IEICE TRANS. INF. & SYST., VOL.E102–D, NO.7 JULY 2019

PAPER

Model Checking in the Presence of Schedulers Using a Domain-Specific Language for Scheduling Policies*

Nhat-Hoa TRAN[†a)], **Yuki CHIBA**[††b)], *Nonmembers*, *and* **Toshiaki AOKI**[†††c)], *Member*

SUMMARY A concurrent system consists of multiple processes that are run simultaneously. The execution orders of these processes are defined by a scheduler. In model checking techniques, the scheduling policy is closely related to the search algorithm that explores all of the system states. To ensure the correctness of the system, the scheduling policy needs to be taken into account during the verification. Current approaches, which use fixed strategies, are only capable of limited kinds of policies and are difficult to extend to handle the variations of the schedulers. To address these problems, we propose a method using a domain-specific language (DSL) for the succinct specification of different scheduling policies. Necessary artifacts are automatically generated from the specification to analyze the behaviors of the system. We also propose a search algorithm for exploring the state space. Based on this method, we develop a tool to verify the system with the scheduler. Our experiments show that we could serve the variations of the schedulers easily and verify the systems accurately.

key words: *concurrent system, model checking, domain-specific language, system behaviors, scheduler*

1. Introduction

The behaviors of a system depend on its scheduler, which determines the execution orders of the processes. In model checking, a scheduling policy is closely related to the algorithm that searches all of the system states. If we consider all the behaviors of the processes by interleaving them when verifying a concurrent system, some spurious counterexamples may be found because following the scheduling policy these behaviors may not happen. This is an over-approximation approach and can produce false positives. To avoid this problem, the scheduling policy needs to be taken into account during the verification to limit the search space and increase the accuracy.

In reality, many kinds of schedulers, which adopt different strategies, are used in practical systems. These policies are often different from 'textbook' strategies. For example, in the automotive operating system (OS) named OSEK/VDX [5], an application includes multiple concurrent tasks that are run based on fixed priority with mixed preemption scheduling policy; the Linux scheduler supports different policies for non-real-time and real-time jobs based on their priorities, such as *round-robin* and *first-in-first-out*. There are methods, such as [6]–[8], for verifying sequential or concurrent software systems. However, these methods are difficult to apply directly to verify systems with schedulers because the behaviors that these methods deal with are different from those of practical systems.

In fact, with the existing modeling languages like Promela [10], which is used by Spin model checker [3], the scheduling problems require the encoding of the whole system (i.e., the processes and the scheduler) [1], [2]. This approach is both time-consuming and error-prone. Moreover, in current studies, the scheduling policies are only realized in the model of the system and difficult to change.

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 scheduling policies 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 system.

We propose a method to verify the system under different scheduling policies. The method includes 1) a specification language for the processes, 2) a DSL for specifying the scheduling strategy, and 3) an algorithm to explore the state space to verify the system behaviors. In our approach, the main aim of the DSL is to provide a high-level support for the succinct specification of various scheduling policies. All of the necessary information for the subsequent analysis of the system is generated automatically. According to this method, we have implemented a tool named SSpinJa with the back-end extended from SpinJa [9], a re-implementation of the core of Spin in Java. Several experiments were conducted with our approach. The results show that our framework can verify the systems with different scheduling policies accurately, flexibly, and easily.

The main contributions included in this approach are 1) proposing a DSL for specifying the scheduling policies, 2)

Manuscript received December 2, 2017.
Manuscript revised December 4, 2018.
Manuscript publicized March 29, 2019.

[†]The author is with Software Engineering Department, National University of Civil Engineering, 55 Giai Phong Street, Hanoi, Vietnam.
[††]The author is with the Advanced R & D Dept., Tokyo Office, DENSO CORPORATION, Tokyo, 103–6015 Japan.
[†††]The author is with the School of Information Science, Japan Advanced Institute of Science and Technology (JAIST), Nomi-shi, 923–1292 Japan.

*This paper is an extension of [29]

a) E-mail: hoatn@nuce.edu.vn
b) E-mail: yuki_chiba@denso.co.jp
c) E-mail: toshiaki@jaist.ac.jp

DOI: 10.1587/transinf.2017EDP7391

proposing a search algorithm based on the behaviors of the scheduler, and 3) implementing a tool for verifying systems under various scheduling policies. This paper is an extension of our work originally reported in [29]. In particular, the DSL is described in more detail to specify the behaviors of the system with the formal definitions and the semantic of the language; we also introduce a new case study to verify the schedulability of real-time systems.

The rest of the paper is organized as follows: Sect. 2 gives the detail of the approach. Section 3 introduces the DSL for specifying the scheduling policies. Section 4 presents our approach to model checking systems with schedulers and the implementation of our method. In Sect. 5, we introduce four case studies with several experiments. The discussion is shown in Sect. 6. Section 7 presents the related work. Finally, the conclusion and future work are given in Sect. 8.

2. The Approach

We propose a method to verify the system behaviors under different scheduling policies. The approach is shown in Fig. 1. Our ideas are as follows.

Firstly, to model the system, we separate the scheduler from the processes. This approach is different from the existing ones, such as [1], [2], which encode both the scheduler and the processes into the same model. With this approach, we can flexibly change the policy and reuse the behaviors of the processes and the scheduling policy.

Secondly, we use Promela as the based modeling language to specify the concurrent behaviors of a set of processes in the system. In order to deal with the scheduling policy, several API functions are introduced in this modeling language (1) to support the interaction between the processes and the scheduler. These functions are for 1) executing a new process with the initial values for the process attributes (e.g., priority), 2) accessing the information managed by the scheduler[†], and 3) performing the user-defined functions (called interface functions) declared in *interface* part (*) to define the process behaviors related to the scheduling policy.

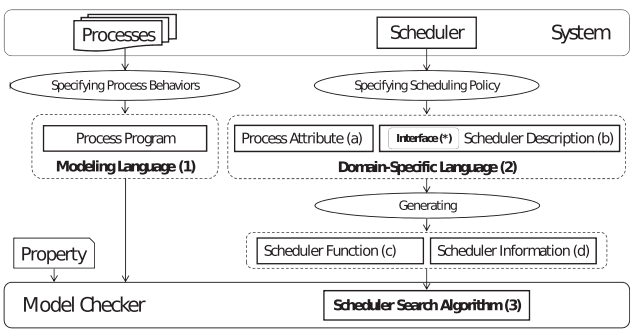

Fig. 1 An approach to verifying a system with the scheduling policy.

[†]These functions are necessary for implementing the algorithms like slack stealing [11] in the scheduling policy.

Thirdly, to facilitate the scheduling strategies, we propose a DSL (2) to describe the policies with the attributes of the processes to handle the scheduling tasks. The main role of the scheduler is a) to select a process for the execution, b) to manage the processes using their attributes (e.g. priority, deadline), c) to change their execution statuses (e.g. blocks a process), and d) to manage the time. In our approach, the DSL aims to provide high-level language to specify various policies easily. By focusing on a domain, using a DSL is easier than using an existing modeling language (e.g. Promela). In the DSL, we separate the attributes of the processes and the behaviors of the scheduler for flexibly changing them. The attributes of the processes are defined in *process attribute* (a), and the behaviors of the scheduler are defined in *scheduler description* (b).

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 define collections for storing the processes. At a time, there may be many candidates for the execution (e.g. the processes with the same priority). We deal with this problem by ordering the processes based on their attributes. The processes are partially ordered for the selection. The order of the processes in a collection is either defined by a comparison function, follows the `LIFO/FIFO` strategy, or uses both of these ordering methods. The operations *getting* a process from and *putting* a process into a collection rely on the ordering method used by the collection. Of course, we also support the collection without using any ordering method (that means the processes in this collection have the same order).

We deal with the behaviors of the scheduler based on handling events (called scheduling events), which are specified in *scheduler description* (b). Three predefined events (`new_process`, `select_process`, and `clock`) are introduced. The event `new_process` occurs when a new process arrives to the system. The event `select_process` is for selecting a process to run. The event `clock` is a timer event, which happens following the occurrence of each action of the process. Each event above is handled by an event handler. We can also define other events raised by the current process. These events are specified in the *interface* part (called interface functions). To handle the scheduling events, several statements are introduced to change the running status of the processes, to change the attributes of the processes, and to select a process for the execution.

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

icy to accurately verify the system.

