Model Checking Conformance of Design Model to Its Formal Specification

Dieu-Huong Vu, Yuki Chiba, Kenro Yatake, Toshiaki Aoki

School of Information Science,
Japan Advanced Institute of Science and Technology
{huongvd, chiba, k-yatake, toshiaki}@jaist.ac.jp

Abstract. Verification of a design with respect to its requirement specification is important to prevent errors before constructing an actual implementation. Existing works focus on the verifications where specifications are described using temporal logics or using the same languages as that used to describe designs. In this paper, we consider cases where specifications and designs are described using different languages. For verifying such cases, we propose a framework to check if a design conforms to its specification based on their simulation relation. Specifically, we define the semantics of specifications and designs commonly as labelled transition systems (LTS), and check if a design conforms to its specification based on the simulation relation of their LTS. In this paper, we present our framework specialized for the verification of reactive systems, and we present the case where specifications and the designs are described in Event-B and Promela/Spin, respectively. As a case study, we show an experiment of applying our framework to the conformance check of the specification and the design of OSEK/VDX OS.

Keywords: Specification, Design, Simulation Relation, Model Checking

1 Introduction

In a general process of software development, we start from informal requirements which target software is expected to satisfy. Such requirements are translated into formal specifications in order to describe them properly. We then develop system designs as models of implementations. Finally, we construct implementations based on the designs using programming languages. In this development process, it is expected that designs satisfy requirements described by formal specifications, because this allows incorrect designs to be revised before significant investments are paid for actual implementations.

For verifying designs with respect to their specifications, existing works focus on the cases where specifications are described using temporal logics or using the same languages as that used to describe designs. In this paper, we consider cases where specifications and designs are described using different languages. For verifying such cases, we propose a framework to check if a design conforms to its specification based on their simulation relation [17]. Specifically, we define the

semantics of specifications and designs commonly as labelled transition systems (LTS), and check if a design conforms to its specification based on the simulation relation of their LTS.

In this paper, we present our framework specialized for the verification of reactive systems, and present the case where specifications and designs are described in Event-B [1] and Promela/Spin [9], respectively. Event-B is effective for describing event driven systems, and Promela can also describe functions of reactive systems as a set of (inline) functions. Figure 1 summarizes our approach. Firstly, we describe specification formally in Event-B model [19]. This is to remove ambiguity and inconsistency in the specification which is written in a natural language. Then, we generate execution sequences from this formal specification. This is performed by a tool called execution sequence generator. Execution sequences are represented as an LTS, and from each state, verification conditions which must be met by the corresponding state of the design are generated. Finally, we apply model checking [3] to the design in combination with the execution sequences to check the verification conditions. By this, we can check the correspondence of state transitions, or simulation relation, between the execution sequences and the design. This ensures that the design conforms to the specification.

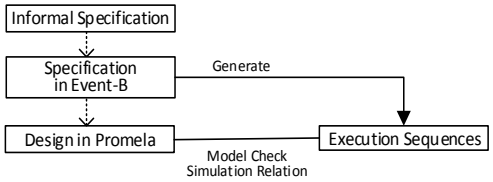

Fig. 1: Scenario for the formal verification of the design

In this paper, we present the formal definition of our framework, and as a case study, we show an experiment of applying our framework to the conformance check of the specification and the design of OSEK/VDX OS [15] (OSEK OS, for short).

The paper is organized as follows: In Section 2, we show the definitions of LTS and simulation relation. In Section 3 and 4, we present the definitions of specifications and designs, respectively. In Section 5, we present the definition of our verification framework. In Section 6 and 7, we present a case study of verifying OSEK OS, and discuss the effectiveness of our framework. In Section 8, we cite related work. In Section 9, we conclude this paper.

2 Preliminaries

In this section, we introduce the definitions of LTS and simulation relation.

First, we show the definition of LTS which is used to represent semantics of specifications and designs.

Definition 1. *(LTS). A* labeled transition system *(*LTS*, for short) is a tuple* $\langle Q, \Sigma, \delta, I \rangle$ *where* Q *is a non-empty set of states,* Σ *is a set of actions,* $\delta \subseteq$

$Q \times \Sigma \times Q$ *is a transition relation, and* $I \subseteq Q$ *is a set of initial states. We write* $(p, a, p') \in \delta$ *as* $p \xrightarrow{a} p' \in \delta$.

Then, we define n-steps transition relation.

Definition 2. *(n-steps transition relation). Let* $M = \langle Q, \Sigma, \delta, I \rangle$ *be an LTS,* Σ^+ *the set of non-empty strings of* Σ*, and* p *and* p' *states. We say* p' *is* reachable from p *with respect to a string* $a_1 a_2 ... a_n \in \Sigma^+$ *by* δ *(denoted* $p \xrightarrow{a_1 a_2 ... a_n} p' \in \delta^+$*), if there exist states* $p_1, p_2, ..., p_{n-1} \in Q$ *such that* $p \xrightarrow{a_1} p_1 \in \delta$, $p_{i-1} \xrightarrow{a_i} p_i \in \delta$ *for* $2 \leq i \leq n-1$, *and* $p_{n-1} \xrightarrow{a_n} p' \in \delta$.

Finally, we define the simulation relation based on the above definitions.

Definition 3. *(Simulation relation). Let* $M_1 = \langle Q_1, \Sigma_1, \delta_1, I_1 \rangle$ *and* $M_2 = \langle Q_2, \Sigma_2, \delta_2, I_2 \rangle$ *be LTSs, and* $f : \Sigma_1 \to \Sigma_2^+$ *a function from* Σ_1 *to* Σ_2^+*. Suppose a relation* $\preceq \subseteq Q_1 \times Q_2$ *is given.* $M2$ simulates $M1$ *with respect to* $\preceq$ *if for all* $q_1, q_1' \in Q_1$, $q_2 \in Q_2$, $a \in \Sigma_1$ *such that* $q_1 \preceq q_2$ *and* $q_1 \xrightarrow{a} q_1' \in \delta_1$, *there exist* $q_2' \in Q_2$ *such that* $q_1' \preceq q_2'$ *and* $q_2 \xrightarrow{f(a)} q_2' \in \delta_2^+$*. If* $M2$ *simulates* $M1$ *with respect to* $\preceq$*, we denote* $M1 \preceq M2$.

