PAPER *Special Section on Formal Approach*

A Framework for Verifying the Conformance of Design to Its Formal Specifications*

Dieu-Huong VU†a), **Yuki CHIBA**†b), **Kenro YATAKE**†c), ***Nonmembers*****,** ***and*** **Toshiaki AOKI**†d)**,** ***Member***

SUMMARY Verification of a design with respect to its requirement specification is important to prevent errors before constructing an actual implementation. The existing works focus on verifications where the specifications are described using temporal logics or using the same languages as that used to describe the designs. Our work considers cases where the specifications and the designs are described using different languages. To verify 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 the specifications and the designs commonly as labelled transition systems (LTSs). We appreciate LTSs since they could interpret information about the system and actions that the system may perform as well as the effect of these actions. Then, we check whether a design conforms to its specification based on the simulation relation of their LTS. In this paper, we present our framework for the verification of reactive systems, and we present the case where the specifications and the designs are described in Event-B and Promela/Spin, respectively. We also present two case studies with the results of several experiments to illustrate the applicability of our framework on practical systems.

key words: *formal specification, design model, formal verification, model checking, simulation relation*

1. Introduction

A software development process begins with informal requirements which the target software is expected to meet. The informal requirements are described either in natural languages or UML [21]; and, they are translated into formal specifications to ensure their consistency. Then, system designs are developed as models for implementation. Finally, the implementation is done according to the designs using programming languages. In this development process, we should verify the fact that the designs satisfy the requirements described by formal specifications since incorrect designs likely lead to significant costs caused by back track of the developments.

We focus on the development of reactive systems. The reactive systems are systems which must continually respond to the stimuli from their environment. Environments are the external systems which invoke the services of the target systems, e.g. software applications running on the operating systems. The specification of such a system represents its externally visible behaviors. That is, the specification represents what the system does in response to the invocations of its environments. Formal specification languages such as VDM [16], Z [18] and Event-B [1] allow us to formally describe the specification. On the other hand, the design represents the collaboration of internal components to realize observable behaviors described in the specification. It usually contains complex data structures such as record types, flags, and hash tables. We consider that imperative specification languages like Promela/Spin [9] are appropriate to describe the design since the data structures and behavior based on them can be straightforwardly described. The problem is how to verify designs with respect to their specifications when they are described by different specification languages.

Existing works focus on cases (i) where the user requirements are translated into temporal logic formulas [22] and the design is described in imperative specification languages like Promela [7] and (ii) where the specification and the design are described in the same specification language [1], [6]. We can see drawbacks when straightforwardly applying the existing approaches to verify the reactive systems. It is well-known that correctly describing properties in temporal logic is difficult [8] and the consistency of properties is not guaranteed. Whereas the formal specification languages with rich notions (e.g. sets and relations) facilitate describing the properties to be checked against the design. In addition, the tool of such formal specification language provides a function to verify the consistency and correctness of the properties. By following the second approach, we can describe the specification in an appropriate specification language; then, we derive the behaviors of the design from the higher-level specification by applying refinement functionality in Event-B [1]. We tried applying this approach to verify operating systems (OS). The behaviors appearing in the OS design are described based on complex data structures, e.g., a record type for `TASK` including elements such as `priority`, `state`, `type`, etc., and an array for `queue`. In Event-B, each element of the record type must be defined as a relation, e.g. $priority \in TASK \rightarrow N$, and $state \in TASK \rightarrow STATE$. Also, `queue` is defined as a set, i.e. $queue \subseteq TASK$ and the order of items in `queue` must be defined using relations, e.g., $queueItem \in 1..queuesize \rightarrow queue$. For each relation in Event-B, a lot of proof obligations are generated. Moreover, the OS design

---

Manuscript received August 19, 2014.
Manuscript revised December 19, 2014.
Manuscript publicized February 13, 2015.

†The authors are with the School of Information Science, Japan Advanced Institute of Science and Technology (JAIST), Nomi-shi, 923–1292 Japan.

*This paper is an extension of [24]

a) E-mail: huongvd@jaist.ac.jp
b) E-mail: chiba@jaist.ac.jp
c) E-mail: k-yatake@jaist.ac.jp
d) E-mail: toshiaki@jaist.ac.jp

DOI: 10.1587/transinf.2014FOP0004

contains not only `TASK` and `queue` but also many other complex data structures. In Promela, such data structures could be easily described. In addition, the OS design may contain sequential actions, which are straightforwardly described in Promela but not in Event-B because actions in Event-B are performed in parallel. In this case, one usually has to introduce more elaborate control structures to derive efficient sequential behaviors from event-based specifications in Event-B [1], [5]. Consequently, deriving the highly optimized behaviors of the design from the highly abstracted specification in Event-B results a lot of proof obligations. This requires much interactive proof to show the consistency between high-level and low-level descriptions. Therefore, this approach is not appropriate to verify the systems with complex data structures and highly optimized behaviors like the operating systems. Our idea is to use appropriate specification languages to describe the specification and the design, e.g. Event-B for the specification and Promela for the design. We propose a method to verify designs against their formal specifications where the specifications and the designs are described in different specification languages. We adopt Event-B for the specification and Promela/Spin for the design and commonly use LTS to interpret them. Our approach to check the design against the specification is based on simulation relation [13], [15] between their LTSs. Firstly, we formally describe the specification in Event-B to remove ambiguity and inconsistency in the specification [23]. Then, we generate execution sequences from this formal specification. 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. In this way, 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. There is a possibility that this approach is applicable not only for Event-B and Promela but also the other specification languages as long as we could interpret them as LTSs.