Finally, to deal with the time related to the behaviors of the system, we use discrete time by considering a transition from one state of the system to another state as taking one time unit (each transition corresponds to one time unit). We follow the timed Kripke structure [12] to model the system. During exploring the state space, the search can reach a state that has already been visited. At this time, a period (a loop) of the system is determined. For example, suppose that the system has only one periodic process with period 4; this process takes 2 time units to finish its tasks; the period considered is only 4 time units because the state of the system at time 4 is the same as that at time 0. In fact, the search can reach several states that have been visited. That means the Kripke structure may have several loops.

We follow the compilation approach to explore the system state space. A converter was built to generate *scheduler function* (c) and *scheduler information* (d) from the specification of the scheduling policy. The *scheduler function* is used to perform the scheduling tasks and the *scheduler information* is for determining the state of the system. Based on the description of the scheduler, a search algorithm (3) is realized. The states to be searched are changed according to the description of the scheduler.

3. DSL for Specifying the Scheduling Policies

A DSL is a limited expressive language that focuses on a particular domain. Using a DSL has several advantages, such as simplification and productivity in comparison with using a general purpose language. In our approach, we propose a DSL to specify the scheduling policy. The grammar of the language is shown in Appendix. In this section, we use the *priority* strategy as an introductory example, which is depicted in Fig. 2. 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.

3.1 The Process Behaviors

The behaviors of the process are specified in the *process program* using the modeling language. To support the interaction between the processes and the scheduler, we introduce several API functions, such as `sch_exec`, `sch_api_get`, `sch_api_set`, `sch_api` and `sch_api_self`. The function `sch_exec` is used for executing a new process, possibly with the initial values for the process attributes. The functions `sch_api_get` and `sch_api_set` are used for accessing the information managed by the scheduler. The functions `sch_api` and `sch_api_self` are used for performing the interface functions to define the process behaviors related to the scheduling policy.

