+`' ⟨*Expr*⟩] '`)`'

Figure 5.12: The grammar for the test generation

To support the test generation, firstly, we introduce two more events: `pre_take` and `post_take`. These events are for dealing with the pre-processing and post-processing of each behavior (action) of a process. For instance, we can display the current value of a variable before taking an action (`pre_take`) and the expected value of this variable after taking this action (`post_take`) in a test case. Secondly, in order to generate the tests, we introduce statements `gen` and `genln` for generating the code following the behaviors of the scheduler.

⟨*Gen*⟩ ::= '`gen`' [⟨*ID*⟩ '`,`'] ⟨*Expr*⟩ '`;`'

⟨*GenLn*⟩ ::= '`genln`' [⟨*ID*⟩ '`,`'] ⟨*Expr*⟩ '`;`'

In the example (as shown in Figure 5.11), we specify the *test cases* generation. Each test case contains two parts (`header` and `behaviors`). The structure of the tests and the template of each component are defined. We use the string operator to produce each component. The code generated is specified in the events `pre_take` and `post_take` using the `genln` statements. The value of the variable `cnt` defined in the *process program* can be get using the function `Sys()`. The codes generated indicate the current value of the `cnt` variable before taking an action of the current process and the expected value of this variable after taking this action.

```
pre_take (){
    genln 'Current process count = ' + Sys(cnt) ;
}
post_take (){
    genln 'Expected process count = ' + Sys(cnt) ;
}
```

We use the string operations to concatenate the codes and the components (in text string) with functions `getStep()` and `getTotalStep()` (the index of the step and the number of steps in a test case). This is defined in the `behavior` part as indicated below.

```
system {
    behavior=('Step '+getStep()+'/'+getTotalStep()+'\\n'+pre_take) +
        ('Process'+<PID>+<InstanceID> + ' action: ' + action +
        ', then cnt = ' + Sys(cnt) + '\\n') +
        (post_take + '\\n') ;
}
```

5.3.2 Formal Definitions

We give some formal definitions for generating the tests from the test generation and the model of the system.

Definition 5.1 (Generation function): $genCode : T_s \rightarrow String$ is a function from the set of transitions of the system T_s to the set of strings.

This function is determined by the test generation specification and the `gen`, `genln` statements used for defining the code generated. We note that the structure and the template of the components of the tests are also determined by this function. In the example indicated in Figure 5.11, the structure of the test defined in the *configuration* part, the template of the behaviors of the system, and the statements `genln`(s) used in the events `pre_take` and `post_take` are to construct function *genCode* above.

Definition 5.2 (Set of Input/Output): Set of input/output $\mathcal{IO} \subseteq D(x_0) \times D(x_1) \times ... \times D(x_n)$, where $x_i \in \mathcal{X}, i \in \mathbb{N}$ is a variable used by the system and $D(x_i)$ is the domain of x_i.

$\mathcal{IO}$ is the set of values for the variables used by the system. We define function $getIO : \mathcal{S}_{sys} \rightarrow \mathcal{IO}$ to get the values of the variables at a state of the system. These values represent the information of the system, and the function $getIO$ is used to get this information at a specific state. In the example, only a variable name `cnt` is used for storing the number of processes. This variable has `integer` as its domain. The function $getIO$ is to get the value of this variable at a specific state. It is presented by function `Sys` in the description of the test generation.

Definition 5.3 (Transition Input/Output): The input/output corresponding to a transition $t = \langle \Sigma, a, \Sigma' \rangle \in T_s$ is a pair $\langle i_t, o_t \rangle$ where $i_t = getIO(\Sigma)$ and $o_t = getIO(\Sigma')$.

The input and output corresponding to a transition represent the information of a system before taking an action and after taking this action. This information is extracted from the corresponding states of the system using the function $getIO$. For instance, $t_0 = \langle \Sigma_0, a, \Sigma_1 \rangle \in T_s$ is a transition from the initial state Σ_0 to state Σ_1 by performing action `d_step{sch_api_self(terminate);cnt--}`. We have $i_{t_0}[\![cnt]\!] = 2$ and $o_{t_0}[\![cnt]\!] = 1$ (corresponding to the values of variable `cnt`).

Definition 5.4 (Generation step): A generation step following a transition $t = \langle \Sigma, a, \Sigma' \rangle \in T_s$ is a tuple $\langle \langle i_t, o_t \rangle, c \rangle$, where $\langle i_t, o_t \rangle$ is the transition input/output of t and $c = genCode(\langle \Sigma, a, \Sigma' \rangle)$ is the code generated following to this transition.

A generation step indicates the code generated following a transition. The information used for the generation is determined by the input and output of the transition. For instance, the generation step corresponding to the transition $\langle \Sigma_0, a, \Sigma_1 \rangle$ above is $\langle \langle 2, 1 \rangle$, "*...Current process count = 2...*"$\rangle$. We use $\mathcal{GT}$ to denote the set of generation steps.

Definition 5.5 (Trail): A trail $[\langle \Sigma_0, a_0, \Sigma_1 \rangle, \langle \Sigma_1, a_1, \Sigma_2 \rangle, .., \langle \Sigma_{n-1}, a_{n-1}, \Sigma_n \rangle]$ is a sequence of transitions, where $\langle \Sigma_i, a_i, \Sigma_{i+1} \rangle \in T_s, i \in \mathbb{N}$ and Σ_0 is the initial state of the system.

For generating the tests, we indicate a sequence of transitions (called a trail). A trail represents a sequence of behaviors of the system and is determined by an execution order of the system. For example, with the system indicated in Figure 5.11, we have the following trail:

```
[<2, d_step{sch_api_self(terminate); cnt--}, 1>,
 <1, d_step{sch_api_self(terminate); cnt--}, 0>]
```

Definition 5.6 (Test sequence): A test sequence $[st_0, .., st_n]$ derived from a trail $[\langle \Sigma_0, a_0, \Sigma_1 \rangle, \langle \Sigma_1, a_1, \Sigma_2 \rangle, .., \langle \Sigma_{n-1}, a_{n-1}, \Sigma_n \rangle]$, $n \in \mathbb{N}$, where $st_j = \langle \langle i_{t_j}, o_{t_j} \rangle, genCode(t_j) \rangle \in \mathcal{GT}$, $j \leq n$ is a generation step corresponding to transition $t_j = \langle \Sigma_j, a_j, \Sigma_{j+1} \rangle$.

For instance, the test sequence derived from the trail above is:

```
[<2, "Current process count = 2
     Expected process count = 1", 1>,
 <1, "Current process count = 1
     Expected process count = 0", 0>]
```

Let $\mathcal{TS}$ be the set of test sequences. We define function $genTest : \mathcal{TS} \rightarrow String$ to combine the codes generated from the generation steps in a test sequence.

Definition 5.7 (Test): A test (test case, test program) derived from a test sequence is a string $test = genTest(ts)$, where $ts \in \mathcal{TS}$.

5.3.3 Generating the Tests

This section introduces the method to generate the tests following the search on the state space and following the executions of the system that satisfy a corresponding property.

Test generation following the search. We introduce an algorithm to generate the tests following the search using the scheduling strategy and the test specification. To deal with the scheduling strategy, the behaviors of the scheduler are also considered. The algorithm is shown in Algorithm 4, which is an extension of the algorithm introduced in Chapter 4. Our idea for the generation is that the codes generated corresponding to the behaviors of the system are recorded in the search and are used to generate the tests.

To generate the tests, the data structures corresponding to a *test sequence* ($\mathcal{TS}$) and the result of the *test generation* ($\mathcal{TG}$) are used ($\mathcal{TS}$ is an ordered set of *generation step*s and $\mathcal{TG}$ is an unordered set of strings representing the result of the test generation). We introduce the following functions.

- Function *Add_step* (line 17, 30) is used to add a *generation step* to the *test sequence*.
- Function *Remove_last_step* (line 42) is for removing the last *generation step* from the *test sequence*.
- Function *Add_test* (line 23, 36) is used to add a test derived from the *test sequence* (*TS*) to the set of tests ($\mathcal{TG}$). This function is called when the search reaches to a visited state.

Algorithm 4 *Test generation algorithm following the search with scheduling policy*

1: Input: Σ_0 ▷ initial state
2: Output: $\mathcal{TG}$ ▷ test generation
3: **procedure** START
4: Stack: $\mathcal{ST} = \emptyset$
5: State space: $\mathcal{SP} = \emptyset$
6: Test sequence: $\mathcal{TS} = \emptyset$
7: Test generation: $\mathcal{TG} = \emptyset$
8: $Push(\mathcal{ST}, \Sigma_0)$
9: $Add\_state(\mathcal{SP}, \Sigma_0)$
10: SEARCH
11: **end procedure**
12: **procedure** SEARCH
13: $\Sigma = Top(\mathcal{ST})$
14: P = SCH_SELECT(Σ)
15: **if** $P == \emptyset$ **then**
16: Σ^t = SCH_CLOCK(Σ)
17: $Add\_step(\mathcal{TS}, \langle\langle getIO(\Sigma), getIO(\Sigma^t)\rangle, genCode(\langle\Sigma, clock, \Sigma^t\rangle)\rangle)$
18: **if** $Contains(\mathcal{SP}, \Sigma^t) == false$ **then**
19: $Push(\mathcal{ST}, \Sigma^t)$
20: $Add\_state(\mathcal{SP}, \Sigma^t)$
21: SEARCH
22: **else**
23: $Add\_test(\mathcal{TG}, genTest(\mathcal{TS}))$
24: **end if**
25: **else**
26: **for** $p \in P$ **do**
27: **for** $a \in p.L_p$ **do**
28: Σ_a =SCH_TAKE(a, Σ)
29: Σ_a^t =SCH_CLOCK(Σ_a)
30: $Add\_step(\mathcal{TS}, \langle\langle getIO(\Sigma), getIO(\Sigma_a^t)\rangle, genCode(\langle\Sigma, a, \Sigma_a^t\rangle)\rangle)$
31: **if** $Contains(\mathcal{SP}, \Sigma_a^t) == false$ **then**
32: $Push(\mathcal{ST}, \Sigma_a^t)$
33: $Add\_state(\mathcal{SP}, \Sigma_a^t)$
34: SEARCH
35: **else**
36: $Add\_test(\mathcal{TG}, genTest(\mathcal{TS}))$
37: **end if**
38: **end for**
39: **end for**
40: **end if**
41: $Pop(\mathcal{ST})$
42: $Remove\_last\_step(\mathcal{TS})$
43: **end procedure**

We note that, if the error is determined at the current state, e.g. taking an action that leads to the violation of a property, we can generate a test from the current test sequence. That fact is not shown in this algorithm.

An example for generating test cases with FIFO scheduling policy is depicted in Figure 5.13. With this example, the behaviors of the processes are specified in *process program* (a); the specification for the test generation is shown in *test generation description* (b); a system execution is shown in *execution order* (c); the code generated corresponding to this execution by mapping the behaviors of the system is shown in *code generated* (d).

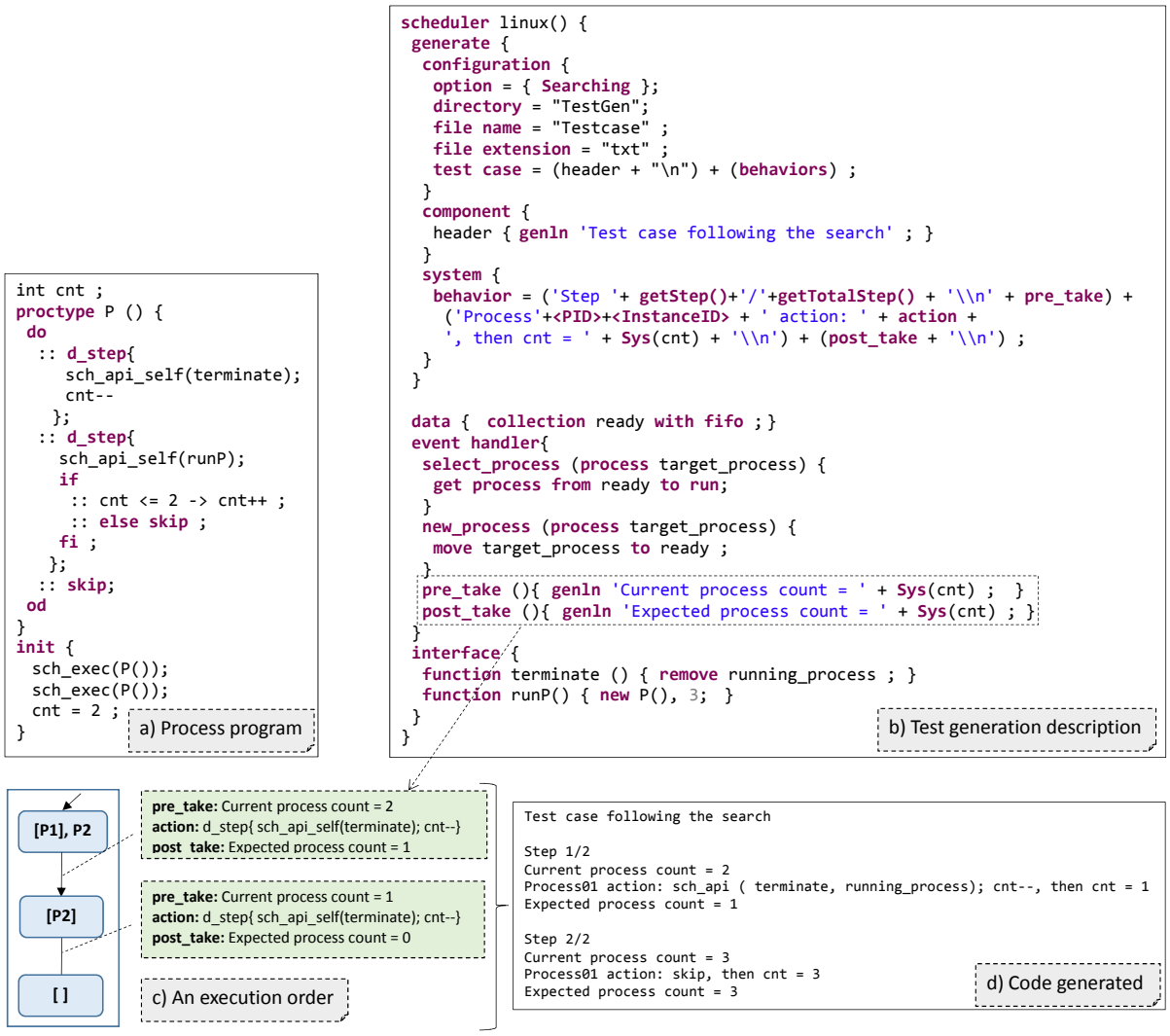


Figure 5.13: An example for generating a test case

With this example, at the initial time, the system has two process `P1` and `P2`, and process `P1` runs first. The execution order of the processes is indicated below.

```
P1.d_step{sch_api_self(terminate); cnt--}
P2.d_step{sch_api_self(terminate); cnt--}
```

The corresponding code generated following this execution is as follows. This code produces a test case following the execution above.

```
Test case following the search
Step 1/2
Current process count = 2
Process01 action: sch_api(terminate,running_process);cnt--,then cnt=1
Expected process count = 1
Step 2/2
Current process count = 1
Process01 action: skip, then cnt = 0
Expected process count = 0
```

Test generation following a property. To generate a test following a property, we do the two steps:

1. Finding the executions (trails) that satisfy the input formula, and
2. Generating the test corresponding to each trail found in the first step.

The property is specified as a CTL/RTCTL formula (an example is shown in Figure 5.14). To find the trails, we label the state graph (see Chapter 4 for more details) to indicate whether the corresponding formula is satisfied or not. If the CTL/RTCTL formula is satisfied, we explore the state graph again to find the trails that satisfy this formula. We can easily see that each trail corresponds to a test sequence and the test generated is now determined using this test sequence (that means we adopt the function *genTest* for this test sequence to generate the tests).

```
scheduler linux() {
 generate {
  configuration {
   option = { Property };
   …
  }
  …
 }
 data {
  collection ready with fifo;
 }
 event handler{
  …
 }
 interface {
  …
 }
}

verify {
 AF (Sys(cnt) == 0)
}
```

Figure 5.14: The specification of a property

5.4 Summary

We have introduced a method to test the scheduling strategy specified in the DSL with the implementation of a scheduler in a real OS. The purpose is to check the correspondence between the policy and the behaviors of a real scheduler. In this work, we apply MBT techniques to generate the tests. The summary of this approach for testing the scheduling strategy is depicted in Figure 5.15. Our method is as follows.

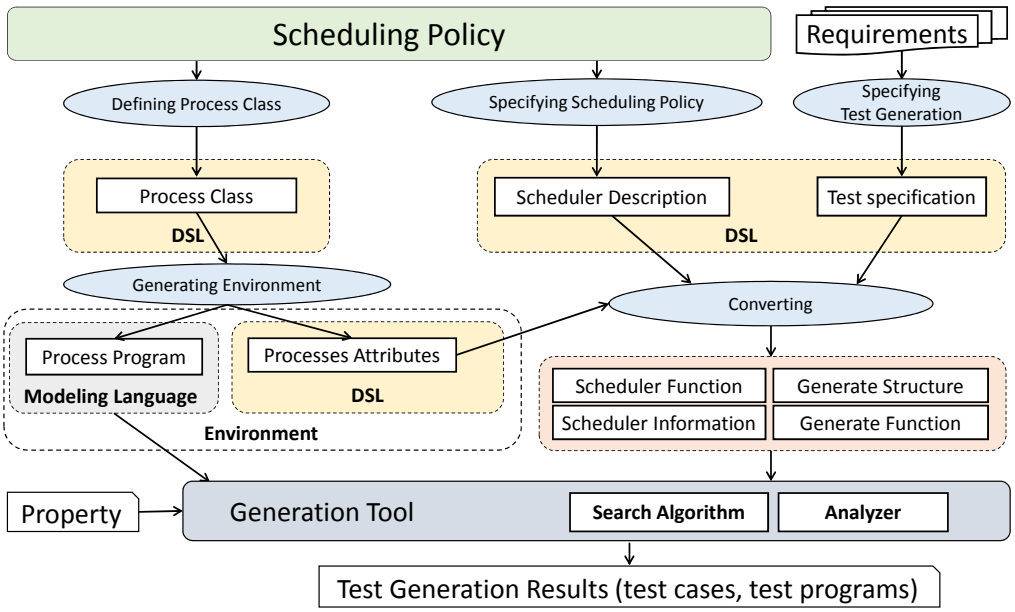


Figure 5.15: Summary of the testing approach

- First, we prepare the environments needed for performing the scheduling strategy. This is done by defining the *process class* in our DSL and generating the environments from this definition.
- Second, we define the mapping between the behaviors of the system and the code generated in the DSL. We propose an algorithm to handle the test generation. With this algorithm, we can generate the tests by searching the state space.
- Third, we perform the tests to check the scheduling strategy implemented in a real system.

We give a case study to test the FIFO real-time scheduling policy of Linux OS. The readers can refer to Chapter 7 for the detail of the description of the tests in the DSL with the approach to testing the scheduling strategy specified in the DSL and the implementation of the scheduler in a real OS.