3 Specifications

3.1 Specification in Event-B

A reactive system is the system that responds to external events. A reactive system is captured as a collection of services which respond to the invocations from the outside. Event-B is the specification language which is useful to model the event-driven systems like the reactive systems. Formal specification described in Event-B is as a highly abstracted level description of the systems. This description mainly consists of state variables, operations on the variables, and state invariant. The variables are typed using set theoretic constructs such as sets, relations, and functions. The events (operations) specify substitutions, which allow both deterministic and nondeterministic state transitions. Figure 2 illustrates a part of the formal specification of OSEK OS described in Event-B: (i) the variables represent sets of entities managed by OSEK OS for instance tasks, resources, and the ready queue; and (ii) the events describe system services such as task activation, task termination, and resource getting. System services represent the interface between the operating system and the outside. Behavior of events is described by value assignments to variables.

3.2 Formalization

We introduce the notion of models for specifications in our framework. Our model of specifications is based on Event-B, but not restricted to it.

$\mathcal{V}$ is the set of *variables*. $\mathcal{D}$ is the *domain*, which is the set of values. Exp is the set of expressions in specifications. An *expression* may contain variables in $\mathcal{V}$, values in $\mathcal{D}$, arithmetic operator, and set operators. BExp is the set of boolean

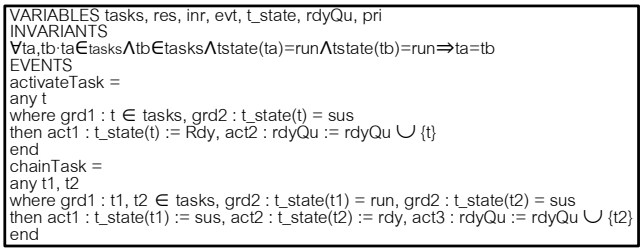

Fig. 2: A specification of OSEK OS in Event-B

expressions ($BExp \subset Exp$). A *substitution* is a mapping from $\mathcal{V}$ to Exp. We note that value assignments are also substitutions because $\mathcal{D} \subseteq Exp$. ACT is the set of substitutions for specifications. A *guard* is a boolean expression. GRD is the set of guards. An *event* is a pair $\langle g, a \rangle$ of a guard g and a substitution a. $\mathcal{E}$ is the set of events. If $e = \langle g, a \rangle$ then we write $grd(e) = g$ and $act(e) = a$. A *state* is a value assignment. $[exp]_\sigma$ denotes the interpretation of the value of an expression exp in a state σ. We say a guard g holds in a state σ iff $[g]_\sigma = tt$. Init is the initialization, which is the set of events whose guards hold for any states and actions are value assignments. We denote $\sigma \xrightarrow{e} \sigma'$ for an event $e = \langle g, a \rangle$ and states σ and σ' if $\sigma(g)$ holds and $\sigma' = \{v \mapsto [a(v)]_\sigma \mid v \in V\}$.

Sematics of specifications is provided by *specification models*.

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

4 Designs and Environments of target system

4.1 Design in Promela

In Promela, service functions of reactive systems can be described by using inline functions. The left hand side of Figure 3 illustrates a design of OSEK OS. We call this model a *design model*. It is constructed based on the informal specification of OSEK OS and described in about 2800 lines of Promela code, according to the approach in [2]. It first defines data structures such as `tsk`, `res`, and `ready` which represent an array of tasks, an array of resources, and ready queues, respectively. Following these data structure, a set of functions are defined. For example, `_ActivateTask` and `_TerminateTask` are the functions to perform activation and termination of tasks, respectively.

Since a reactive system only defines a set of functions, it cannot operate by itself. To operate it, we need an environment which calls functions to the reactive system. In the case of operating systems, an environment means a software

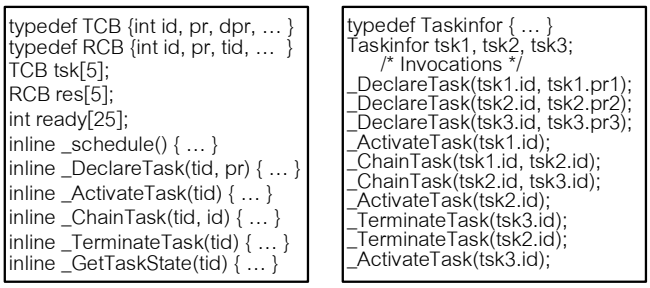

Fig. 3: A design model of OSEK OS and its environment in Promela

application running on it. The right hand side of Figure 3 shows an example of an environment for the OSEK OS design. We call this model an *environment model*. It first defines entities in the environment such as tasks and resources. Then, it defines a sequence of function calls to the OSEK OS. By combining the design and environment, we can make a closed system which can operate by itself. We call this model a *combination model*. In terms of Promela, a combination model can be obtained by including the Promela code of the design into that of the environment model. As we explain later, an environment model is constructed from the specification model, and input to Spin to check simulation relation.

4.2 Formalization

$\mathcal{P}$ is the set of *parameters* (function arguments). A *parameterized expression* may contain constants, variables, parameters and arithmetic operators. The set of parameterized expression is denoted as PExp. A *p-substitution* (function body) is a mapping from $\mathcal{V}$ to PExp. The set of p-substitution is denoted as PSubst. Id is the set of *identifiers* (used as function names). For the simplicity, we assume that functions has only one parameter. Design models are defined as follows.

