A Formal Modeling and Analysis Framework for Software Product Line of Preemptive Real-Time Systems

Jin Hyung Kim, Axel Legay, Louis-Marie Traonouez, Mathieu Acher, Sungwon Kang

A Formal Modeling and Analysis Framework for Software Product Line of Preemptive Real-Time Systems

Jin Hyun Kim, Axel Legay
Louis-Marie Traonouez
INRIA/IRISA,
Rennes Cedex, France

Mathieu Acher
University of Rennes 1
France

Sungwon Kang
KAIST
Daejeon, Republic of Korea

ABSTRACT

Adapting real-time embedded software for various variants of an application and usage contexts is highly demanded. However, the question of how to analyze real-time properties for a family of products (rather than for a single one) has not drawn much attention from researchers. In this paper, we present a formal analysis framework to analyze a family of platform products w.r.t. real-time properties. To this end, we first propose an extension of the widely-used feature model, called Property Feature Model (PFM), that distinguishes features and properties explicitly, so that the scope of properties restricted to features can be explicitly defined. Then we present formal behavioral models of components of a real-time scheduling unit, i.e. tasks, resources, and resource schedulers, such that all real-time scheduling units implied by a PFM are automatically composed with the components to be analyzed against the properties given by the PFM. We apply our approach to the verification of the schedulability of a family of scheduling units using the *symbolic* and *statistical* model checkers of UPPAAL.

1. INTRODUCTION

Software Product Line Engineering (SPLE) allows reusing software assets by managing the commonality and variability of products. Recently, SPLE has gained a lot of attention as an approach for developing a wide range of software products from non-critical to critical software products [4, 5, 11–15, 19, 22–24, 26, 27], and from application software to platform software products [25]. Real-time software products (such as real-time operating systems) are a class of systems for which SPLE techniques have not drawn much attention from researchers, despite the need to efficiently reuse and customize real-time artifacts.

A real-time system is a time and resource-constrained system, in which the performance and the correctness of the system depend not only on individual capabilities of its components but also on their composition under given resources. For this reason, it is indispensable to check if a complete system guarantees its composability over timing requirements concerning resource constraints *whenever* it is deployed with varying sets of resources. The same constraints hold for an SPL of real-time system, such that all the products generated from the SPL should satisfy various real-time properties. In general, a real-time system product is not verifiable until its configuration under given resources is fixed.

The overall challenge is to analyze a family of real-time systems (rather than a single one) against real-time properties, depending on varying sets of resources. Two main issues are raised for the verification of an SPL of real-time systems. 1) The specification method must link individual features of an SPL to the corresponding real-time properties that must be verified. 2) The analysis method must verify all products generated from an SPL against all real-time properties imposed upon individual features of each product. If the products of an SPL are safety-critical, this analysis method should be rigorous enough to guarantee the safety of all the products.

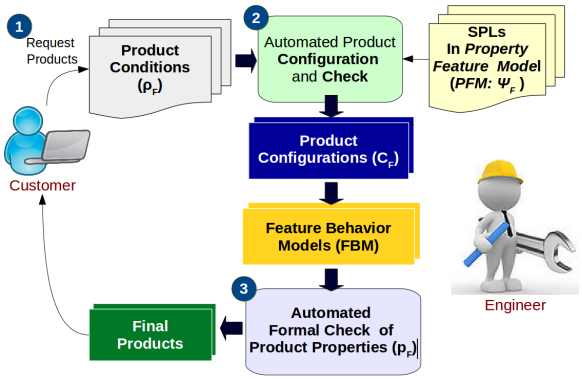

Figure 1: Our formal SPLE framework

When analyzing real-time systems, formal methods such as model-checking are often used to obtain safe and reliable proofs that the system satisfies the expected properties. The research in [5, 6, 23, 24, 26, 27] consider various properties, including structural and behavioral properties of SPLs. However, they are limited to functional properties, and rarely considered real-time properties or non-functional properties, that are related to the real behavior of a complete system.

This paper proposes a formal SPLE framework for real-time scheduling units[1] and demonstrates its efficiency and feasibility. It focuses on the formal analysis of real-time properties of an SPL in terms of resource sharing with time dependent functionalities. Our framework is depicted in Figure 1. It provides a structural description method of the variability and the properties of a real time system, and behavioral models to verify the properties using formal techniques and the tools UPPAAL symbolic model checker (MC) [8] and UPPAAL statistical model checker (SMC) [16].

For the specification of an SPL, we propose an extension of a feature model, called Property Feature Model (PFM), shown on the right side of Figure 1. A PFM explicitly distinguishes

[1]A scheduling unit consists of tasks and a scheduling mechanism.

features and properties associated with features in a FM so that properties are analyzed in the context of the relevant features. In the second step of Figure 1, we define a *non-deterministic decision process* that automatically configures the products of an SPL that satisfy the constraints of a given PFM and the product conditions of customers. In the third step in Figure 1, we analyze the products against the associated properties. For analyzing real-time properties, we provide *feature behavioral models* of the components of a scheduling unit, i.e. tasks, resources and schedulers. Using these feature behavioral models, a family of scheduling units of an SPL is formally analyzed against the designated properties with model checking techniques.

The rest of the paper is organized as follows: Section 2 discusses some background concerning SPL specification and analysis of real-time systems. Section 3 presents a new extension of a feature model, Property Feature Mode (PFM). In addition, we define product conditions to express customer's requests. Also, we define the semantics of the PFM to configure the SPL according to the customer's requests. In Section 4, we provide feature behavioral models of the components of a scheduling unit, using Timed Automata and its extensions Stopwatch Automata. In Section 5, we present the results from a case study. Finally Section 6 discusses related work and Section 7 concludes this paper.

2. BACKGROUND

This section discusses our basic formal models, analysis techniques, and a basic model of a product family.

2.1 Specification methods

In our framework, a real-time system is considered in terms of resource sharing, where a scheduling unit consisting of a set of tasks and a set of shared resources is organized to support the execution of applications. It is given timing requirements, such as schedulability conditions, performance, etc. Behaviors of a SPL are captured to analyze real-time properties using *Timed Automaton* (TA) [3], that is a classical formal model for designing real-time systems. It consists of:

- A set of real-time clocks. The model uses a continuous time semantics meaning that the clocks are evaluated to real numbers.
- A set of locations, possibly labeled with an invariant constraint over clocks, which restricts the time spent in the location.
- A set of transitions between pairs of locations, possibly labeled with a guard constraint over clocks. This constraint specifies from which values of the clocks the transition may be taken. The transition may also be labeled with a synchronization channel and an update of clocks.