In the next chapter, we will introduce our implementation for the approach. Our tool contains three main parts: a converter for translating the specification in the DSL into the executable code, a model checker, and an analyzer to verify and analyze the behaviors of the system to export the results.

Chapter 6

Implementation

In this chapter, we introduce the implementation of our method. We used the SpinJa model checker as the back-end for our framework. The language for scheduling was implemented in Xtext framework. We also introduce the approach to generate the information for the scheduling tasks. With this information, our tool can perform the search and explore the system states.

6.1 SSpinJa Tool

We designed a framework and implemented a tool named SSpinJa[1] for verifying and analyzing systems with the scheduling strategies. Our design is depicted in Figure 6.1. In our work, we extended SpinJa for our tool and used Promela as the base language for the processes. The DSL is implemented in Xtext framework as an external DSL with its own syntax (see Chapter 3).

In our implementation, we adopt the compilation approach to prepare all necessary information beforehand. To realize the new API functions, we updated both of the parser and the compiler of SpinJa tool. The input of our tool is the *process program*, which is then compiled into a model in Java. This model links to the libraries of SSpinJa for the verification.

To generate the information necessary from the scheduling strategy, we built a converter under Xtext framework. The information includes:

1. *scheduler model*, which contains *scheduling information* for performing the scheduling tasks,
2. *test information* for the test generation to test the scheduling strategy, and
3. *analyzing information*, which is used for analyzing the behaviors of the system.

We also implemented an algorithm, which uses the scheduling information generated from the policy in the DSL, and an analyzer in our tool.

6.2 Generating Information

The scheduling strategy specified in the DSL is used for generating necessary information for scheduling tasks and the test generation is converted from the test specification

[1] SSpinJa stands for 'Scheduling SpinJa'.

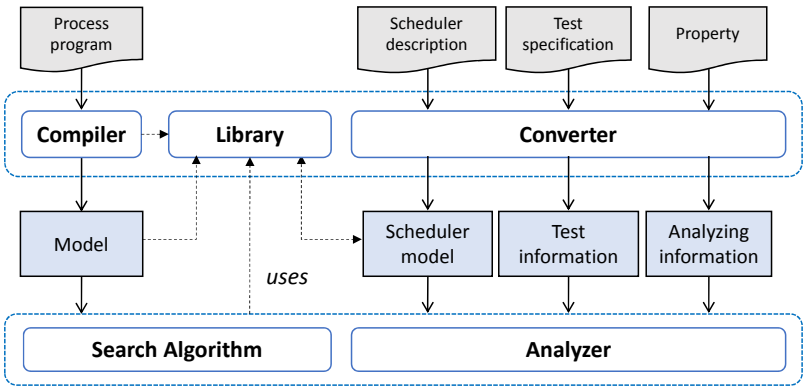


Figure 6.1: The architecture of the framework

including a) the processes implementation, b) the collections implementation with ordering methods, c) the scheduler implementation, and d) test information. This approach is depicted in Figure 6.2.

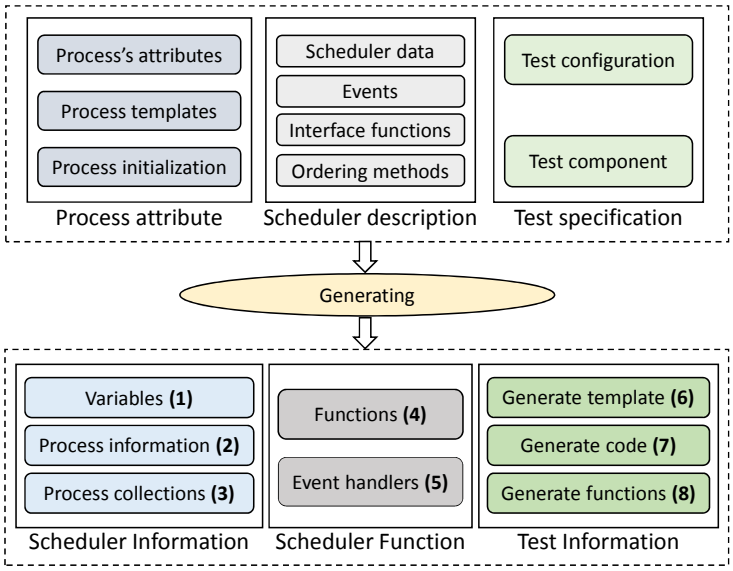


Figure 6.2: Generating the scheduling information

- The generation for *scheduler information* and *scheduler function* for the scheduling policy have been explained in Section 4.2 of Chapter 4.
- The *test component* and the *test configuration* in the *test specification* define a) the *template* (6) of the tests, b) the structure of the *code* (7) to be generated and c) the corresponding *functions* (8) for capturing the behaviors of the system to generate the code for each component of the tests.

An example of the generation is demonstrated in Section 6.3.

6.3 Verifying and Analyzing Systems with SSpinJa

The input of our tool includes 3 files (*process program* (*.pml), *process attribute* (*.proc) and *scheduling policy*(*.sch)). The corresponding property can be specified in the *process program* and/or in the *scheduling policy*. To perform the verification and analysis with SSpinJa tool, we do these steps.

- *First*, we convert the description of the scheduling strategy (including the *process attributes* and the *scheduler description*) into the information necessary for performing the scheduling tasks using the following command.

  ```
  java -jar schedulerDSL.jar -proc pri.proc -sch pri.sch
  ```

  where `schedulerDSL.jar` is the SSpinJa library file, with this example, `pri.proc` and `pri.sch` is the *priority* policy in the DSL (as introduced in Figure 3.2 of Chapter 3). As a result, the following files are generated.

  ```
  schedulerinfo.dat
  CTLFormula.java
  ProcessCollection.java
  ProcessCollection_priorityOrder.java
  ProcessCollectionBase.java
  ProcessSet.java
  RunningSet.java
  SchedulerObject.java
  SchedulerObject_Priority.java
  SchedulerProcess.java
  SchedulerProcess_Priority.java
  SchedulerProcessBase.java
  SchedulerState.java
  SortedProcessCollectionBase.java
  StaticProperty.java
  StaticProperty_Priority.java
  ```

  These *java* files implement the following things:

  - the processes of the system (files: *SchedulerProcess*, *SchedulerProcess_Priority*, *SchedulerProcessBase*, *StaticProperty* and *StaticProperty_Priority*) to specify the information of the processes (e.g. the attributes of the processes),
  - the process collections (files: *ProcessCollection*, *ProcessCollection_priorityOrder*, *ProcessCollectionBase*, *ProcessSet*, *RunningSet*, and *SortedProcessCollectionBase*) to define the collections with the ordering methods used by these collections,
  - the scheduler (files: *SchedulerObject* and *SchedulerObject_Priority*) to implement the behaviors of the scheduler (i.e. handling the scheduling tasks),
  - the analyzing information (files: *CTLFormula* and *SchedulerState*) for the analysis behaviors of the system ; these files express the property to be checked and the information of the system state.

The behaviors of the scheduler (i.e. handling the *scheduling event*s) are implemented in *SchedulerObject* file. An example for generating the code of this Java file from the scheduling strategy is shown in Appendix B. The main functionalities of the scheduler are defined and represented by the following the data structures and a set of methods defined in the *SchedulerObject* as follows.

- The variables (e.g. clock variables) and data structures (e.g. process collections);
- The methods to handle the scheduling tasks (e.g. the events `select_process`, `new_process`, `preTake`, `postTake`, etc.);
- Interface methods (e.g. `sch_api`, `sch_get`) for the interaction between the processes and the scheduler;
- The methods for handling the time managed by the scheduler (e.g. `inc_time`, `dec_time`, `time_out`, etc.);
- Encoding methods (e.g. `encode`, `decode`) for storing the states of the scheduler;
- Analysis methods (e.g. `schedulerCheck`, `stateCheck`, etc.) for analysis the behaviors of the system ;
- Other utilities methods (e.g. `getInstance`, `isTime`, etc.) used by the scheduler.

We note that these methods (functions) are used by the search algorithm to search the system states. The states of the system to be visited are determined by this algorithm based on the scheduling strategy specified in the DSL.

- *Second*, the *process program* (*.pml) in Promela is compiled into the process model in Java using the following command. The result of this step is the model of the processes in Java (`SchedulerPanModel.java`).

  ```
  java -cp sspinja.jar sspinja.Compile example.pml
  ```

  where `sspinja.jar` is the library file and `example.pml` is a *process program* in Promela.

- *Third*, we compile the process model in Java into Java bytecode. This step automatically compiles and links the `SchedulerPanModel.java` with the files generated from the scheduling strategy description in step 1.

  ```
  javac -cp sspinja.jar sspinja/SchedulerPanModel.java
  ```

- *Fourth*, we perform this command to do the verification and the analysis

  ```
  java -cp sspinja.jar;. sspinja.SchedulerPanModel
  ```

Our tool will indicate the verification result, which contains the information: the error (if exist, with the counterexample), number of states, memory usage, and the time for verifying as shown in Figure 6.3. For verifying the corresponding property, our tool will indicate the satisfaction and the witness (as depicted in Figure 6.4).

```
...
assertion violated (balance >= 0)
0.(proc 0 trans 59): run VerificationCase2_0()
1.(proc 1 trans 55): printf("config3");; balance = 10; amount = 15; run login_deposit_0();
run login_withdraw_0(); run logout_0(); run deposit_0(); run withdraw_0()
2.(proc 3 trans 62): (status == 0); status = 2
3.(proc 3 trans 62): (status == 0); status = 2
4.(proc 3 trans 62): (status == 0); status = 2
5.(proc 3 trans 62): (status == 0); status = 2
6.(proc 3 trans 62): (status == 0); status = 2
7.(proc 3 trans 62): (status == 0); status = 2
8.(proc 3 trans 62): (status == 0); status = 2
9.(proc 3 trans 62): (status == 0); status = 2
10.(proc 3 trans 62): (status == 0); status = 2
11.(proc 6 trans 65): (status == 2); _bwithdraw0_0 = false; if; balance = (balance -
amount); status = 0; if
-----------------------------------------------

State-vector 49 byte, depth reached 11, errors: 21
      89 states, stored
      68 states, matched
     157 transitions (= stored+matched)
       0 atomic steps
8.00584 memory usage (Mbyte)

sspinja: elapsed time 22.00 miliseconds
sspinja: rate     4045 states/second
8.75444 real memory usage (Mbyte)
```

Figure 6.3: Result of the verification (counterexamples)

6.4 Summary

We have introduced the implementation of our method in the tool named SSpinJa. We extended SpinJa in our approach. SpinJa is written in Java and follows the object-oriented design principle; thus, it is easy to extend. In addition, we adopted the compilation approach to preparing the information necessary beforehand. In fact, the DSL in our framework was implemented in Xtext framework to prepare the information for performing the scheduling tasks (handling the scheduling events) and for the verification (analyzing the behaviors of the system). We also implemented a new algorithm in the tool using this information to search the system states. With the states visited, we can verify the behaviors of the system with the scheduling strategies. The evaluation of our method is considered in the next chapter.

```
...
+ Check system (start state: 705970) with: AG<=2 (balance >= 5): Satisfied
-----------------------------------------------
AG<=2 (balance >= 5)
0. (proc 0 trans 43): run VerificationCase0_0()
1. (proc 1 trans 41): printf("config1");; balance = 10; amount = 5; run login_deposit_0();
run login_withdraw_0(); run logout_0(); run deposit_0(); run withdraw_0()
2. (proc 3 trans 46): (status == 0); status = 2
3. (proc 6 trans 49): (status == 2); _bwithdraw0_0 = false; if; balance = (balance -
amount); status = 0; if
-----------------------------------------------
-----------------------------------------------
AG<=2 (balance >= 5)
0. (proc 0 trans 43): run VerificationCase0_0()
1. (proc 1 trans 41): printf("config1");; balance = 10; amount = 5; run login_deposit_0();
run login_withdraw_0(); run logout_0(); run deposit_0(); run withdraw_0()
2. (proc 3 trans 46): (status == 0); status = 2
3. (proc 4 trans 47): (status != 0); status = 0
-----------------------------------------------
-----------------------------------------------
AG<=2 (balance >= 5)
0. (proc 0 trans 43): run VerificationCase0_0()
1. (proc 1 trans 41): printf("config1");; balance = 10; amount = 5; run login_deposit_0();
run login_withdraw_0(); run logout_0(); run deposit_0(); run withdraw_0()
2. (proc 2 trans 45): (status == 0); status = 1
3. (proc 5 trans 48): (status == 1); balance = (balance + amount); status = 0
-----------------------------------------------
-----------------------------------------------
AG<=2 (balance >= 5)
0. (proc 0 trans 43): run VerificationCase0_0()
1. (proc 1 trans 41): printf("config1");; balance = 10; amount = 5; run login_deposit_0();
run login_withdraw_0(); run logout_0(); run deposit_0(); run withdraw_0()
2. (proc 2 trans 45): (status == 0); status = 1
3. (proc 4 trans 47): (status != 0); status = 0
-----------------------------------------------

State-vector 49 byte, depth reached 11, errors: 5
      39 states, stored
      30 states, matched
      69 transitions (= stored+matched)
       0 atomic steps
8.00270 memory usage (Mbyte)

sspinja: elapsed time 54.00 miliseconds
sspinja: rate      722 states/second
8.74844 real memory usage (Mbyte)
```

Figure 6.4: Result of the analysis (witnesses)

Chapter 7

Case Studies

To evaluate the method, we conducted experiments on systems ranging from simple to the real ones (e.g. the dining philosopher problem, benchmark for explicit model checker [66], OSEK/VDX OS [62], and Linux OS with real-time scheduling policies). In this chapter, we introduce the case studies and presents the experiment results[1]. Our experiments were carried out on 3.4 GHz CPU Intel Core i7 with 32G RAM.

7.1 Verifying Systems with Scheduling Policies

7.1.1 Dining Philosopher Problem

We considered the dining philosopher problem with *round-robin* (RR) and *priority* (FP) policies. Each philosopher was modeled as a process in the modeling language and each behavior was represented as an atomic action. The numbers of philosophers considered were 2, 4, 8, 16, and 32. With the FP policy, we assigned different priorities to the processes. For the execution of each process with RR policy, we set the time slice to 3 indicating that each philosopher can take 3 actions in his turn. Only four philosophers are considered to check the absence of starvation. The results of the experiments are listed in Table 7.1.

Table 7.1: Deadlock and starvation verification results

Scheduling policy	**Deadlock**	**Starvation**
Without scheduler	Yes	-
RR, time slice = 3	No	No
FP	No	Yes

Without using any scheduling policy, deadlocks occurred; therefore, we did not consider the starvation. In addition, within 60 seconds, the search was incomplete with the number of philosophers being equal to 16 or 32. With FP policy, the starvation occurred although the deadlock was resolved. With RR policy, both deadlock and starvation were absent. The detailed results are shown in Table 7.2 with the number of philosophers (N), the number of states (S), time (T) in seconds, and memory usage (M) in MB. In this table, T.O. means timeout (incomplete within 60 seconds); this happened without using the scheduling strategy.

[1]The results were first introduced in [83, 84]

Table 7.2: Dining philosopher problem verification results

N	Without using scheduling policy			RR			FP		
	S	T	M	S	T	M	S	T	M
2	9	0.01	8.64	11	0.01	17.0369	4	0.01	17.0327
4	115	0.03	8.66	17	0.02	17.0343	4	0.01	17.0339
8	12319	18.79	9.58	33	0.03	17.0403	4	0.01	17.0368
16	-	T.O.	-	65	0.05	17.0588	4	0.01	17.0478
32	-	T.O.	-	129	0.1	17.1211	4	0.01	17.0515

7.1.2 Synchronization in OSEK/VDX OS

We considered a problem for scheduling resources in synchronization mechanisms. The problem indicates that a lower-priority process can delay the execution of the higher one in a system using priority scheduling policy. For instance, let a system with three tasks/processes T_1, T_2 and T_3 (as depicted in Figure 7.1.a) with preemptive policy. Semaphore S is occupied by process T_3, which has the lowest priority and is currently running. Process T_1 preempts this process and then requests the semaphore. However, because S is already used by T_3, T_1 is denied and then enters the waiting state. Now T_2 is executed. We can see that the highest priority process T_1 can only run until the others have been terminated and the semaphore S released. Actually, T_2 delays process T_1 although it does not use S.

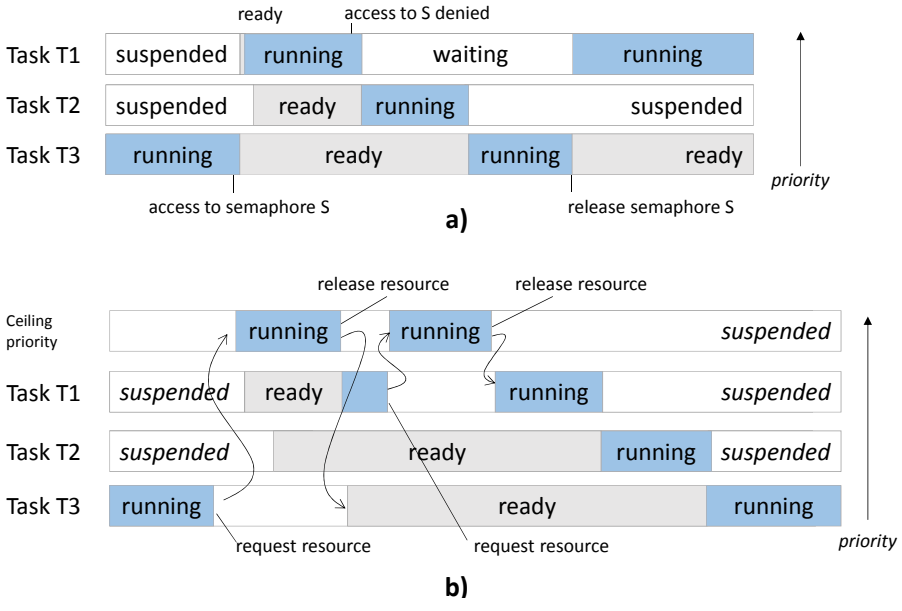


Figure 7.1: Synchronization mechanism problem

To overcome this problem, OSEK OS applies the Priority Ceiling Protocol; the method is as follows. The priority of a process will be raised to the value of the priority of the resource when it occupies this resource. The priority of the process will be reset after it releases the resource (as depicted in Figure 7.1.b). We note that all processes access the resource have lower priority than this value.