This paper presents our framework and its applications to verify practical systems. This is an extension of our work originally reported in [24]. In particular, we add more details to present the specification versus the design in the early parts of Sect. 2. The algorithm to generate the LTS from the specification is added in Sect. 3. A description of our own generator is also added in the last part of Sect. 3. In addition, a case study on vending machines is used in this paper as a simple example of reactive systems for readability.

The paper is organized as follows: In Sect. 2, we present definitions of specifications and designs of reactive systems. In Sect. 3, we present the definition of our verification framework. In Sects. 4 and 5, we present two case studies with the results of several experiments and discuss the practicality of our framework. In the last sections, we present related works and conclusions.

2. Specification and Design of Reactive System

Specifications generally describe the desirable properties and the external behaviors of the systems based on the mathematical data structures using notions of set, relation, and function. Designs must be close to the implementation. The mathematical data structures must be replaced by the data structures implementable on a computer and underspecified design decisions must be introduced. Generally, designs of reactive systems describe implementation of functions which realize the observable behaviors appearing in the specifications. We can see that there exists a gap between the specification and the design: the specification defines results of functions based on abstract data structures; however, the design defines details of how to make the results based on implementable data structures. For such a gap, we intendedly use different languages to facilitate describing the specification and the design. In this framework, we adopt Event-B for the specification and Promela for the design. This section presents the specifications and the designs of reactive systems described in Event-B and Promela, respectively, as well as their formal semantics. To verify the design of reactive systems, environment models are important; they describe possible entities and behaviors in communication with the target system. In this section, we also present the environment of reactive systems. We use a simple example, a vending machine, to demonstrate the specifications, the designs, and the environments of reactive systems.

2.1 Specifications in Event-B

A vending machine is a machine which dispenses items such as snacks, beverages, cigarettes, lottery tickets, etc. to customers automatically, after the customer inserts currency or credit into the machine. The specification of vending machines describes their external behaviors including (SF1) switching the machine on, (SF2) switching the machine off, (SF3) inserting credit into the machine, (SF4) returning credit, (SF5) restocking an item, and (SF6) dispensing an item. Each of them is a so-called service function. The essential properties of the vending machine refer to preconditions and post-conditions of the service functions. We demonstrate some of them as follows:

- Pushing a button shall vend a soda of the type corresponding to that button
- The machine shall retain exactly item cost for each item vended
- The machine shall return all deposited money in excess of item cost
- The machine shall flash the light for a selected item while vending is in progress to indicate acceptance of a selection to the buyer

The properties above could be defined in the LTL formulas and checked by Promela/Spin; however, the LTL

```
VARIABLES         INVARIANTS             INITIALISATION
  avail             avail⊂ PRODUCT         avail:= {}
  cred              cred∈N                 cred:=0
  state             state∈{0,1}            state:=0
  card              card∈N                 card:=0
```

```
switchon=
  when state=0
  then state:=1
```

```
switchoff=
  when state=1
  then state:=0
```

```
insert= any cr
  where state=1
  then cred:=cred+cr
```

```
restock= any item
  when item∈PRODUCT
       state=0
       card<MAX
  then avail:=avail∪{item}
       card:=card+1
```

```
dispense= any item
  when item∈avail
       state=1
  then avail:=avail\{item}
       cred:=cred-PRICE
       card:=card-1
```

Fig. 1 Specification.

formulas have a tendency to be complicated. For example, applying the patterns of [8] to define the last property, the LTL formula may be defined in the following form: <> *dispense* → (!*dispense U* (*insert* && !*dispense* && (!*dispense U select*)). This form of the LTL formula is complicated and prone to mistakes. Our idea is to describe the specification of the vending machine in Event-B and generate verification conditions from the Event-B specification, which represents desirable behaviors at a highly abstracted level.

Formal specification described in Event-B is regarded as a highly abstracted level description of the systems. This description mainly consists of state variables, operations (events) on the variables, and state invariants. The variables are typed using set theoretic constructs such as sets, relations, and functions. The events are defined with their guard conditions and substitutions (so-called before and after predicates), which allow both deterministic and non-deterministic state transitions.