When considering preemptive real-time systems, it is necessary to keep track of the execution time of a running process. For this reason, *Stopwatch Automata* (SWA) are TA that use a stopwatch mechanism to stop and resume the execution of a clock

A simple example of SWA is shown in Figure 2. It depicts an abstract model of a periodic task. A transition may be taken when its guard and all the invariant hold. The transition between JobDone and Ready is thus taken as soon as x is

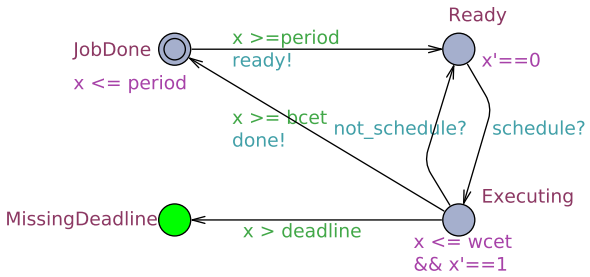

Figure 2: SWA for an abstract task

equal to period. At Ready the SWA starts executing the task if it receives the event schedule?. Then, it can send the event done! as soon as the clock x reaches the best case execution time (bcet) and before it reaches the worst case execution time (wcet). Otherwise it joins the location MissingDeadline if the clock exceeds the deadline. Finally, it returns to location JobDone and waits for the next period. The running task at location Executing can be preempted by the event not_schedule?, and then it returns to the Ready location. Notice that the clock x at the location Ready is associated to a stopwatch (x'==0) meaning that the clock stops progressing, whereas at location Executing the stopwatch x'==1 means that the clock is progressing. This model will be refined in Section 4.

Even if a real-time system synchronizes with a clock, there might be a delay caused by various reasons, such as a noise or a jitter, etc. Such non-deterministic timed events cannot be physically and precisely measured with ease, but they can be abstracted by giving probabilities to events and actions.

For this reason, Probabilistic Timed Automata (PTA) extends TA with probability transitions. In PTA, like in TA, a transition is enabled as long as the relevant guards and invariants hold, but the time to make an enabled transition depends on a certain probability distribution [17]. If multiple transitions leaving the same location are enabled at the same time, one of them is taken according to another probability distribution [21]. In the tool UPPAAL, if transitions and locations are not specified with any probability, then the relevant transitions are guarded by uniform distributions, i.e. all possible transitions have the same probability.

In our framework, the correctness of a system is specified using formal logics that define the admissible executions of the system. We use a subset of the Computational Tree Logic (CTL) as defined by the model-checker UPPAAL. The grammar of this subset is $\varphi ::= \mathsf{A}[]P \mid \mathsf{A}<>P \mid \mathsf{E}[]P \mid \mathsf{E}<>P$, where A and E are paths operators, meaning, respectively, "for all path" and "there exists a path". [] and <> are state operators, meaning, respectively, "all states of the path" and "there exists a state in the path". P is an atomic proposition that is valid in some state. For example the formula "A[] not deadlock" specifies that in all the paths and all the states on these paths we will never reach a deadlock state.

2.2 Analysis methods

Model Checking (MC) is an automated verification technique that explores all the possible executions of a model to verify if it satisfies a property expressed in a logic like CTL. While this technique can guarantee that the model satisfies a given property, it has several limitations:

- It is susceptible to state-space explosion if the model

is too large, which can prevent analysis due to a lack of memory.

- It can give only approximated results when verifying SWA, since the exact computation is not possible.

When MC is undergoing state-space explosion, *Statistical Model Checking* (SMC) techniques [16] can be used as an efficient alternative, but with the sacrifice of the absolute certainty provided by MC verification.

SMC is also used to analyze scheduling systems with stochastic features, using PTA. In this context, the scheduling problem becomes to compute, for a given set of tasks, the probabilities of schedulability and the possible response time of individual tasks. We briefly explain the principles of SMC hereafter.

SMC combines formal verification and techniques from the statistic area. For instance, the Monte-Carlo algorithm computes N executions of the model ρ and estimates the probability γ that ρ satisfies a logical formula φ using the following equation:

$$\tilde{\gamma} = \frac{1}{N} \sum_{i=1}^{N} \mathbf{1}(\rho \models \varphi)$$

where $\mathbf{1}$ is an indicator function that returns 1 if φ is satisfied and 0 otherwise. It guarantees that the estimate $\tilde{\gamma}$ is close enough to the true probability γ with the probability of error that is controlled by the number N of simulations.

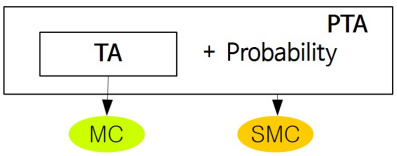

Figure 3: Model analysis

Fig. 3 shows how to analyze TA and its extensions. MC accepts TA models, while SMC accepts PTA. UPPAAL MC accepts PTA models but ignores all stochastic aspects of the model. Meanwhile, UPPAAL SMC accepts a TA model, that is interpreted using uniform probability distribution whenever non-deterministic choices have to be made.

For this reason, we apply both techniques, MC and SMC, to the same model. First, we check a model with SMC and repeat until the certainty of our verification of a model is close to 100%. Then, we apply MC to our model to be sure that the system satisfies a given property.

2.3 Feature Model

A Feature Model (FM) is a well-known specification and analysis model for SPLs. It is used to manage the commonality and the variability of products in a simple and easy-to-understand way [9]. A feature is a distinguished aspect, quality, property, or characteristic of a product. A FM organizes features of products together with constraints among them. A product generated from a FM is then a set of included features that satisfies all the feature constraints.

We adopt the syntax shown in Figure 4 for property specification. The root is the representative feature of a family of products. The child nodes of the root are features either included or excluded in the products. Individual features are linked to its parent node by connectors that represents their

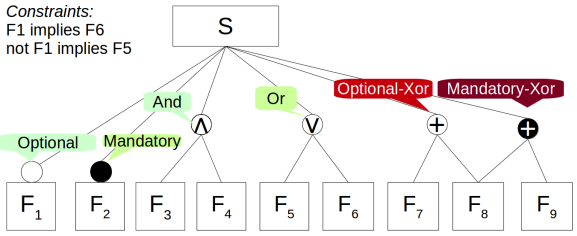

Figure 4: Feature operators of a FM

modality, such as optional/mandatory inclusions, and/or compositions, optional-xor/mandatory-xor inclusions.

3. PROPERTY FEATURE MODEL

We analyze SPLs of real-time systems with respect to the following properties:

- Deadlock properties, that concern all the components in the system.
- Schedulability properties, such that tasks with deadlines do not exceed them.
- Performance properties, that evaluate the response time of tasks.