We demonstrated the mechanism by describing the OSEK scheduler in our DSL. The process now has two attributes (*PRIORITY* and *CEILING_PRIORITY*) corresponding to the static priority and the dynamic priority. When the process accesses to a resource, its dynamic priority is changed. We define two interface functions correspond to the service APIs *GetResource* and *ReleaseResouce*. These functions are to change the priority following that of the resource. To execute and terminate a process, we defined two other functions `ActivateTask` and `TerminateTask` for the two service APIs. The system in the example above was modeled with three processes (from `t1` to `t3`) (as depicted in Figure 7.2). We set the value `true` for the attribute *AUTOSTART* of `t3` to make it execute at the starting time; `t1` and `t2` will be in the suspended state.

```
int x = 0 ;
proctype t1() {
  sch_api_self(ActivateTask, t2) ;
  sch_api_self(GetResource, 1); //resource S
  sch_api_self(ReleaseResource, 1) ;
  assert (x == 0) ;
  x = 1 ;
  sch_api_self(TerminateTask) ;
}

proctype t2() {
  assert (x == 1) ;
  x = 2 ;
  sch_api_self(TerminateTask) ;
}

proctype t3() {
  sch_api_self(GetResource, 1);
  sch_api_self(ActivateTask, t1) ;
  sch_api_self(ReleaseResource, 1) ;
  assert (x == 2) ;
  x = 3 ;
  sch_api_self(TerminateTask) ;
}

init {
  sch_exec( t1()) ;
  sch_exec( t2()) ;
  sch_exec( t3()) ;
}
```

Figure 7.2: Modeling the example for synchronization problem

We verified the execution order of the processes by conducting the experiments with and without using the protocol. The results are indicated in Table 7.3. The violation of assertion corresponding to the assert statement `(x==1)` in process `t2` was found. This indicates that `t1` cannot terminates before `t2` ends.

Table 7.3: Priority ceiling protocol verification result

	State	**Time** (s)	**Memory (MB.)**	**Error**
Not using protocol	5	0.01	17.0751	Yes
Using protocol	16	0.01	17.0688	No

7.1.3 Linux Scheduling Policies

The scheduling strategies in Linux are based on the priorities of the processes. Linux supports three types of strategies for a non-real-time process; the priority now is assigned with 0. These policies are `SCHED_OTHER` (for time-sharing executions), `SCHED_BATCH` (for batch executions), and `SCHED_IDLE` (for background processes). With the real-time policies, the priority of a process is assigned from 1 to 99; Linux uses `SCHED_FIFO` and `SCHED_RR` policies for managing the executions. The difference between these two policies is that with the processes that have the same priority, `SCHED_RR` uses the round-robin with a certain time slice for the processes; with `SCHED_FIFO`, each process can run until it releases the processor.

We applied our method to verifying systems on Linux OS. We used a system with two process (`P` and `Q`) described in Chapter 3 (as shown in Figure 7.3) for the experiments. For the verification, we applied several scenarios using the strategies with different priorities values. We considered the properties expected and checked them with the strategies. In these experiments, process `P` was run before process `Q` did. We also conducted the experiments with the related policies (RR and FIFO) and compared the experiment results.

```
int a, b;

proctype P() {
next:
 if
  :: (a+b) < 100000 -> a ++; goto next;
  :: else -> sch_api_self(terminate)
 fi;
}
proctype Q() {
next:
 if
  :: (a+b) < 100000 -> b ++; goto next;
  :: else -> sch_api_self(terminate)
 fi;
}

init {
 sch_exec(P());
 sch_exec(Q());
}
```

Figure 7.3: A system with two processes

We also implemented the programs corresponding to these systems in Linux. The results of running these programs were matched with the results of the verification. Table 7.4 indicates these results with the number of states (S), time (T) in seconds, and memory usage (M) in MB. Each scenario indicates the strategy and the relation of the priorities of the processes; for instance, "`OTHER, P.pri > Q.pri`" means we used `SCHED_OTHER` policy and the priority of `P` was higher than `Q`. Table 7.5 depicts the results of the verification with the related strategies (RR: round-robin, FIFO: first-in-first-out). The execution results for the program in Linux system are shown in Table 7.6.

In every case, with the scheduling strategies that are described, the expected properties hold. These results also match with the results of the execution of the programs in Linux. However, in some cases, the policies related to the Linux policies show that the properties did not hold.

Table 7.4: Linux tasks verification results

No.	Scenario	Property	Result	S	T	M
1	`OTHER, P.pri == Q.pri`	$(a > 0) \wedge (b > 0)$	holds	240627	0.41	52.15
2	`FIFO, P.pri > Q.pri`	$(a > 0) \wedge (b == 0)$	holds	300008	0.42	57.40
3	`FIFO, P.pri < Q.pri`	$(a > 0) \wedge (b > 0)$	holds	200108	0.29	48.31
4	`FIFO, P.pri == Q.pri`	$(a > 0) \wedge (b == 0)$	holds	300008	0.46	57.42
5	`RR, P.pri > Q.pri`	$(a > 0) \wedge (b == 0)$	holds	300008	0.49	57.40
6	`RR, P.pri < Q.pri`	$(a > 0) \wedge (b > 0)$	holds	240617	0.46	52.15

Table 7.5: Linux tasks verification with related policies results

No.	Scenario	Policy	Property	Result	S	T	M
1	`OTHER, P.pri == Q.pri`	RR	$(a > 0) \wedge (b > 0)$	holds	240616	0.4	50.34
2	`FIFO, P.pri > Q.pri`	FIFO	$(a > 0) \wedge (b == 0)$	holds	300008	0.42	52.85
3	`FIFO, P.pri < Q.pri`	FIFO	$(a > 0) \wedge (b > 0)$	not hold	300003	0.33	53.15
4	`FIFO, P.pri == Q.pri`	FIFO	$(a > 0) \wedge (b == 0)$	holds	300008	0.42	52.85
5	`RR, P.pri > Q.pri`	RR	$(a > 0) \wedge (b == 0)$	not hold	240603	0.35	51.64
6	`RR, P.pri < Q.pri`	RR	$(a > 0) \wedge (b > 0)$	holds	240616	0.4	50.34

7.2 Analyzing the Behaviors of the System

We analyzed the behaviors of real-time systems quantitatively and qualitatively with considering the schedulability problem using different policies and different configurations of a system.

7.2.1 Different Scheduling Policies

For real-time systems, each task/process has several attributes, such as best-case execution time (BCET), worst-case execution time (WCET), the time between task releases (PERIOD), deadline (DEADLINE), and priority (PRIORITY). The schedulability problem is specified as all processes in the system will not violate their deadlines.

In the first experiment, we considered a system with four different processes. Table 7.7 shows the configurations of these processes. Here, the BCET was equal to WCET. We used three scheduling policies for the evaluation: *fixed-priority* (FP), *first-in-first-out* (FIFO) and *earliest-deadline-first* (EDF). The deadline violation for this system was analyzed.

We also conducted the experiments with the framework introduced in [28]. This framework[2] uses UPPAAL and follows the model-based approach for the schedulability analysis. This framework aims at analyzing the resource sharing problem with real-time behaviors and a scheduling policy for each resource. In this experiment, the scheduler was considered as a system resource. The attributes of the processes were set up as indicated above. We compared the results using this framework with that resulted using our approach. Table 7.8 shows the evaluation results with time (T) in seconds and memory usage (M) in MB. In this table, "satisfied" means there is no deadline; "unsatisfied" means the deadline violation occurs; "may not be satisfied" means we do not know whether the deadline violation happens or not.

[2]The framework is taken from `http://www.uppaal.org/`

Table 7.6: Execution results in Linux

No.	Scenario	Property	Result	Linux execution result
1	`OTHER, P.pri == Q.pri`	$(a > 0) \wedge (b > 0)$	holds	(a, b) = (99976, 25)
2	`FIFO, P.pri > Q.pri`	$(a > 0) \wedge (b == 0)$	holds	(a, b) = (100000, 0)
3	`FIFO, P.pri < Q.pri`	$(a > 0) \wedge (b > 0)$	holds	(a, b) = (42837, 57164)
4	`FIFO, P.pri == Q.pri`	$(a > 0) \wedge (b == 0)$	holds	(a, b) = (100000, 0)
5	`RR, P.pri > Q.pri`	$(a > 0) \wedge (b == 0)$	holds	(a, b) = (100000, 0)
6	`RR, P.pri < Q.pri`	$(a > 0) \wedge (b > 0)$	holds	(a, b) = (54020, 45981)

Table 7.7: The configuration of the processes

Process	PERIOD	Initial OFFSET	TIME	PRIORITY	DEADLINE
t1	20	6	5	2	16
t2	20	9	5	3	11
t3	20	11	5	2	8
t4	20	10	5	5	20

In the next experiment, different numbers of the processes, which had the same configuration, were used to analyze the performance. The configuration of the processes is (PERIOD, BCET, WECT, DEADLINE, PRIORITY) = (20, 5, 5, 20, 1). The numbers of processes considered in this experiment were 2, 3, 4, and 5. Table 7.9 shows the analysis results with the average time (T_a) in seconds and the average memory usage (M_a) in MB. Here, "satisfied" means that there is no deadline violation with all of the scheduling strategies; "may not be satisfied" means that we cannot know the deadline violation with all of the scheduling strategies; "NA." means "not available" (that occurs when the system reaches the maximum number of processes and our tool cannot determine the result).

7.2.2 Different Configurations

We considered the system containing two processes (`t1` and `t2`) with priority scheduling policy (as depicted in Figure 7.4). For this experiment, three configurations based on the priorities of these two processes were used. We indicated the analysis properties that were `AG(a > b)` and $\texttt{AG}^{\leq 3}\texttt{(a > b)}$ meaning that the value of variable `a` is always greater than that of `b`, and the value of variable `a` is always greater than that of `b` within 3 time units. Statements *checkpoint* with label `exec_t1` in the *select_process* event handler and *checkpoint* with label `end_t2` in the *interface function* named *terminate* were set (as shown in Figure 7.4.c) to realize the executions of these statements (`exec_t1` is for checking the execution of process `t1` and `end_t2` is for checking the termination of process `t2`).

Table 7.10 shows the analysis results for these properties. Property $\texttt{AG}^{\leq 3}\texttt{(a > b)}$ always holds, while property `AG(a > b)` only holds if the priority of `t1` is greater than that of `t2`. The analysis results corresponding to the *checkpoint* statements `exec_t1` and `end_t2` (*min, max*) are different according to each configuration.

Table 7.8: Analysis results with four processes

Policy	Expected	Scheduling framework in UPPAAL			SSpinJa		
		T	*M*	*Result*	*T*	*M*	*Result*
FP	unsatisfied	0.002	41.46	may not be satisfied	0.06	19.692	unsatisfied
FIFO	unsatisfied	0.002	41.176	may not be satisfied	0.02	19.675	unsatisfied
EDF	satisfied	0.01	41.8	satisfied	0.03	19.709	satisfied

Table 7.9: Analysis results with different number of processes

N	Expected	Scheduling framework in UPPAAL			SSpinJa		
		T_a	M_a	*Result*	T_a	M_a	*Result*
2	satisfied	0.004	40.94	satisfied	0.02	19.6891	satisfied
3	satisfied	0.015	40.98	may not be satisfied	0.03	19.6906	satisfied
4	satisfied	0.141	41.29	may not be satisfied	0.063	19.6984	satisfied
5	unsatisfied	1.064	43.54	may not be satisfied	0.203	25.2627	NA.

Table 7.10: Analysis results with priority policy

Configuration	T	M	AG($a > b$)	AG$^{\leq 3}$($a > b$)	*exec_t*1		*end_t*2	
					min	*max*	*min*	*max*
`t1.priority > t2.priority`	0.02	19.6704	satisfied	satisfied	0	0	10	10
`t1.priority == t2.priority`	0.02	19.6704	unsatisfied	satisfied	0	8	8	10
`t1.priority < t2.priority`	0.02	19.6764	unsatisfied	satisfied	8	8	8	8

7.3 Testing Scheduling Policies

In this experiment, we checked the correspondence between the policy specified in the DSL with the implementation of the real scheduler using the test generation approach based on the specifications the scheduling strategy in our DSL. Real-time FIFO scheduling policy on Linux OS was used in this experiment.

7.3.1 Preparing the Environments

We aimed at verifying the scheduling strategy in Linux OS. To prepare the behaviors of the processes and the configuration of the system for testing, we define the *process class* in our DSL[3] as depicted in Figure 7.5.

For checking the selection of the scheduler, the scheduling tasks are performed by the behaviors of the processes using the following methods a) `terminate` for terminating the current process and b) `runP` for executing another process. The result for generating the environment from the definition of the *process class* is represented in Table 7.11.

[3]see Chapter 5 for the details of the test generation

```
int a, b;

proctype t1() {
next:
 if
  :: (a+b) < 5 -> a++; goto next;
  :: else -> sch_api_self(terminate)
 fi;
}

proctype t2() {
next:
 if
  :: (a+b) < 5 -> b++; goto next;
  :: else -> sch_api_self(terminate)
 fi;
}

init {
 sch_exec(P());
 sch_exec(Q());
}
```

a) Process program

```
def process Priority {
 attribute{
  var byte priority;
 }
 proctype t1() {
  this.priority = 5;
 }
 proctype t2() {
  this.priority = 3;
 }
}
init {
 [{t1(), t2()}]
}
```

b) Process attribute

```
scheduler Priority () {
 data {
  collection ready using priorityOrder;
 }

 event handler {
  select_process (process p) {
   get process from ready to run;
   if (p.hasName("t1")) checkpoint exec_t1 ;
  }

  new_process (process target) {
   move target to ready;
   if (!running_process.isNull()) {
    if (target.priority > running_process.priority) {
     move running_process to ready;
    }
   }
  }
 }

 interface {
  function terminate(process target) {
   if (target.hasName("t2")) checkpoint end_t2;
   remove target;
  }
 }
}

comparator {
 variable {
  int x;
 }
 comparetype priorityOrder(process p_n, p_o) {
  x = p_n.priority - p_o.priority;
  if (x>0) return greater;
  else if (x==0) return equal;
   else return less;
 }
}

verify {
 AG<=3 (Sys(a) > Sys(b))
}
```

c) Scheduler description

Figure 7.4: A system with priority policy

7.3.2 Test Cases Generation

The specification of Linux scheduling strategies indicates that the process with higher priority will preempt the current process. Therefore, we only used the processes with the same priority to check the selection of the scheduler. With the real-time policies, the *nice* attribute of the processes is ignored. Thus, we used only one configuration for the testing. The processes with this configuration have the same priority.

In this experiment, we generated test cases corresponding to the behaviors of the processes defined in the *process program* with `FIFO` scheduling policy. The description of the policy with the test generation in the DSL is depicted in Figure 7.6.

The test generation was indicated following the search to cover all the states of the system. With this experiment, 14 test cases were generated. One of these tests is depicted in Figure 7.7. The results for the generation are listed in Table 7.12.

```
process linux {
 attribute {
  priority: type = byte, value = [0..2], default = 0; //static priority; 0: normal, 1..99 real time
  nice: type = int, value = [0..2], default = 0; //dynamic priority for SCHED_OTHER -20..19
  schtype: type = byte, value = [0,1,2], default = 0 ; //SCHED_OTHER, SCHED_FIFO, SCHED_RR
  preempt: type = byte, value = [0,1], default = 1 ;

  constraint {
   (schtype == 0) => (priority == 0) ;  //normal process
   (schtype > 0) => (nice == 0) ;
   (priority == 0) => (schtype == 0) ;
  }
 }

 behavior {
  constructor : P(byte priority = 0; byte nice = 0; byte schtype = 0) ;
  method : terminate () ;
  method : runP(byte priority; byte nice; byte schtype){
   nice: value = [0..1];
   priority: value = [0..2] ;
   schtype: value = [0..2];

   constraint {
    (schtype == 0) => (priority == 0) ;  //normal process
    (schtype > 0) => (nice == 0) ;
    (priority == 0) => (schtype == 0) ;
   }
  }
 }
}

configuration {
 init {
  [{P(?,?,?), P(?,?,?)}]
 }
}
```

Figure 7.5: Defining the process class

Table 7.11: Generating environment result

No. configurations	Time (s)
49	0.207

Table 7.12: Test case generation result

No. test cases	States	Memory (MB.)	Time (s)
14	16	21.119	0.03

7.3.3 Test Programs Generation

In the next experiment, we generated test programs following the scheduling specify in the DSL to check the correspondence between the description of the policy with the implementation in a real system. Linux Ubuntu version 12.04.5 with real-time FIFO policy was used in this experiment.

In this experiment, the selection of the scheduler for the execution was tested. Basing on the specification of Linux scheduling policy, there is a case that when a process arrives, it may preempt the current process if they have the same priority. That means

a) Process program

```
int cnt ;
proctype P () {
  do
    :: d_step{
        sch_api_self(terminate);
        cnt--
      };
    :: d_step{
       sch_api_self(runP);
       if
         :: cnt <= 2 -> cnt++ ;
         :: else skip ;
       fi ;
      };
    :: skip;
  od
}
init {
   sch_exec(P());
   sch_exec(P());
   cnt = 2 ;
}
```

b) Process attribute

```
def process linux_0  {
  proctype P () {}
}

init {
  [{P(),P()}]
}
```

c) Scheduler description with test generation specification

```
scheduler linux() {
  generate {
    configuration {
      option = { Searching };
      directory = "TestGen";
      file name = "Testcase" ;
      file extension = "txt" ;
      test case = (header + "\n") + (behaviors) ;
    }
    component {
      header {
        genln 'Test case following the search' ;
      }
    }
    system {
      behavior =('Step '+ getStep()+'/'+getTotalStep() + '\\n' + pre_take) +
        ('Process'+<PID>+<InstanceID> + ' action: ' + action +
        ', then cnt = ' + Sys(cnt) + '\\n') +
        (post_take + '\\n') ;
    }
  }

  data {
    collection ready with fifo ;
  }
  event handler{
    select_process (process target_process) {
      get process from ready to run;
    }
    new_process (process target_process) {
      move target_process to ready ;
    }
    pre_take (){
      genln 'Current process count = ' + Sys(cnt) ;
    }
    post_take (){
      genln 'Expected process count = ' + Sys(cnt) ;
    }
  }
  interface {
    function terminate () {
      remove running_process ;
    }
    function runP() {
      new P(), 3;
    }
  }
}
```

Figure 7.6: The specification of a test case generation

we can not determine which process will be executed next. In this experiment, only the case that the processes had the same priority was considered. Therefore, in testing, we created the processes with the same priority and checked which process was selected to run. For the design of the test programs, we used a variable named `pointpassed` to count the actions of the processes. After performing an action, we increased the value of `pointpassed` to indicate that an action is performed (the testing also passes 1 more step). A test program is passed (in testing) if all the actions are performed and the corresponding steps are passed.

To generate the test programs, we indicated the scheduling tasks in the *process program* (e.g. terminates itself or executes another process). The specification of the strategy and the test generation for this experiment is depicted in Figure 7.8. In comparison with the previous experiment (the test cases generation), the attributes of the processes, and the behaviors of the scheduler are kept. The specification for the test generation was changed. Firstly, the `configuration` part was updated to support the

```
Test case following the search
Step 1/2
Current process count = 2
Process01 action: sch_api ( terminate, running_process); cnt--, then cnt = 1
Expected process count = 1
Step 2/2
Current process count = 3
Process01 action: skip, then cnt = 3
Expected process count = 3
```

Figure 7.7: A test case

Table 7.13: Test program generation result

No. test programs	States	Memory (MB.)	Time (s)
29	16	21.2529	0.05

test generation. The `pre_take` and `post_take` events were updated to specify the code generated for checking the value of `pointpassed` variable. The structure of the programs was defined. The template of `init` component and the template of the processes were determined in the test specification. Two other components (`header` and `declare`) were added to construct the program structure. We note that the `declare` component is updated following the behaviors of the scheduler using the `genln` statement in the handler for the `new_process` event. In this experiment, 29 programs were generated. One of the programs is depicted in Figure 7.9. The detail result for the generation is represented in Table 7.13.