The introductory example indicates a system with two processes (as shown in Fig. 2 a)). In this example, when `a + b` < 100000, process `P` repeatedly increases the value of variable `a`, while process `Q` increases the value of variable `b`. Statement `sch_api_self` is used to call an interface function named `terminate` defined in the `interface` part of the *scheduler description* (Fig. 2 c)) to terminate itself. The `sch_exec` statements used in `init` part determine that at the starting time, process `P` and `Q` are executed.

```
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 {
 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

Fig. 2 An introductory example.

3.2 Specifying the Scheduling Policy

To specify the scheduling policy, our DSL provides two types of specifications[†]: one for the process attributes and the other for the scheduler behaviors as <*ProcDSL*> and <*SchDSL*> of the grammar.

Process attributes: The process attributes are used to deal with the scheduling tasks (especially for selecting a process to run). The specification for the process attributes (<*ProcDSL*>) includes (a) the definition of the attributes (<*ProcAttr*>), (b) the initial values for the attributes of each process (<*Process*>), (c) the declaration of the periodic/sporadic processes[††] (<*ProcConf*>), and (d) the execution order of the processes at the starting time (<*ProcInit*>).

In the introductory example, only an attribute

[†]See the Appendix for the detailed language grammar.

[††]Each periodic process (<*PeriodicP*>) is determined using a period, while a sporadic process (<*SporadicP*>) is defined using the length of time that the process becomes ready and a number indicating the maximum instances of the process. We handle these processes by generating the corresponding variables and manage their execution following the time changing by the `clock` event.

(`priority`) is defined (as shown in Fig. 2 b)). The initial value of this attribute for each process is determined by a template (`proctype`). In this example, the priority of `P` is higher than that of `Q`. The `init` part indicates the execution order of the processes following a partially ordered set. This example shows that process `P` and `Q` are executed at the same time. We note that the `init` part in the *process program* determines the existence of the processes at the starting time (using the `sch_exec` statement); the `init` part of the *process attribute* defines the execution order of these processes. This order affects the selection of the scheduler.

Scheduler behaviors: The scheduler behaviors defined as <*SchDSL*> of the grammar are specified in the *scheduler description*, which consists of the definition of the scheduler (<*SchDef*>) and the definition of the methods for ordering the processes (<*OrdDef*>). The definition of the scheduler includes: (a) the variables used by the scheduler (<*VarDef*>), (b) the data used by the scheduler (<*DatDef*>), which contains the definition of the process collections (<*ColDef*>), (c) the handlers for the events (<*HandlerDef*>) and (d) the interface functions (<*InterDef*>).

Several statements (defined as <*Stm*> of the grammar) are introduced to specify the behaviors of the scheduler.

- The values of the variables used by the scheduler and the values of the attributes of the processes can be changed using statements: <*SetTime*> (to indicate the running time for the current process), <*SetCol*> (to determine the collection that contains the process after its execution time) and <*Change*> (to change the value of a variable or the value of an attribute of a process).
- We can update the running status of the process using statements <*Move*>, <*Remove*> and <*Get*> by changing the collection containing the process.
- Statement <*New*> is for executing a new process.
- The language also supports conditional statement (<*If*>) and loop over a collection statement (<*Loop*>).
- Statements <*Assert*> and <*Print*> are used for tracking the behaviors of the scheduler.

The processes selected from a collection for the execution (using statement <*Get*>) is determined by the ordering method used by the collection. We provide a mechanism to order the processes by defining a comparison function (<*CompDef*>), which is used to compare two processes in the 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. Based on this fact, the processes in the collection are ordered.

In the introductory example, a collection named `ready` is used to store the processes, which are ordered following a function named `priorityOrder` (Fig. 2 c)). The process with higher priority will be placed in front of the other. If two processes have the same priority, they will have the same order. In this example, the processes with the highest priority will be selected for the execution.

In order to perform the scheduling tasks, we handle the scheduling events. Three predefined events (`new_process`, `select_process`, and `clock`) are introduced as explained before. The corresponding event handlers for these events are specified as <*EventDef*> of the grammar. Other events raised by the current process using the interface functions (<*InterFunc*>) can be defined in the `interface` part (<*InterDef*>). The scheduling events are specified using the DSL statements (<*Stm*>).

In the example, the scheduler handles two events (`new_process` and `select_process`). When a new process (indicated by `target`) arrives, if its priority is greater than that of the current process (indicated by `running_process`), the current process will be preempted by putting it to the `ready` collection (using <*Move*> statement). This makes the scheduler select another process to run. To do that, the scheduler obtains a process from this collection. That behavior is specified in the `select_process` event handler using statement <*Get*>. A function named `terminate` is declared in the `interface` part for terminating a process (using statement <*Remove*>). This function is called by function `sch_api_self` in the *process program*.

3.3 Formal Definitions

We now give the formal definitions for specifying 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.

Definition 1 (Process state): A process state is a tuple $\langle \sigma_g, \sigma_l \rangle$, where $\sigma_g : \mathcal{X}_g \to \mathcal{V}$ and $\sigma_l : \mathcal{PID} \to (\mathcal{X}_l \to \mathcal{V})$.

In this definition, σ_g is the mapping from the set of the global variables ($\mathcal{X}_g \subset \mathcal{X}$) to the set of values ($\sigma_g$ is called the global state of a process), and σ_l is the mapping from the set of process identifiers to the mappings from the local variables of a process ($\mathcal{X}_l \subset \mathcal{X}$) to the set of values ($\sigma_l$ is called the local state of a process). These variables are defined in the *process program* (e.g. in Fig. 2 a)), `a` and `b` are the global variables; there is no local variable). We use $\mathcal{S}_{proc}$ to denote the set of process states.

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

- *normal* is a set of normal actions described in Promela.
- *get* is an action that corresponds to function `sch_api_get` to access the information of the scheduler.
- *scheduling* = $\{api, exec\}$ is a set of actions performed by the current process using the API functions.
  - An *api* action corresponds to the `sch_api` (or `sch_api_self`) function called by the current process (e.g. `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())` represents an *exec* action).

These functions are handled by the event handlers and

the interface functions specified in the *scheduler description*. For instance, in the introductory example, function `sch_api_self(terminate)` is handled by the interface function named `terminate`.

Definition 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.

We use $\Sigma_p = \langle \sigma_g, [\sigma_{l_1}, \ldots, \sigma_{l_i}, \ldots, \sigma_{l_m}]\rangle$ to denote the state of the set of processes in the system, where $\langle \sigma_g, \sigma_{l_i}\rangle \in \mathcal{S}_{proc}$ is the state of ith process; $[\sigma_{l_1}, \ldots, \sigma_{l_i}, \ldots, \sigma_{l_m}]$ is a sequence of the local states of the processes, and $m \in \mathbb{N}$ is the number of the processes in the system. These local states are ordered by the identifiers of the processes.