Inspired by [20, 27], we propose a new extension of feature model called Property Feature Model (PFM) that distinguishes between features and properties using property-specific operators. It states two pieces of important information: the scope of a property and a list of properties that individual features must satisfy. A property can either be local when associated to a leaf feature, or global when associated to the root feature. The association between a feature f and a property p instantiates a satisfiability relation $f \models p$, which can be represented by a CTL formula.

3.1 Syntax for PFM

A PFM is described using the similar notations of a FM. The root of a PFM is a feature, that has child features or properties. A property node can have another property node as its child, but not a feature node. A property can be represented by the composition of multiple properties.

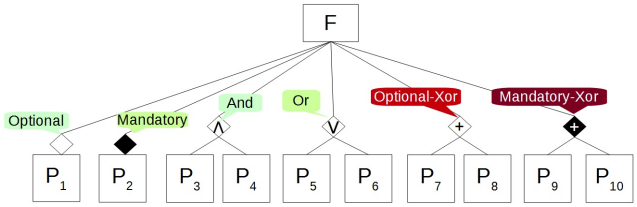

Figure 5: Property-specific operators PFM

Figure 5 shows property-specific operators of a PFM in graphical notations:

- Optional: the child property may or may not be satisfied,
- Mandatory: the child property must be satisfied,
- And: the parent feature must satisfy all child properties,

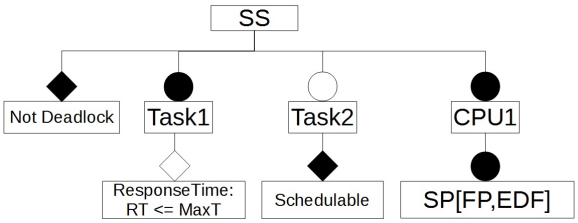

Figure 6: An SPL of a scheduling unit

- Or: the parent feature must satisfy one or more child properties,
- Optional-XOR: at most one property may be included,
- Mandatory-XOR: at most one property must be included.

Similar to the optional feature of a feature model, an optional property of a PFM can represent two products: one that satisfies the property and one that does not. Feature and property nodes can be quantified or given parameters for their products.

Figure 6 shows an example of PFM. The SPL has the root feature SS representing a scheduling unit. It is composed of two mandatory features Task1 and CPU1, one optional feature Task2, and one mandatory property "Not Deadlock." The feature CPU1 is mandatory and quantified by two schedulers, FP (Fixed-Priority) or EDF (Earliest Deadline First). The property node denoted by Not Deadlock states a global property that requires that the root feature SS is never in deadlock when it operates. The property node denoted by Schedulable imposed upon Task2 is a mandatory and local property specifying that Task2 can never miss its deadline. Note that the property node ResponseTime requires a single product satisfying RT<=MaxT, while the feature SP[FP, EDF] requires two products, each of which comes along with the feature SP[FP] or SP[EDF]. A complete example of PFM is presented in Figure 7.

A PFM can be represented by a propositional logic formula with Boolean variables [9]. Each Boolean variable corresponds to a single feature f stating whether the feature is included or not or the satisfiability relation $f \models p$. We allow numeric and arrayed features in propositional logic formulas, like F[A, B, C], instead of Boolean variables [22]. F[A, B, C] abstracts three features: F[A], F[B], and F[C].

A feature and property can be given a parameter to instantiate a product. For instance, WCRT<= MaxT is imposed upon WCRT, restricting WCRT to be less than or equal to MaxT.

Definition 1. *A Property Feature Model (PFM) is a quintuple* $PFM = \{\mathcal{F}, \mathcal{P}, \rightarrow, \models, \psi_F\}$ *such that*

- $\mathcal{F} = \{f_0, ..., f_n\}$ *is a set of features,* f_0 *being the root feature,*
- $\mathcal{P} = \{p_1, ..., p_m\}$ *is a set of properties,*
- $\rightarrow \in 2^{\mathcal{F}}$ *is a parent to child feature relation that encodes the feature structure of the PFM,*
- $\models \in \mathcal{F} \times \mathcal{P}$ *is a satisfiability relation ($f \models p$) meaning that a feature f satisfies a property p.*
- ψ_F *is a propositional logic formula over features and properties that represents the constraints of the PFM.*

Notice that ψ_F includes both a relation between parent and child features and a relation between features that are not in the parent-child relation. In the case where a feature is in association with another feature that is neither parent nor child, an additional proposition logic formula is given to define such a relation.

Example 1. *The PFM in Figure 6 can be defined as:*

$$
\begin{aligned}
\mathcal{F} &= \{SS, Task1, Task2, CPU1, SP.EDF, SP.FPs\} \\
\mathcal{P} &= \{\text{“}Not\ Deadlock\text{”}, \text{“}Schedulable\text{”}, \text{“}WCRT <= MaxT\text{”}\} \\
\rightarrow &= \{(SS, Task1), (SS, Task2), (SS, CPU1), (CPU1, SP.FP), \\
&\quad (CPU1, SP.EDF)\} \\
\models &= \{(SS, \text{“}Not\ Deadlock\text{”}), (Task1, \text{“}WCRT <= MaxT\text{”}), \\
&\quad (Task2, \text{“}Schedulable\text{”})\} \\
\psi_F &= (SS \implies Task1) \wedge (Task1 \implies SS) \\
&\wedge\ (Task2 \implies SS) \\
&\wedge\ (CPU1 \implies SS) \wedge (SS \implies CPU1) \\
&\wedge\ (SP[FP, EDF] \implies SS) \wedge (SS \implies SP[FP, EDF]) \\
&\wedge\ (SS \models NotDeadlock) \\
&\wedge\ (Task1 \models WCRT \leqslant MaxT \implies Task1) \\
&\wedge\ (Task2 \implies Task2 \models Schedulable)
\end{aligned}
$$

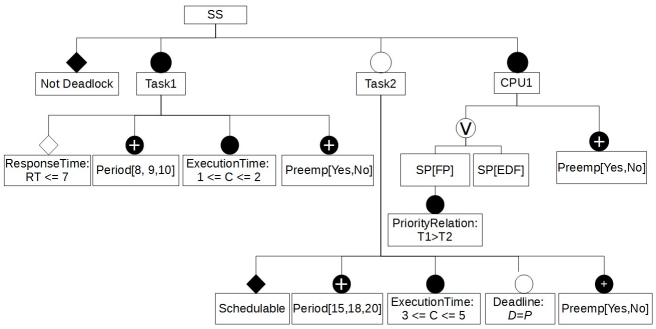

Figure 7: The refined PFM of a scheduling unit

3.2 Product Configuration

A product generated from a PFM is set of included features that satisfy the constraints of the PFM.