For executing the test programs, there is no specification indicating which process will be selected to run if a process with the same priority as the current one arrives at the system (the current process may be preempted). In this situation, we make the assumption that for an implementation of Linux OS, the scheduler can select any process among these processes to run. In fact, with our approach, a test only represents an execution of the system. Thus, it cannot handle all the possible executions of these process. To validate the implementation of the scheduling strategy, our method is based on executing a test program multiple times and checking the execution orders of its processes. With this experiment, we examined the value of `pointpassed`. The step indicated by the value of this variable was compared with the execution order of the processes. If the results are matched, the test is passed, otherwise, we execute the test program again with the limited times to run. Based on this idea, with the test programs generated, we wrote a test script to execute the tests and handle the results of the execution. The bound of times to try for each program was set to 1000000. In this experiment, all the tests were passed after 1388702 times to try in total 2371.407 seconds. The detail results are shown in Table 7.14.

In the next experiment, we only dealt with a limited number of behaviors of the processes. The number of behaviors considered was 2. Therefore, in the `verify` part of the specification, we specified a dummy property `AF AF (true)` to indicate that we need to check all the execution orders containing only 2 steps. We performed the generation again and 13 programs were generated (the result is represented in Table 7.15). For executing the test programs, we also set the number of times to try to 1000000. With this experiment, all the tests were passed after 20780 times to try in total 36.423 seconds. The detail result is represented in Table 7.16.

```
scheduler linux() {
   generate {
      configuration {
         option = { Searching };
         directory = "TestGen";
         file name = "Program" ;
         file extension = "c" ;
         test program = (header + "\n")+(declare + "\n")+(init  + "\n")+(processes);
      }
      component {
         header {
            genln '/* header */' ;
            genln '#define _GNU_SOURCE' ;
            …
            genln 'int pointpassed;';
            …
         }
         declare {
            genln '/* declaration */' ;
         }
         init {
            template = 'void init_thread() {\\n' + <behaviors> +'}';
            behavior = (select_process) ;
         }
      }
      process {
         template = 'void *process_'+<PID>+<InstanceID>+'() {\\n'+<behaviors>+'}\\n';

         behavior = (pre_take)+(select_process)+("//action: "+action)+
                (new_process)+(post_take+'\\n');
      }
   }

   data {
         collection ready with fifo;
   }
   event handler{
      select_process (process target_process) {
            get process from ready to run ;
      }
      new_process (process target_process) {
            genln   declare, 'void* process_' + target_process.getPID() +
            target_process.getInstanceID()+'() ;';
               …
            move target_process to ready ;
      }
      pre_take(){
         if (!running_process.isNull()) {
            genln '\t if (pointpassed == ' + getStep() + ' - 1) {' ;
            …
            genln '\t }' ;
         } else {
            genln '\t\t exit(0) ;' ;
         }
      }
   }
   interface {
         function terminate () {
               remove running_process ;
         }
         function runP() {
               new P(), 3;
         }
      }
}
```

Figure 7.8: The specification of a test program generation

```c
/* header */
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <sched.h>
#include <stdbool.h>
#include <assert.h>

int pointpassed;
pthread_attr_t attr ;
struct sched_param parm ;

void init_thread();
void create_thread(pthread_t thread, void* process){
    pthread_attr_init(&attr);
    pthread_attr_setschedpolicy(&attr, SCHED_FIFO);
    parm.sched_priority = sched_get_priority_min(SCHED_FIFO);
    pthread_attr_setschedparam(&attr, &parm);
    pthread_create(&thread, &attr, process , NULL);
    pthread_detach(thread) ;
}

int main(int argc, char *argv[]) {
    int num_CPUs = 0;
    cpu_set_t mask;
    CPU_ZERO(&mask);
    CPU_SET(num_CPUs, &mask);
    if (sched_setaffinity(0, sizeof(mask), &mask) == -1) {
    printf("Could not set CPU Affinity");
   }
    init_thread();
    pthread_exit(0);
}

/* declaration */
void* process_01() ;
pthread_t thread_01;
void* process_11() ;
pthread_t thread_11;

void init_thread() {
    create_thread(thread_01, (void*) process_01 );
    create_thread(thread_11, (void*) process_11 );
}
void *process_01() {
    if (pointpassed == 1 - 1) {
       pointpassed = 1;
       if (pointpassed == 3 - 1) exit(1) ;
    } else {
       exit(0) ;
    }
   if (pointpassed == 2 - 1) {
       pointpassed = 2;
       if (pointpassed == 3 - 1) exit(1) ;
    } else {
       exit(0) ;
    }
}
void *process_11() {
}
```

Figure 7.9: A test program

Table 7.14: Test program execution results

Test program	Passed	No. steps	Times to try	Testing time (s)
Program_0.c	Yes	4	674	1.147
Program_1.c	Yes	2	1	0.004
Program_2.c	Yes	4	81	0.214
Program_3.c	Yes	2	1	0.004
Program_4.c	Yes	4	89	0.215
Program_5.c	Yes	7	813	2.429
Program_6.c	Yes	7	264453	451.564
Program_7.c	Yes	4	52284	89.027
Program_8.c	Yes	8	159024	272.397
Program_9.c	Yes	2	12140	20.730
Program_10.c	Yes	4	595	1.138
Program_11.c	Yes	3	15904	27.686
Program_12.c	Yes	6	52659	92.072
Program_13.c	Yes	6	6023	10.440
Program_14.c	Yes	8	52506	88.996
Program_15.c	Yes	8	400181	681.109
Program_16.c	Yes	4	676	1.149
Program_17.c	Yes	8	107358	182.041
Program_18.c	Yes	5	18433	31.164
Program_19.c	Yes	8	635	1.118
Program_20.c	Yes	3	629	1.084
Program_21.c	Yes	3	14044	23.764
Program_22.c	Yes	4	25	0.074
Program_23.c	Yes	8	209274	355.802
Program_24.c	Yes	4	11002	18.633
Program_25.c	Yes	7	923	1.695
Program_26.c	Yes	7	8135	15.364
Program_27.c	Yes	6	1	0.008
Program_28.c	Yes	6	139	0.339

Table 7.15: Test program generation result with 2 steps

No. test programs	States	Memory (MB.)	Time (s)
13	16	21.1623	0.03

7.4 Summary

We have introduced the case studies to evaluate our method. In fact, the behaviors of the system now are limited by the scheduling strategies in comparison without using the scheduler. This fact makes the verification more accurate because we can remove the spurious counter-examples that appear outside of the execution indicated by the

Table 7.16: Test program execution results with 2 steps

Test program	**Passed**	**Times to try**	**Testing time (s)**
Program_0.c	Yes	1	0.004
Program_1.c	Yes	1	0.005
Program_2.c	Yes	1	0.009
Program_3.c	Yes	1	0.004
Program_4.c	Yes	1	0.004
Program_5.c	Yes	2817	4.999
Program_6.c	Yes	1	0.004
Program_7.c	Yes	1	0.004
Program_8.c	Yes	1	0.006
Program_9.c	Yes	8724	15.249
Program_10.c	Yes	1	0.004
Program_11.c	Yes	9229	16.126
Program_12.c	Yes	1	0.004

scheduler. Moreover, beside the qualitative analysis, following the notation of time (as proposed in [36]) we can analyze the behaviors of the system quantitatively. In addition, with the approach to generating the tests, our method can be used to test the scheduling strategy specified in the DSL with the implementation of the scheduler in an OS. In the next chapter, we discuss the experimental results in more details with the advantages and disadvantages of our framework to show the flexibility and the accuracy of our tool for verifying systems with scheduling policies.

Chapter 8

Discussion

Based on the results of the experiments shown in the previous chapter, we discuss the advantage and the disadvantage of our method. The main issues for evaluating our results are as follows. Some remaining problems are also discussed at the end of this chapter.

- Accuracy and reliability: The verification results are matched with the expected results or with the results produced by a real system.
- Simplicity: It is easy to use the language to describe the policy in comparison with implementing the scheduling from scratch.
- Flexibility: It is flexible to change the behaviors of the scheduler in the description of the policy.
- Reusability: The specifications in the DSL can be reused.
- Scalability and practicality: Our method scale well in comparison with other tools and it is possible to apply our method to a real system.

8.1 Accuracy and Reliability

Verifying systems using different strategies and without scheduling lead to different results because they perform different behaviors. That means for the accurate verification, the policies need to be considered. The experiments show that our tool can realize the behaviors of the system following the scheduling policies.

As indicated in Section 7.1.1, the dining philosopher problem caused the deadlock when the scheduler was not used (with SpinJa tool) while they were absent with RR policy (with our tool, SSpinJa). If the priorities of the philosophers are different, the opportunities to eat for the philosophers are different. In this situation, no deadlock occurs; however, starvation happens because the low-priority philosophers have no chance to eat. The results above are as expected.

In Section 7.1.3, with the strategies used by Linux OS, the system can be accurately verified if the specification of the policies conforms to the real one (the scheduling policies used by an OS). We can see that some properties do not hold with the related policies (i.e. RR and FIFO). Therefore, to verify a real system accurately, we need to specify the behaviors of the scheduler conforming to the real one. This experiment indicates that we can accurately verify the behaviors of Linux processes.

The experiment in Section 7.2.1 shows that our method is more accurate than that of the scheduling framework in UPPAAL. Actually, we can determine that deadline violation occurs when the system uses either FP or FIFO policy; however, it does not occur with EDF strategy. In the first experiment in this section for the schedulability analysis of a real-time system shows that our tool can realize the deadline violation with either FP or FIFO policy. That result is different from that of the framework using UPPAAL model checker. In fact, this framework uses an over-approximation approach to analyze the schedulability problem. Therefore, the analysis results can be "may not be satisfied". That means the framework cannot determine the satisfaction. With the second experiment, although we can easily realize that the deadline violation does not occur with the number of the processes being less than 5, the analysis results of this framework were also "may not be satisfied". Moreover, this framework only focuses on the time constraints. Thus, considering both the behaviors of the processes and the behaviors of the scheduler is challenging. As indicated in Section 7.2.2, although using the same strategy (*priority* policy), when the configuration was changed, the analysis results for the behaviors of the system were also changed. Therefore, both the scheduling strategies and the attributes of the processes need to be considered to verify the behaviors of the system.

In addition, before making any verification, we take an assumption that the policy specified in the DSL must be correct. Actually, by applying our method, we can check the correspondence between the specification of the policy with the strategy implemented in a real OS as indicated in Section 7.3. It helps us to increase the confidence of the policy in the DSL.

8.2 Simplicity, Flexibility, and Reusability

The synchronization mechanism described in Section 7.1.2 relates to the scheduling strategies. The relation is not easy to describe using the language for the process because the whole system with the relations needs to be encoded. With our approach, because we provide a mechanism to define an interface for the communication between the processes and the scheduler, the relations are easy to specify.

For the analysis of the system, using discrete time with defining the `checkpoint` statements to label the state graph can easily handle the behaviors of the system quantitatively. The behaviors of the system (the processes with the scheduler) can be captured using the `checkpoint` statements. With this approach, labeling the graph helps to realize the occurrence of the scheduling events. Therefore, the property represented by a CTL/RTCTL formula can be verified simply. We note that the original tool (SpinJa) can use the statement merging technique to reduce the number of states. However, with the assumption that each transition taking one time unit, to determine the satisfaction of a quantitative property, we can not use this technique in our approach with our tool (SSpinJa).

Our framework can handle different configurations by only changing the values of the attributes of the processes. Besides, with the collection for storing the processes and specifying the event, we provide a flexible way to describe the behaviors of the scheduler. The experiments indicate that our framework can easily deal with the variation of the schedulers. With the ordering approach for selecting a process to run, we can easily specify the common scheduling policies. In fact, the description code for specifying

the policy (with the *verify* part) used in the experiments is really small in comparison with the number lines of code generated (as indicated in Table 8.1). For instance, with a GPL, if we want to deal with FIFO policy, we need to implement the queue data structure with the corresponding operations; of course, the implementation needs a lot of work, time-consuming and error-prone.

Table 8.1: The number lines of the code generated from the scheduling policy

Scheduling policy	**No. Lines Specification**	**No. Lines Code**
First-in-first-out (FIFO)	13	1668
Priority	30	1787
Earliest deadline first (EDF)	30	1775
Round-robin (RR)	15	1475
OSEK/VDX Priority Ceiling Protocol	68	2061
Linux (SCHED_OTHER, SCHED_FIFO, SCHED_RR)	55	1868

In our approach, we separate the specification of the policy and the processes. Considering the strategy as an input of the verification, with the same behaviors of the processes we can apply different policies, and with a strategy, we can apply different process models. This means that the processes and the scheduling policy can be reused completely.

In addition, for the approach to generating the tests, the specification of the policy and the test specification in the DSL is also reusable and flexible to handle the variation of the behaviors of the scheduler.

- The number lines of code for the policy and for the test generation used in the experiments is really small in comparison with the results generated (as indicated in Table 8.2). In another word, using the DSL is an effective way for the specification.

Table 8.2: The number lines of the code generated for the testing

Experiment	**Number of lines of the specification**	**Test generation results**
Test cases generation	- Process program: **18** lines - Process attributes: **6** lines - Scheduling and test generation: **48** lines	- No. tests: **14** test cases - No. lines codes: **317** lines
Test programs generation	- Process program: **18** lines - Process attributes: **6** lines - Scheduling and test generation: **100** lines	- No. tests: **29** test programs - No. lines codes: **3058** lines
Test programs generation with limited behaviors	- Process program: **18** lines - Process attributes: **6** lines - Scheduling and test generation: **105** lines	- No. tests: **13** test programs - No. lines codes: **936** lines

- With the specification of the test generation in the DSL, all the tests were generated automatically. The *process program* models the behaviors of the processes and we use the search to explore the system states, therefore, all the states are covered and represented in the tests. The execution orders of the system affect the code generated. For instance, which process selected among the processes with the same priority will lead to the different code generated in comparison with the others.

- In the implementation, all of the duplicate results in the test generation will be removed. Therefore, although using the same *process program* and the same policy, the number of test cases (in the experiment shown in Section 7.3.2) and the number of test programs (in the experiment shown in Section 7.3.3) are different.

- Our method also supports finding the trails that satisfy a property by labeling the system graph; thus, it is easy to generate the tests following a corresponding property. In the experiment shown in Section 7.3.3, we limited the behaviors of the processes (only 2 actions were performed) using the property indicated in the *verify* part of the *scheduler description*. Therefore, the number of the test programs generated were also limited.

8.3 Performance

As indicated in Section 7.1.1, although the number of states visited by SSpinJa was smaller than that visited by SpinJa when the search was completed, our tool needed more memory because we consider the behaviors of the scheduler during the verification, therefore, our tool needs more memory to store the information of the scheduler. In this experiment, the number of states and the verification time for different scheduling policies are also different. For instance, with the dining philosopher problem using the FP strategy, only the highest priority process can be selected; therefore, the number of states is unchanged. That is different from the RR strategy (as depicted in Figure 8.1.a). It also makes the running times for the scheduling policies different (Figure 8.1.b).

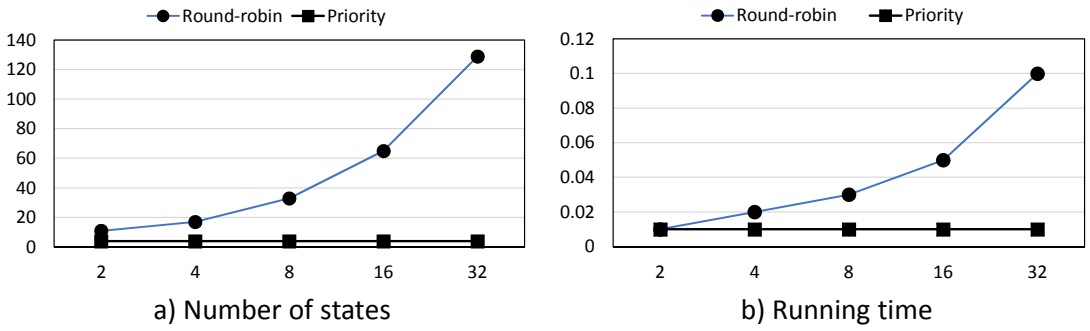


Figure 8.1: Dining philosopher problem results

In the experiment for schedulability analysis (Section 7.2), the running time of the scheduling framework using UPPAAL showed a significant increase in comparison with SSpinJa, and SSpinJa used less memory than this framework (as depicted in Figure 8.2). It means that our tool has better performance than the scheduling framework in UPPAAL does. In addition, the framework using UPPAAL model checker only focuses on the time constraints. Therefore, considering both the behaviors of the processes and the behaviors of the scheduler is also challenging.

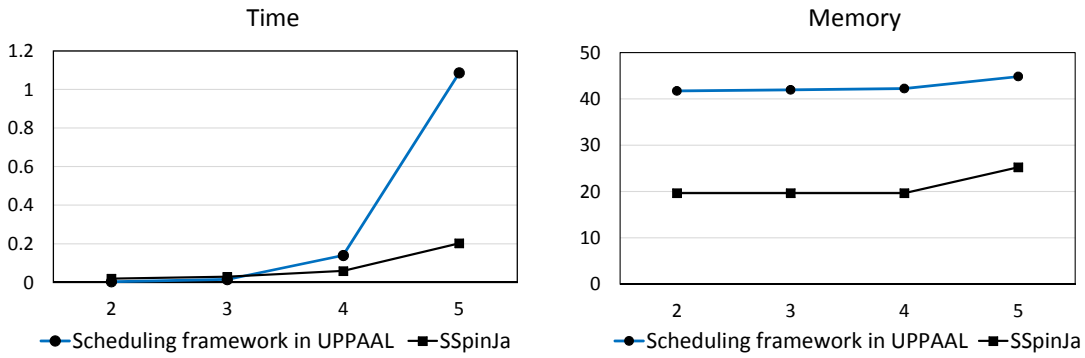


Figure 8.2: Running time and memory usage for the analysis

8.4 Practicality

For verifying the system with the scheduling strategies and testing the scheduler implemented in a real system, the experiments also showed that our approach is practical.

- Firstly, we can accurately verify the program executed in Linux OS based on the specification of the policy used by the system (as shown in Section 7.1.3).
- Secondly, we can generate the test programs for testing the policy in our DSL with the scheduler implementation of Linux OS (as described in Section 7.3.3). This is done easily with the support of the DSL for determining the structure of and the template of the programs.
- Thirdly, with the automated test generation for testing scheduling policies, we can increase the confidence of the specification of the policy in the DSL. It helps us to easily handle the quality assurance of the system with the scheduling strategies.

8.5 Remaining Problems

8.5.1 Improving the Performance

The experiments show that with the policy, our tool can limit the number of system states to be visited. However, in some cases, the memory usage of our tool is greater than that used by the original tool (SpinJa) without using the scheduler (as described in Section 7.1.1). That fact is because we store the scheduling information during the verification. How to optimize the memory usage is a problem needed to be solved in the future for this research.