Definition 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 status of the processes (such as *ready*, *blocked*). In the example (as shown in Fig. 2), 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. 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 the set of collections.

Definition 4 (Scheduler state): A scheduler state is a tuple $\langle \sigma_s, (C_1, \ldots, C_i, \ldots, C_k), P_r\rangle$, where

- $\sigma_s : \mathcal{X}_s \cup \mathcal{X}_c \cup \{run, tslice, rcol\} \to \mathcal{V} \cup \{\perp\}$ is a mapping from the set of normal variables ($\mathcal{X}_s \subset \mathcal{X}$) defined in the policy, the set of clock variables ($\mathcal{X}_c \subset \mathcal{X}$) used by the scheduler, and the set of predefined 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 stores the process after finishing its execution time.
- $C_i \in \mathcal{COL}$ is a collection, $i = \overline{1..k}, k \in \mathbb{N}$;
- $P_r \subseteq \mathcal{PID}$ is a set of process identifiers represent the set of processes that can be run.

The variables used in the scheduling policy can be defined in the *scheduler description*. The variables: *run*, *tslice*, and *rcol* are predefined. In some states, the values of these variables are non-determined ($\perp$). For instance, $run = \perp$ means that there is no currently running process.

The set of collections $\{C_i\}, i = \overline{1..k}$ are defined in the *scheduler description*. P_r is predefined 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 an action of the process) in searching the state space (see Sect. 4 for more details).

In the example (as depicted in Fig. 2 c)), only one collection (`ready`) is defined; there is no variable. The *time slice* and the collection containing the currently running process after its execution time are not determined (i.e. using statement <*SetTime*> and <*SetCol*>). One of the scheduler states 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 5 (System state): A system state is a tuple $\Sigma = \langle \Sigma_p, \Sigma_s\rangle$, where $\Sigma_p = \langle \sigma_g, [\sigma_{l_1}, \ldots, \sigma_{l_i}, \ldots, \sigma_{l_m}]\rangle$ is the state of the set processes in the system and $\Sigma_s = \langle \sigma_s, (C_1, \ldots, C_k), P_r\rangle$ is a scheduler state.

The system state is derived from the set of processes and the state of the scheduler. For the convenience of writing, we use both $\langle \Sigma_p, \Sigma_s\rangle$ and $\langle\langle \sigma_g, [\sigma_{l_1}, \ldots, \sigma_{l_i}, \ldots, \sigma_{l_m}]\rangle, \Sigma_s\rangle$ to denote a system state. We use $\mathcal{S}_{sys}$ to represent the set of system states.

Let $\mathcal{L}_{sch} = \{select, clock, new, inter\}$ be a set of labels that represent 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 corresponds to the event raised by an interface function called by the current process.

Definition 6 (System): A system is a tuple $\langle \Sigma_{sys}, L_{sys}, T_{sys}, \Sigma_0\rangle$, where $\Sigma_{sys} \subseteq \mathcal{S}_{sys}$ is a set of system states, $L_{sys} \subseteq (\mathcal{L}_{proc} \cup \mathcal{L}_{sch})$ is a set of labels, $T_{sys} \subseteq \Sigma_{sys} \times L_{sys} \times \Sigma_{sys}$ is a set of transitions, and $\Sigma_0 \in \Sigma_{sys}$ is the initial state.

Note that L_{sys} 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_{sys} represents the transition relation between the system states. Each 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 get the information of the system.

- Function $getCol : \mathcal{PID} \to \mathcal{COL}$ to determine the collection that contains the process. For instance, 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 $x \gtrsim y$ with $\gtrsim = \succ \cup \sim$.
  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 uses $priority$ 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**

$$\langle \Sigma_p, \langle \sigma_s, (C_1, \ldots, C_k), P_r\rangle\rangle \xrightarrow{v=\langle exp\rangle} \langle \Sigma_p, \langle \sigma_s[[\![exp]\!]/v], (C_1, \ldots, C_k), P_r\rangle\rangle$$

  where v is a variable used by the scheduler, $\sigma_s[[\![exp]\!]/v]$ means replacing the value of v by the value of $[\![exp]\!]$ obtaining 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 policy.

- **Remove a process**

$$\langle \Sigma_p, \langle \sigma_s, (C_1, \ldots, C_i, \ldots, C_k), P_r\rangle\rangle \xrightarrow{rem(p)} \langle \Sigma'_p, \langle \sigma_s, (C_1, \ldots, C'_i, \ldots, C_k), P'_r\rangle\rangle$$

  where $\Sigma_p = \langle \sigma_g, [\sigma_{l_1}, \ldots, \sigma_{l_{j-1}}, \sigma_{l_j}, \sigma_{l_{j+1}}, \ldots, \sigma_{l_m}]\rangle$ with σ_{l_j} is the local state of process p, $\Sigma'_p = \langle \sigma_g, [\sigma_{l_1}, \ldots, \sigma_{l_{j-1}}, \sigma_{l_{j+1}}, \ldots, \sigma_{l_m}]\rangle$, $getCol(p) = C_i$ and $C'_i = \langle P_i - \{p\}, \succ_i, \sim_i\rangle$ with $C_i = \langle P_i, \succ_i, \sim_i\rangle$ 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.

- **Move a process to a collection**

  - If p is a new process[†] then

$$\langle \Sigma_p, \langle \sigma_s, (C_1, \ldots, C_i, \ldots, C_k), P_r\rangle\rangle \xrightarrow{mov(p,C_i)} \langle \Sigma_p, \langle \sigma_s, (C_1, \ldots, C'_i, \ldots, C_k), P_r\rangle\rangle$$

  where $C'_i = \langle P_i \cup \{p\}, \succ_i, \sim_i\rangle$ with $C_i = \langle P_i, \succ_i, \sim_i\rangle$.
  - If p is the currently running process then

$$\langle \Sigma_p, \langle \sigma_s, (C_1, \ldots, C_i, \ldots, C_k), P_r\rangle\rangle \xrightarrow{mov(p,C_i)} \langle \Sigma_p, \langle \sigma_s[\perp /run], (C_1, \ldots, C'_i, \ldots, C_k), P_r\rangle\rangle$$

  where $C'_i = \langle P_i \cup \{p\}, \succ_i, \sim_i\rangle$ with $C_i = \langle P_i, \succ_i, \sim_i\rangle$.
  - If p belongs to collection C_i then

$$\langle \Sigma_p, \langle \sigma_s, (C_1, \ldots, C_i, \ldots, C_j, \ldots, C_k), P_r\rangle\rangle \xrightarrow{mov(p,C_j)} \langle \Sigma_p, \langle \sigma_s, (C_1, \ldots, C'_i, \ldots, C'_j, \ldots, C_k), P_r\rangle\rangle$$

  where $getCol(p) = C_i$ with $C_i = \langle P_i, \succ_i, \sim_i\rangle$, $C'_i = \langle P_i - \{p\}, \succ_i, \sim_i\rangle$, and $C'_j = \langle P_j \cup \{p\}, \succ_j, \sim_j\rangle$ with $C_j = \langle P_j, \succ_j, \sim_j\rangle$.
  The function is defined by the statement <Move> of the grammar.

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

$$\langle \Sigma_p, \langle \sigma_s, (C_1, \ldots, C_i, \ldots, C_k), P_r\rangle\rangle \xrightarrow{select(C_i)} \langle \Sigma_p, \langle \sigma_s, (C_1, \ldots, C_i, \ldots, C_k), P'_r\rangle\rangle$$

  where $C_i = \langle P_i, \succ_i, \sim_i\rangle$ and $P'_r = max(C_i)$.
  The function is defined by the statement <Get> of the grammar.

The statements <Move>, <Remove> and <Get> are used to update the execution 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 to represent the transition relation. Here, the relation $(\xrightarrow{s})^+$ indicates the closure of $\xrightarrow{s}$ (i.e., the rule $\xrightarrow{s}$ is applied following the set of statements in the block statement s until it can not proceed any more).

- **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 $\Sigma(\xrightarrow{st_2})^+\Sigma_2$.

- **Loop over a collection**
  A loop statement is represented as $for(p, [p_1, \ldots, p_k], st)$, where p is the loop variable, $[p_1, \ldots, p_k]$ is a sequence of process identifiers that represents a total order of the process in a collection[††], and st is a statement[†††]. We have $\Sigma \xrightarrow{for(p,[p_1,\ldots,p_k],st)} \Sigma_k$, 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 the sequence of two statements.

The behaviors of the scheduler are defined in the *scheduler description*. These behaviors are handled by the scheduler using the events corresponding to these actions: *new,*

[†] A new process does not belong to any collection.

[††] we use the identifiers of the processes to define the order