We define a product condition that is used to describe requirements of product features requested by the customer. A *product condition* ρ_F is a propositional formula that is a conjunction of condition variables corresponding to individual features in $\{f_0, f_1, ..., fn\}$. It is defined by the following grammar:

$$
\begin{aligned}
\rho_F &::= c \mid \neg c \mid \rho_F \wedge \rho_F \mid e \\
e &::= x == d \mid x > d \mid x < d \mid x \leqslant d \mid x \geqslant d \mid x[f] \\
f &::= d, f \mid d
\end{aligned}
$$

where c is a Boolean variable, x is a numeric or constant variable that are not allowed to contain negation, and d is a numeric or constant value.

Example 2. *A product condition ρ_F for the SPL of the running example in Figure 6 can be:*

$$\rho_F = (SS \wedge Task1 \wedge Task2 \wedge CPU1 \wedge SP[FP, EDF])$$

It describes two components Task1 and Task2, exploiting CPU1 served by two different scheduling policies FP or EDF.

A product condition is checked against the PFM to see if the proposed products are producible from the PFM specified by the product condition. This check can be performed by SMT-solvers [14, 18, 22].

To derive products from a PFM, we define a (product) *configuration* that is a set of condition variables that imply the inclusion, exclusion, or valuation of the corresponding features. Compared to a product condition, it is used to generate all possible products of an SPL, which should satisfy all product conditions that customers require.

Definition 2. *(Configuration): A* configuration γ *is a set of condition variables* $c_i \in \{true, false, v\}$*, each corresponding to a feature* $f_i \in \mathcal{F}$ *or a property* $p_i \in \mathcal{P}$*, such that*

- $c_i = true$ *represents the inclusion of* f_i *or* p_i*,*
- $c_i = false$ *represents the exclusion of* f_i *or* p_i*,*
- $c_i = v$ *represents the assignment of* f_i *to a value* v *in any type.*

For a given PFM, a configuration of a product is created by assigning c_i to one of *true*, *false* or a value v, where c_i has a corresponding feature or property in the PFM. A configuration γ is "*determined*" if no variable c_i remains undetermined, i.e. not included in γ. Then $|\gamma|$ is equal to $|\mathcal{F}| + |\mathcal{P}|$.

Definition 3. *(Propositional Logic Formula Projection): The projection of* ψ_F *over a configuration* γ*, denoted by* $\psi_F|_\gamma$*, returns the formula* ψ_F *in which every variable* v_i *corresponding to a feature* f_i *or a property* p_i *has been substituted with the value of the corresponding condition variable* c_i *in* γ *[22].*

A configuration γ is said to be "*valid*" if $\psi_F|_\gamma$, holds, i.e. the configuration is producible from a feature model ψ_F. Otherwise, the configuration γ is said to be "*invalid.*" Formally, a product is a *valid* and *determined* configuration.

Now, we define a non-deterministic decision process that allows to construct all the products of a PFM compatible with the product condition ρ_F expressed by the customer. The process starts from the configuration $\gamma_0 = \{c_0 = true\}$ that only includes the root feature of the PFM, and it recursively extends this configuration until all the features and all the properties have been determined. Therefore, from a configuration γ a new configuration γ' is produced by extending γ with a feature condition c_i, according to the following rules:

1. $\gamma' = \gamma \cup c_i$,
2. $\exists c_j \in \gamma$ such that either $f_j \rightarrow f_i$, which means that f_i is a child feature of f_j that has already been determined to be included, or $p_i \vDash f_j$, which means that p_i is a property of f_j already determined,
3. $\psi_F|_{\gamma'}$ and $\rho_F|_{\gamma'}$ hold.

The first rule produces a new configuration by including the condition variable c_i corresponding to the decision on the feature f_i or the property p_i. The second rule restricts the decision process to make it follow the order from parent to child defined in the PFM. The last rule checks if a new configuration (γ') satisfies both feature constraints (ψ_F) and customer's requests (ρ_F).

4. FEATURE BEHAVIORAL MODEL

A SPL of a scheduling unit is analyzed to see if the products generated from the SPL satisfy their properties. To this end, all products from an SPL are represented by behavioral models of real-time scheduling units. We model them using timed automata such that properties of an SPL can be analyzed using the behavioral models.

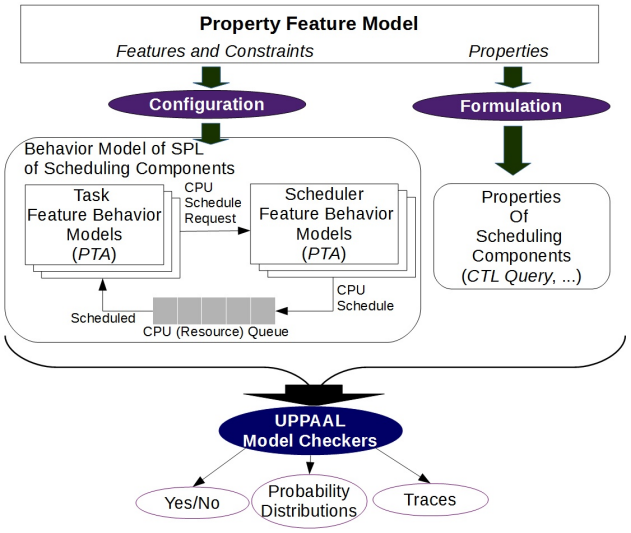

Figure 8: The process of analyzing an SPL of scheduling units using UPPAAL tools

The process of analyzing an SPL of a real-time scheduling unit is presented in Figure 8. With TA and SWA, we build reconfigurable task and resource scheduler models whose properties are instantly updated. The task model executes according to its timing properties, such as a period, an execution time, and a deadline, and its execution depends on the availability of specific resources. The resource scheduler model schedules resource-requiring tasks by managing the status and owner of a resource according to a scheduling policy. Once a configuration of products is formulated from a PFM, the properties of tasks and resource schedulers included by the configuration are instantiated and a scheduling unit model composed of the tasks and the scheduler models is checked against properties from the same PFM. It repeats until all possible configurations of the PFM are checked.

The properties to check are also extracted from the PFM and translated to CTL formulae. They are analyzed with UPPAAL MC and UPPAAL SMC. These analyses would return either a "Yes/No" answer or a probability distribution when UPPAAL SMC is used. It can also extract specific traces from the system behavior if it does not satisfy a property.