Definition 5. *(Design model). A* design model *is a tuple* $D = \langle \mathcal{V}_D, \mathcal{D}_D, \mathcal{P}_D, F, \Sigma_D, I_D \rangle$ *where* $\mathcal{V}_D \subseteq \mathcal{V}$ *is the set of variables used in* D, $\mathcal{D}_D \subseteq \mathcal{D}$ *is the domain of* D, $\mathcal{P}_D \subseteq \mathcal{P}$ *is a finite set of parameters for* D, F *is a set of function identifiers defined as* $F = \{id(p) \mid id \in \mathrm{Id}, p \in \mathcal{P}_D\}$, Σ_D *is a relation such that* $\Sigma_D \subseteq F \times \mathrm{PSubst}$, *and* $I_D \subseteq \{\sigma \mid \sigma : \mathcal{V}_D \to \mathcal{D}_D\}$ *a set of value assignments from* $\mathcal{V}_D$ *to* $\mathcal{D}_D$.

Environment models are defined as follows.

Definition 6. *(Environment model). An* environment model for a design model D *is a tuple* $E = \langle \mathcal{V}_E, \mathcal{D}_E, \Sigma_E, I_E \rangle$ *where* $\mathcal{V}_E \subseteq \mathcal{V}$ *is a set of variables used in* E, $\mathcal{D}_E = \mathcal{D}_D$ *is the domain of* E, Σ_E *is a set of* invocations to D *such that* $\Sigma_E \subseteq \{id(v) \mid id \in \mathrm{Id}, v \in \mathcal{V}_E\}$, *and* I_E *is a set of value assignments from* $\mathcal{V}_E$ *to* $\mathcal{D}_D$. *An LTS derived from the environment model* E *is defined as* $M_E = \langle Q_E, \Sigma_E, \delta_E, I_E \rangle$ *where* $Q_E = \{\sigma \mid \sigma : \mathcal{V}_E \to \mathcal{D}_E\}$, $\delta_E \subseteq Q_E \times \Sigma_E \times Q_E$, *and* $I_E \subseteq Q_E$.

A combination of a design and an environment describes the execution of the design according to the environment. An expression in combination contains

constants from $\mathcal{D}$, variables in $\mathcal{V}$, and arithmetic operators. The set of expressions in combinations is denoted as Exp$'$. A substitution for combinations is a mapping from $\mathcal{V}$ to Exp$'$. The set of substitutions for combinations is denoted as SubstDE. For a mapping π from $\mathcal{P}$ to $\mathcal{V}$ and a parameterized expression $pexp \in$ PExp, $pexp_\pi$ is the result of replacing each parameter p appearing in $pexp$ by $\pi(p)$.

Combination models are defined as LTS as follows.

Definition 7. *(Combination model). Let $D = \langle \mathcal{V}_D, \mathcal{D}_D, \mathcal{P}_D, F, \Sigma_D, I_D \rangle$ be a design model and $E = \langle \mathcal{V}_E, \mathcal{D}_E, \Sigma_E, I_E \rangle$ an environment model.*

1. *We denote $\sigma \xrightarrow{id(v)} \sigma'$ for an invocation $id(v) \in \Sigma_E$ and states σ and σ' if there exist $(id(p), a) \in \Sigma_D$ and a mapping $\pi : \mathcal{P}_D \to \mathcal{V}_E$ such that $\pi(p) = v$ and $\sigma' = \{v \mapsto [a(v)_\pi]_\sigma \mid v \in \mathcal{V}_D \cup \mathcal{V}_E\}$.*
2. *The* combination model *of D and E (denoted as $D{\cdot}E$) is an LTS $\langle Q_{D\cdot E}, \Sigma_{D\cdot E}, \delta_{D\cdot E}, I_{D\cdot E} \rangle$ where $Q_{D\cdot E}$ is the set of value assignments from $\mathcal{V}_D \cup \mathcal{V}_E$ to $\mathcal{D}$ (*states for the combination model*), $\Sigma_{D\cdot E} = \Sigma_E$, $\delta_{D\cdot E} = \{\sigma \xrightarrow{id(v)} \sigma' \mid \sigma, \sigma' \in Q_{D\cdot E}, id(v) \in \Sigma_E\}$, and $I_{D\cdot E} = I_D \cup I_E$ is the set of initial states of D and E.*

5 Verifying Simulation Relations by Model Checkers

In this section, we present an approach for checking designs against their formal specifications based on simulation relations. We first present an overview, then, we present formal definitions.

5.1 Overview

In our framework, the initial inputs are pairs of formal specifications and designs of reactive systems. Suppose that M1 and M2 be two LTSs. We define M2 simulates M1 based on semantics of LTSs by extending the given relation on the states. The states are value assignments which are mappings from the variables to the values. Therefore, the relation on states of M1 and those of M2 are established based on mappings R and C where R is the mapping from variables of M1 to those in M2, C is the mapping from values in M1 to those in M2. The left hand side of Figure 4 shows a relation between state p of M1 and state q of M2. p relates to q based on R and C because $u = sus$ in state p corresponds to $v = 1$ in state q with mappings $R(u) = v$ and $C(sus) = 1$. M2 simulates M1 if for each transition in M1 from state p to state p' and p relates to state q of M2, there exists state q' and a corresponding transition in M2 from q to q' such that p' relates to q'. In the right hand side of Figure 4, a line arrow connecting p to p' represents a one-step transition from p to p', and a dashed arrow connecting q to q' represents an n-step transition from q to q'. To check whether M2 simulate M1, we check if there exists a reachable state q' from q such that $v = 2$ corresponding to $u = rdy$ in p' with mappings $R(u) = v$ and $C(rdy) = 2$.

Figure 5 shows steps to verify the simulation between a specification and a design using model checkers Spin: (step 1) Giving bounds for the verification

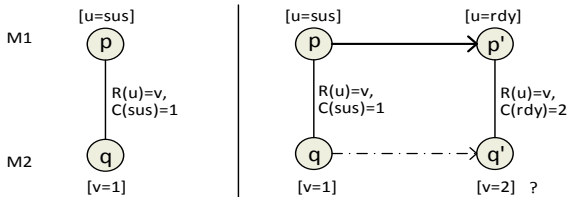

Fig. 4: Simulation Relation

and generate execution sequences from Event-B specification within the bounds, (step 2) Generating environments of the target system and assertions from the execution sequences and the given mappings between elements in the design and those in the specification, and (step 3) Model checking simulation relation in Spin.