In addition, for the analysis system with the scheduling strategies, our tool can raise a problem when the state space is not explored completely, it can return a "non-determined" result (as indicated in Section 7.2.1). This is because the number of processes is now limited in the current implementation of the tool.

8.5.2 Testing Non-deterministic Behaviors

Following the specification, there are some options for the implementation of the behaviors of the scheduler in an OS. We call that fact as *non-deterministic behaviors*. Actually, the approach for testing the non-deterministic behaviors by executing a test program many times to check the satisfaction is an ineffective approach because it only shows the satisfaction indicated by the program and can not prove the dissatisfaction. In the case of testing the policy, the average times to try and the average time for executing the programs corresponding to the number of the steps in the experiments shown in Section 7.3.3 are varied (as shown in Figure 8.3). The problem is now how to determine the bound of times for the execution. To handle this problem, the design of the tests needs to be considered.

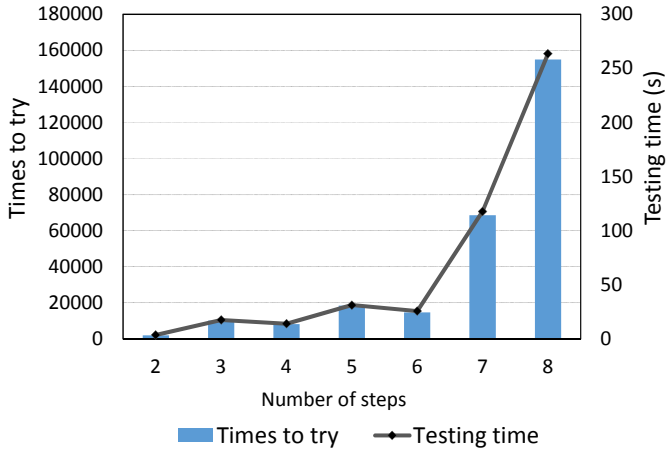


Figure 8.3: Experimental results for testing the scheduling policy

One idea to deal with this problem is that we need to find another method to generate the tests which can cover the non-deterministic behaviors following the specification of the scheduling policy of an OS. Another approach to handle the non-deterministic behaviors of the scheduler is based on checking the behaviors of the real scheduler (of an OS) are accepted by the description of the policy. The idea for testing includes two following steps.

- From the model of a system (with the scheduler and the processes), we generate the test data that represents the execution of the processes in the system. Besides, we write a test program, which uses this data as its input. Then, executing the test program with this input. As a result, we can determine the execution orders of the processes in a real system (called *execution result*).

- We then check whether this *execution result* is accepted by the model of this system (by searching on the graph). That means if there is a path on the graph corresponding to the *execution result*, the test is passed. Otherwise, the test fails.

8.5.3 Multi-core Scheduling Policies

Another problem we need to consider is that how to deal with the multi-core scheduling policies. In fact, a multi-core system can use a scheduling policy to control its processes.

The policy determines the executions of the processes for each core. Several problems need to be recognized to deal with multi-core systems, such as assigning the processes for cores, shared processes between cores, the time related to behaviors of the system, specifying the policy, etc.

To solve these problems, firstly we need to specify the scheduling policy of the multi-core system. Besides, some other issues related to the implementation need to be considered, such as determining the strategies to balance the load of each core. Secondly, to verify the system, the algorithm to explore the state space also needs to be changed to determine the possible executions of the processes in the system.

Chapter 9

Related Work

This chapter introduces the related work. We category the researches into four main related issues: a) verifying concurrent systems with considering the algorithm to search the system states, b) analyzing the schedulability of real-time systems, c) specifying the scheduling strategies, and d) methods for testing.

9.1 Verifying Concurrent Systems

Clarke et al. [24] proposed an algorithm to verify the behaviors of the system based on labeling the state graph realized from state space. There are two steps in this approach: first, building the state graph and second, label the graph following the property. In this research, we focus on verifying concurrent systems executed under scheduling policies. We realized that our research can adopt this approach by changing the first step, i.e. building the state graph. Actually, the search space is now limited by the scheduler. We dealt with that fact by proposing a DSL for the schedulers and introducing an algorithm based on the scheduling strategy specified in the DSL to search the system states. We now can build the state graph using the search algorithm and then adopt existing algorithms for checking the corresponding property. The main difference is that the search algorithm will limit the executions of the system. Therefore, the state space is also limited. After that, the results of the analysis can indicate the satisfaction of the property following the policy.

There are related works for guiding the search called directed model checking [35]. Several techniques were proposed to reduce the number of state using abstractions [4, 51] or calculating the distance [34, 33, 87] from the current state to the error states. The main purpose is to fast find the errors and short counterexamples. To achieve this objective, the well-known algorithm A^* [45] is used for performing the search. The process with the best value for the transition (lowest/highest) is selected to run. Our work is different because we use the scheduler to select a process, while they choose the best one to run.

Some works are to verify the behaviors of the system. Liu et al. [54] proposed a tool named PAT to build multi-domain tools; Pan et al. [64] proposed a method to deal with CAN network protocol. However, these works do not deal with scheduling strategies. The work Bogor [32] introduced a framework named for flexibly implementing the model checking tools flexibly. Our research is different from these works because we propose a method to flexibly verify different kinds of concurrent behaviors based on the scheduling policies.

9.2 Analyzing Real-Time Systems

There are some ways to handle the schedulability problem in scheduling domain. The main purpose is to indicate whether the design of the processes satisfies the scheduling conditions. There are some works using constraints solving [42], fixed scheduling [41] or worst-case assumption [82] to deal with this problem. Moreover, some tools were introduced, such as TimeWiz from Time Sys Corporation and RapidRMA from TriPacific. These tools are based on rate monotonic analysis, which uses fixed priority for the process. Actually, there are several differences between these approaches and ours.

- Firstly, our method uses model checking techniques.
- Secondly, we focus on the variation of the scheduler in analyzing the behaviors of the system by proposing a DSL for the scheduling strategies.
- Thirdly, the worst-case assumption is not used in our approach because the relation between the scheduling policy and the processes will be missed. Some behaviors of the processes will never happen. That means analyzing the behaviors of the system with a worst-case assumption is too pessimistic.

In our work, we consider individual behaviors of processes. Therefore, to handle the time related to the behaviors of the system, we use discrete time by considering an action of a process as taking one time unit. We used the timed-Kripke structure [36] to model the system. This allows us to apply existing algorithms introduced in [18, 24, 36] to verify the systems qualitatively and quantitatively.

There are researches with tools using timed automata to verify real-time systems. One of these tools is UPPAAL. With UPPAAL, to deal with the policies, we can build timed automata for the scheduling policy and the behaviors of the processes. With this approach, the scheduling framework introduced in [28] follows a model-based approach for the schedulability analysis. This framework aims at analyzing the resource sharing problem with real-time behaviors and a scheduling policy. Some fixed policies (i.e. FP, FIFO, and EDF) are implemented in this framework. However, this framework uses an over-approximation approach to deal with the schedulability problem. Therefore, in some situations, the framework cannot determine the satisfaction of the property. Moreover, one of the limitations of UPPAAL is that this tool does not support to access the internal information of a process. Therefore, the information of the process must be defined in public scope. This approach lacks flexibility. Moreover, it is challenging to deal with the facilitating the scheduling strategies with timed automata.

Another appropriate tool which allows the schedulability analysis is TIMES [3]. This tool can specify various attributes for tasks/processes, such as periodic, sporadic, and priority. However, TIMES provides limited scheduling policies and we cannot extend this tool to deal with other policies.

For real-time systems, the work UML profile for MARTE [55] introduced a DSL to analyze the time constraints. Nonetheless, the language cannot carry out the behaviors of processes and the scheduler as our work does.

9.3 Specifying Scheduling Policies

To deal with scheduling strategies, the approaches in model checking like [56, 5], use the existing modeling languages (e.g. Promela) to encode the scheduler and the processes

into a model of the system, then use an existing tool (e.g. Spin) for the verification. With these approaches, there are several limitations. Firstly, the strategy is fixed in the model and it is difficult to handle another polices. Secondly, the capability of this approach is limited because much redundant information is stored, and many behaviors of the scheduler are checked. Thus, the problem of state space explosion can easily occur. To overcome this problem, Zhang et al. proposed methods to remove the information of the scheduler from the model of a system [90], [91]. Nevertheless, the disadvantage of these methods is that the scheduler must be deterministic and fixed in tools. In addition, because the approach [91] relies on SMT and bounded model checking techniques [15], it is not efficient for verifying applications which contain many branches and loops.

To easily handle various kinds of schedulers, we propose a DSL for the scheduling policies. There are some existing languages for describing the scheduling strategies, such as Bossa [10] and Catapults [71]. In comparison with these languages, we have two main differences as follows.

- These studies apply to particular systems and rely on their techniques. Therefore, only limited types of policies are supported. In contrast, our approach does not apply to any system and can support a variation of policies.
- Their aim is for implementing the scheduler in OSs; ours is to ensure the correctness of a system using model checking techniques. With this purpose, we need to consider all possible behaviors based on the corresponding specification while a specific case for the implementation is enough for their works.

9.4 Conformance Testing and Model-Based Testing

Conformance testing techniques aim to verify the system implementation follows its specification. The research proposed by Chen and Aoki [17] introduces the scheduler to generate the test cases with Spin model checker for conformance testing OSEK/VDX OS. In fact, Spin can export the information during the checking phase using C functions. Their work uses this functionality to produce the log corresponding to the invoked system services and the current state of the system. Using that log, another tool will generate test cases for the system. However, because this work uses the support of an existing model checker, the approach has several limitations as follows.

- Firstly, the scheduler is modeled and embedded in the model of the system. Thus, the checking phase needs to consider many behaviors and store unnecessary information of the scheduler.
- Secondly, we need to build another tool to complete the generation.
- Thirdly, it is difficult to handle other scheduling policies for generating the tests.

In our case, the purpose of testing the strategy is different because we want to check that the description of the policy can specify the behaviors of the scheduler in an implementation. We can assume that the policy in the DSL is correct and then use it to test the behaviors of the scheduler in the implementation as in the conformance testing. However, the results of the testing have different meanings. In conformance testing, if the test fails that means the implementation does not follow the design (or the specification). In our case, if the tests passed, we are confident about the scheduling

policy in the DSL; otherwise, if any test fails, we cannot conclude anything because the test may be outside of the executions of the system.

To prepare the tests for testing the policy automatically, we apply MBT techniques [6]. There are some works using MBT for verifying the behaviors of the system. However, these works are different from ours. Actually, they do not deal with the scheduling policy [37], are not for testing the policy implemented in a real OS[74], or do not focus on the test generation for the testing [25].

To handle the test generation, recent researches use designed models taken from UML diagrams to generate the tests [65, 76, 80, 86]. Our work focuses on generating the tests automatically with model checking techniques. There are also several works to handle this problem using verification technologies [2, 67, 75, 30]. With the behaviors of each process representing in the model of the system, we can find the executions that lead to the violation of a property (counterexamples) or satisfy the property (witness) using model checking approach. In addition, there are tools, such as SAL [44] and STG [22] to deal with the test generation. SAL uses a specification with *trap variables* to represent the goals to be tested. STG uses symbolic techniques to overcome the problem of state space explosion. In fact, our work is based on the idea that uses the search to explore the system states and generate the tests following the behaviors of the system as these approaches do. However, these works and tools did not deal with the scheduling strategies as our work does.

In our research, we propose a DSL for the scheduling strategies and for the test generation. There are some works introducing the DSL for the test generation. The work [48] extracts the test cases from use case definitions using a DSL for the operations of a system. However, it only focuses on automating the system test process and does not deal with the scheduling policy. Paiva et al. [63] proposed a DSL for automatic test cases generation from the specification of a system. This work focus on interactive components and lack of its behaviors. To verify the event-based systems, Cyrille Artho et al. presented a tool named Modbat [8], which provides a DSL for constructing the state machine based on the explicit representation of system states. That means the behaviors of the system are represented as finite state machine explicitly in the DSL proposed. Our research is different from these works because our DSL is for the scheduling strategies with the test generation and building the state space on-the-fly during the verification.

Chapter 10

Conclusion and Future Directions

In this research, we aim at verifying concurrent applications (systems) which run on OSs using model checking techniques. We address the following problems.

1. The scheduler controls the executions of the system,
2. There is a variation of the policies uses by the OSs, and
3. Existing approaches are difficult to handle the variation.

The objective of this research is proposing a method to facilitate the variation of schedulers in model checking. The originality of our method is proposing a DSL for facilitating the variation of the schedulers. For verifying the system with the scheduling policy, we generate all of the necessary information automatically from the specification of the scheduler in the DSL to perform the search on the state space for the verification. In addition, we can exhaustively and automatically generate the tests to check the correspondence between the policy and the implementation of the scheduler in a real OS. This helps us to increase the confidence of the policy specified in the DSL and accurately verify the behaviors of the system.

Following this method, we implemented SSpinJa tool and conducted the experiments to show the accuracy, reliability, simplicity, flexibility, and reusability of our approach. We have the following results for this research.

1. Proposing a DSL for the scheduling strategies;
2. Proposing a model checking algorithm based on the policy to verify the system under the scheduling policy;
3. Proposing a method to analyze the behaviors of the system under the scheduling strategies qualitatively and quantitatively;
4. Proposing a method to test the correspondence between the specification of the policy in the DSL and the implementation of the scheduler in a real OS;
5. Implementing a tool following the method proposed to verify the behaviors of a system with scheduling strategies and generate the tests to check the scheduling policy.

The advantages of our approach are as follows.

1. The language for scheduling policies is simple;
2. The behaviors of the system can be extended and captured easily for the analysis;
3. The scheduling strategies can be reused;
4. The framework can analyze the behaviors of the system accurately;
5. The method is practical.

The limitation of our tool is memory usage. To overcome this problem, in the future, some optimization techniques, such as partial order reduction will be applied. Moreover, in this research, we only deal with the scheduling for the uniprocessor that has one core and at most one process can run. In fact, the concurrency can be taken place on multi-core processors and the scheduling strategies are also applied. This is one direction for us to extend our work. In addition, in this research, we use ordering methods to deal with the selection of the scheduler. Therefore, some policies without using the ordering, such as lottery policy, can not be handled by our method. To deal with this problem, we intend to support customizing the behaviors of the scheduler by adding user-defined libraries to handle the selection. Moreover, we plan to adopt testing techniques with multiple options for the implementation of the scheduling policies.

Bibliography

[1] P. Ammann and J. Offutt. *Introduction to software testing*. Cambridge University Press, 2016.

[2] P. E. Ammann, P. E. Black, and W. Majurski. Using model checking to generate tests from specifications. In *Formal Engineering Methods, 1998. Proceedings. Second International Conference on*, pages 46–54. IEEE, 1998.

[3] T. Amnell, E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi. TIMES: a tool for schedulability analysis and code generation of real-time systems. In *International Conference on Formal Modeling and Analysis of Timed Systems*, pages 60–72. Springer, 2003.

[4] A. S. Andisha, M. Wehrle, and B. Westphal. Directed model checking for PROMELA with relaxation-based distance functions. In *Model Checking Software*, pages 153–159. Springer, 2015.

[5] T. Aoki. Model checking multi-task software on real-time operating systems. In *2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*, pages 551–555. IEEE, 2008.

[6] L. Apfelbaum and J. Doyle. Model based testing. In *Software Quality Week Conference*, pages 296–300, 1997.

[7] K. Arnold, J. Gosling, and D. Holmes. *The Java programming language*. Addison Wesley Professional, 2005.

[8] C. V. Artho, A. Biere, M. Hagiya, E. Platon, M. Seidl, Y. Tanabe, and M. Yamamoto. Modbat: A model-based API tester for event-driven systems. In *Haifa Verification Conference*, pages 112–128. Springer, 2013.

[9] C. Baier, J.-P. Katoen, and K. G. Larsen. *Principles of model checking*. MIT press, 2008.

[10] L. P. Barreto and G. Muller. Bossa: a language-based approach to the design of real-time schedulers. *Proceedings of the 23rd IEEE Real-Time Systems*, 2002.

[11] B. Beizer. *Black-box testing: techniques for functional testing of software and systems*. John Wiley & Sons, Inc., 1995.

[12] S. Bennett, S. McRobb, and R. Farmer. *Object-oriented systems analysis and design using UML*, volume 2. McGraw-Hill New York, 1999.

[13] L. Bettini. *Implementing domain-specific languages with Xtext and Xtend*. Packt Publishing Ltd, 2016.

[14] D. Beyer, A. J. Chlipala, T. A. Henzinger, R. Jhala, and R. Majumdar. Generating tests from counterexamples. In *Proceedings of the 26th International Conference on Software Engineering*, pages 326–335. IEEE Computer Society, 2004.

[15] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu. Bounded model checking. *Advances in computers*, 58:117–148, 2003.

[16] P. Bjesse. What is formal verification? *ACM SIGDA Newsletter*, 35(24):1, 2005.

[17] J. Chen and T. Aoki. Conformance testing for OSEK/VDX operating system using model checking. In *Software Engineering Conference (APSEC), 2011 18th Asia Pacific*, pages 274–281. IEEE, 2011.

[18] A. M. Cheng. *Real-time systems: scheduling, analysis, and verification*. John Wiley & Sons, 2003.

[19] E. Christen and K. Bakalar. VHDL-AMS-a hardware description language for analog and mixed-signal applications. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, 46(10):1263–1272, 1999.

[20] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: a new symbolic model checker. *International Journal on Software Tools for Technology Transfer*, 2(4):410–425, 2000.

[21] A. Cimatti, A. Micheli, I. Narasamdya, and M. Roveri. Verifying SystemC: a software model checking approach. In *Formal Methods in Computer-Aided Design (FMCAD), 2010*, pages 51–59. IEEE, 2010.

[22] D. Clarke, T. Jéron, V. Rusu, and E. Zinovieva. STG: A symbolic test generation tool. In *TACAS*, volume 2280, pages 470–475. Springer, 2002.

[23] E. M. Clarke. The birth of model checking. In *25 Years of Model Checking*, pages 1–26. Springer, 2008.

[24] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(2):244–263, 1986.

[25] A. Claudi and A. F. Dragoni. Testing Linux-based real-time systems: Lachesis. In *Service-oriented computing and applications (SOCA), 2011 IEEE International Conference on*, pages 1–8. IEEE, 2011.

[26] L. Cordeiro and B. Fischer. Verifying multi-threaded software using SMT-based context-bounded model checking. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 331–340. ACM, 2011.

[27] C. J. Date and H. Darwen. *A guide to the SQL Standard: a user's guide to the standard relational language SQL*. Addison-Wesley, 1989.

[28] A. David, J. Illum, K. G. Larsen, and A. Skou. Model-based framework for schedu-

lability analysis using UPPAAL 4.1. *Model-based design for embedded systems*, 1(1):93–119, 2009.

[29] M. de Jonge and T. C. Ruys. The SpinJa model checker. In *International SPIN Workshop on Model Checking of Software*, pages 124–128. Springer, 2010.

[30] R. DeMilli and A. J. Offutt. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, 1991.

[31] B. Dushnik and E. W. Miller. Partially ordered sets. *American journal of mathematics*, 63(3):600–610, 1941.