4.1 Preemptive Task Model

The scheduling units that we consider in this paper are preemptive, so that the execution of a task can be interrupted by other tasks according to a scheduling policy. Preemption is implemented using stopwatch clocks in SWA models and it is known that the model-checking of SWA is undecidable [2]. However, preemption is one of the main features of real-time tasks. Our solution is to use SMC to check SWA models in order to guarantee the termination of the analysis of preemptive scheduling units. Figure 9 shows the feature

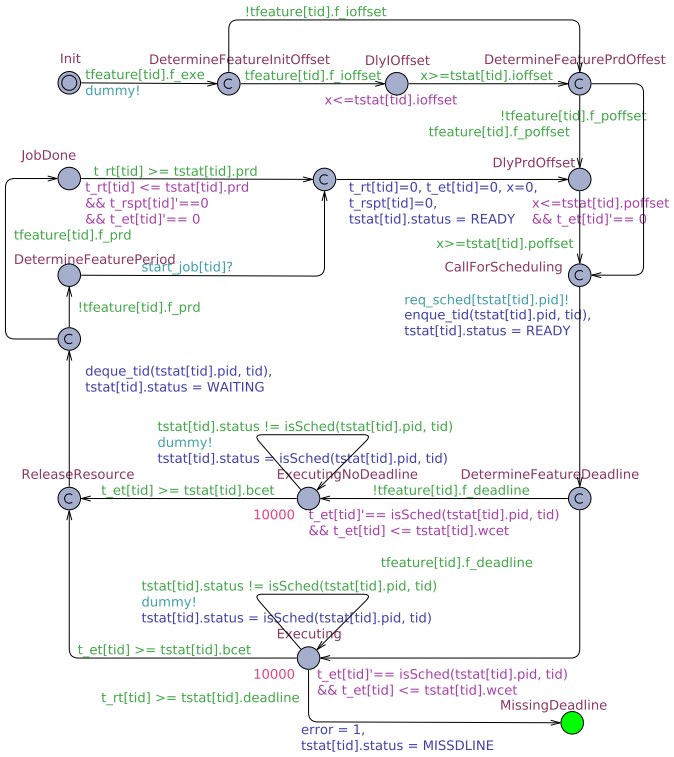

Figure 9: A SWA task model for a family of tasks

behavioral model of a real-time task with preemption. This SWA model refines the basic model presented in Figure 2, inspired by the work in [10]. We have extended this model with variables that encode the enabling, disabling or valuation of the features.

The SWA task model in Figure 9 is a generic model that can be configured to execute any configuration of task producible from the PFM. It captures the behavior of the task after the feature variables have been configured at initialization. Several behaviors are then possible depending on the value of the feature variables. For instance, the location DetermineFeatureInitOffset has two out-going transitions: one to DlyOffset, and the other to DetermineFeaturePrdOffset. The transitions are labeled with a guard that distinguishes a feature and the property of a task is determined by a set of enabled guards. Thus, the transition guard tfeature[tid].f_ioffset is set to true if the feature InitOffset is included, but set to false when the feature is excluded. The other feature variables are tfeature[tid].f_poffset, tfeature[tid].f_deadline and tfeature[tid].f_prd are associated to the features PeriodOffset, Deadline, and Period, respectively.

After the initialization phase, the task eventually reaches a location that corresponds to the execution of the task. The location ExecutingNoDeadline does not consider a deadline, on the contrary to location Executing, that allows to reach location MissedDeadline if the deadline is missed.

While executing, the task may be preempted by the resource scheduler. The preemption is implemented by a stopwatch clock t_et[tid] that can stop and resume. It represents the remaining execution time of the task tid and it should progress only when the CPU resource is available to the task. This stopwatch t_et[tid] is constrained by an invariant that is associated with a function isSched(). The preemption mechanism is as follows: when the task must be preempted, the function isSched() is manipulated by the scheduler such that it returns 0, which indicates that the resource is no longer available to the running task, and then the clock t_et[tid] stops. When the resource is available again, the function isSched() returns 1, and then the clock t_et[tid] resumes its progress. Finally when the execution is completed, the task reaches the location JobDone and awaits the next period.

We present in Appendix A a detailled PFM of a real-time task, and the feature behavioral models for real-time resources and schedulers

5. EVALUATION

This section presents results of analyzing the SPL of Figure 7. Using UPPAAL MC we check the schedulability of the tasks and deadlock freedom as well. UPPAAL SMC is used to estimate the worst-case execution time of tasks, individually.

In addition to the feature behavioral models of tasks and resources, we provide a configuration template that generates configurations of real-time systems out of a given PFM before the execution of the system. A configuration template simulates the non-deterministic decision process presented in Section 3 and selects features from a PFM in a non-deterministic way to make a configuration of the system under analysis

5.1 Analysis of the Running Example

In this section, we present the analysis results of Figure 7. The running example of Figure 7 has only 2 tasks and no constraints over configurations. The feature Task1 has 6 configurations, the feature Task2 has 12 configurations, and the feature CPU1 has 4 configurations. SS has 24 configurations without Task2, and 288 configurations with Task2.

Table 1: Timing analysis results for the SPL in Figure 7

Feature	Query for Property	Results	Time
SS	A[] not deadlock	Yes	28.43s
$Task1$	E[<=10000;100](max:t_rspt[1])	6.80	3.22s
$Task2$	A[] (tstat[2].status ! = MISSDLINE)	Yes	29.85s

Table 1 shows the results of analyzing the properties included in the SPL. First, the property of SS "Not Deadlock" is formulated as the CTL query "A[] not deadlock" stating that the system is deadlock-free. The property is proven to hold in the system. Second, the schedulability of Task2 is analyzed. The CTL query, "A[] (tstat[2].status ! = MISSDLINE)", is used as a specification, meaning that the state variable tstat[2].status can never be the same as "MISSDLINE" while the system is running. UPPAAL MC verified that Task2 never misses the deadline. Third, we analyzed the performance, i.e. the response time of a task, of configurations from the SPL. The property RT<=7 upon Task1 in the SPL is represented by a SMC query, E[<=10000;100](max:t_rspt[1]), requiring UPPAAL SMC to compute the average of the maximum value of t_rspt[1] for 10,000 simulation times by 100 simulation rounds.

UPPAAL SMC produces a probability distribution, as the answer to the query, shown in Figure 10. It shows that the response times of the task is at most 6.80 time units during the simulation and validates that the worst-case response time of Task1 is less than 7.

In Appendix B we present the analysis results of another

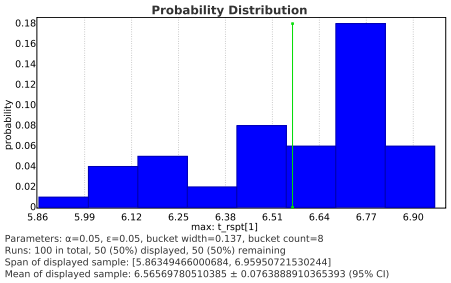

Figure 10: Probability distribution of Task1's response time

case-study containing 5 real-time tasks and 2 resource schedulers.