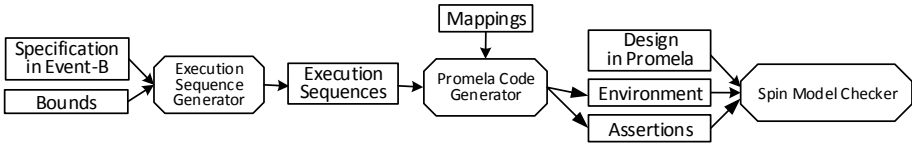

Fig. 5: Checking simulation relation of the design and its formal specification (steps)

Giving Bounds In a specification, there may be infinitely many states and transitions of target system because variables in Event-B obtain values in unbounded domains. Model checking does an exhaustive check of the system. It needs a representation of the system as a finite set of all possible states. So, abstract types in Event-B must be replaced by concrete types. Also, types have infinite ranges of values like Int and Nat must be restricted as finite ranges. By such restriction, the state space and the set of transitions explored from Event-B specification become finite sets. This makes the LTS explored from the specification finite. We define such restrictions as bounds of the verification.

Generating execution sequences from specification In order to generate execution sequences, or LTS, from the specification and bounds, the generator computes all possible transitions and reachable states. Every computed state must satisfy the invariant. Started at the initialization, the generator enumerates all possible values for the constants and variables of the specification that satisfy the initialization and the invariant to compute the set of initial states. To compute all possible transitions from a state, the generator finds all possible values for event parameters of an individual event to evaluate the guard of that event. If the guard holds in the given state, the generator computes the effect of the event based on substitution of that event. When new states are generated, we repeat this process to these states until no new states are generated.

Generating Evironment In order to verify that designs simulate their formal specifications, environments of the target systems are constructed and combined

with designs. As the result of such combination, we obtain combination models which become the inputs of model checkers. Since environments trigger specific behavior of designs by calling functions to the design, we construct environments by representing possible behavior described in their specifications. For this, we generate execution sequences as an LTS from the specification and generate the environment by translating the LTS into Promela. The LTS represents possible execution sequences of the specification within the input bounds as illustrated in Figure 6(a)).

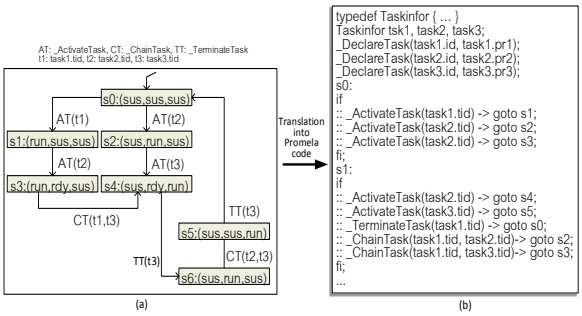

Fig. 6: Generation of environment from LTS

In Figure 6(a), the rectangles represent the states and the labeled arrows represent the events that are enabled in each states. For example, three events `AT(t1)`, `AT(t2)`, and `AT(t3)` are enabled in state $s0$, and five events `TT(t1)`, `AT(t2)`, `AT(t3)`, `CT(t1,t2)`, and `CT(t1,t3)` are enabled in state $s1$. In our framework, the states are defined as the value assignments; however, we show them here as values, e.g. `(sus, sus, sus)`, for readability.

The LTS is translated into the environment (e.g. from (a) to (b) of Figure 6). For this translation, we define a mapping from the events in the LTS to the function calls in the environment. In general, it is one-to-many mapping; however, in the case of reactive systems like OSEK OS, it is one-to-one mapping. For example, in the sample case, we map the event `ActivateTask(t1)` in the LTS to a function call `_ActivateTask(task1.tid)` in the environment. The states and transitions in the LTS are represented by labels and if-statements in the environment. The combination between the design and the environment becomes a direct input to the model checker in step 3 of the framework.

Generating Assertions The desirable behaviors of the target system obtained from the Event-B specification are encoded in imperative assertion statements which will be added into appropriate locations in the environment and justified by Spin model checker. Assertions are generated using the mapping R from the variables in the specification to those in the design, and the mapping C from the values in the specification to those in the design.

Assertions represent the constraints on the simulation relation between the specification and the design. In Figure 7, the above illustrates an execution sequence explored from specification and the below illustrates that of the design. Here, t, u are variables, and sus, rdy, run, $wait$ are values, which are defined in the specification. a, b are variables, and 1, 2, 3, 4 are values, which are defined in the design.

Value of variables in states p_0, p_1, p_2, $\cdots$ of the specification are evident from the LTS. Value of variables in state q_0 of the design is assigned evidently. It is obvious that q_0 relates to p_0 because values of variables in these two states preserve given R and C. However, the other states q_1, q_2, $\cdots$ reached from q_0 are not evident. We use a model checker to explore these states of the design and check whether q_1, q_2, $\cdots$ relate to p_1, p_2, $\cdots$, respectively. In order to ensure the simulation relation, for example for state q_1, we expect that $a = 2$ and $b = 1$ to comply with $R(t) = a$, $R(u) = b$, and $C(sus) = 1$, $C(rdy) = 2$. Hence, in order to check if q_1 relates to p_1, an assertion for state q_1 is defined by $a = 2$ and $b = 1$. Such assertions are automatically generated from the interpretations of states explored from the specification (step 2 of the framework), and they are embedded in the corresponding states of the environment as an assert statement of Promela.