[32] M. B. Dwyer and J. Hatcliff. Bogor: A flexible framework for creating software model checkers. In *Testing: Academic & Industrial Conference-Practice And Research Techniques (TAIC PART'06)*, pages 3–22. IEEE, 2006.

[33] S. Edelkamp, A. L. Lafuente, and S. Leue. *Protocol verification with heuristic search*. Bibliothek der Universität Konstanz, 2001.

[34] S. Edelkamp, S. Leue, and A. Lluch-Lafuente. Directed explicit-state model checking in the validation of communication protocols. *International journal on software tools for technology transfer*, 5(2-3):247–267, 2004.

[35] S. Edelkamp, V. Schuppan, D. Bošnački, A. Wijs, A. Fehnker, and H. Aljazzar. Survey on directed model checking. In *International Workshop on Model Checking and Artificial Intelligence*, pages 65–89. Springer, 2008.

[36] E. A. Emerson, A. K. Mok, A. P. Sistla, and J. Srinivasan. Quantitative temporal reasoning. *Real-Time Systems*, 4(4):331–352, 1992.

[37] L. Fang, T. Kitamura, T. B. N. Do, and H. Ohsaki. Formal model-based test for AUTOSAR multicore RTOS. In *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*, pages 251–259. IEEE, 2012.

[38] M. Fowler. *Domain-specific languages*. Pearson Education, 2010.

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

[40] R. Gerth. Concise PROMELA reference, 1997.

[41] H. Gomaa. Designing concurrent, distributed, and real-time applications with UML. In *Proceedings of the 23rd international conference on software engineering*, pages 737–738. IEEE Computer Society, 2001.

[42] R. Gorcitz, E. Kofman, T. Carle, D. Potop-Butucaru, and R. De Simone. On the scalability of constraint solving for static/off-line real-time scheduling. In *International Conference on Formal Modeling and Analysis of Timed Systems*, pages 108–123. Springer, 2015.

[43] I. S. Graham. *The HTML sourcebook*. John Wiley & Sons, Inc., 1995.

[44] G. Hamon, L. De Moura, and J. Rushby. Automated test generation with SAL. *CSL Technical Note*, page 15, 2005.

[45] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[46] G. J. Holzmann. *The SPIN model checker: Primer and reference manual*, volume 1003. Addison-Wesley Reading, 2004.

[47] G. J. Holzmann, D. A. Peled, and M. Yannakakis. On nested depth first search. *The Spin Verification System*, 32:23–32, 1996.

[48] K. Im, T. Im, and J. D. McGregor. Automating test case definition using a domain specific language. In *Proceedings of the 46th Annual Southeast Regional Conference on XX*, pages 180–185. ACM, 2008.

[49] N. Juristo, A. M. Moreno, and S. Vegas. Reviewing 25 years of testing technique experiments. *Empirical Software Engineering*, 9(1-2):7–44, 2004.

[50] M. Kattenbelt, T. C. Ruys, and A. Rensink. An object-oriented framework for explicit-state model checking. In *Proceedings of the 3rd European Symposium on Verification and Validation of Software Systems (VVSS 2007)*. Eindhoven University of Technology, 2007.

[51] S. Kupferschmid, J. Hoffmann, H. Dierks, and G. Behrmann. Adapting an AI planning heuristic for directed model checking. In *International SPIN Workshop on Model Checking of Software*, pages 35–52. Springer, 2006.

[52] K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer (STTT)*, 1(1):134–152, 1997.

[53] J. P. Lehoczky and S. Ramos-Thuel. An optimal algorithm for scheduling soft-aperiodic tasks in fixed-priority preemptive systems. In *Real-Time Systems Symposium, 1992*, pages 110–123. IEEE, 1992.

[54] Y. Liu, J. Sun, and J. S. Dong. Pat 3: An extensible architecture for building multi-domain model checkers. In *2011 IEEE 22nd International Symposium on Software Reliability Engineering*, pages 190–199. IEEE, 2011.

[55] MARTE, UML. UML profile for MARTE: modeling and analysis of real-time embedded systems, 2015.

[56] N. Marti, R. Affeldt, and A. Yonezawa. Model-checking of a multi-threaded operating system. In *23rd Workshop of the Japan Society for Software Science and Technology, University of Tokyo, Tokyo, Japan*, 2006.

[57] K. L. McMillan. Symbolic model checking. In *Symbolic Model Checking*, pages 25–60. Springer, 1993.

[58] P. McMinn. Search-based software test data generation: a survey. *Software testing, Verification and reliability*, 14(2):105–156, 2004.

[59] M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. *ACM computing surveys (CSUR)*, 37(4):316–344, 2005.

[60] C. Newham and B. Rosenblatt. *Learning the bash shell: Unix shell programming*. O'Reilly Media, Inc., 2005.

[61] J. Offutt and A. Abdurazik. Generating tests from UML specifications. In *International Conference on the Unified Modeling Language*, pages 416–429. Springer, 1999.

[62] OSEK Group and others. OSEK/VDX Operating System Specification, 2005.

[63] A. C. Paiva, J. P. Faria, and R. M. Vidal. Automated specification-based testing of interactive components with AsmL. In *QUATIC*, pages 119–126, 2004.

[64] C. Pan, J. Guo, L. Zhu, J. Shi, H. Zhu, and X. Zhou. Modeling and verification of CAN bus with application layer using UPPAAL. *Electronic Notes in Theoretical Computer Science*, 309:31–49, 2014.

[65] P. E. Patel and N. N. Patil. Testcases formation using UML activity diagram. In *Communication Systems and Network Technologies (CSNT), 2013 International Conference on*, pages 884–889. IEEE, 2013.

[66] R. Pelánek. BEEM: Benchmarks for explicit model checkers. In *International SPIN Workshop on Model Checking of Software*, pages 263–267. Springer, 2007.

[67] J. Peleska, E. Vorobev, and F. Lapschies. Automated test case generation with SMT-solving and abstract interpretation. In *NASA Formal Methods Symposium*, pages 298–312. Springer, 2011.

[68] J. L. Peterson and A. Silberschatz. *Operating system concepts*, volume 214. Addison-Wesley Reading, MA, 1985.

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

[70] S. Robinson. *Simulation: the practice of model development and use*. Wiley Chichester, 2004.

[71] M. D. Roper and R. A. Olsson. Developing embedded multi-threaded applications with CATAPULTS, a domain-specific language for generating thread schedulers. In *Proceedings of the 2005 international conference on Compilers, architectures and synthesis for embedded systems*, pages 295–303. ACM, 2005.

[72] J. Rushby. Automated test generation and verified software. In *Verified Software: Theories, Tools, Experiments*, pages 161–172. Springer, 2008.

[73] V. Rusu, L. Du Bousquet, and T. Jéron. An approach to symbolic test generation. In *International Conference on Integrated Formal Methods*, pages 338–357. Springer, 2000.

[74] R. Schlatte, B. Aichernig, F. De Boer, A. Griesmayer, and E. B. Johnsen. Testing concurrent objects with application-specific schedulers. In *International Colloquium on Theoretical Aspects of Computing*, pages 319–333. Springer, 2008.

[75] R. Seater and G. Dennis. Automated test data generation with SAT. *URL*

http://groups.csail.mit.edu/pag/6.883/projects/mutant-test-generation.pdf, 2005.

[76] M. Shirole and R. Kumar. UML behavioral model based test case generation: a survey. *ACM SIGSOFT Software Engineering Notes*, 38(4):1–13, 2013.

[77] A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating system concepts essentials*. John Wiley & Sons, Inc., 2014.

[78] J. A. Stankovic. Misconceptions about real-time computing: A serious problem for next-generation systems. *Computer*, 21(10):10–19, 1988.

[79] J. Sun, Y. Liu, J. S. Dong, and C. Chen. Integrating specification and programs for system modeling and verification. In *Theoretical Aspects of Software Engineering, 2009. TASE 2009. Third IEEE International Symposium on*, pages 127–135. IEEE, 2009.

[80] S. K. Swain, D. P. Mohapatra, and R. Mall. Test case generation based on use case and sequence diagram. *International Journal of Software Engineering*, 3(2):21–52, 2010.

[81] A. S. Tanenbaum. *Modern operating system*. Pearson Education, Inc, 2009.

[82] K. Tindell and J. Clark. Holistic schedulability analysis for distributed hard real-time systems. *Microprocessing and microprogramming*, 40(2-3):117–134, 1994.

[83] N.-H. Tran, Y. Chiba, and T. Aoki. Domain-specific language facilitates scheduling in model checking. In *Asia-Pacific Software Engineering Conference (APSEC), 2017 24th*, pages 417–426. IEEE, 2017.

[84] N.-H. Tran, Y. Chiba, and T. Aoki. Qualitative and quantitative analysis with scheduling policies in model checking. In *ACM/SIGAPP Symposium On Applied Computing (SAC), 2018 33rd*, pages 1873–1880. IEEE, 2018.

[85] V. Trenkaev, M. Kim, and S. Seol. Interoperability testing based on a fault model for a system of communicating FSMs. In *IFIP International Conference on Testing of Software and Communicating Systems*, pages 226–242. Springer, 2003.

[86] Y. Wang and M. Zheng. Test case generation from UML models. In *45th Annual Midwest Instruction and Computing Symposium, Cedar Falls, Iowa*, volume 4, 2012.

[87] M. Wehrle, S. Kupferschmid, and A. Podelski. Transition-based directed model checking. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 186–200. Springer, 2009.

[88] Z. Yang, C. Wang, A. Gupta, and F. Ivančić. Model checking sequential software programs via mixed symbolic analysis. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 14(1):10, 2009.

[89] H. Zhang, T. Aoki, and Y. Chiba. A spin-based approach for checking OSEK/VDX applications. In *International Workshop on Formal Techniques for Safety-Critical Systems*, pages 239–255. Springer, 2014.

[90] H. Zhang, T. Aoki, and Y. Chiba. Yes! you can use your model checker to verify OSEK/VDX applications. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–10. IEEE, 2015.

[91] H. Zhang, T. Aoki, K. Yatake, M. Zhang, and H.-H. Lin. An approach for checking OSEK/VDX applications. In *2013 13th International Conference on Quality Software*, pages 113–116. IEEE, 2013.

Publications

[1] N.-H. Tran, Y. Chiba, and T. Aoki. Domain-specific language facilitates scheduling in model checking. In *Asia-Pacific Software Engineering Conference (APSEC), 2017 24th*, pages 417–426. IEEE, 2017.

[2] N.-H. Tran, Y. Chiba, and T. Aoki. Qualitative and quantitative analysis with scheduling policies in model checking. In *ACM/SIGAPP Symposium On Applied Computing (SAC), 33rd*, pages 1873–1880. IEEE, 2018.

Appendix A

Language Grammar

⟨*Model*⟩ ::= ⟨*ProcClass*⟩ | ⟨*ProcDSL*⟩ | ⟨*SchDSL*⟩

⟨*ProcClass*⟩ ::= '`process`' ⟨*ID*⟩ ['`refines`' ⟨*ID*⟩] '`{`' [⟨*DefAttr*⟩] ⟨*DefBehavior*⟩ '`}`' '`configuration`' '`{`' [⟨*ProcessConfig*⟩] ⟨*ProcessInit*⟩ '`}`'

⟨*DefAttr*⟩ ::= '`attribute`' '`{`' (⟨*AttDef*⟩)* [⟨*Constraints*⟩] '`}`'

⟨*AttDef*⟩ ::= ⟨*ID*⟩ '`:`' '`type`' '`=`' ⟨*Type*⟩ ['`,`' '`value`' '`=`' ⟨*ListDef*⟩] '`,`' '`default`' '`=`' ⟨*Value*⟩ '`;`'

⟨*ListDef*⟩ ::= '`[`' ⟨*List*⟩ ('`,`' ⟨*List*⟩) * '`]`'

⟨*List*⟩ ::= ⟨*Range*⟩ | ⟨*BOOL*⟩ | ⟨*ID*⟩

⟨*Range*⟩ ::= ⟨*INT*⟩ '`..`' ⟨*INT*⟩

⟨*Value*⟩ ::= ⟨*BOOL*⟩ | ⟨*INT*⟩

⟨*DefBehavior*⟩ ::= ⟨*ProcType*⟩ | ⟨*ProcBehav*⟩

⟨*ProcType*⟩ ::= (⟨*ProcessType*⟩)*

⟨*ProcessType*⟩ ::= '`proctype`' ⟨*ID*⟩ '`{`' [⟨*Constraints*⟩] (⟨*ProcBehav*⟩)* '`}`'

⟨*ProcBehav*⟩ ::= '`behavior`' '`{`' (⟨*PBehav*⟩)* '`}`'

⟨*PBehav*⟩ ::= ⟨*Constructor*⟩ | ⟨*Method*⟩

⟨*Constructor*⟩ ::= '`constructor`' '`:`' ⟨*ID*⟩ '`(`' [⟨*PramList*⟩]'`)`' '`;`'

⟨*Method*⟩ ::= '`method`' '`:`' ⟨*ID*⟩ (('`(`' '`)`' '`;`') | ('`(`' ⟨*PramList*⟩ '`)`' '`{`' (⟨*AssignPara*⟩)* [⟨*Constraints*⟩] '`}`'))

⟨*AssignPara*⟩ ::= ⟨*ID*⟩ '`:`' '`value`' '`=`' ⟨*ListDef*⟩ '`;`'

⟨*Constraints*⟩ ::= '`constraint`' '`{`' (⟨*Constr*⟩) * '`}`'

⟨*Constr*⟩ ::= ⟨*Or*⟩ '`;`'

⟨*ProcDSL*⟩ ::= '`def`' '`process`' '`{`' [⟨*ProcAttr*⟩] ⟨*Process*⟩* '`}`' [⟨*ProcConf*⟩] [⟨*ProcInit*⟩]

⟨*ProcAttr*⟩ ::= '`attribute`' '`{`' ⟨*PAttr*⟩* '`}`'

⟨*PAttr*⟩ ::= ['`var`'|'`val`'] ⟨*Type*⟩⟨*ID*⟩ ('`,`'⟨*ID*⟩)* ['`=`'⟨*Value*⟩]'`;`'

⟨*Type*⟩ ::= ‘`int`’ | ‘`byte`’ | ‘`clock`’

⟨*Process*⟩ ::= ‘`proctype`’ ⟨*ID*⟩ ‘`(`’[⟨*PramList*⟩]‘`)`’ ‘`{`’ ⟨*AttAss*⟩* ‘`}`’

⟨*PramList*⟩ ::= ⟨*PramAss*⟩ (‘`;`’ ⟨*PramAss*⟩)*

⟨*PramAss*⟩ ::= ⟨*Type*⟩ ⟨*ID*⟩ (‘`,`’ ⟨*ID*⟩)* ‘`=`’ ⟨*Value*⟩

⟨*AttAss*⟩ ::= [‘`this`’ ‘`.`’] ⟨*ID*⟩ ‘`=`’ (⟨*Value*⟩ | ⟨*ID*⟩) ‘`;`’

⟨*ProcConf*⟩ ::= ‘`config`’ ‘`{`’ ⟨*PConf*⟩* ‘`}`’

⟨*PConf*⟩ ::= ⟨*SporadicP*⟩ | ⟨*PeriodicP*⟩

⟨*SporadicP*⟩ ::= ‘`sporadic`’ ‘`process`’ ⟨*Proc*⟩ ‘`in`’ ‘`(`’ ⟨*INT*⟩ ‘`,`’ ⟨*INT*⟩ ‘`)`’ [‘`limited`’ ⟨*INT*⟩] ‘`;`’

⟨*PeriodicP*⟩ ::= ‘`periodic`’ ‘`process`’ ⟨*Proc*⟩ ‘`offset`’ ‘`=`’ ⟨*INT*⟩ ‘`period`’ ‘`=`’ ⟨*INT*⟩ [‘`limited`’ ⟨*INT*⟩] ‘`;`’

⟨*Proc*⟩ ::= ⟨*ID*⟩ ‘`(`’ [⟨*Value*⟩ (‘`,`’ ⟨*Value*⟩)*] ‘`)`’

⟨*ProcInit*⟩ ::= ‘`init`’ ‘`{`’ ‘`[`’ ⟨*PSet*⟩ (‘`,`’ ⟨*PSet*⟩)* ‘`]`’ ‘`}`’‘`;`’

⟨*PSet*⟩ ::= ‘`{`’ ⟨*Proc*⟩ (‘`,`’ ⟨*Proc*⟩)* ‘`}`’

⟨*SchDSL*⟩ ::= ⟨*SchDef*⟩ [⟨*OrdDef*⟩] [⟨*Verify*⟩]

⟨*SchDef*⟩ ::= ‘`scheduler`’ ⟨*ID*⟩ ‘`(`’ [⟨*ParamList*⟩] ‘`)`’ [‘`refines`’ ⟨*ID*⟩] ‘`{`’ [⟨*Generate*⟩] [⟨*VarDef*⟩] [⟨*DatDef*⟩] [⟨*HandlerDef*⟩] [⟨*InterDef*⟩] ‘`}`’

⟨*Generate*⟩ ::= ‘`generate`’ ‘`{`’ ⟨*GenConfig*⟩ ⟨*GenComp*⟩‘`}`’

⟨*GenConfig*⟩ ::= ‘`configuration`’ ‘`{`’ [⟨*GenOption*⟩ ‘`;`’] [⟨*Dir*⟩ ‘`;`’] [⟨*FName*⟩ ‘`;`’] [⟨*FExt*⟩ ‘`;`’] ‘`test`’ (‘`program`’ | ‘`case`’ | ‘`data`’) ‘`=`’ ⟨*TestPart*⟩ ‘`}`’

⟨*GenOption*⟩ ::= ‘`option`’ ‘`=`’ ‘`{`’ ⟨*GenOpt*⟩ (‘`,`’ ⟨*GenOpt*⟩)* ‘`}`’

⟨*GenOpt*⟩ ::= ‘`Searching`’ | ‘`Error`’ | ‘`Property`’ | ‘`All`’

⟨*Dir*⟩ ::= ‘`directory`’ ‘`=`’ ⟨*STRING*⟩ ‘`;`’

⟨*FName*⟩ ::= ‘`file`’ ‘`name`’ ‘`=`’ ⟨*STRING*⟩ ‘`;`’

⟨*FExt*⟩ ::= ‘`file`’ ‘`extension`’ ‘`=`’ ⟨*STRING*⟩ ‘`;`’

⟨*TestPart*⟩ ::= ⟨*GenPart*⟩ (‘`+`’ ⟨*GenPart*⟩)*

⟨*GenPart*⟩ ::= ‘`(`’ [⟨*STRING*⟩ ‘`+`’] (⟨*ID*⟩ | ‘`init`’ | ‘`processes`’ | ‘`behaviors`’ | ‘`error`’) [‘`+`’ ⟨*STRING*⟩] ‘`)`’

⟨*GenComp*⟩ ::= ‘`component`’ ‘`{`’ (⟨*Comp*⟩)* [⟨*InitGen*⟩] [⟨*ProcGen*⟩] ‘`}`’;