6. RELATED WORK

Numerous works addressed the problem of model checking a family of products and considered various kinds of properties including structural or behavioural ones [4, 5, 12–15, 19, 22–24, 26, 27]. Model checking an SPL requires a formalism to encode properties that have to be checked [27]. Numerous approaches rely on computation tree logic (CTL) [12, 19] or linear temporal logic (LTL) [12, 23, 24, 26]. Apel et al. [4] model temporal safety properties. Asirelli et al. [5] propose a branching-time temporal logic. Cordy et al. [15] utilize timed CTL, an extension of CTL with support for modeling real-time properties on FTS. This paper, in contrast, specifically considers schedulability aspects and proposes to leverage formal techniques for verifying a family of products. Moreover, we can verify the CTL properties of all the products in one step. Our framework offer means to model variability (through an extension of feature model) and automated translations to Uppaal and Uppaal SMC for the verification of schedulability properties.

One of the work closely related to this paper is Sabouri et al. [22]. The authors proposed an SPL framework for scheduling units on the application level, in which schedulability is verified by Uppaal using modular schedulability analysis. In contrast with [22], we focus on the platform-level of the scheduling units. Furthermore we consider not only non-preemptive scheduling units, but also preemptive scheduling units. The verification process combines Uppaal MC and Uppaal SMC (hence supporting probabilities).

The verification and validation of scheduling systems using Uppaal is inspired by [10], where a hierarchical scheduling component is specified with timed automata and stopwatch automata. In this paper, we address the problem of verifying a family of scheduling systems. We thus have to adapt at the specification and reasoning level the formalisms and techniques to take variability into account. We modify the model of the scheduling units so that features of the system can be selected and deselected. Our framework allows practitioners to specify variability and benefit from advanced reasoning support.

The formalism of feature models in this paper relies on the basic and classical constructs of [1, 7]. Our extension of FM was inspired by Kang et al. [20] that criticizes the existing feature model by saying that it often specifies one or more concerns of SPL in one FM. Related to the quality of an SPL, they proposed an attribute-based feature model where only qualities of products are separately given as a FM. However, such a representation makes it hard to explicitly figure out the relationship between a feature and the associated quality attributes (i.e. properties). For this reason, this paper extended FM with the related properties so that a verification property is associated to a feature in one FM through specific operators.

7. CONCLUSIONS

SPLE aims to provide efficient engineering solutions for building multiple products that share common features. This paper proposed a formal framework dedicated to the verification of SPLs that should satisfy schedulability properties.

Specifically, we proposed a new formalism for variability modeling, called PFM, to define feature models together with feature properties, and defined the notion of product condition that represents customer's product requests. We formally defined the semantics of PFM so that the SPL modeled in the PFM can automatically generate valid configurations in compliance with customer's requests. In order to analyze the configured products against feature properties, we proposed behavioral models that capture the features of real-time scheduling units defined in the PFM. We then showed how a set of scheduling units in an SPL specification can be automatically verified against the set of required properties by leveraging efficient model checking methods. Throughout the paper we illustrated the formal framework with a family of scheduling units and showed the applicability and efficiency of our techniques.

As future work we plan to investigate the scalability of our proposal w.r.t. large, variability-intensive scheduling systems. We also want to include a wider range of schedulability properties in our verification process.

8. REFERENCES

[1] M. Acher, P. Collet, P. Lahire, and R. B. France. Familiar: A domain-specific language for large scale management of feature models. *Science of Computer Programming (SCP)*, 78(6):657–681, 2013.

[2] R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138:3–34, 1995.

[3] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[4] S. Apel, A. v. Rhein, P. Wendler, A. Grösslinger, and D. Beyer. Strategies for product-line verification: Case studies and experiments. In *Proceedings of the 2013 ICSE*, ICSE'13, pages 482–491, 2013.

[5] P. Asirelli, M. ter Beek, A. Fantechi, and S. Gnesi. A compositional framework to derive product line behavioural descriptions. volume 7609 of *LNCS*, pages 146–161, 2012.

[6] P. Asirelli, M. H. ter Beek, S. Gnesi, and A. Fantechi. Formal description of variability in product families. In *SPLC'11*, pages 130–139, 2011.

[7] D. Batory. Feature models, grammars, and propositional formulas. In *Proceedings of the 9th International Conference on Software Product Lines*, SPLC'05, pages 7–20, 2005.

[8] G. Behrmann, A. David, K. G. Larsen, J. Håakansson, P. Pettersson, W. Yi, and M. Hendriks. UPPAAL 4.0.

In *Third International Conference on the Quantitative Evaluation of Systems*, QEST'06, pages 125–126, 2006.

[9] D. Benavides, S. Segura, and A. Ruiz-Cortes. Automated analysis of feature models 20 years later: a literature review. *Information Systems*, 35(6), 2010.

[10] A. Boudjadar, A. David, J. H. Kim, K. G. Larsen, M. Mikucionis, U. Nyman, and A. Skou. Hierarchical scheduling framework based on compositional analysis using uppaal. In *FACS 2013*, volume 8348 of *LNCS*, pages 61–78. Springer, 2013.

[11] R. Braga, O. Trindade Junior, K. Castelo Branco, L. Neris, and J. Lee. Adapting a software product line engineering process for certifying safety critical embedded systems. In *Computer Safety, Reliability, and Security*, volume 7612 of *LNCS*, pages 352–363. Springer, 2012.

[12] A. Classen, M. Cordy, P. Schobbens, P. Heymans, A. Legay, and J. Raskin. Featured transition systems: Foundations for verifying variability-intensive systems and their application to LTL model checking. *IEEE Trans. Software Eng.*, 39(8):1069–1089, 2013.

[13] A. Classen, P. Heymans, P.-Y. Schobbens, A. Legay, and J.-F. Raskin. Model checking lots of systems: efficient verification of temporal properties in software product lines. In *ICSE'10*, pages 335–344. ACM, 2010.

[14] M. Cordy, P. Schobbens, P. Heymans, and A. Legay. Beyond boolean product-line model checking: dealing with feature attributes and multi-features. In *35th ICSE, ICSE '13*, pages 472–481, 2013.

[15] M. Cordy, P.-Y. Schobbens, P. Heymans, and A. Legay. Behavioural modelling and verification of real-time software product lines. In *Proceedings of the 16th International SPLC - Volume 1*, SPLC '12, pages 66–75. ACM, 2012.