Event-B is appropriate to describe the specification of the vending machine. Service functions are specified in terms of events with high-level operational definition of state changes by guarded substitutions. An event is made of two elements: (1) a guard that states the necessary conditions for the event to occur, and (2) a substitution that defines the state transition associated with the event. The semantics of the events define the overall results of the executions; therefore, represent pre-conditions and post-conditions of the service functions. Figure 1 demonstrates a specification of the vending machine in Event-B. Variable `avail` defines a set of items that are currently available to be dispensed. It has an abstract data type namely `PRODUCT`. Variable `cred` defines the total of money deposited so far and available to make a purchase. Variable `state` defines the state of the vending machine. Variable `card` defines the size of `avail`. External behaviors are specified in terms of events in Event-B namely `switchon`, `switchoff`, `insert`, `restock`, and `dispense`. Set operations (e.g. union, set minus) are used to describe what the system behaves when an item is restocked or dispensed. A mechanism to add an item into set `avail` and remove the corresponding item from the set has not been described. Also, the specification describes what happens when the customers insert cash or credit; however, how to recognize them and compute the total deposited money is postponed to describing the design.

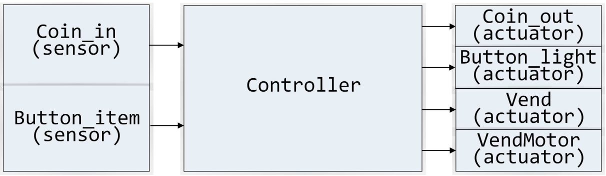


Fig. 2 Architecture design.

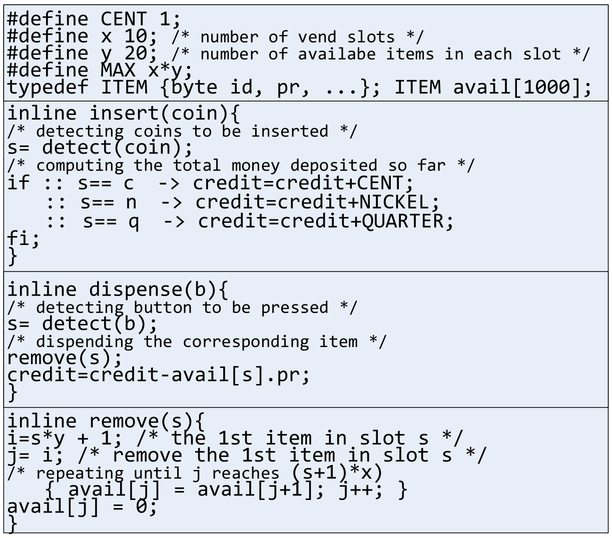
The specification could be also described in VDM or Z because they also provide rich notions like set, relation, and function. In this paper, we adopt Event-B because Event-B models are event-driven models, which are close to reactive systems.

2.2 Designs in Promela

Figure 2 shows an architecture design of the vending machine. The system consists of two sensors Coin_in sensor and Button_item sensor, a controller, and four actuators including Coin_out, Button_Light, Vend and VendMotor. The internal behaviors of the vending machine are as follows. When a coin is inserted, the Coin_in sensor detects the coin to be inserted and then sends an appropriate electrical signal to the Controller. The Controller computes the total of deposited money based on the inserted coin evaluation. When an item is selected, the Button_item sensor detects the item to be selected and then sends a corresponding signal to the Controller. The Controller commands the Button_Light to flash. The Controller compares the item cost with the total of the deposited money. If the item cost is less than the total, the Controller commands the VendMotor and Vend to remove the corresponding item from the set of available items and dispense it. The Controller commands the Button_Light to stop flashing and commands the Coin_out to return the correct change.

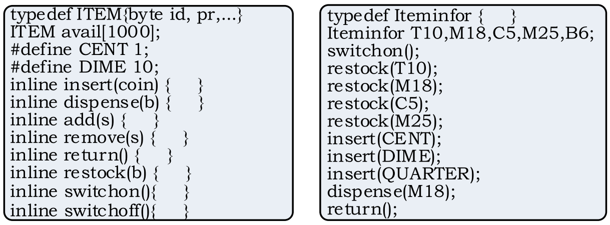
Designs of the vending machine can be straightforwardly described in Promela. The abstract data structures are replaced by the implementable data structures, e.g. array, record type. The behaviors are described using statements of Promela, e.g. expressions, assignment statements. The execution of the statement may change the value of variables. Additional variables and constants may be introduced to explicitly describe statements that must be performed to detect cash and credit for computing the total deposited money. Figure 3 demonstrates a detailed design of the vending machine. In the example, variables having abstract types, e.g. `avail ⊆ PRODUCT`, are replaced by variables having concrete types, e.g. `ITEM avail[1000]`. New constants are introduced, e.g. `CENT` is used in the case that a one cent coin is inserted. Design decisions for how to add a new item into the order set and to remove one from the corresponding position are explicitly described based on the implementable data structures and the control structures, e.g. loop and selection structures.