⟨*Comp*⟩ ::= ⟨*ID*⟩ ‘`{`’ (⟨*Gen*⟩ |⟨*GenLn*⟩)* ‘`}`’

⟨*InitGen*⟩ ::= ‘`init`’ ‘`{`’ ⟨*Template*⟩ ‘`}`’

⟨*ProcGen*⟩ ::= ‘`process`’ ‘`{`’ ⟨*Template*⟩ ‘`}`’

⟨*Template*⟩ ::= [⟨*SetTemplate*⟩] ⟨*Behavior*⟩

⟨*SetTemplate*⟩ ::= ‘`template`’ ‘`=`’ ⟨*Expr*⟩ ‘`;`’

⟨*Behavior*⟩ ::= ‘`behavior`’ ‘`=`’ ⟨*EventTemp*⟩ (‘`+`’ ⟨*EventTemp*⟩)* ‘`;`’

⟨*EventTemp*⟩ ::= ‘`(`’ [⟨*Expr*⟩ ‘`+`’] ⟨*Event*⟩ [‘`+`’ ⟨*Expr*⟩] ‘`)`’

⟨*VarDef*⟩ ::= ‘`variable`’ ‘`{`’ ⟨*VDec*⟩* ‘`}`’

⟨*VDec*⟩ ::= [⟨*IfDef*⟩] (⟨*VBlockDef*⟩ | ⟨*VOneDef*⟩)

⟨*IfDef*⟩ ::= ‘`#`’ ‘`ifdef`’ ‘`(`’ ⟨*Expr*⟩ ‘`)`’

⟨*VBlockDef*⟩ ::= ‘`{`’ ⟨*VOneDef*⟩* ‘`}`’

⟨*VOneDef*⟩ ::= ⟨*Type*⟩ ⟨*ID*⟩ (‘`,`’ ⟨*ID*⟩)* [‘`=`’ ⟨*Value*⟩] ‘`;`’

⟨*DatDef*⟩ ::= ‘`data`’ ‘`{`’ ⟨*DDef*⟩* ‘`}`’

⟨*DDef*⟩ ::= [⟨*IfDef*⟩] ‘`data`’ (⟨*DBlockDef*⟩ | ⟨*DOneDef*⟩)

⟨*DBlockDef*⟩ ::= ‘`{`’ ⟨*DOneDef*⟩* ‘`}`’

⟨*DOneDef*⟩ ::= ⟨*VOneDef*⟩ | ⟨*ColDef*⟩

⟨*ColDef*⟩ ::= [‘`refines`’] ‘`collection`’ ⟨*ID*⟩ [‘`using`’ ⟨*ID*⟩ (‘`,`’ ⟨*ID*⟩)*] [‘`with`’ ⟨*OrdType*⟩] ‘`;`’

⟨*OrdType*⟩ ::= ‘`lifo`’ | ‘`fifo`’

⟨*HandlerDef*⟩ ::= ‘`event`’ ‘`handler`’ ‘`{`’ ⟨*EventDef*⟩* ‘`}`’

⟨*EventDef*⟩ ::= ⟨*Event*⟩ ‘`(`’ [⟨*ID*⟩] ‘`)`’ ‘`{`’ ⟨*IfDefStm*⟩* ‘`}`’

⟨*IfDefStm*⟩ ::= [⟨*IfDef*⟩] ⟨*Stm*⟩

⟨*Event*⟩ ::= ‘`select_process`’ | ‘`new_process`’ | ‘`clock`’ | ‘`pre_take`’ | ‘`post_take`’ | ‘`action`’

⟨*InterDef*⟩ ::= ‘`interface`’ ‘`{`’ ⟨*InterFunc*⟩* ‘`}`’

⟨*InterFunc*⟩ ::= ‘`function`’ ⟨*ID*⟩ ‘`(`’ [⟨*IParamList*⟩] ‘`)`’ ‘`{`’ ⟨*Stm*⟩* ‘`}`’

⟨*IParamList*⟩ ::= ⟨*IParamDec*⟩ (‘`,`’ ⟨*IParamDec*⟩)*

⟨*IParamDec*⟩ ::= ⟨*Type*⟩ ⟨*ID*⟩

⟨*OrdDef*⟩ ::= ‘`comparator`’ ‘`{`’ [⟨*CVarDef*⟩] ⟨*CompDef*⟩* ‘`}`’

⟨*CVarDef*⟩ ::= ‘`variable`’ ‘`{`’ ⟨*VOneDef*⟩* ‘`}`’

⟨*CompDef*⟩ ::= ‘`comparetype`’ ⟨*ID*⟩ ‘`(`’ ‘`process`’ ⟨*ID*⟩ ‘`,`’ ⟨*ID*⟩ ‘`)`’ ‘`{`’ ⟨*Stm*⟩* ‘`}`’

⟨*Stm*⟩ ::= ⟨*SetTime*⟩ | ⟨*SetCol*⟩ | ⟨*Change*⟩ | ⟨*Move*⟩ | ⟨*Remove*⟩ | ⟨*Get*⟩ | ⟨*New*⟩

| ⟨*If*⟩ | ⟨*Loop*⟩ | ⟨*Block*⟩ | ⟨*Assert*⟩ | ⟨*Print*⟩ | ⟨*Return*⟩ | ⟨*Gen*⟩ |⟨*GenLn*⟩

⟨*SetTime*⟩ ::= '`time_slice`' '`=`' ⟨*Expr*⟩ '`;`'

⟨*SetCol*⟩ ::= '`return_set`' '`=`' ⟨*ID*⟩ '`;`'

⟨*Change*⟩ ::= ⟨*ChgUnOp*⟩ | ⟨*ChgExpr*⟩

⟨*ChgUnOp*⟩ ::= ⟨*QualName*⟩ ('`++`' | '`{{`') '`;`'

⟨*ChgExpr*⟩ ::= ⟨*QualName*⟩ '`=`' ⟨*Expr*⟩ '`;`'

⟨*QualName*⟩ ::= ⟨*ID*⟩ ['`.`'⟨*ID*⟩]

⟨*Move*⟩ ::= '`move`' ⟨*ID*⟩ to ⟨*ID*⟩ '`;`'

⟨*Remove*⟩ ::= '`remove`' ⟨*ID*⟩ '`;`'

⟨*Get*⟩ ::= '`get`' '`process`' '`from`' ⟨*ID*⟩ '`to`' '`run`' '`;`'

⟨*New*⟩ ::= '`new`' ⟨*Proc*⟩ ['`,`' ⟨*INT*⟩] '`;`'

⟨*If*⟩ ::= '`if`' '`(`' ⟨*Expr*⟩ '`)`' ⟨*Stm*⟩ ['`else`' ⟨*Stm*⟩]

⟨*Loop*⟩ ::= '`for`' '`each`' '`process`' ⟨*ID*⟩ '`in`' ⟨*ID*⟩ ⟨*Stm*⟩

⟨*Block*⟩ ::= '`{`' ⟨*Stm*⟩* '`}`'

⟨*Assert*⟩ ::= '`assert`' ⟨*Expr*⟩ '`;`'

⟨*Print*⟩ ::= '`print`' ⟨*Expr*⟩ '`;`'

⟨*Return*⟩ ::= '`return`' ⟨*OrderType*⟩ '`;`'

⟨*OrderType*⟩ ::= '`greater`' | '`less`' | '`equal`'

⟨*Gen*⟩ ::= '`gen`' [⟨*ID*⟩ '`,`'] ⟨*Expr*⟩ '`;`'

⟨*GenLn*⟩ ::= '`genln`' [⟨*ID*⟩ '`,`'] ⟨*Expr*⟩ '`;`'

⟨*Expr*⟩ ::= ⟨*Or*⟩

⟨*Or*⟩ ::= ⟨*And*⟩ ('`||`' ⟨*And*⟩)*

⟨*And*⟩ ::= ⟨*Equality*⟩ ('`&&`' ⟨*Equality*⟩)*

⟨*Equality*⟩ ::= ⟨*Equality*⟩ ('`==`' | '`!=`') ⟨*Compar*⟩

⟨*Compar*⟩ ::= ⟨*PlusMinus*⟩ ('`>=`' | '`<=`' | '`>`' | '`<`') ⟨*PlusMinus*⟩

⟨*PlusMinus*⟩ ::= ⟨*MulOrDiv*⟩ ('`+`' | '`-`') ⟨*MulOrDiv*⟩

⟨*MulOrDiv*⟩ ::= ⟨*MulOrDiv*⟩ ('`*`' | '`/`') ⟨*Primary*⟩

⟨*Primary*⟩ ::= '`(`'⟨*Expr*⟩'`)`' | '`!`'⟨*Primary*⟩ | ⟨*Empty*⟩ | ⟨*Null*⟩ | ⟨*InCol*⟩ | ⟨*Exist*⟩ | ⟨*GetID*⟩ | ⟨*HasName*⟩ | ⟨*Atomic*⟩

⟨*Empty*⟩ ::= ⟨*ID*⟩‘.’ ‘`isEmpty`’ ‘(’ ‘)’

⟨*Null*⟩ ::= ⟨*ID*⟩‘.’ ‘`isNull`’ ‘(’ ‘)’

⟨*InCol*⟩ ::= ⟨*ID*⟩‘.’ ‘`containsProcess`’ ‘(’ ⟨*STRING*⟩ ‘)’

⟨*Exist*⟩ ::= ‘`exists`’ ‘(’ ⟨*STRING*⟩ ‘)’

⟨*GetID*⟩ ::= ‘`get_pid`’ ‘(’ ⟨*STRING*⟩ ‘)’

⟨*HasName*⟩ ::= ⟨*ID*⟩ ‘.’ hasName’ ‘(’ ⟨*STRING*⟩ ‘)’

⟨*Atomic*⟩ ::= ⟨*Value*⟩ | ⟨*QualName*⟩ | ⟨*SysVar*⟩

⟨*SysVar*⟩ ::= ‘`Sys`’ ‘(’ ⟨*ID*⟩ ‘)’

⟨*Verify*⟩ ::= ‘`verify`’ ‘`{`’ [⟨*CTL_AT*⟩] ⟨*RTCTL*⟩ ‘`}`’

⟨*CTL_AT*⟩ ::= ‘`@`’ ⟨*Expr*⟩ ‘`:`’

⟨*RTCTL*⟩ ::= ‘(’ ⟨*Expr*⟩ ‘)’ | ‘`not`’ ⟨*RTCTL*⟩ | ‘`or`’ ⟨*RTCTL*⟩ ⟨*RTCTL*⟩ | ‘`implies`’ ⟨*RTCTL*⟩ ⟨*RTCTL*⟩ | ‘`AX`’ ⟨*RTCTL*⟩ | ‘`AF`’ [⟨*LTE*⟩] ⟨*RTCTL*⟩ | ‘`AG`’ [⟨*LTE*⟩] ⟨*RTCTL*⟩ | ‘`EX`’ ⟨*RTCTL*⟩ | ‘`EF`’ [⟨*LTE*⟩] ⟨*RTCTL*⟩ | ‘`EG`’ [⟨*LTE*⟩] ⟨*RTCTL*⟩ | ‘`AU`’ [⟨*LTE*⟩] ⟨*RTCTL*⟩ ⟨*RTCTL*⟩ | ‘`EU`’ [⟨*LTE*⟩] ⟨*RTCTL*⟩ ⟨*RTCTL*⟩

⟨*LTE*⟩ ::= ‘`<=`’ ⟨*INT*⟩

- We note that some terms, such as `<ID>`, `<STRING>`, `<INT>`, `<BOOL>`, are not shown in the grammar.
- The ‘`val`’ (‘`var`’) keyword for defining an attribute of the process indicates that the value of this attribute is unchangeable (changeable). Only the values assigning to the changeable attributes are stored in the system state.
- The `<IfDef>` statement is used for initializing the scheduler based on the condition `<Expr>`. This statement allows us to deal with parameterizing the scheduling policy.
- We also support reusing the specification by introducing ‘`refines`’ keyword. If scheduler `B` ‘`refines`’ scheduler `A`, all of the data structures and the event handlers of `A` are inherited by `B`; however, `B` can redefine them, add more data structures and handle its new events. It is similar to the inheritance in object-oriented programming. With a collection, ‘`refines`’ means redefining its ordering method.

Appendix B

Code Generated for Scheduling Policy

1. The description of Round-Robin scheduling policy in the DSL

```
def process roundrobin {
 proctype P()  {}
}

init {
 [{P()}, {P()}]
}
```

a) Process attribute

```
scheduler roundrobin() {
 data {
  collection ready with fifo ;
 }
 event handler{
  select_process (process target_process) {
   get process from ready to run ;
   time_slice = 3 ;
   return_set = ready ;
  }
  new_process (process target_process) {
   move target_process to ready ;
  }
 }
}
```