[16] A. David, K. Larsen, A. Legay, M. Mikučionis, and D. Poulsen. Uppaal smc tutorial. *International Journal on Software Tools for Technology Transfer*, pages 1–19, 2015.

[17] A. David, K. Larsen, A. Legay, M. Mikučionis, D. Poulsen, J. van Vliet, and Z. Wang. Statistical model checking for networks of priced timed automata. In U. Fahrenberg and S. Tripakis, editors, *Formal Modeling and Analysis of Timed Systems*, volume 6919 of *LNCS*, pages 80–96. Springer Berlin Heidelberg, 2011.

[18] V. Ganesh. *Decision Procedures for Bit-vectors, Arrays and Integers*. PhD thesis, Stanford, CA, USA, 2007. AAI3281841.

[19] J. Greenyer, A. Molzam Sharifloo, M. Cordy, and P. Heymans. Features meet scenarios: modeling and consistency-checking scenario-based product line specifications. *Requirements Engineering*, 18(2):175–198, 2013.

[20] K. Kang and H. Lee. Variability modeling. In *Systems and Software Variability Management*, pages 25–42. Springer, 2013.

[21] G. Norman, D. Parker, and J. Sproston. Model checking for probabilistic timed automata. *Formal Methods in System Design*, 43(2):164–190, 2013.

[22] H. Sabouri, M. Jaghoori, F. de Boer, and R. Khosravi. Scheduling and analysis of real-time software families. In *Computer Software and Applications Conference (COMPSAC), 2012 IEEE 36th Annual*, pages 680–689, July 2012.

[23] H. Sabouri and R. Khosravi. Modeling and verification of reconfigurable actor families. *J. UCS*, 19(2):207–232, 2013.

[24] I. Schaefer, D. Gurov, and S. Soleimanifard. Compositional algorithmic verification of software product lines. In *Formal Methods for Components and Objects*, volume 6957 of *LNCS*, pages 184–203. 2012.

[25] R. F. Scheidt, K. Schmidt, G. M. Pessoa, M. A. Viera, and M. Dantas. A software product line approach to enhance a meta-scheduler middleware. *Journal of Physics: Conference Series*, 341(1):012030, 2012.

[26] M. H. ter Beek, A. L. Lafuente, and M. Petrocchi. Combining declarative and procedural views in the specification and analysis of product families. In *Proceedings of the 17th SPLC Co-located Workshops*, SPLC '13 Workshops, pages 10–17. ACM, 2013.

[27] T. Thüm, S. Apel, C. Kästner, I. Schaefer, and G. Saake. A classification and survey of analysis strategies for software product lines. *ACM Comput. Surv.*, 47(1):6:1–6:45, June 2014.

APPENDIX

A. FAMILIES OF REAL-TIME COMPONENTS

A SPL of real-time scheduling units and its behavior model are designed to encompass representative features and properties of real-time system components. Put simply, a family of scheduling units consists of a family of tasks and a family of resources. The behavior of the SPL of scheduling units is captured by a model composed of a behavior model of the family of tasks and a behavior model of the family of resources. Figure 11 shows a PFM of real-time tasks featured with representative real-time features and properties.

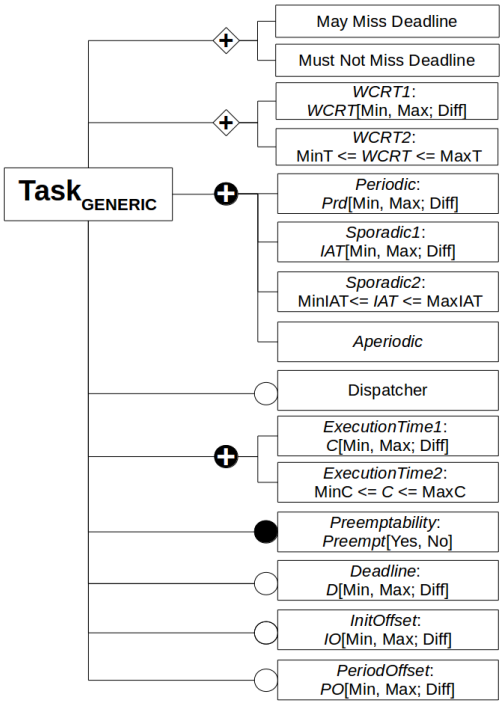

Figure 11: PFM of a real-time task

The task behavior model of Figure 9 is designed to encompass all the features and properties of the task family of Figure 11.

A.1 Feature Behavioral Model of Resources

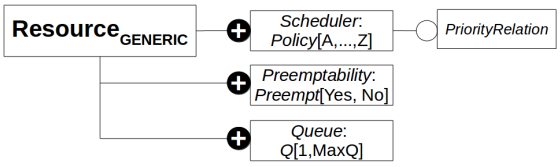

Figure 12: PFM of a real-time resource

Figure 12 shows PFMs of resources of real-time systems. A resource always comes along with a scheduler observing a scheduling policy. The owner of a resource can change even though the current owner hasn't yet finish using it. A resource can have a queue where the resource-requesting tasks wait for the use of it.

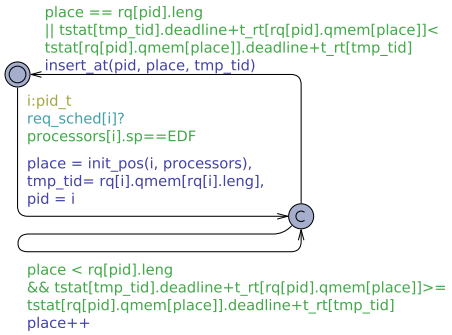

Figure 14: EDF scheduler

The feature behavioral model of an SPL resource is composed of a resource queue and a resource scheduler. The resource queue is realized with the "typedef" construction from UPPAAL's specific language, as shown in Listing 1. The resource scheduler model is constructed with a TA and C-style function definition, as shown in Figure 12.

Listing 1: Resource queue

```
typedef struct {
pri_arr    leng;              % The length of queue
tid_arr    qmem[tid_t];       % The member of tasks
} queue_t;
queue_t    rq[pid_t] ;          % Resource queue
```

Listing 1 shows the resource queue model implemented with a "typedef" construction. The variable leng denotes the length of a queue. The array qmem sorts the identities of the jobs requesting the resource according to their priorities. A resource is instantiated by the declaration of the corresponding queue variable (rq[pid_t]), where pid_t is an array of resources.

Listing 2: Resource feature

```
typedef struct{
bool active;                % Usability
bool preemptive;            % Preemptiveness
schedpolicy_t sp;           % Scheduling policy
} processor_t;
```

The feature of resources is implemented by using the "typedef" construction, as shown in Listing 2. The availability of the resource is indicated by a Boolean variable active. The preemptability of the resource is realized by a Boolean variable preemptive. A specific scheduling policy is associated with the resource by the variable sp.