We can see a gap between the specifications and the designs. The observable behaviors appearing in the specifications are



```
#define CENT 1;
#define x 10; /* number of vend slots */
#define y 20; /* number of availabe items in each slot */
#define MAX x*y;
typedef ITEM {byte id, pr, ...}; ITEM avail[1000];
inline insert(coin){
/* detecting coins to be inserted */
s= detect(coin);
/* computing the total money deposited so far */
if :: s== c  -> credit=credit+CENT;
   :: s== n  -> credit=credit+NICKEL;
   :: s== q  -> credit=credit+QUARTER;
fi;
}
inline dispense(b){
/* detecting button to be pressed */
s= detect(b);
/* dispending the corresponding item */
remove(s);
credit=credit-avail[s].pr;
}
inline remove(s){
i=s*y + 1; /* the 1st item in slot s */
j= i; /* remove the 1st item in slot s */
/* repeating until j reaches (s+1)*x)
   { avail[j] = avail[j+1]; j++; }
avail[j] = 0;
}
```

Fig. 3 Design in promela.



```
typedef ITEM{byte id, pr,...}
ITEM avail[1000];
#define CENT 1;
#define DIME 10;
inline insert(coin) {    }
inline dispense(b) {    }
inline add(s) {    }
inline remove(s) {    }
inline return() {    }
inline restock(b) {    }
inline switchon(){    }
inline switchoff(){    }
```

```
typedef Iteminfor {    }
Iteminfor T10,M18,C5,M25,B6;
switchon();
restock(T10);
restock(M18);
restock(C5);
restock(M25);
insert(CENT);
insert(DIME);
insert(QUARTER);
dispense(M18);
return();
```

Fig. 4 Design and environment in promela.

realized by the optimized behaviors appearing in the designs. The specifications can be described in a declarative manner whereas the design can be described in an imperative manner. Our objective is to verify the conformance between such specifications and designs by using a simulation relation between them.

2.3 Communication of System and Environment

Figure 4 illustrates the overall structure of the design (left) and the environment (right) of the reactive systems. The design defines data structures and a collection of inline functions; it cannot operate by itself. To operate it, we need an environment which calls the functions of the target system. Essentially, the reactive systems need to be verified in the combination with their environments. The environment defines entities such as items, coins and a sequence of function calls to the target system. By combining the design and the environment, we can make a closed system which can operate by itself. We call this 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. As explained later, the environment is constructed from the specification, and input to Spin to check the simulation relation.

2.4 Formal Semantics

We first present a model of specifications based on Event-B. $\mathcal{V}$ is the set of *variables*. $\mathcal{D}$ is the *domain*, which is the set of values. Exp is the set of expressions in the specifications. An *expression* may contain variables in $\mathcal{V}$, values in $\mathcal{D}$, arithmetic operators, logical operators, and set operators. BExp is the set of boolean expressions (BExp $\subset$ Exp). A *substitution* $a : \mathcal{V} \to \mathrm{Exp}$ is a mapping from $\mathcal{V}$ to Exp. We note that value assignments are also substitutions because $\mathcal{D} \subseteq \mathrm{Exp}$. 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 set of special initialization events that have no guard. We denote $\sigma \xrightarrow{e} \sigma'$ for an event $e = \langle g, a\rangle$ and states σ and σ' if $\sigma(g)$ holds and $\sigma' = \{v \mapsto [a(v)]_\sigma \mid v \in V\}$.

Definition 1: (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, $\Sigma_S \subseteq \mathcal{E}$ is the set of events, $\mathrm{Init}_S \in \mathrm{Init}$ 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 $Q_S = \{\sigma \mid \sigma : \mathcal{V}_S \to \mathcal{D}_S\}$ is a non-empty set of states, $\delta_S = \{\sigma \xrightarrow{e} \sigma' \mid \sigma, \sigma' \in Q_S,\ e \in \Sigma_S\}$ is a transition relation, and $I_S = \{act(e) \mid e \in \mathrm{Init}_S\}$ is a set of initial states.

In Event-B, a substitution can be deterministic or non-deterministic. We regard a non-deterministic substitution as multiple deterministic substitutions. Therefore, we assume that the LTS is deterministic.

We now define model of designs in Promela. $\mathcal{P}$ is the set of *parameters* (function arguments). In the design, an expression may contain constants, variables, parameters and arithmetic operators, therefore, a so-called *parameterized expression*. The set of parameterized expressions is denoted as PExp. A function body is defined as a substitution. The substitution may contain the parameterized expressions. We use *p-substitution* to denote the substitution in the design. *p-substitution* is a mapping from $\mathcal{V}$ to PExp. The set of p-substitutions is denoted as PSubst. Id is the set of *identifiers* (used as function names). For the simplicity, we assume that functions have only one parameter. The design also includes an initialization function which assigns the initial values for the variables. Design models are defined as follows.

Definition 2: (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 signatures 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 is a set of value assignments of the initialization function such that $I_D \subseteq \{\sigma \mid \sigma : \mathcal{V}_D \to \mathcal{D}_D\}$.

We assume that the functions in the design are deterministic to have a unique successor state for each current state and each called function. This assumption is realistic for the implementation of the reactive systems like the automotive operating systems. On the other hand, it is generally non-deterministic to select a function applicable in each state. This is described in environment models. Environment models are defined as follows.

Definition 3: (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, 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 $\mathcal{D}_D$.