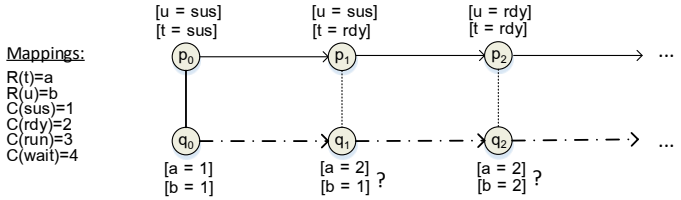

Fig. 7: Execution sequences of specification and design

Checking of simulation relation In the last step, we input the combination model and the assertions to Spin to check the simulation relation of the specification and the design. If no counter example is found, then we say the design conforms to the formal specification within the input bounds.

5.2 Formalization

We now give formal definitions of the relation between states, the bound, simulation relation of two LTSs within the given bound, and steps in the framework.

Definition 8. *(Relation between states). Let* $S = \langle \mathcal{V}_S, \mathcal{D}_S, \Sigma_S, \mathrm{Init}_S, Inv \rangle$ *be a specification model,* $M_S = \langle Q_S, \Sigma_S, \delta_S, I_S \rangle$ *the LTS derived from by* S*,* $D = \langle \mathcal{V}_D, \mathcal{D}_D, \mathcal{P}_D, ID(\mathcal{P}_D), \Sigma_D(\mathcal{P}_D), I_D \rangle$ *a design model,* $E = \langle \mathcal{V}_E, \mathcal{D}_E, \Sigma_E, I_E \rangle$ *an environment model for* D*, and* $D{\cdot}E = \langle Q_{D\cdot E}, \Sigma_{D\cdot E}, \delta_{D\cdot E}, I_{D\cdot E} \rangle$ *the combination model of* D *and* E*. We say a state* $\sigma_{D\cdot E} \in Q_{D\cdot E}$ *relates to a state* $\sigma_S \in Q_S$ *based on mappings* $R : \mathcal{V}_S \to \mathcal{V}_D$ *and* $C : \mathcal{D}_S \to \mathcal{D}_D$ *(denoted* $\sigma_S \preceq_{R,C} \sigma_{D\cdot E}$*), if for any* $x \in \mathcal{V}_S$ *and* $y \in \mathcal{V}_D$*,* $R(x) = y$ *implies* $C(\sigma_S(x)) = \sigma_{D\cdot E}(y)$*.*

It is obvious that $\preceq_{R,C}$ is a relation between the states. We omit R and C from $\preceq_{R,C}$ if they are clear from the context.

Definition 9. *(Bounds).* Bounds for LTS $\langle Q, \Sigma, \delta, I\rangle$ *are defined as a pair* $B = \langle G, H\rangle$ *of mappings* G *and* H *where* $G : 2^Q \to 2^Q$, $G(Q) \subseteq Q$, *and* $Q' \subseteq Q''$ *implies* $G(Q') \subseteq G(Q'')$ *and* $H : Q \times \Sigma \to \{tt, ff\}$ *and for any state* $p \in Q$, *there exist finitely many actions* $a \in \Sigma$ *such that* $H(p, a) = tt$.

For specifications in Event-B, we set bounds for range of values that variables can obtain to finite sets. By this restriction, the state space and the set of events applicable in each state become finite. We give mapping X for implementing such bounds to generate LTSs. X is mapping from variables to finite sets of values. We use $\mathrm{ES}_X(\sigma)$ to denote the set of all events which are applicable to σ and satisfy restrictions defined by X.

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

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

Definition 11. *(Simulation relation of two LTSs within bounds). Let* M_1 *and* M_2 *be two LTSs, and* B *be bounds. The simulation relation of* M_1 *and* M_2 *within bounds* B *is defined as* $M_1 \preceq_B M_2$ *if* $M_1{\downarrow_B} \preceq M_2$. *If* $M_1 \preceq_B M_2$ *holds, we say* M_2 *satisfies* M_1 *within* B.

Generating execution sequences from specification The algorithm to compute execution sequences from a specification model is presented in (Algorithm 1). Inputs of the algorithm are a specification model S, and bounds $B = \langle G, H\rangle$ which is implemented by X. Output is a bounded LTS. The algorithm uses two data structures: $QUEUE$ storing reachable states, and $VISITED$ storing visited states. It uses two routines to access $QUEUE$: Push($QUEUE$,$\langle\sigma\rangle$) adds state σ as an element into $QUEUE$, Pop($QUEUE$) returns the head of $QUEUE$. In each step of **while** loop, one state is removed from $QUEUE$, and reachable states from the state are computed. The algorithm terminates when $QUEUE$ becomes an *empty* set.

Generating Environments An environment is generated from the LTS of a specification model. First, we define correspondence of states between the specification and the environment. Let $S = \langle \mathcal{V}_S, \mathcal{D}_S, \Sigma_S, \mathrm{Init}_S, Inv\rangle$ be a specification model, $M_S = \langle Q_S, \Sigma_S, \delta_S, I_S\rangle$ the LTS derived from S, $E = \langle \mathcal{V}_E, \mathcal{D}_E, \Sigma_E, I_E\rangle$ an environment model, and $M_E = \langle Q_E, \Sigma_E, \delta_E, I_E\rangle$ the LTS derived from E. We say a state $\sigma_E \in Q_E$ corresponds to a state $\sigma_S \in Q_S$ based on mappings

Algorithm 1 Generating $S{\downarrow_B} = \langle \widehat{S}, \widehat{\Sigma}, \widehat{\delta}, \widehat{I} \rangle$ from S and X where $B = \langle G, H \rangle$, $G(Q_S) = \{\sigma \in Q_S \mid \forall v \in \mathcal{V}_S.\sigma(v) \in X(v))\}$, $G(I_S) \subset G(Q_S)$, and $H(\sigma, e) = tt$ iff $e \in \mathrm{ES}_X(\sigma))$

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

$R' : \mathcal{V}_S \to \mathcal{V}_E$ and $C : \mathcal{D}_S \to \mathcal{D}_D$ (denoted $\sigma_S \preceq_{R',C} \sigma_E$), if for any $x \in \mathcal{V}_S$ and $y \in \mathcal{V}_E$, $R'(x) = y$ implies $C(\sigma_S(x)) = \sigma_E(y)$.