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

select, clock, and *inter* 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**

$$\langle\langle\sigma_g, [\sigma_{l_1}, \ldots, \sigma_{l_m}]\rangle, \Sigma_s\rangle(\xrightarrow{new})^+ \langle\langle\sigma_g, [\sigma_{l_1}, \ldots, \sigma_{l_m}]\rangle, \Sigma'_s\rangle$$

  This event is handled by `new_process` event handler when a new process arrives to the system. It happens when the current process executes another process or the system initialize the processes (i.e. using `sch_exec` statements).

- **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 the system with a collection `ready`: $C = \langle Pid, \succ, \sim\rangle$ with $Pid = \{P, Q\}$, $\succ = \{P, Q\}$, and $\sim = \{\}$. The handler for the event `select_process` uses the following statement "`get process from ready to run`" to select a process to run. If there is no current process, the scheduler will perform the `select` action. The state of the system is changed as follows.

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

- **Inter event**

$$\langle\Sigma_p, \Sigma_s\rangle(\xrightarrow{inter})^+\langle\Sigma'_p, \Sigma'_s\rangle$$

  This event is defined by the *interface functions*, which is performed by the current process. 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).

- **Clock event**

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

  This event is defined in `clock` event handler 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 policy is increased by 1 when this event is handled: $\sigma'(c) = \sigma(c) + 1$, where $c \in \mathcal{X}_c$ is a clock variable used by the scheduler.

Suppose that process P_i is selected by the scheduler for the execution. Following an action of this process, a *sequence-action* happens as follows.

- Let $a \in normal \cup \{get\}$ be an action[†] of process P_i and $\langle\langle\sigma_g, \sigma_{l_i}\rangle, a, \langle\sigma'_g, \sigma'_{l_i}\rangle\rangle \in T_{p_i}$ be a transition of this process, two corresponding actions happen in the following sequence:

$$\langle\langle\sigma_g, [\sigma_{l_1}, \ldots, \sigma_{l_i}, \ldots, \sigma_{l_m}]\rangle, \Sigma_s\rangle \xrightarrow{a} \langle\langle\sigma'_g, [\sigma_{l_1}, \ldots, , \sigma'_{l_i}, \ldots, \sigma_{l_m}]\rangle, \Sigma_s\rangle\rangle \quad (1)$$

$$\langle\langle\sigma'_g, [\sigma_{l_1}, \ldots, , \sigma'_{l_i}, \ldots, \sigma_{l_m}]\rangle, \Sigma_s\rangle(\xrightarrow{clock})^+ \langle\langle\sigma'_g, [\sigma_{l_1}, \ldots, , \sigma'_{l_i}, \ldots, \sigma_{l_m}]\rangle, \Sigma'_s\rangle \quad (2)$$

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

$$\langle\langle\sigma_g, [\sigma_{l_1}, \ldots, \sigma_{l_i}, \sigma_{l_{i+1}}, \ldots, \sigma_{l_m}]\rangle, \Sigma_s\rangle \xrightarrow{exec} \langle\langle\sigma_g, [\sigma_{l_1}, \ldots, \sigma_{l_i}, \sigma_{l_n}, \sigma_{l_{i+1}}, \ldots, \sigma_{l_m}]\rangle, \Sigma_s\rangle \quad (1)$$

$$\langle\langle\sigma_g, [\sigma_{l_1}, \ldots, \sigma_{l_i}, \sigma_{l_n}, \sigma_{l_{i+1}}, \ldots, \sigma_{l_m}]\rangle, \Sigma_s\rangle(\xrightarrow{new})^+ \langle\langle\sigma_g, [\sigma_{l_1}, \ldots, \sigma_{l_i}, \sigma_{l_n}, \sigma_{l_{i+1}}, \ldots, \sigma_{l_m}]\rangle, \Sigma'_s\rangle \quad (2)$$

$$\langle\langle\sigma_g, [\sigma_{l_1}, \ldots, \sigma_{l_i}, \sigma_{l_n}, \sigma_{l_{i+1}}, \ldots, \sigma_{l_m}]\rangle, \Sigma'_s\rangle(\xrightarrow{clock})^+ \langle\langle\sigma_g, [\sigma_{l_1}, \ldots, \sigma_{l_i}, \sigma_{l_n}, \sigma_{l_{i+1}}, \ldots, \sigma_{l_m}]\rangle, \Sigma''_s\rangle \quad (3)$$

  where σ_{l_n} is the local state of the new process.
  We note that the new process may be assigned the identifier that has been used by previous instance of this process type (`proctype`). For example, at the initial time, an instance of process type P with identifier 1 runs. After that, this process ends. Later, another instance of P is spawned by another process. Because the identifier 1 is not used by any process, it is assigned to the new process. Now, the sequence of the local states of the processes is updated. This mechanism helps to produce the same state of the set of processes in the system.
- If the current process performs an *api* action that raises an *inter* action taken by the scheduler, two actions happen in the following sequence:

$$\langle\Sigma_p, \Sigma_s\rangle \xrightarrow{inter} \langle\Sigma'_p, \Sigma'_s\rangle \quad (1)$$

$$\langle\Sigma'_p, \Sigma'_s\rangle(\xrightarrow{clock})^+\langle\Sigma'_p, \Sigma''_s\rangle \quad (2)$$

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, 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 in this example, we do not use any clock variable and the handler for the `clock` event is not defined, the *clock* action does nothing. After that, because the running process is not determined, the scheduler will select another process to run (a *select* action).

[†]Statement *get* is considered as an action to change the value of a variable.

4. Model Checking Systems with Schedulers

In this section, we propose an algorithm for exploring the state space to verify the system with the scheduler. We also present our approach to generate the necessary information for the scheduling policy and introduce the implementation of our framework.

Scheduler Search Algorithm: DFS and BFS are the search algorithms used by explicit-state model checkers. Because the execution orders of the processes defined by the scheduler are different from that defined by the current algorithms, we need another algorithm to deal with the scheduling policies. The algorithm using the scheduling policy is shown in Algorithm 1. It is an extension of the DFS algorithm. However, it has two main characteristics. Firstly, the process is selected to run by the scheduler. Secondly, the behaviors of the scheduler need to be considered in the algorithm.

Two data structures are used in this algorithm: a state space $\mathcal{SS}$ and a stack $\mathcal{ST}$. The state space is an unordered set of states. Two routines are used to update the contains of the state space: $Add\_state(\mathcal{SS}, \Sigma)$ to add state Σ as an element to the state space, and $Contains(\mathcal{SS}, \Sigma)$ to check whether element Σ exists in the state space. The stack, which is used to record the search steps, is an ordered set of states. The operations *Push* (adds a state to the stack), *Top* (returns the top element of the stack) and *Pop* (removes the top element from the stack) are performed at the head of the stack.

The algorithm performs a search starting from function START (line 3) to visit every state that is reachable from the initial state Σ_0. In the algorithm, function SELECT (line 12) is used to obtain the processes from a relevant collection for the execution. This function corresponds to the *select* action of the scheduler (it happens when the system has no running process). This action is handled by the `select_process` event handler. Based on the ordering method used by the collection, a set of processes is returned. This function can return an empty set (line 13) indicating that no process can be executed. In this case, the system only performs the *clock* action determined by the `clock` event handler, which is done by function CLOCK (line 14). Otherwise, all actions of the processes selected (line 21, 22) are considered. Function TAKE (line 23) performs an action a of the process to change the system state: Σ_a = TAKE(a, Σ). This function represents the following behaviors:

- the behavior of the process defined by the transition $\langle\langle\sigma_g, \sigma_l\rangle, a, \langle\sigma'_g, \sigma'_l\rangle\rangle \in T_p$, where $\langle\sigma_g, \sigma_l\rangle$ and $\langle\sigma'_g, \sigma'_l\rangle$ 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 an action $a \in \{api, exec\}$ of the process (i.e. handling the scheduling action).

The functions SELECT, TAKE and CLOCK determine the transition relation $\langle\Sigma, a, \Sigma'\rangle \in T_{sys}$ of the system. From the state space determined by the search algorithm, we can verify the system behaviors.

Algorithm 1 *Scheduler depth-first search algorithm.*