A combination of a design and an environment describes the execution of the design according to the environment. An expression in the 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 $pexp \in \mathrm{PExp}$, $pexp_\pi$ is the result of replacing each parameter p appearing in $pexp$ by $\pi(p)$. In other words, if $a(v)$ is an expression in D then $a(v)_\pi$ is an expression in the combination obtained by replacing each parameter p appearing in $a(v)$ by $\pi(p)$. Combination models are defined as LTSs as follows.

Definition 4: (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 $Q_{D\cdot E} = \{\sigma \mid \sigma : \mathcal{V}_D \cup \mathcal{V}_E \to \mathcal{D}_D\}$ is a set of states, $\Sigma_{D\cdot E} = \Sigma_E$, $\delta_{D\cdot E} = \{\sigma \xrightarrow{id(v)} \sigma' \mid \sigma, \sigma' \in Q_{D\cdot E}, id(v) \in \Sigma_E\}$ is a transition relation, and $I_{D\cdot E} = I_D \cup I_E$ is a set of initial states of D and E.

3. Verification Framework

This section presents a framework to verify designs in Promela against specifications in Event-B. Our approach is based on a simulation relation between the specification and the design. As demonstrated in Fig. 1, the specification defines state variables, invariants and events which trigger state transitions. Formally, the execution of the specification is represented as an LTS. Also, Fig. 3 describes variables and functions appearing in the design in Promela. The variables represent information about the system (states) at certain moments. The execution of statements changes the values of variables. Therefore, the design can be interpreted as an LTS if we consider that the variables are states and each function call is a label to make transitions on the states.

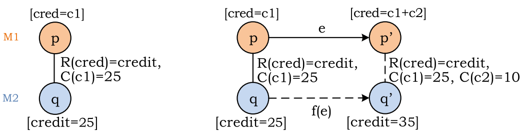
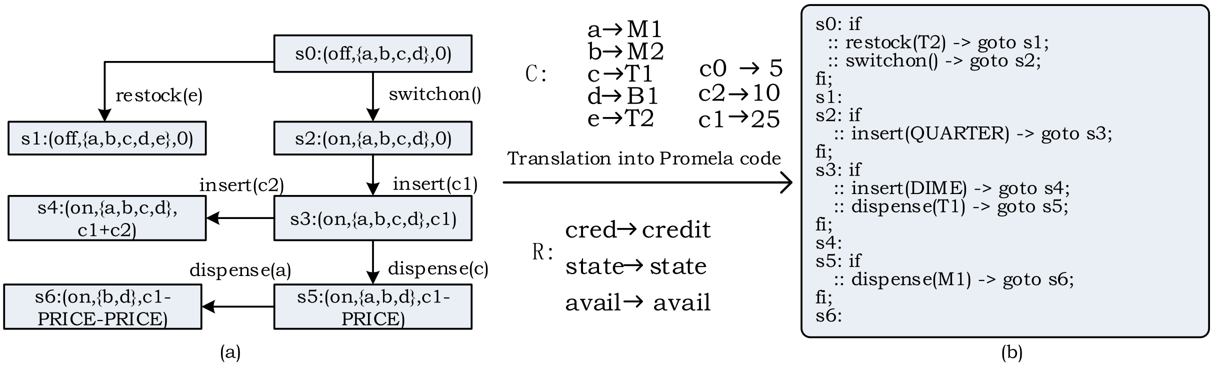


Fig. 5 Simulation relation.

3.1 Preliminary

We now present the simulation relation between two LTSs. Supposing that M1 and M2 be two LTSs. We define M2 simulating 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. Figure 5 (left) shows a relation between state p of M1 and state q of M2. p relates to q based on R and C because $cred = c1$ in state p corresponds to $credit = 25$ in state q with mappings $R(cred) = credit$ and $C(c1) = 25$. 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 Fig. 5 (right), 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 simulates M1, we check if there exists a reachable state q' from q such that $credit = 35$ corresponding to $cred = c1 + c2$ in p' with mappings $R(cred) = credit$, $C(c1) = 25$, and $C(c2) = 10$.

This definition of the simulation relation is similar to the refinement of [13] and the simulation of [15], [20]; however, in our definition, a one-step transition in M_1 may correspond to an n-step transition in M_2. This is appropriate for a simulation from a specification to its design because the behaviors in the design are usually more concrete than those in the specification. Considering bi-simulation in [15], a bi-simulation $\mathcal{S}$ between M_1 and M_2 requires that M_1 simulates M_2 by $\mathcal{S}$ and M_2 simulates M_1 by the reverse of $\mathcal{S}$. In this framework, we check one direction of simulation, that is checking whether the design simulates the specification. We considered that this is sufficient to detect bugs in the design when we apply model checking. This is an important objective of our framework. Considering refinement in Event-B, the refinement in Event-B focuses on the reverse direction, that is verifying whether the specification simulates the design. Verifying whether the specification simulates the design is sufficient to show there is no extra behavior added into the design with respect to the specification; on the other hand, its reverse is sufficient to show every behavior in the



```
s0: if
  :: restock(T2) -> goto s1;
  :: switchon() -> goto s2;
fi;
s1:
s2: if
   :: insert(QUARTER) -> goto s3;
fi;
s3: if
   :: insert(DIME) -> goto s4;
   :: dispense(T1) -> goto s5;
fi;
s4:
s5: if
   :: dispense(M1) -> goto s6;
fi;
s6:
```

Fig. 7 Generation of environment from LTS.