Then, we define correspondence between events in the specification and functions in the design. For the target is a reactive system, this is defined by a mapping $f : \Sigma_S \to \Sigma_{D \cdot E}$. This means that an event in the specification corresponds to a function call in environment.

Finally, we define an LTS for the environment model E based on the specification S as $M_E = \langle Q_E, \Sigma_E, \delta_E, I_E \rangle$ where $Q_E = \{\sigma_E \mid \exists \sigma_S \in Q_S, \sigma_S \preceq_{R',C} \sigma_E\}$, $\Sigma_E = \{a \mid a \in F, \exists e \in \Sigma_S, f(e) = a\}$, $\delta_E = \{\sigma_E \xrightarrow{a} \sigma'_E \mid \sigma_E, \sigma'_E \in Q_E,\ a \in \Sigma_E, \exists \sigma_S, \sigma'_S \in Q_S, \sigma_S \preceq_{R',C} \sigma_E, \sigma'_S \preceq_{R',C} \sigma'_E\}$, and $I_E = \{\sigma_E \mid \exists \sigma_S \in I_S, \sigma_S \preceq_{R',C} \sigma_E\}$.

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

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

After this step, these conditional expressions are defined as inputs of model checkers. During the execution of the combination, the model checker will verify the reachable states of the combination against these conditions. At the end of the verification, one therefore can conclude that for any reachable state σ_S of the specification within the given bounds, there exists a reachable state $\sigma_{D \cdot E}$ of the combination such that $\sigma_S \preceq_{R,C} \sigma_{DE}$. As a result, the verification of simulation between the design and the specification has completed within the bounds.

6 Case study

We implemented our framework as a generator that produces (i) the LTS of the bounded specification, (ii) the environment in Promela, and (iii) the assertions. As an application of our framework to a practical system, we conducted an experiment to verify that a design of OSEK OS in Promela conforms to its formal specification in Event-B. We have illustrated these two models partially in Figure 3 and Figure 2.

Firstly, bounds are set for the verification by restricting range of values for every variables in Event-B specification. As shown in Figure 2, variables *tasks*, *res*, *evt*, and *inr* define entities managed by OSEK OS such as tasks, resources, events, and interrupt routines; variable *pri* defines the priority assigned to tasks, resources, and interrupt routines; and variable *t_state* defines the task state. Table 1 illustrates examples of the bounds in which each row shows values of the bounds that can be used in separate experiment. Here, because of the space limitation, we show restricted ranges of values for *tasks*, *pri*, *res*, *evt*, and *inr*, respectively.

According to our framework, we can set ranges so that they contain different entities such as tasks, resources, events, and interrupt routines as example No.3 of Table 1. In our experiments, we perform separate experiments in which some cases deal with distinct groups of system services of the target system; the other cases check the relation between different groups. This helps us to avoid the state explosion but preserve important behavior of the target system.

All experiments are conducted on an Intel(R) Core(TM) i7 Processor at 2.67GHz running Linux. Verification results outputted by Spin are shown in

Table 1: Examples of the bounds

No.	Restricted Ranges for $tasks$, $pri.$, $res.$, $evt.$, $inr.$
1	$\{t1, t2\}$, {1, 2}, {}, {}, {}
2	$\{t1, t2\}$, {1}, {}, $\{evt1\}$, {}, {}
3	$\{t1, t2, t3\}$, $\{1, 2, 3, 6, 4, 7\}$, $\{res1\}$, {evt1}, $\{inr1, inr2\}$
4	$\{t1, t2\}$, {1}, $\{res1\}$, {}, {}
5	$\{t1, t2\}$, {1}, $\{evt1\}$, {}, {}
6	$\{t1, \cdots, t9\}$, {1}, {}, {}, {}
7	$\{t1, \cdots, t10\}$, {1}, {}, {}, {}
8	$\{t1\}$, {1}, {}, {}, {}

Table 2. Here, the first column ("No.") represents experiment number. The next column presents size of ranges for variables $task$, pri, res, evt, and inr. Values in this column express bounds of the verification. Column "LTS Generation" shows statistic of the execution sequence generator. Here, columns "#State", and "#Trans" present the number of distinct states and that of transitions appearing in the execution sequences; column "Time" present the time taken (s) for the generation. Column "Model Checking" presents statistic of the model checker including total actual memory usage for distinct verification, the time taken (s), and the verification result in which $\surd$ indicates the successful result - neither error returned. As mentioned in Section 3, groups of system services of OSEK OS are task management, resource management, event mechanism, and interruption management. In the table, experiments No.1-No.15 are performed to check the task management of OSEK OS. In these cases, we set ranges for $tasks$ and pri. Experiments No.16-No.20 are performed to check relation between task management, resource management, event mechanism, and interruption management; therefore, we set ranges for $task$, pri, res, evt, and inr. Since OSEK OS schedules the tasks based on task priorities, it behaves differently with different patterns of assignment of task priorities. The best scenario to ensure that every possible behavior is checked is to use all patterns to assign the task priorities in such a way that each pattern is used in a distinct verification. For each value for pri, except for the case where pri includes only one element (e.g $pri = \{1\}$), there are several patterns to assign the priority for the tasks. For example, if there are two tasks with identifiers a and b; and the value domain for the task priorities is defined as [1..2], it is obvious that there are 4 patterns to assign the priority for the tasks. They are (1,1), (1,2), (2,1), and (2,2). Generally, let n be size of the value domain for task priorities, there are n^m patterns to assign priorities for m tasks. When the value bounds increase, the number of patterns becomes significantly larger. For preliminary verification, we categorize the patterns into groups where OSEK OS with one pattern behaves almost the same as other patterns in the group. Then we select typical patterns in groups to use in our experiments. In sample case above, patterns (1,1) and (2,2) are categorized in the same group; and both patterns (2,1) and (1,2) belong to

another group. We choose patterns (1,1) and (1,2) as typical patterns to use in experiments. This is reflexed in cases No.1-No.15 of the table.