A.2 Feature Behavioral Model of Schedulers

To share a resource between several tasks a scheduling mechanism is needed. This scheduling mechanism decides according to a scheduling policy at each instant which task is allowed to exploit the resource. Figure 14 shows the TA model of the EDF (Earliest Deadline First) resource scheduler, which sorts a set of tasks according to the slack time, i.e. the remaining time to a deadline. In a similar way, different types of resource scheduler, Fixed-Priority and FIFO, are modeled with TA [10].

B. A CASE STUDY

To see the feasibility of our verification capability, we conduct a further case study, as shown in Figure 13, where a

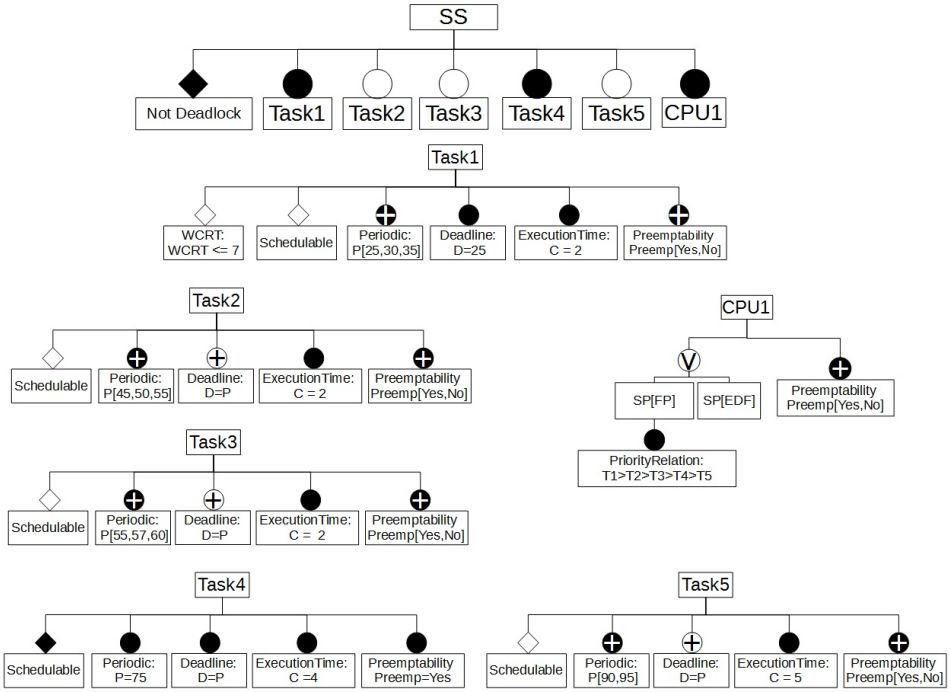

Figure 13: PFM of scheduling units with 5 tasks

Table 2: Constraints of the SPL of Figure 13

ID	Constraints
c1	$Task1 \wedge \neg Task2toCPU1.Preempt = false$
c2	$Task1 \wedge Task2toCPU1.Preempt = true$
c3	$Task1 \wedge \neg Task2 \rightarrow (Task1.P = 25 \wedge Task1.Preemp = No)$
c4	$Task1 \wedge (Task2 \vee Task3) \rightarrow (Task1.P = 30 \wedge Task1.Preemp = No)$
c5	$Task1 \wedge ((Task2 \wedge Task3 \wedge Task4) \vee (Task3 \wedge Task4 \wedge Task5)) \rightarrow (Task1.P = 35 \wedge Task1.Preemp = yes)$
c6	$\neg Task5 \rightarrow Task2 = true$
c7	$Task2 \wedge Task1.P <= 30 \rightarrow (Task2.P = 45 \wedge Task2.deadline = false \wedge Task2.Preempt = false)$
c8	$Task2 \wedge Task1.P < 30 \rightarrow (Task2.P = 50 \wedge Task2.P = Task2.D \wedge Task2.Preempt = false)$
c9	$Task2 \wedge Task1.P \leqslant 35 \rightarrow (Task2.P = 55 \wedge Task2.P = Task2.D \wedge Task2.Preempt = true)$
c10	$\neg Tasl2 \rightarrow Task3 = false$
c11	$CPU1 \rightarrow Task3 = true$
c12	$Task3 \wedge Task1.P \leqslant 25 \rightarrow Task3.P = 55 \wedge Task3.deadline = flase \wedge Task3.Preempt = true$
c13	$Task3 \wedge Task1.P \leqslant 35 \rightarrow Task3.P = 57 \wedge Task3.deadline = flase \wedge Task3.Preempt = true$
c14	$Task3 \wedge Task1.P \geqslant 35 \rightarrow Task3.P = 60 \wedge Task3.D = Task3.P \wedge Task3.Preempt = true$
c15	$Task5 \rightarrow Task1.P \leqslant 25 \rightarrow Task5.P = 90 \wedge Task5.deadline = false \wedge Task5.Preempt = true$
c16	$Task5 \rightarrow Task1.P \leqslant 35 \rightarrow Task5.P = 95 \wedge Task5.D = Task5.P \wedge Task5.Preempt = true$

SPL of scheduling units is composed of 5 tasks and 2 resource schedulers with 16 constraints. We checked the schedulability of 4 hard real-time tasks and estimated the worst-case response times of some of the tasks.

Table 3 shows the results of the analysis. We analyzed this model using Uppaal MC and Uppaal SMC on Intel 4-Core CPU 2.90GHz with 8 GB Memory in Window OS 64 Bits.

The first line of Table 3 shows that the system is free from deadlock. The fourth line shows that no task misses the deadline. The last line is the result returned by Uppaal SMC, which shows that the probability of missing the deadline of tasks is between 0 and 0.0199955 with the confidence of 0.99. It shows that even though a SPL of a preemptive scheduling system cannot be checked by a MC technique, SMC can verify the system with a quantified analysis result withs a limited confidence.

Table 3: Timing analysis results for the SPL of scheduling units with 5 tasks with 16 constraints

Feature	Query for Property	Results	Time
SS	A[] not deadlock	Yes	25,322.8 s
$Task1$	E[<=10000;100](max:t_rspt[1])	66.6	0.28s
$Task4$	E[<=10000;100](max:t_rspt[4])	71.75	0.31s
SS	A[] not miss_deadline	Yes	25,134.51 s
SS	Pr [<=1000000] <> miss_deadline	[0,0.0199955]	302.51 s