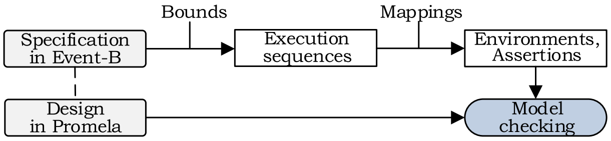


Fig. 6 Checking simulation of design and its specification.

specification are actually realized in the design.

3.2 Overview

Figure 6 shows the steps to verify the simulation relation between a specification and a design using the Spin model checker. Firstly, we give bounds for the verification and explore execution sequences from the Event-B specification within the bounds. From these execution sequences, we generate environments of the target system and assertions based on the given mappings between elements of the specification and those of the design. Finally, we check the simulation relation in Spin.

Bounds. Model checking does an exhaustive check of the system. It needs a representation of the system as a finite set of all possible states. Firstly, abstract types in Event-B must be replaced by concrete types. Also, types having infinite ranges of values like Int and Nat must be restricted as small ranges. Then, by studying the properties of interest, we can restrict the behaviors will be checked. Such restrictions are to reduce the size of execution sequences explored from the Event-B specification and produce a finite LTS associated to the restricted specification. We define such restrictions as bounds of the verification.

Exploring Execution Sequences. In order to explore the execution sequences, or LTS, from the specification and bounds, the execution sequence explorer computes all possible transitions and reachable states. Every value used in the computation must be within the bounds. Starting at the initialization, the explorer enumerates all possible values for the constants and variables of the specification that satisfy the initialization to compute the set of initial states. To compute all possible transitions from a state, the explorer 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 explorer 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 Environments and Assertions. The environments trigger specific behaviors of the target system; therefore, it is essential to construct such comprehensive environments that representing all possible behaviors in the specification. In the previous step, we explored the execution sequences as an LTS of the specification. In this step, we generate the environment by translating the LTS into Promela such that the enabled events in LTS are translated to the corresponding function calls in Promela.

Figure 7 (a) illustrates an LTS associated to the specification. Here, the rectangles represent the states and the labeled arrows represent the events that are enabled in each states. For example, two events `restock(e)`, `switchon()` are enabled in state $s0$. In our framework, the states are defined as the value assignments; however, we show them here as values for readability. For example, `s2:(on, {a,b,c,d}, 0)` describes that the machine is on; there are 4 items `a`, `b`, `c`, and `d` available for buying; and the currently deposited money is 0, in state labeled `s2`. The LTS is translated into Promela to generate the environment (Fig. 7 (b)). For this generation, we give a mapping from the enabled events in the LTS to the function calls in the environment. In general, it is a one-to-many mapping. In the sample case of the figure, it is a one-to-one mapping. For example, event `restock(e)` in the LTS is mapped to function call `restock(T2)` in the environment. The states and transitions in the LTS are represented by labels and if-statements in the environment. There may be more than one function call applicable in each state. For example, `insert(DIME)` and `dispense(T1)` are applicable in state $s3$; which function call actually applied is nondeterministic. The environment is combined with the design to make a combination model.

Verification conditions represent constraints on the simulation relation between the specification and the design. They are encoded as assertions in Promela/Spin. For generation of assertions, we define mappings R and C from the variables, the values in the specification to those in the design. Figure 7 demonstrates some mappings used in verification of vending machines: $R(cred) = credit$, $C(a) = M1$, $C(c) = T1$, $C(c1) = 25$, $C(c2) = 10$, etc. From each state of the LTS, an assertion, which must be met by the corresponding states of the designs, is generated. In Fig. 5, for example, from state p' where $cred = c1 + c2$ of the top with mappings $R(cred) = credit$, $C(c1) = 25$ and $C(c2) = 10$, the generator outputs an assertion $credit = 35$ to check state q' of the bottom.

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. Even though there exists a gap between the specification and the design, our framework can verify the correspondence between state transitions, or simulation relation, of the specification and the design. Specifically, each state transition in the specification leads to a function call, which in turn triggers multiple state transitions in the design; after these state transitions, the design reaches a state where the verification conditions are asserted. If no counter-example is found, we say the design conforms to the formal specification within the input bounds.

3.3 Formalization

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

Definition 5: (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 S, $D = \langle \mathcal{V}_D, \mathcal{D}_D, \mathcal{P}_D, F, \Sigma_D, I_D\rangle$ a design model, $E = \langle \mathcal{V}_E, \mathcal{D}_E, \Sigma_E, I_E\rangle$ an environment model, 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 $R : \mathcal{V}_S \rightarrow \mathcal{V}_D$ and $C : \mathcal{D}_S \rightarrow \mathcal{D}_D$ (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 $C(\sigma_S(x)) = \sigma_{D\cdot E}(y)$.