```
 1: Input: Σ0                                              ▷ initial state
 2: Output: SS                                             ▷ state space
 3: procedure START()
 4:     Stack: ST = {}
 5:     State space: SS = {}
 6:     Push(ST, Σ0)
 7:     Add_state(SS, Σ0)
 8:     SEARCH()
 9: end procedure
10: procedure SEARCH()
11:     Σ = Top(ST)
12:     P = SELECT(Σ)
13:     if P == {} then
14:         Σ^t = CLOCK(Σ)
15:         if Contains(SS, Σ^t) == false then
16:             Push(ST, Σ^t)
17:             Add_state(SS, Σ^t)
18:             SEARCH()
19:         end if
20:     else
21:         for p ∈ P do
22:             for a ∈ p.T do                            ▷ a is an action of p
23:                 Σ_a =TAKE(a, Σ)
24:                 Σ^t_a = CLOCK(Σ_a)
25:                 if Contains(SS, Σ^t_a) == false then
26:                     Push(ST, Σ^t_a)
27:                     Add_state(SS, Σ^t_a)
28:                     SEARCH()
29:                 end if
30:             end for
31:         end for
32:     end if
33:     Pop(ST)
34: end procedure
```

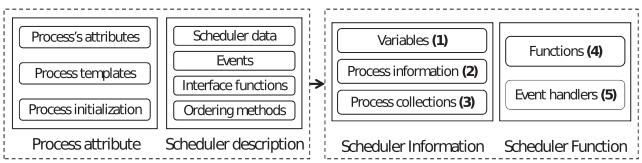

Fig. 3 Generation approach.

Generate Scheduling Information: In our framework, the information needed for performing scheduling tasks is generated from the description of the scheduling policy. Our approach for the generation is shown in Fig. 3. The generation is as follows. All the variables (1) are generated from *scheduler data*, which contains the definition of the scheduler variables and the collections. The template(s) (`proctype`) for the process(es) and the attributes of the process are converted to *process information* (2), which is managed by the scheduler. The definition of the collections in the *scheduler data* is converted to *process collections* (3), which use the *ordering methods* defined in the *scheduler description*. The *process initialization* (corresponding to the *init* part of the *process attribute*) determines the initial function in (4). The other functions in (4) are realized from the *interface functions* specified in the *scheduler description*. The definitions of the events in this description are

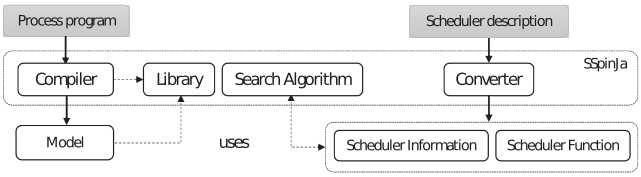

Fig. 4 Architecture of the framework.

converted to *event handlers* (5). Using these functions and the information above, the framework performs the search algorithm with the scheduling policy defined in the DSL.

Implementation: The design of our framework is shown in Fig. 4. Promela is used as the base modeling language for the process. We extended SpinJa for the back-end tool in our framework. XText framework [13] was used to implement the DSL.

A tool named SSpinJa[†] has been implemented according to the method proposed. We follow the compilation approach as in Spin and SpinJa model checkers to prepare all necessary information beforehand. The parser and the compiler of SpinJa were updated to realize the new API functions in the modeling language. The *process program* is compiled into a model (in Java), which uses the libraries of SSpinJa. We built a converter to generate the *scheduler information* and the *scheduler function* from the description of the scheduling policy under XText framework. The generated code includes (a) the implementation of the processes, (b) the implementation of the collections with ordering methods, and (c) the implementation of the scheduler. A new search algorithm, which uses the information generated from the description of the scheduler, was implemented in the tool.

5. Case Studies

In order to evaluate our method, many experiments were conducted on several systems ranging from simple to practical ones, such as the dining philosopher problem, the benchmark for explicit model checkers [14], the OSEK/VDX OS, and the Linux OS with real-time scheduling policies to show the accuracy, flexibility, and scalability. This section introduces four case studies and presents the experiment results[††].

5.1 Dining Philosopher Problem

We used *round-robin* (RR) and *priority* (FP) strategies to deal with the dining philosopher problem. Numbers of philosophers considered were 2, 4, 8, 16, and 32. The philosopher was modeled as a process. Each behavior of the philosopher was represented as an atomic action. With RR policy, we set the time slice to 3 indicating that the number of actions taken for each process turn was 3. With FP policy, different priorities were assigned for the processes. To check the absence of starvation, we considered the problem with only four philosophers. The results for checking deadlock and starvation are listed in Table 1.

[†]SSpinJa stands for 'Scheduling SpinJa'.

[††]The experiments were conducted on Intel Core i7, 3.4 GHz CPU with 32G RAM.

Table 1 Checking deadlock and starvation results.

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

Table 2 Results of dining philosopher problem.

N	Without scheduler			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

Table 3 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 these experiments, deadlock occurred when we did not use the scheduler; therefore, the starvation problem was not considered. We note that every error was counted during the search. Without using the scheduler, the search was incomplete within the limited time (60 seconds) if the number of philosophers was equal to 16 or 32. Using FP policy, the deadlock was resolved, but the starvation occurred. Both deadlock and starvation were absent with RR policy. Table 2 shows the detailed results for checking deadlock with the number of philosophers (N), the number of states (S), time (T) in seconds, and memory usage (M) in Mb. In the results, T.O. means timeout (the search was incomplete within 60 seconds); this only happened without using the scheduler.

5.2 Schedulability for Real-Time Systems

In a real-time system, each task (process) has several attributes, such as priority (*PRIORITY*), best-case execution time (*BCET*), worst-case execution time (*WCET*), the time between task releases (*PERIOD*), and deadline (*DEADLINE*). The schedulability means that all tasks will meet their deadlines. We conducted two experiments based on these attributes to deal with the schedulability problem.

In the first experiment, four processes, which had different configurations, were considered. The attributes of these processes are shown in Table 3 with *TIME* indicating both of the *BCET* and the *WCET*; *Initial OFFSET* determining the time before the first instance of the process can

Table 4 Analysis results with four processes.

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

Table 5 Analysis results with different number of processes.

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

be activated. For the evaluation, we used three scheduling policies: *priority* (FP), *first-in-first-out* (FIFO) and *earliest-deadline-first* (EDF). Deadline violation was analyzed for this system.

We also conducted the experiments using the framework introduced in [25]. This study aims at schedulability analysis following the model-based approach using UPPAAL model checker [4]. The resource sharing problem with real-time behaviors and a scheduling policy for each resource is considered. To apply this study, we considered the scheduler to be a system resource. The attributes of the processes were set up as above. The analysis results of using this framework and the results of using our approach were compared, which are shown in Table 4 with time (T) in seconds and memory usage (M) in Mb. In this table, "satisfied" means no deadline violation; "unsatisfied" means that deadline violation occurs; "may not be satisfied" means that the deadline violation was not determined.

In the second experiment, different numbers of the processes with the same configuration for each process were used to analyze the performance. The configuration was (PERIOD, BCET, WECT, DEADLINE, PRIORITY) = (20, 5, 5, 20, 1). The numbers of processes considered were 2, 3, 4, and 5. The analysis results are shown in Table 5 with the average time (T_a) in seconds and the average memory usage (M_a) in Mb. for three scheduling policies above. We note that the *result* column indicates the results for all of the scheduling policies (e.g., "satisfied" means that no deadline violation was found for all of the scheduling policies). In this table, "ND." means "not determined"; that occurs when the system reaches the maximum number of processes† determined by the tool.

5.3 Synchronization Mechanism in OSEK/VDX OS

We considered a typical problem with common synchronization mechanisms for scheduling resources. The problem

†The maximum number of simultaneously running processes is 255.

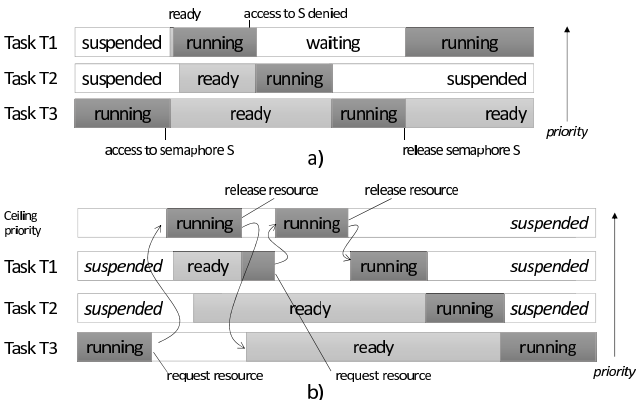

Fig. 5 Synchronization mechanism problem.

relates to the system using priority scheduling policy with the fact that a lower-priority process (task) can delay the execution of the higher-priority process. For example, assume that the system with preemptive policy has three processes from T_1 to T_3 (as shown in Fig. 5 a)). Process T_3, which has the lowest priority, is currently running and occupies the semaphore S. Process T_1 with the highest priority preempts this process and requests the same semaphore. However, T_1 is denied because S is already used by T_3; therefore, T_1 enters the waiting state. Now process T_2 runs. Process T_1 can only run until all lower-priority processes have been terminated and the semaphore S has been released. Process T_2 does not use S but it delays process T_1.

To avoid this problem, OSEK/VDX OS [5] uses the Priority Ceiling Protocol. When a process occupies a resource and the priority of the process is lower than that of the resource, the priority of the process will be raised to this value. It will be reset after the process releases the resource (as shown in Fig. 5 b)). Note that the ceiling priority of a shared resource is lower than the lowest priority of all processes that do not access the resource and higher than the priorities of all processes that access the resource.

To demonstrate the mechanism, we described the OSEK/VDX scheduler in the DSL. Each process has two attributes (*PRIORITY* and *CEILING_PRIORITY*) to indicate the static priority and the dynamic priority of the process. The dynamic priority is changed when the process accesses to a resource. Two service APIs *GetResource* and *ReleaseResouce* were described as two interface functions, which change the priority of the process following that of the resource. Two other functions `ActivateTask` and `TerminateTask` were described for executing and terminating a process corresponding to the two related service APIs of OSEK/VDX OS. We modeled the system in the example above with three processes (from `t1` to `t3`) (as shown in Fig. 6). We note that at the starting time, only process `t3` can be executed because we set the value `true` for the attribute *AUTOSTART* of this process; the other processes will be in suspended state.

We conducted the experiments with and without using the protocol to verify the the occurrence of the behaviors

of the processes. The results are shown in Table 6. Without using the protocol, we found the violation of assertion corresponding to the assert statement (x==1) in process t2. This error means that process t1 can not terminates before t2 ends.

5.4 Linux Scheduling Policies

In this section, we show the accuracy of the framework for verifying a practical system. The Linux scheduling policies were used for the experiments.

The primary scheduling mechanism in Linux OS is based on the priority of the process. For a non-real-time process, Linux supports three kinds of normal scheduling policies (the priority of the process is set to 0) that are *SCHED_OTHER* (for round-robin, time-sharing policy), *SCHED_BATCH* (for batch executions), and *SCHED_IDLE* (for background processes). For each real-time process (the priority is set from 1 to 99), Linux uses two policies that are *SCHED_FIFO* and *SCHED_RR*. The difference between *SCHED_FIFO* and *SCHED_RR* is that among the processes with the same priority, *SCHED_RR* performs the round-robin method with a certain time slice; *SCHED_FIFO*, instead, needs the process to explicitly yield the processor. Since version 3.14, an additional policy called *SCHED_DEADLINE* was available in the Linux kernel. This policy implements the EDF scheduling algorithm. Each process under this policy is assigned a deadline and the earliest-deadline process is selected to run.