b) Scheduler description

2. Java code (scheduler object) generated from the description of the scheduling policy

```
1  package sspinja ;

3  import java.io.PrintWriter;
   import java.util.ArrayList;
5  import java.util.HashMap;
   import java.util.Iterator;
7  import spinja.util.DataReader;
   import spinja.util.DataWriter;
9  import spinja.util.StringUtil;
   import spinja.util.Util;
11 import spinja.util.Log;
   import spinja.exceptions.*;
13 import spinja.promela.model.PromelaProcess;
   import sspinja.scheduler.search.SchedulerSearchAlgorithm;
15 import spinja.util.ByteArrayStorage;

17 import sspinja.scheduler.promela.model.SchedulerPromelaModel;
   import sspinja.SchedulerPanModel;
```

```
19 import sspinja.Generate;
   //Automatic generation
21 public class SchedulerObject_roundrobin {
   public static ArrayList<StaticProperty> staticPropertyList =
23     new ArrayList<StaticProperty>() ;
   public static ArrayList<String> processList = new ArrayList<String>() ;
25 public static boolean [] processInScheduler = new boolean [128];
   public static byte pcount = 0 ;
27 public static ArrayList<Byte> pcnt = new ArrayList<Byte>();
   public String _action = "";
29 public static ArrayList<String> initprocesslist = new ArrayList<String>();
   public static SchedulerPromelaModel panmodel ;
31 public int _schselopt ;
   public int _schnumopt ;
33 public int [][] _opt;

35 private static int newP = -1, endP = -1;
   public static int getnewP(){
37     int result = newP;
       newP = -1;
39     return result;
   }
41 public static int getendP(){
       int result = endP;
43     endP = -1;
       return result;
45 }

47 public int switchCore(int lastcore) throws ValidationException {return -1;}
   public int selCore(int lastcore) throws ValidationException {return -1;}
49
   public static void setPcnt(ArrayList<Byte> pcount) {
51     pcnt.clear();
       pcnt.addAll(pcount) ;
53 }
   public static ArrayList<Byte> getPcnt() {
55     return pcnt ;
   }
57 public void setSchedulerSelOption(int sel){
       _schselopt = sel;
59 }

61 public int nextSchedulerOption(int lastschopt) {
       if (lastschopt == -1) {
63         lastschopt = 0 ;
       }
65     if (lastschopt < _schnumopt - 1) {
           _schselopt = lastschopt + 1 ;
67     } else {
           _schselopt = 0 ;
69         return -1; //no more scheduler option
       }
71     return _schselopt;
   }
73
   public int firstSchedulerOption() {
75     _schselopt = 0 ;
       return 0;
```

```
77  }

79  public boolean hasGenTemplate = false ;

81  public void setAction(String act) {
        _action = act ;
83  }
    public String getAction() {
85      return _action ;
    }
87  public int getRefID(String pName) {
        int i = 0 ;
89      for (StaticProperty sP : staticPropertyList) {
            if (sP.pName.equals(pName))
91              return i ;
            i ++ ;
93      }
        return -1 ;
95  }
    public static StaticProperty getStaticPropertyObject(int refID) {
97      for (StaticProperty sP : staticPropertyList)
            if (sP.refID == refID)
99              return sP ;
        return null ;
101 }

103 public ArrayList<SchedulerProcess> findProcessByAlias(String alias) {
        ArrayList<SchedulerProcess> result=new ArrayList<SchedulerProcess>();
105
        if (alias.trim().equals("running_process")) {
107         if (running_process != null) {
                result.add(running_process);
109         }
        } else {
111         int idx = 0 ;
            for (String procN : processList) {
113             if (procN.trim().equals(alias.trim())) {
                    SchedulerProcess target_process = findProcessByID(idx) ;
115                 if (target_process != null)
                    result.add(target_process) ;
117             }
                idx ++ ;
119         }
            for (StaticProperty stP : staticPropertyList) {
121             if (stP.pName.trim().equals(alias.trim())) {
                    int refID = stP.refID ;
123                 ArrayList<SchedulerProcess> resultStP =
                    new ArrayList<SchedulerProcess>();
125                 resultStP = findProcessByrefID(refID) ;
                    if (!resultStP.isEmpty()) {
127                     for (SchedulerProcess p1 : resultStP) {
                            boolean add = true ;
129                         for (SchedulerProcess p2 : result) {
                                if (p1.processID == p2.processID)
131                                 add = false;
                            }
133                         if (add) {
                                result.add(p1) ;
```

```
135                         }
                        }
137                 }
                }
139         }
        }
141     return result ;
    }
143 public ArrayList<SchedulerProcess> findProcess(String pName) {
        return findProcessByAlias(pName) ;
145 }
    public int existsProcess (String pName) {
147     ArrayList<SchedulerProcess> aP = findProcess(pName) ;
        return aP.size() ;
149 }
    public int existsProcess (int pID) {
151     SchedulerProcess p = findProcessByID(pID) ;
        if (p == null) return 0 ;
153     else return 1 ;
    }
155 public void updateProcessInSchedulerList() {
        for (int i = 0; i < 128; i ++) {
157         if (processInScheduler[i]) {
                processInScheduler[i] = (findProcessByID(i) != null);
159         }
        }
161 }

163 //--------------- constructor-------------------------------
    public SchedulerObject_roundrobin() {
165     //default constructor
        _opt = new int[2][3];
167     int index = 0 ;
        for (int i = 1; i <= 1 ; i++)
169         for (int j = 1; j <= 1 ; j++)
                for (int k = 1; k <= 1 ; k++) {
171             index ++ ;
                _opt[index][0] = i ; //new
173             _opt[index][1] = j ; //select
                _opt[index][2] = k ; //clock
175         }
        _schnumopt = 1;
177 }

179 public boolean InitSchedulerObject(String args) {
        _runningSet = new RunningSet() ;
181     running_process = null ;
        genStaticProcessProperty() ;
183     //initial the variables
        //initial the scheduler variables
185     //initial the collections
        //ensure the collections are not null
187     if (ready == null) {
            ready = new ProcessCollection_fifo() ;
189     }
        return true ;
191 }
```

```
193 public int get_init_process_count() {
    int pcnt = 0 ;
195     pcnt += 1 ;
    pcnt += 1 ;
197     return pcnt;
}
199
public void init_order() throws ValidationException {
201     //initial the order of processes (using order defined in process DSL)
    ArrayList<SchedulerProcess> procList=new ArrayList<SchedulerProcess>();
203     {
        int processID = getProcessID("P") ;
205         if (processID >= 0) {
            //create new process in model
207             //SchedulerPanModel.p
            //create new process information in scheduler
209             SchedulerProcess P = new SchedulerProcess() ;
            //P.processID = (byte) processID ;
211             P.processID = processID ;

213             while (pcnt.size() < processID + 1) pcnt.add((byte) 0) ;
            pcnt.set(processID, (byte) (pcnt.get(processID) + 1));
215
            P.refID = getRefID("P") ;
217             P.P() ;
            //processList.set(processID, "P_0") ;
219             processList.set(processID, "P") ;

221             procList.add(P) ;
        }// else ignore this initial process
223     }

225     if (!procList.isEmpty()) {
        addProcessList(procList) ;
227         procList.clear() ;
    }
229     //---------------------------------------

231     {
        int processID = getProcessID("P") ;
233         if (processID >= 0) {
            //create new process in model
235             //SchedulerPanModel.p
            //create new process information in scheduler
237             SchedulerProcess P = new SchedulerProcess() ;
            //P.processID = (byte) processID ;
239             P.processID = processID ;

241             while (pcnt.size() < processID + 1) pcnt.add((byte) 0) ;
            pcnt.set(processID, (byte) (pcnt.get(processID) + 1));
243
            P.refID = getRefID("P") ;
245             P.P() ;
            //processList.set(processID, "P_0") ;
247             processList.set(processID, "P") ;

249             procList.add(P) ;
        }// else ignore this initial process
```

```
251         }
            if (!procList.isEmpty()) {
253             addProcessList(procList) ;
                procList.clear() ;
255         }
            //---------------------------------------
257     }

259     public void init() {
        }
261
        //------------------------ event handler -------------------------
263     public int select_process(int lastProcessID) throws ValidationException {
            SchedulerProcess target_process ;
265         {//GetProcess statement
                SchedulerProcess previous_running = running_process;
267             //1. Select process set
                if (lastProcessID < 0) {
269                 ArrayList<SchedulerProcess> runSet = ready.getProcessSet();
                    if (runSet != null) {
271                     _runningSet.dataSet = runSet ; //only get no remove
                        _putColIndex = (byte) getCollectionIndex("ready") ;
273                 } else {
                        if (_runningSet != null)
275                     _runningSet.clear() ;
                        return - 1;
277                 }
                }
279             //2 Get first process which has different processID to run
                int processID = select_process_to_run(lastProcessID) ;
281             if (processID < 0) {
                    return -1 ;
283             }
                //SchedulerProcess target_process = running_process ;
285             target_process = running_process ;
                if (lastProcessID >= 0) {
287             replace_running_process(_putColIndex,running_process,previous_running) ;
                }
289             //remove it from collection
                ready.removeProcess(target_process.processID) ;
291             //3 change properties
                //4 set running parameters
293                 _time_count = 0 ;
                    _time_slice = 0 ;
295         }//GetProcess statement

297         {//SetExecTime
                _time_slice = 3 ;   //runtime = true
299         }
            {//SetReturnCol
301             _putColIndex = (byte) getCollectionIndex("ready");//runtime = true
            }
303
            if (running_process != null)
305             return running_process.processID ;
            return -1;
307     }
```

```
309 public SchedulerProcess new_process(String procName) throws ValidationException {
        int index = 0 ;
311     for (String pName : SchedulerObject.processList) {
            if (pName.equals(procName)) {
313             if (!SchedulerObject.processInScheduler[index])
                break ;
315         }
            index ++ ;
317     }
        if (index >= SchedulerObject.processList.size()) {
319         SchedulerObject.processList.add(procName) ;
        }
321
        if (index > 255) {
323         for (int i = 0 ; i < 255 ; i ++) {
                if ( SchedulerObject.processList.get(i).equals("")) {
325                 index = i ;
                    SchedulerObject.processList.set(index, procName) ;
327                 break ;
                }
329         }
        }
331
        try {
333         new_process(procName, index, null) ;
            SchedulerObject.processInScheduler[index] = true ;
335         newP = (byte) index ;
        } catch (ValidationException e) {
337         e.printStackTrace();
        }
339
        return new_process(procName, -1, null) ;
341 }

343 public SchedulerProcess new_process(String procName, int processID,
        ArrayList<String> para) throws ValidationException {
345     //Util.print("--> new_process(" + procName + ")") ;
        SchedulerProcess new_process_target_process = new SchedulerProcess();
347     if (processID >= 0) {
            //target_process.processID = (byte) processID ;
349         new_process_target_process.processID = processID ;
            while (pcnt.size() < processID + 1) pcnt.add((byte) 0) ;
351         pcnt.set(processID, (byte) (pcnt.get(processID) + 1));
        } else {
353         return null ;
        }
355     new_process_target_process.initProcess(procName, para) ;
        config_new_process(new_process_target_process) ;
357     return new_process_target_process ;
    }
359
    public void config_new_process (SchedulerProcess target_process)
361     throws ValidationException {
        {//MoveProcess target_process
363         if (target_process != null) {
                remove_process(target_process.processID) ;
365             ready.put(target_process) ;
```

```
367             if (running_process != null) {
                    if (running_process.processID == target_process.processID){
369                     running_process = null;
                    }
371             }
            }
373     }
    }
375
    public void addProcessList(ArrayList<SchedulerProcess> procList)
377     throws ValidationException { //called by init_order
        ArrayList<SchedulerProcess> AL_ready=new ArrayList<SchedulerProcess>();
379     for (SchedulerProcess target_process : procList) {
            {//MoveProcess target_process
381             AL_ready.add(target_process) ;
            }
383
            //initProcessList = false
385         initprocesslist.add(getInstance(target_process)) ;
        }
387
        if (!AL_ready.isEmpty() )
389     ready.put(AL_ready) ;
    }
391 /*
    public void clock(GenerateCode _code) throws ValidationException{
393     this._code = _code ;
        clock() ;
395 }
    */
397 public void clock() throws ValidationException{
        inc_time() ; //increase all clock including _time_count
399     check_running_time_to_put_running_process() ; //to end the time slice
        if (_runningSet != null) {
401         _runningSet.clear();
        }
403     if (!hasGenTemplate) {
            if (running_process == null) {
405             if (select_process(-1) < 0) {
                //Util.print("No running process");
407             }
            }
409     }
    }
411 public void preTake() throws ValidationException{}
    public void postTake() throws ValidationException {}
413 public int terminate_process(String procName) throws ValidationException {
        //default missing handler
415     SchedulerProcess target_process = null ;
        Util.print("--> Terminate process: " +procName+" default processing");
417     int id = 0 ;
        for (String procN : processList) {
419         if (procN.contains(procName)) {
                SchedulerProcess terminate_target_process=findProcessByID(id);
421             target_process = terminate_target_process ;
                if (terminate_target_process != null) {
423             return terminate_process(id) ;
                }
```

```
425             }
                id ++ ;
427         }
            Util.print("--> Cannot find process: " + procName) ;
429         return -1 ;
        }
431     public int terminate_process(int processID) throws ValidationException {
            endP = (byte) processID ;
433         if (running_process != null) {
                if (running_process.processID == processID) {
435                 SchedulerObject.processInScheduler[processID] = false ;
                    running_process = null ;
437                 return processID;
                }
439         }
            ready.removeProcess(processID);
441
            return processID ;
443     }

445     //--------------- encoding function -----------------------------
        public void encode(DataWriter _writer) {
447         //_writer.writeInt(_schselopt);
            if (running_process == null)
449             _writer.writeBool(false);
            else {
451             _writer.writeBool(true);
                running_process.encode(_writer);
453         }
            _writer.writeByte(_putColIndex) ; //for running process (selection)
455
            //could be duplicated!
457         _writer.writeInt(_time_count);
            _writer.writeInt(_time_slice);
459
            ready.encode(_writer) ;
461     }
        public boolean decode(DataReader _reader) {
463         //_schselopt = _reader.readInt();
            clearProcessInScheduler() ;
465         if (_reader.readBool()) {
                if (running_process == null)
467                 running_process = new SchedulerProcess() ;
                running_process.decode(_reader) ;
469             processInScheduler[running_process.processID] = true ;
            } else {
471             running_process = null ;
            }
473         _putColIndex = (byte) _reader.readByte() ; //for running process
            //could be duplicated!
475         _time_count = _reader.readInt();
            _time_slice = _reader.readInt();
477
            ready.decode(_reader) ;
479         return true;
        }
481     public int getRunningSetSize(){
            return _runningSet.getSize();
```

```
483 }
    public void encodeRunningSet(DataWriter _writer){
485     _runningSet.encode(_writer);
    }
487 public boolean decodeRunningSet(DataReader _reader) {
        _runningSet.decode(_reader);
489     return true ;
    }
491 protected void clearProcessInScheduler() {
        for (int i = 0 ; i < 128; i ++)
493     processInScheduler[i] = false ;
    }
495
    /* ---------------------- utility function */
497 public String getInstance(SchedulerProcess process) {
        return pcnt.get(process.processID) + "";
499 }
    public int getRunningInstance() {
501     if (running_process == null)
            return -1 ;
503     else
            return pcnt.get(running_process.processID);
505 }
    public int getRunningID() {
507     if (running_process == null)
            return -1 ;
509     else
            return running_process.processID ;
511 }
    public static void printProcessInScheduler(){
513     for (int i=0 ; i< 128 ; i++){
            System.out.print(processInScheduler[i] + ", ");
515     }
        System.out.println();
517 }
    public void printProcessInstance(){
519     for (int i=0 ; i < pcnt.size() ; i++){
            System.out.print(pcnt.get(i) + ", ");
521     }
        System.out.println();
523 }
    public int addProcessList(String pName){
525     //return the index of new process name in process list
        processList.add(pName) ;
527     return processList.size() - 1 ;
    }
529 public int isNull(SchedulerProcess process) {
        if (process == null)
531         return 1 ;
        else
533         return 0 ;
    }
535 public static int getProcessID(String procName){
        int id = 0;
537     for (String pName : processList){
            if (pName.equals(procName) && !processInScheduler[id]){
539             //processInModel[id] = true ;
                processInScheduler[id] = true ;
```

```
541                 return id ;
            }
543             id ++ ;
        }
545     if (processList.size() < 128) {
            processList.add(procName) ;
547         id = processList.size() - 1 ;
            processInScheduler[id] = true;
549         return id ;
        }
551     return -1 ;
    }
553 public int getCollectionIndex(String collectionName) {
        int numCol = 0 ;
555     switch (collectionName) {
            case "ready" :
557             return numCol + 0 ;
            default :
559             Util.print("Put back collection error") ;
                return -1 ;
561     }
    }
563 public int getNumberProcessCollection() {
        int result = 0 ;
565     //ready : 0
        result += 1 ;
567     return result ;
    }
569 //return collection contains process -> needs to be considered
    public int getProcessCollectionID(int processID) {
571     int numCol = 0 ;

573     if (ready.hasProcess(processID) > 0)
        return 0 + numCol ;
575     return -1 ;
    }
577 public boolean isTimer() {
        //boolean hasClockEventHandler = false ;
579     //boolean hasPeriodicProcess = false ;
        //boolean runTime = true ;
581     //has clock data type = false
        if (_time_slice != 0)
583     return true ; //(_time_slice > 0)
        else
585     return false ;
    }
587 public SchedulerProcess findProcessByID(int processID) {
        SchedulerProcess proc = null ;
589     if (running_process != null)
            if (running_process.processID == processID)
591             return running_process ;
        proc = ready.getProcess(processID);
593     if (proc != null) return proc ;
        return null ;
595 }
    public ArrayList<SchedulerProcess> findProcessByrefID(int refID) {
597     ArrayList<SchedulerProcess> result = new ArrayList<SchedulerProcess>();
        if (running_process != null)
```

```
599             if (running_process.refID == refID)
                    result.add(running_process) ;
601         ArrayList<SchedulerProcess> temp = new ArrayList<SchedulerProcess>();
            temp = ready.findProcessByrefID(refID);
603         if (temp != null)
                result.addAll(temp);
605         return result ;
        }
607     public void remove_process(int processID) {
            ready.removeProcess(processID);
609     }
        public int isEmpty() {
611         if (ready.isEmpty() > 0)
                return 1 ;
613         return 0 ;
        }
615     public int hasProcess(String processName) {
            if (running_process != null)
617             if (getStaticPropertyObject(running_process.refID).pName.trim()
                .equals(processName.trim()))
619                 return 1 ;

621         int result = 0 ;
            int processID = 0 ;
623         for (String pName : processList) {
                if (pName.trim().equals(processName.trim())) {
625                 result = ready.hasProcess(processID);
                    if (result > 0) return result ;
627             }
                processID ++ ;
629         }
            return result ;
631     }
        public String getSchedulerName() {
633         return "roundrobin_roundrobin" ;
        }
635     public void print_all(){
            Util.print("- SCH OPT: " + _schselopt + "/" + (_schnumopt - 1));
637         Util.print("- Time_count/Time_slice: "+_time_count +"/"+_time_slice) ;
            Util.print("- Running process: " ) ;
639         if (running_process == null)
                Util.print("Null") ;
641         else {
                running_process.print();
643         }
            Util.print("- Running set: " ) ;
645         if (_runningSet == null)
                Util.print("Null") ;
647         else {
                _runningSet.print() ;
649         }
            System.out.println(processList);
651         printProcessInScheduler() ;
            System.out.print("- Collection: ready : ");
653         ready.print() ;
        }
655     //--------------- running function -------------------------------
        public void executeProcess(PromelaProcess proc, int processID,
```

```
657  ArrayList<String> para) throws ValidationException {
        SchedulerProcess p = new SchedulerProcess() ;
659     //p.processID = (byte) processID ;
        p.processID = processID ;
661     newP = processID ;

663     //initialize the process
        p.initProcess(proc.getName(), para);
665     //Util.print("New process " + p) ;
        processInScheduler[processID] = true ;
667     while (pcnt.size() < processID + 1) pcnt.add((byte) 0) ;
        pcnt.set(processID, (byte) (pcnt.get(processID) + 1) ) ;
669     config_new_process(p) ;
    }
671 public int select_process_to_run(int lastProcessID) {
        if (_runningSet.isEmpty() == 1)
673     return -1 ;

675     SchedulerProcess temp = _runningSet.getFirstProcess(lastProcessID) ;
        if (temp != null) {
677         running_process = temp ;
            _runningSet.getNextProcess(running_process) ;
679         return running_process.processID ;
        }
681     else return - 1;
    }
683 public boolean replace_running_process(byte collectionIndex,
     SchedulerProcess running_process, SchedulerProcess previous_running) {
685     byte numCol = 0 ;
        if (collectionIndex == (byte) (0 + numCol)) {
687         if (ready != null)
                ready.replace(running_process, previous_running) ;
689             return true ;
        }
691     return false ;
    }
693 public void put_running_process(byte collectionIndex) {
        if (running_process != null) {
695         put_process(running_process, collectionIndex) ;
        }
697 }
    public boolean put_process(SchedulerProcess proc, byte collectionIndex) {
699     byte numCol = 0 ;
        if (collectionIndex == (byte) (0 + numCol)) {
701         if (ready != null)
                ready.put(proc);
703             return true ;
            }
705     return false ;
    }
707 //--------------- timed function -------------------------------
    public void time_out(){
709     //just only add time
        add_time(_time_slice - _time_count) ;
711     _time_count = _time_slice ;
        //check_running_time_to_put_running_process() ;
713 }
    public void inc_time() {
```

```
715     if (_time_slice != 0) {
            if (_time_count == _time_slice)
717             _time_count = 1 ;
            else
719             _time_count ++ ;
        }
721     add_time(1) ;
    }
723 public void dec_time() {
        if (_time_slice != 0) {
725         if (_time_count == 0)
                _time_count = _time_slice - 1 ;
727         else
                _time_count -- ;
729     }
        sub_time(1) ;
731 }
    public void add_time(int time) {
733     //clock for periodic process
        ready.add_time(time) ;
735     if (running_process != null)
            running_process.add_time(time) ;
737 }
    public void sub_time(int time) {
739 //clock for periodic process
        ready.sub_time(time) ;
741     if (running_process != null)
        running_process.sub_time(time) ;
743 }
    public boolean check_running_time_to_put_running_process(){
745     if (_putColIndex != -1) {
            //for putting the running process to the destination collection
747         //need to call select_process_set() to get other process
            if (_time_count == _time_slice && _time_slice > 0) {
749             if (running_process != null){
                    put_running_process(_putColIndex) ;
751                 //_runningSet.dataSet.clear();
                    running_process = null ;
753                 return true ;
                }
755         }
        }
757     return false ; //process still running
    }
759 //---------------interface function---------------------
    public boolean sch_api(String funcName, String paraList)
761     throws ValidationException {
        switch (funcName) {
763         default:
            System.out.println("Error calling Scheduler API function");
765         return false ;
        }
767 }
    public boolean sch_get(String processName, String property) {
769     if (processName.trim().equals("scheduler")) {
            //scheduler data variable
771         switch (property) {
                default:
```

```
773                 System.out.println("Error getting scheduler property");
            }
775         } else {
            ArrayList<SchedulerProcess> aProcess = findProcess(processName) ;
777         if (aProcess.size() == 1) {
                SchedulerProcess process = aProcess.get(0) ;
779             switch (property) {
                    default:
781                 System.out.println("Error getting process property");
                }
783         }
        }
785     return false ;
    }
787
    //----------------- genStaticProcessProperty ------------------
789 public void genStaticProcessProperty(){
        StaticProperty sP ;
791     //0
        sP = new StaticProperty() ;
793     sP.refID = 0 ;
        sP.pName = "P" ;
795     staticPropertyList.add(sP) ;
    }
797
    //--------------- data structure ---------------------------
799 public SchedulerProcess running_process ;


801
    public int size = 0 ;
803 public byte _putColIndex = -1; //for replacement
    public RunningSet _runningSet; //temporary store the running set
805 public int _time_count = 0 ;
    public int _time_slice = 0 ;
807 //public int _time = 0 ;
    ProcessCollectionBase ready ;
809 //--------------- genUtilitiesFunctions --------------------------
    public boolean[] getProcessCheckList() {
811     boolean result[] = new boolean[128] ;
        if (running_process != null)
813         result[running_process.processID] = true ;
        for (ArrayList<SchedulerProcess> procList : ready.dataSet)
815         for (SchedulerProcess proc : procList)
                result[proc.processID] = true ;
817     return result ;
    }
819 public ProcessSet getProcessSet(String psName) {
        //scheduler data
821     if (psName == "ready")
            return ready ;
823     return null ;
    }
825 public int getSize() {
        size = 0; //4 ; //schselopt
827     //scheduler variables data
        size += 8 ; //_time_count, _time_slice (system)
829     size += 1 ; //running_process != null ?
        if (running_process != null)
```

```
831             size += running_process.getSize() ;
        size += 1 ; //_putColIndex -> for replacement
833     //no contains refines collections
        size += ready.getSize() ;
835     return size ;
    }
837 //--------------- verification structure --------------------------
    public boolean schedulerCheck() {return false ; }
839 public void initSchedulerState(SchedulerState schState, int depth) {}
    public boolean stateCheck() {return false;}
841 public boolean collectState() { return false ;}
    public void printAnalysisResult(PrintWriter out) {
843     if (out != null) out.println("No Analysis result");
            System.out.println("No Analysis result") ;
845     }
    }
```