Table 2: Experiment Outputs

No.	Size of Ranges					LTS Generation			Model Checking		
	$tasks$	pri	res	evt	inr	#State	#Trans	Time(s)	Memory(Mb)	Time(s)	Result
1	1	1	0	0	0	2	2	1	128	3	√
2	2	1	0	0	0	4	10	1	128	3	√
3	2	2	0	0	0	4	10	1	128	3	√
4	3	1	0	0	0	8	36	1	128	3	√
5	3	3	0	0	0	8	36	1	128	3	√
6	4	1	0	0	0	16	112	1	129	4	√
7	4	3	0	0	0	16	112	1	129	4	√
8	5	1	0	0	0	32	320	1	133	5	√
9	5	3	0	0	0	32	320	1	130	4	√
10	6	1	0	0	0	64	864	1	164	10	√
11	6	3	0	0	0	64	864	1	132	10	√
12	7	1	0	0	0	128	2240	1	380	26	√
13	7	3	0	0	0	128	2240	1	324	26	√
14	8	3	0	0	0	256	5632	2	382	99	√
15	9	3	0	0	0	512	13824	3	430	362	√
16	5	7	0	0	2	128	1536	2	133	17	√
17	2	1	1	0	0	8	22	1	130	7	√
18	2	1	0	1	0	10	27	1	129	4	√
19	3	6	1	0	2	80	520	1	129	8	√
20	3	6	1	1	2	152	1036	2	132	14	√

From the experiment results, we can see that the time taken and the total actual memory usage for the generation of the execution sequences from Event-B specification and the verification of the simulation relation are reasonable. For the model checking result, no errors were returned in all cases of experiments. This is because the design of OSEK OS has already been reviewed carefully by a lot of people including researchers and engineers. Still, this result offers a confidence on the conformance of the OSEK OS design with respect to its specification within input bounds.

7 Discussion

OSEK OS is the operating system which is widely used in the automotive systems. Our approach is applied to verify a design of a practical system, that is, OSEK OS design model. The approach directly checks the design against the formal specification. Although we show the experiments, when our approach is

applied to the operating system, it is not limited to this application. In the approach, the simulation relation is defined based on the given relation between states of the formal specification and those of the combination (of the design and the environment). In models, the states are all interpreted as value assignments. The design is described as a collection of functions which update the value assignments. The environment is described as a collection of invocations. This style of the models will be adopted not only for the operating systems but also other reactive systems. In our case study, Promela is used as a specification language to describe the design and the environment; however, our framework can be applied for designs described in not only Promela but also other languages as long as they can deal with a collection of functions for the design and sequences of invocations for the environment.

We introduce a formalization of the bounds for verifying the simulation relation of the design and the formal specification with Event-B. The bounds are used to limit set of states and possible transitions of LTS associated to Event-B model. This bound can be applied generally to any design and its formal specification as long as the formal models of the inputs are defined as LTSs. In Section 5, we present the interpretation of the bound in a concrete model, that is, Event-B model. In the first step of interpreting the bounds in the specification, we restrict the range of values for every variable, constant, and parameter defined in the specification. Next, we regard the typical bugs that can be found in the verification with large value domain. For finding such bugs of the target system, in addition to restrict the range of values, one can restrict system services of target system. Intention of such additional restriction is to exclude transitions not relevant to the bugs but to reduce size of model for which model checking is feasible.

In order to apply our framework to practical systems, we need to define initial states of the system by assigning initial values for every variable. Even though ranges of values for every variable have been restricted to finite sets, there are a lot of variations for initial states. In our case study, we assign initial states for distinct cases by manually. It is necessary to find a mechanism to assign initial states systematically.

8 Related Works

Verification of systems using model checking There are a number of works on verification of systems using model checking. Commonly, in model checking, a system is modeled with a finite automaton and various desired properties with temporal logic formulae. In [7], the authors translate Trampoline OS into Promela and define the safety properties in temporal logic formulae. Spin is used to check the OS model against these formulae. In [20], the authors model OSEK/VDX applications and the kernel in timed automata. Time properties are focused and verified by model-checker UPPAAL in these works. Separately, our input models are an OS model and its formal specification which are described in different specification language. We ensure that the design conforms to its

specification by utilizing Spin model checker to check the simulation relation of the design and its formal specification. In context of bounded model checking which is applied in [16][8], bounds are defined as the depth of the execution paths of the target model. In our work, bounds are defined as restrictions of ranges of values in Event-B specification so that every type has infinite range of values must be restricted as small range.

Verification of systems based on simulation relation FDR [6] is a refinement checker for the process algebra CSP. Inputs of FDR are the specifications and the implementations written in the same language. Our framework accepts the inputs written in different languages.

Theorem proving has been applied as a technique to verify systems based on simulation relation. [18] verifies the simulation relation between run-time environments layers including application layer, operating system, and FlexRay. [10] proves the simulation relation between abstract specification, executable specification, and C program for the seL4 micro-kernel. In these two existing works, Isabelle/HOL is used as a theorem prover for verification. Theorem proving ensures the correctness of system in universal scope; however, it generally requires many interactive proofs with human-machine collaboration. Separately, we use model checking as a technique of our verification. Range for types in the system model is bounded due to limitation of model checking; however, we get completely automatic verification and confidence on verification results. Additionally, our approach utilize Spin model checker to accept input models described in various specification languages.

The simulation of two LTSs was introduced in [13] in which the key point is to find a relation between state spaces of two LTSs such that the simulation rule holds. In our approach, the relation between state spaces of two LTSs is given. Then, we need to check whether the simulation rule holds.