```
int x = 0 ;
proctype t1() {
  sch_api_self(ActivateTask, t2) ;
  sch_api_self(GetResource, 1);
  //get 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()) ;
}
```

Fig. 6 The example program.

Table 6 Results of verifying priority ceiling protocol.

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

The system introduced in the example in Sect. 3 was used for the experiments. Several scenarios based on the combinations of the scheduling policies with different priorities of the processes were used. Some properties expected to be held were checked under these policies. The experiments with the policies (RR and FIFO) related to the Linux scheduling were also conducted. We implemented the programs corresponding to these experiments in Linux. The results of verifying the system with Linux scheduling policies and the results of verifying the system with the related policies were compared; they were then matched with the results of the executions of the programs in Linux. The experiment results are shown in Table 7 with the number of states (S), time (T) in seconds, and memory usage (M) in Mb. In this table, each scenario follows the form (*scheduling policy, priority condition*); for example, the scenario (*SCHED_OTHER, P.pri > Q.pri*) means that the *SCHED_OTHER* policy was used, the priority of process *P* was higher than that of *Q*. For each scenario, the process *P* is set to run first. The results of verifying the system with the related policies are shown in Table 8.

We can see that, in every case, the expected property holds with the scheduling policy described. It conforms with the results of the execution of the programs in Linux. However, in some cases, the policies related to Linux scheduler indicate that these properties do not hold.

Table 7 Results of verifying Linux tasks.

No.	Scenario	Property	Result	S	T	M	Linux execution result
1	SCHED_OTHER, *P.pri == Q.pri*	$(a > 0) \wedge (b > 0)$	holds	240627	0.41	52.15	(a, b) = (99976, 25)
2	SCHED_FIFO, *P.pri > Q.pri*	$(a > 0) \wedge (b == 0)$	holds	300008	0.42	57.40	(a, b) = (100000, 0)
3	SCHED_FIFO, *P.pri < Q.pri*	$(a > 0) \wedge (b > 0)$	holds	200108	0.29	48.31	(a, b) = (42837, 57164)
4	SCHED_FIFO, *P.pri == Q.pri*	$(a > 0) \wedge (b == 0)$	holds	300008	0.46	57.42	(a, b) = (100000, 0)
5	SCHED_RR, *P.pri > Q.pri*	$(a > 0) \wedge (b == 0)$	holds	300008	0.49	57.40	(a, b) = (100000, 0)
6	SCHED_RR, *P.pri < Q.pri*	$(a > 0) \wedge (b > 0)$	holds	240617	0.46	52.15	(a, b) = (54020, 45981)

Table 8 Results of verifying tasks with related scheduling policies.

No.	Scenario	Related policy	Property	Result	S	T	M
1	SCHED_OTHER, *P.pri == Q.pri*	round-robin	$(a > 0) \wedge (b > 0)$	holds	240616	0.4	50.34
2	SCHED_FIFO, *P.pri > Q.pri*	first-in-first-out	$(a > 0) \wedge (b == 0)$	holds	300008	0.42	52.85
3	SCHED_FIFO, *P.pri < Q.pri*	first-in-first-out	$(a > 0) \wedge (b > 0)$	not hold	300003	0.33	53.15
4	SCHED_FIFO, *P.pri == Q.pri*	first-in-first-out	$(a > 0) \wedge (b == 0)$	holds	300008	0.42	52.85
5	SCHED_RR, *P.pri > Q.pri*	round-robin	$(a > 0) \wedge (b == 0)$	not hold	240603	0.35	51.64
6	SCHED_RR, *P.pri < Q.pri*	round-robin	$(a > 0) \wedge (b > 0)$	holds	240616	0.4	50.34

6. Discussion

Verifying the systems using different scheduling policies and without using the scheduler produces different results because they lead to different behaviors of the system. That means that in order to verify system accurately, the scheduling policy needs to be taken into account during the verification. As shown in Sect. 5.1, the dining philosopher problem caused the deadlock when the scheduler was not used while deadlock and starvation were absent in the case of using RR policy. If the priorities for the philosophers are different as in FP policy, their opportunities to eat are different. In this case, there is no deadlock; however, starvation occurs. This is because the philosophers with low priorities will have no chance to eat. The number of visited states and the running time for different scheduling policies are different. For example, the dining philosopher problem under the FP policy, only the process with the highest priority can be selected to run; therefore, the number of states is unchanged. That is different from the result of using RR policy (as shown in Fig. 7 a)). The running times for each scheduling policies are also different (Fig. 7 b)) because the larger number of states to visit, the more time needed to explore.

The first experiment in Sect. 5.2 for verifying the schedulability of a real-time system shows that deadline violation occurs when the system uses either FP or FIFO policy; however, it does not occur with EDF strategy. 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 with the framework using the UPPAAL model checker were still "may not be satisfied" meaning that the framework can not determine the satisfaction. That is due to the over-approximation approach adopted by this work. In addition, this framework only focuses on the time constraints. Therefore, considering both the behaviors of the process and the behaviors of the scheduler is also challenging. Moreover, we can see that the running time of the scheduling framework showed a significant increase in comparison with SSpinJa, and SSpinJa used less memory than this framework (as shown in Fig. 8).

In Sect. 5.4, with the scheduling policies used by Linux OS, we can see that the system can be verified accurately if the specification of the scheduler conforms with the policies used by the system. In fact, the scheduling policies used by practical systems are often different from 'textbook' strategies. Therefore, to verify a practical system, we need to specify the behaviors of the scheduler conforming with that of the system.

The synchronization mechanism shown in Sect. 5.3 relates to the scheduling policies. This relation is not easy to handle with using only the modeling language for the process because we need to model the whole system including the processes and the scheduler, then encode the relations between them. With our framework, these relations are easy to define because an interface for the interaction between the processes and the scheduler is already provided.

By changing the initial values of the process attributes, our framework can deal with different configurations of the system. Moreover, using appropriate data structures to store the processes and specifying the handler for each system event is a flexible way to deal with describing the behaviors of the scheduler. The description code for specifying each scheduling policy used in the experiments is really small in comparison with the number lines of code generated (as indicated in Table 9). For example, with a general language, 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.

In our framework, the specification of the scheduling

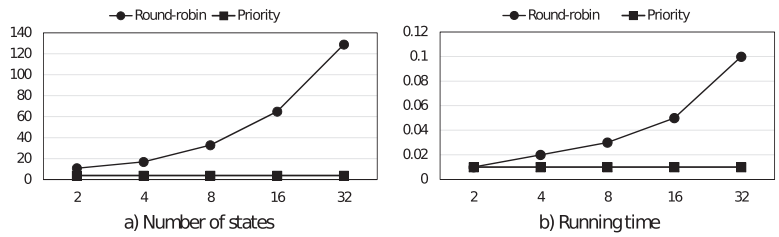

Fig. 7 Results for the dining philosopher problem.

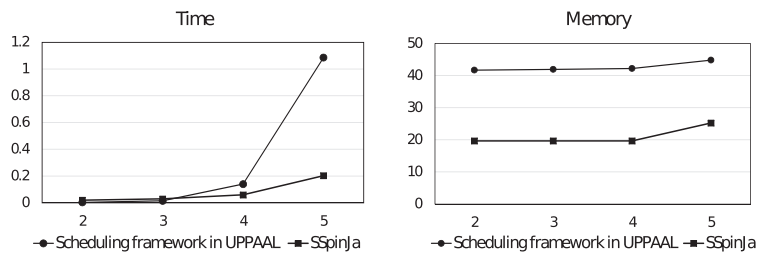

Fig. 8 Running time and memory usage.

Table 9 The number lines of code generated from the scheduling policy.

Scheduling policy	No. Lines Specification	No. Lines Code Generated
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

policy and the process model are separate. Considering the scheduling strategy as an input of the verification, with the same process model we can apply different scheduling polices, and with the same scheduling policy we can apply various process models. This means that the process model and the description of the scheduler can be reused completely.

As indicated in Sect. 5.1, when the search was completed, although the number of states visited by SSpinJa was smaller than the number of states visited by SpinJa, SSpinJa needed more memory than the original tool (SpinJa) did. Because our tool considers the behaviors of the scheduler during the verification, it needs more memory to store the information of the scheduler for the computation.

7. Related Work

The related work for search algorithms is directed model checking, which focuses on several techniques to reduce the search space based on the abstractions of the system state [15], [16] or based on estimating the distance [17], [18] from the current state to the error states. The process that has the transition corresponding to the best value (lowest/highest) will be chosen for the execution. This means guiding the search for particular aims. In comparison with these techniques, we use the scheduler for selecting the process, while directed model checking techniques choose the best process for the execution.

There are several studies that analyzed the behaviors of systems without considering the scheduling policies, such as [19], [20]. To easily implement a model checking tool, the work [21] proposed a framework named Bogor with an extensible input language for the implementation. Our research is different because we do not deal with the implementation; instead, we propose a method for flexibly verifying different kinds of concurrent behaviors based on the scheduling policies.

In the scheduling domain, there are several ways to deal with schedulability problem. One of these is schedulability analysis, which aims to find the set of conditions on a task (process) design that determines whether the scheduling problem is feasible or not. These works in this research field are based on constraints solving [22], [23], using fixed scheduling [24] or worst-case assumption [26]. Moreover, there are tools based on the rate monotonic analysis that uses fixed priority for the process, such as TimeWiz from Time Sys Corporation and RapidRMA from TriPacific. In comparison with these works, ours have the following differences. Firstly, we deal with facilitating the variation of the scheduler in model checking techniques. Secondly, we do not use any assumption because the behaviors of the system will be missed. For example, with the worst-case assumption, some of the combinations of the conditions never occur. Thus, verifying the system based on worst-case assumption is very pessimistic.

To analyze the time constraints, the work UML profile for MARTE [27] introduced a DSL for modeling a real-time system with timing properties. However, this language only focuses on the time relations and can not deal with the behavior of processes and the behaviors of the scheduler as our work does.

One of the model checkers for verifying real-time systems is UPPAAL. To deal with the scheduling policies, with UPPAAL, we can create a timed automata to capture the behaviors of the scheduler and the processes. However, the limitation is that UPPAAL does not support accessing the internal information of a process, which is described as a template. Therefore, we must declare all internal information of the process as public variables to implement the scheduling algorithms or synchronization mechanisms that use these attributes, such as the slack stealing algorithm or the priority ceiling protocol. We can see that this approach lacks of flexibility. Moreover, dealing with the facilitating the variation of the scheduler with timed automata is challenging.