We omit R, C from $\preceq_{R,C}$ if they are clear from the context.

As we mentioned, the bounds are introduced to obtain a finite LTS from the Event-B specification. A finite LTS is obtained from an infinite LTS when we restrict the state space and the set of actions that trigger the state transitions. The bounds are defined as follows:

Definition 6: (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 $G : 2^Q \rightarrow 2^Q$, $G(Q) \subseteq Q$, and $Q' \subseteq Q''$ implies $G(Q') \subseteq G(Q'')$ and $H : Q \times \Sigma \rightarrow \{tt, ff\}$ and for any state $p \in Q$, there exist finitely many actions $a \in \Sigma$ such that $H(p, a) = tt$.

Definition 7: (Bounded LTS). An LTS obtained by restricting an LTS $M = \langle Q, \Sigma, \delta, I\rangle$ within 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 $\widehat{I} = G(I)$.

To implement the bounds for LTS associated to the Event-B specification, we restrict the range of the variable values. When every range of the variable values has been restricted, the state space and the set of actions of the LTS become finite. The restriction is represented by a mapping X from variables to finite sets of values, i.e. $X : V_S \rightarrow 2^{\widehat{D_S}}$, where $\widehat{D_S}$ is the restricted range for the variable values. For example, Figure 9 illustrates bounds to be used for verification of vending machines where values of `cred` is restricted to [1..100]. Formally, this restriction is defined as $X(cred) = [1..100]$. $\mathrm{ES}_X(\sigma)$ is used to denote the set of events which are applicable to state 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 $\langle Q_S, \Sigma_S, \delta_S, I_S\rangle$ an 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 $e \in \mathrm{ES}_X(\sigma)$.

We now define a simulation relation between two LTSs. In general, a transition step in the specification is followed by a n-step transition in the design. In the definition, Σ^+ denotes the set of non-empty strings of Σ, δ^+ denotes a n-step transition relation, and $p \xrightarrow{a_1 a_2 \ldots a_n} p' \in \delta^+$ denotes a n-step transition from state p to state p'.

Definition 8: (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 $f : \Sigma_1 \rightarrow \Sigma_2^+$ 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 $M1 \preceq M2$.

Definition 9: (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 simulates M_1 within B.

Exploring Execution Sequences. The algorithm to compute execution sequences from a specification model is presented in (Algorithm 1). Inputs of the algorithm are a specification model $S = \langle \mathcal{V}_S, \mathcal{D}_S, \Sigma_S, \mathrm{Init}_S, Inv\rangle$, and bound $B = \langle G, H\rangle$ which is implemented by X. Output is a finite 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. The environment is generated from the LTS of the specification model. Let $S = \langle \mathcal{V}_S, \mathcal{D}_S, \Sigma_S, \mathrm{Init}_S, Inv\rangle$ be a specification model and $M_S =$

Algorithm 1 Generating $S{\downarrow_B} = \langle \widehat{S}, \widehat{\Sigma}, \widehat{\delta}, \widehat{I} \rangle$ from $S = \langle \mathcal{V}_S, \mathcal{D}_S, \Sigma_S, \mathrm{Init}_S, Inv \rangle$ and X

```
$QUEUE = empty$
$VISITED = empty$
$\widehat{S} = \widehat{\Sigma} = \widehat{\delta} = \widehat{I} = empty$
for each $\sigma_0 \in \{act(e) \mid e \in \mathrm{Init}_S\}$ do
    if $\forall v \in V_S, \sigma_0(v) \in X(v)$ then
        Push($QUEUE, \langle\sigma_0\rangle$)
        $\widehat{S} = \widehat{S} \cup \{\sigma_0\}$
        $\widehat{I} = \widehat{I} \cup \{\sigma_0\}$
    end if
end for
while $QUEUE \neq empty$ do
    $\langle\sigma\rangle$ = Pop($QUEUE$)
    $VISITED = VISITED \cup \{\sigma\}$
    $\widehat{E} = \{e \mid e \in \mathrm{ES}_X(\sigma)\}$
    if $\widehat{E} \neq empty$ then
        for each event $e = (g, a) \in \widehat{E}$ do
            $\sigma' = \{v \mapsto [(act(e))(v)]_\sigma | v \in \mathcal{V}_S\}$
            if $\sigma' \notin VISITED$ then
                Push($QUEUE, \langle\sigma'\rangle$)
                $\widehat{S} = \widehat{S} \cup \{\sigma'\}$
            end if
            $\widehat{\Sigma} = \widehat{\Sigma} \cup \{e\}$
            $\widehat{\delta} = \widehat{\delta} \cup \{\sigma \xrightarrow{e} \sigma'\}$
        end for
    end if
end while
return $S{\downarrow_B}$
```

$\langle Q_S, \Sigma_S, \delta_S, I_S \rangle$ be the LTS derived from S. Based on the given mapping $f : \Sigma_S \to \Sigma^+_{D\cdot E}$ from the events in the LTS to the function calls in the environment, mapping $R' : \mathcal{V}_S \to \mathcal{V}_E$ and mapping $C : \mathcal{D}_S \to \mathcal{D}_D$, the environment model $E = \langle \mathcal{V}_E, \mathcal{D}_E, \Sigma_E, I_E \rangle$ with $\mathcal{D}_E = \mathcal{D}_D$ is generated such that $\Sigma_E = \{f(e) \mid e \in \Sigma_S\}$ and $I_E = \{f(e) \mid e \in I_S\}$.