Generation of LTS from Event-B model [11] presents the ProB tool which support interactively animating B models. Using ProB, users can see the current state and set an upper limit on the number of ways that the same operation can be executed. In our works, we firstly set finite ranges for types in Event-B specification, then, explore all possible execution sequences within defined ranges. [4] and [5] define the semantic of Event-B model as labeled transition systems to reason about behavioral aspects of specifications in Event-B. For a further objective, that is to check if all behaviors which are possible in Event-B specification, are also possible in the design, we generate all possible behaviors from Event-B specification within defined ranges. We precisely define the restriction of ranges of values in Event-B specification as bounds of our verification; then generate the environment and the verification conditions based on the explored execution sequences of Event-B specification within the input bound.

Construction of the environment of the operating system In previous works, we verified the OSEK OS by constructing a general model of the envi-

ronment from scratch. The environment model is firstly described using UML [21] then translated into Promela. In the current work, we construct the environment based on the information obtained from the specification in Event-B and we propose a flexible framework to check the design with respect to its formal specification so that it can accept different specification languages of the input models.

Combination of Event-B model and model checking The currently available tools such as ProB and Eboc [12] aim at the integration of model-checking and Event-B to benefit from both approaches in the same development process. In this context, ProB and Eboc are both model checkers for Event-B. For combination of model-checking using the Spin model checker and Event-B, [14] translate Event-B model into Promela model. We have not directly translated Event-B code into Promela but generated the environment and assertions from the execution sequences of Event-B specification, then, use Spin to check the design in combination with the environment against assertions.

9 Conclusion

In this paper, we proposed an approach to verify the design against its formal specification. Our framework includes (1) formalization of the simulation relation between two models of the same system; (2) formalization of the bound for the verification of the simulation relation using model checking; (3) the formal models of the specification, the design model, the environment, and the combination; (4) generation of the LTS associated with the Event-B specification within the given bound; and (5) verification of the designs against their formal specifications within input bounds using model checking. Based on our experiments, we discussed advantages and the applicability of the proposed approach in the engineering. The coverage of this verification is evaluated as how much of the specification satisfied by the design. In the future work, we aim at enlarging the verification coverage to make our framework effective for the practical verifications.

References

1. Abrial, J.R.: Modeling in Event-B: system and software engineering. Cambridge University Press (2010)
2. Aoki, T.: Model checking multi-task software on real-time operating systems. In: Proceedings of the 11th IEEE International Symposium on Object Oriented Real-Time Distributed Computing (ISORC '08). pp. 551–555 (2008)
3. Baier, C., Katoen, J.P.: Principles of Model Checking (Representation and Mind Series). The MIT Press (2008)
4. Bert, D., Cave, F.: Construction of finite labelled transition systems from B abstract systems. In: Proceedings of the 2nd International Conference on Integrated Formal Methods (IFM '00). Lecture Notes in Computer Science, vol. 1945, pp. 235–254 (2000)

5. Bert, D., Potet, M.L., Stouls, N.: Genesyst: a tool to reason about behavioral aspects of B event specifications. application to security properties. CoRR abs/1004.1472 (2010)
6. Broadfoot, P.J., Roscoe, A.W.: Tutorial on FDR and its applications. In: Proceedings of the 7th International SPIN Workshop. Lecture Notes in Computer Science, vol. 1885, pp. 322–322. Springer (2000)
7. Choi, Y.: Model checking trampoline os: a case study on safety analysis for automotive software. Softw. Test., Verif. Reliab. 24(1), 38–60 (2014)
8. Cordeiro, L., Fischer, B., Marques-Silva, J.: Smt-based bounded model checking for embedded ansi-c software. IEEE Trans. Software Eng. 38(4), 957–974 (2012)
9. Holzmann, G.J.: The SPIN Model Checker - primer and reference manual. Addison-Wesley (2004)
10. Klein, G., Andronick, J., Elphinstone, K., Heiser, G., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: Formal verification of an operating-system kernel. Communications of the ACM 53(6), 107–115 (2010)
11. Leuschel, M., Butler, M.: ProB: An automated analysis toolset for the B method. International Journal on Software Tools for Technology Transfer 10(2), 185–203 (2008)
12. Matos, P., Fischer, B., Marques-Silva, J.: A lazy unbounded model checker for event-b. In: Formal Methods and Software Engineering, Lecture Notes in Computer Science, vol. 5885, pp. 485–503. Springer Berlin Heidelberg (2009)
13. Milner, R.: Communication and concurrency. PHI Series in computer science, Prentice Hall (1989)
14. MULLER, T.: Formal methods, model-cheking and rodin plugin development to link event-b and spin (2009)
15. OSEK/VDX Group: OSEK/VDX operating system specification 2.2.3, http://portal.osek-vdx.org/
16. Pradella, M., Morzenti, A., San Pietro, P.: Refining real-time system specifications through bounded model- and satisfiability-checking. In: Automated Software Engineering, 2008. ASE 2008. 23rd IEEE/ACM International Conference on. pp. 119–127 (Sept 2008)
17. Reeves, S., Streader, D.: Guarded operations, refinement and simulation. Electron. Notes Theor. Comput. Sci. 259, 177–191 (2009)
18. In der Rieden, T., Knapp, S.: An approach to the pervasive formal specification and verification of an automotive system: Status report. In: Proceedings of the 10th International Workshop on Formal Methods for Industrial Critical Systems. pp. 115–124. FMICS '05, ACM, New York, NY, USA (2005), http://doi.acm.org/10.1145/1081180.1081195
19. Vu, D.H., Aoki, T.: Faithfully formalizing OSEK/VDX operating system specification. In: Proceedings of the 3rd Symposium on Information and Communication Technology (SoICT '12). pp. 13–20 (2012)
20. Waszniowski, L., Hanzálek, Z.: Formal verification of multitasking applications based on timed automata model. Real-Time Syst. 38(1), 39–65 (2008)
21. Yatake, K., Aoki, T.: Model checking of OSEK/VDX os design model based on environment modeling. In: Proceedings of the 9th International Colloquium on Theoretical Aspects of Computing (ICTAC '12). pp. 183–197 (2012)