The appropriate model checker using the schedulers is TIMES [28], which allows performing scheduling analysis with various properties for tasks, such as period, priority, and deadline. Nevertheless, TIMES only supports five scheduling policies that are *rate-monotonic*, *deadline-monotonic*, *fixed-priority*, *earliest-deadline-first*, and *first-come first-served*. There is no way to extend this tool for other policies.

To deal with using the scheduling policy in model checking techniques, the approaches like [1], [2], use simulation methods for combining the scheduler with the processes into a system model, then use an existing tool for the verification. However, the checking capability is limited. This is because much redundant information of the system is stored in the system state and many behaviors of the scheduler are checked during the verification. Therefore, the state space explosion problem will occur easily.

It is difficult to extend current approaches to deal with various kinds of schedulers. To solve this problem, we pro-

pose a DSL to facilitate the variation of the scheduler. The most closely related languages in this area are *Bossa* and *Catapults*, which are introduced in [30] and [31]. In comparison with these studies, our approach has two main differences, which are based on the target systems and the purpose. Firstly, their studies focus on specific systems and rely on the techniques of these systems: *Bossa* is based on the Linux OS and *Catapults* is used for embedded systems, although it can support different target platforms and languages. Therefore, they only support limited types of scheduling strategies. In contrast, our approach does not apply to any particular system and can support a variation of policies. Secondly, their aim is for implementing the scheduler in practical systems; ours is for ensuring the correctness of software systems using model checking techniques.

8. Conclusion

This paper presents an approach to verify concurrent behaviors under various kinds of scheduling policies in model checking techniques. We have the following contributions for this paper: 1) We propose a DSL to specify the scheduling policies; 2) We propose a search algorithm based on the scheduler; 3) We developed a tool for verifying systems under different scheduling policies.

The DSL provides a high-level support for the succinct specification of scheduling strategies. The necessary artifacts are automatically generated for the subsequent analysis of the system. The advantages of this approach are: 1) the specification language is simple, 2) the behaviors of the systems can be extended easily, and 3) the descriptions can be reused completely. The demonstration shows that our approach facilitates the variation of schedulers and verifies systems accurately.

The limitation of the framework is memory usage, which is a problem for an explicit-state model checker. In the future, some optimization techniques, such as partial order reduction, will be applied to overcome this limitation. In addition, we plan to study on the scheduling optimization techniques for the specification of the scheduler and use model-based testing approach to check that the specification in the DSL can specify the practical one.

Acknowledgments

This work was supported by JSPS KAKENHI Grant Number 18H03220.

References

[1] N. Marti, R. Affeldt, and A. Yonezawa, "Model-checking of a Multithreaded Operating System," 23rd Workshop of the Japan Society for Software Science and Technology, University of Tokyo, Tokyo, Japan, http://staff.aist.go.jp/reynald.affeldt/documents/marti-jssst2006-en.pdf, 2006.
[2] T. Aoki, "Model checking multi-task software on real-time operating systems," 2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC), IEEE, pp.551–555, 2008.
[3] G.J. Holzmann, The SPIN model checker: Primer and reference manual, Addison-Wesley Reading, vol.1003, 2004.
[4] K.G. Larsen, P. Pettersson, and W. Yi, "UPPAAL in a nutshell," International Journal on Software Tools for Technology Transfer (STTT), vol.1, no.1-2, pp.134–152, 1997.
[5] OSEK Group and others, "OSEK/VDX Operating System Specification," http://portal.osek-vdx.org, 2005.
[6] A. Cimatti, A. Micheli, I. Narasamdya, and M. Roveri, "Verifying SystemC: a software model checking approach," Formal Methods in Computer-Aided Design (FMCAD), 2010, IEEE, pp.51–59, 2010.
[7] L. Cordeiro and B. Fischer, "Verifying multi-threaded software using smt-based context-bounded model checking," Proceedings of the 33rd International Conference on Software Engineering, ACM, pp.331–340, 2011.
[8] 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), vol.14, no.1, pp.1–26, 2009.
[9] M. de Jonge and T.C. Ruys, "The SpinJa model checker," International SPIN Workshop on Model Checking of Software, Springer, vol.6349, pp.124–128, 2010.
[10] R. Gerth, "Concise PROMELA reference, 1997," http://spinroot.com/spin/Man/Quick.html, accessed: 20-April-2017.
[11] J.P. Lehoczky and S. Ramos-Thuel, "An optimal algorithm for scheduling soft-aperiodic tasks in fixed-priority preemptive systems," Real-Time Systems Symposium, 1992, IEEE, pp.110–123, 1992.
[12] E. Emerson, A. Mok, A. Sistla, and J. Srinivasan, "Quantitative temporal reasoning," Computer-Aided Verification, Springer, pp.136–145, 1991.
[13] L. Bettini, "Implementing Domain-Specific Languages with Xtext and Xtend," Packt Publishing Ltd, 2013.
[14] R. Pelánek, "BEEM: Benchmarks for explicit model checkers," International SPIN Workshop on Model Checking of Software, Springer, pp.263–267, 2007.
[15] A.S. Andisha, M. Wehrle, and B. Westphal, "Directed Model Checking for PROMELA with Relaxation-Based Distance Functions," Model Checking Software, Springer, vol.9232, pp.153–159, 2015.
[16] S. Kupferschmid, J. Hoffmann, H. Dierks, and G. Behrmann, "Adapting an AI planning heuristic for directed model checking," International SPIN Workshop on Model Checking of Software, Springer, vol.3925, pp.35–52, 2006.
[17] 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, vol.5, no.2-3, pp.247–267, 2004.
[18] M. Wehrle, S. Kupferschmid, and A. Podelski, "Transition-based directed model checking," International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Springer, vol.5505, pp.186–200, 2009.
[19] 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, vol.309, pp.31–49, 2014.
[20] Y. Liu, J. Sun, and J.S. Dong, "Pat 3: An extensible architecture for building multi-domain model checkers," 2011 IEEE 22nd International Symposium on Software Reliability Engineering, IEEE, pp.190–199, 2011.
[21] M.B. Dwyer and J. Hatcliff, "Bogor: A flexible framework for creating software model checkers," Testing: Academic & Industrial Conference-Practice And Research Techniques (TAIC PART'06), IEEE, pp.3–22, 2006.
[22] S. Nejati, S. Di Alesio, M. Sabetzadeh, and L. Briand, "Modeling and analysis of CPU usage in safety-critical embedded systems to support stress testing," International Conference on Model Driven Engineering Languages and Systems, Springer, vol.7590,

pp.759–775, 2012.

[23] S. Di Alesio, S. Nejati, L. Briand, and A. Gotlieb, "Stress testing of task deadlines: A constraint programming approach," 2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE), IEEE, pp.158–167, 2013.

[24] H. Gomaa, "Designing concurrent, distributed, and real-time applications with UML," Proceedings of the 23rd international conference on software engineering, IEEE Computer Society, pp.737–738, 2001.

[25] A. David, J. Illum, K.G. Larsen, and A. Skou, "Model-based framework for schedulability analysis using UPPAAL 4.1," Model-based design for embedded systems, vol.1, no.1, pp.93–119, 2009.

[26] K. Tindell and J. Clark, "Holistic schedulability analysis for distributed hard real-time systems," Microprocessing and microprogramming, vol.40, no.2-3, pp.117–134, 1994.

[27] MARTE, UML, "UML profile for MARTE: model and analysis of real-time embedded systems," http://www.omgwiki.org, 2015.

[28] T. Amnell, E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi, "TIMES: a tool for schedulability analysis and code generation of real-time systems," International Conference on Formal Modeling and Analysis of Timed Systems, Springer, vol.2791, pp.60–72, 2003.

[29] N.-H. Tran, Y. Chiba, and T. Aoki, "Domain-specific language facilitates scheduling in model checking," 2017 24th Asia-Pacific Software Engineering Conference (APSEC), pp.417–426, IEEE, 2017.

[30] 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, pp.19–31, 2002.

[31] M.D. Roper and R.A. Olsson, "Developing embedded multithreaded applications with CATAPULTS, a domain-specific language for generating thread schedulers," Proceedings of the 2005 international conference on Compilers, architectures and synthesis for embedded systems, ACM, pp.295–303, 2005.

Appendix: Language Grammar

⟨*Model*⟩ ::= ⟨*ProcDSL*⟩ | ⟨*SchDSL*⟩
⟨*ProcDSL*⟩ ::= 'def' 'process' '{' [⟨*ProcAttr*⟩] ⟨*Process*⟩* '}' [⟨*ProcConf*⟩] [⟨*ProcInit*⟩]
⟨*ProcAttr*⟩ ::= 'attribute' '{' ⟨*PAttr*⟩* '}'
⟨*PAttr*⟩ ::= ['var'|'val'] ⟨*Type*⟩⟨*ID*⟩ (',' ⟨*ID*⟩)* ['=' ⟨*Value*⟩]';'
⟨*Type*⟩ ::= 'int' | 'byte' | 'clock'
⟨*Value*⟩ ::= ⟨*BOOL*⟩ | ⟨*INT*⟩
⟨*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*⟩]
⟨*SchDef*⟩ ::= 'scheduler' ⟨*ID*⟩ '(' [⟨*ParamList*⟩] ')' ['refines' ⟨*ID*⟩] '{' [⟨*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'
⟨*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*⟩
⟨*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'
⟨*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*⟩ ')'

- 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 as the inheritance in object-oriented programming. With a collection, `'refines'` means redefining its ordering method.

Nhat-Hoa Tran received the B.S., M.S. degrees in Information Technology from Hanoi University of Science and Technology, Vietnam (2003, 2008), and Ph.D. degree from Japan Advanced Institute of Science and Technology (2018). He is currently a lecturer at national University of Civil Engineering, Vietnam (NUCE). His primary research interests are software engineering, formal verification, formal method and model checking.

Yuki Chiba received his B.S., M.S., and Ph.D. degrees from Tohoku University (2003, 2005, 2008). He is currently a researcher at DENSO CORPORATION. His research interests include program transformation, term rewriting system, automated theorem proving, and model checking.

Toshiaki Aoki is a professor, JAIST (Japan Advanced Institute of Science and Technology). He received B.S. degree from Science University of Tokyo (1994), M.S. and Ph.D. degrees from (1996, 1999). He was an associate at JAIST from 1999 to 2006, and a researcher of PRESTO/JST from 2001-2005. His research interests include formal methods, formal verification, theorem proving, model checking, object-oriented design/analysis, and embedded software.