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

- For initialization of the combination, the assertion is: $\bigwedge_{x \in \mathcal{V}_S, y \in \mathcal{V}_D, y = R(x)} (\sigma^0_{D\cdot E}(y) = C(\sigma^0_S(x)))$,
- 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 $\sigma_S \xrightarrow{e} \sigma'_S \in \delta_{S\downarrow_B}$, $\sigma_{D\cdot E} \xrightarrow{f(e)} \sigma'_{D\cdot E} \in \delta^+_{D\cdot E}$, and $\sigma_S \preceq_{R,C} \sigma_{D\cdot E}$, the assertion is $\bigwedge_{x \in \mathcal{V}_S, y \in \mathcal{V}_D, y = R(x)} (\sigma'_{D\cdot E}(y) = C(\sigma'_S(x)))$.

After this step, the assertions are input to the model checker. During the execution of the combination, the model checker will verify the reachable states of the combination against conditions in the assertions. At the end of the verification, one can conclude that, within the bounds, for each reachable state σ_S of the specification, there exists a reachable state $\sigma_{D\cdot E}$ of the combination such that $\sigma_S \preceq_{R,C} \sigma_{D\cdot E}$. As a result, the verification of simulation between the design and the specification has completed within the bounds.

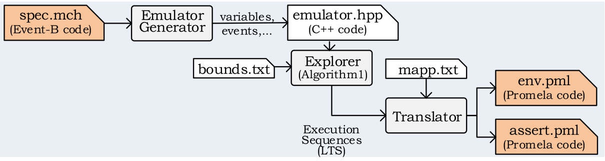


Fig. 8 Architecture of generator.

```
PRODUCT {a1,b2,c1,d3,e1,k4,d2,x5,u7,e3,p9,s1}
nitem   200
cred    [1..100]
```

Fig. 9 Bounds used in verification of vending machines (“bounds.txt”).

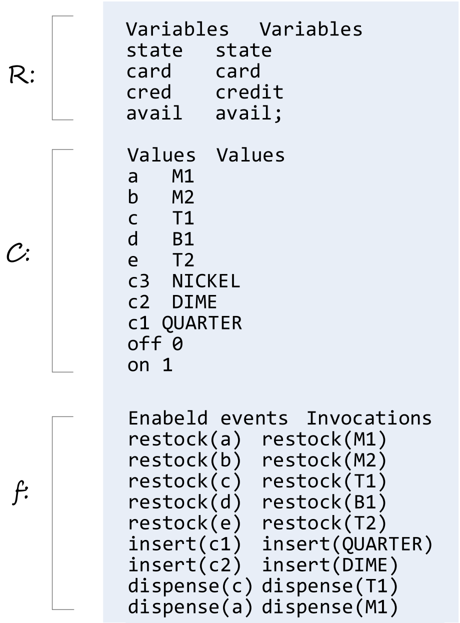
3.4 Generator

We implemented a generator that produces the environments and the assertions from the specification. The architecture of our tool is shown in Fig. 8. The core of our tool consists of three modules: Emulator Generator, Explorer and Translator. They are all implemented in the C++ programming language. The initial input is the specification in Event-B. The emulator generator performs the lexical and syntactic analysis to emulate the behaviors of the specification in C++. The explorer implements (Algorithm 1); it invokes functions which emulate events in Event-B and use the given bounds to outputs the execution sequences of the specification within the bounds. The execution sequences are represented as an LTS of the specification. The enabled events appearing in the LTS are sources to generate sequences of invocations in the environments. The states appearing in the LTS are used to generate the assertions. The translator uses mappings between elements of the Event-B specification and those of the Promela design to output the environments and the assertions in Promela code.

Inputs are provided by users. Inputs produced by the users include:

- The Event-B specification file with “.mch” extension, as shown in Fig. 1
- The bounds described in the file with “.txt”. The bounds of vending machines are demonstrated in Fig. 9.
- The mappings (from elements in the Event-B specification to those in the Promela design) described in the file with “.txt”. Each correspondence from the source to the target is presented in a distinct row; the source is separated from the target by a tab character, as shown in Fig. 10.

Generating Description of Specification in C++. The emulator generator analyzes syntactic structures of the Event-B specification. They are variables, types, events, guards, substitutions, expressions, set operators, arithmetic operator, etc. These structures are translated into C++ by following the correspondences presented in Table 1.

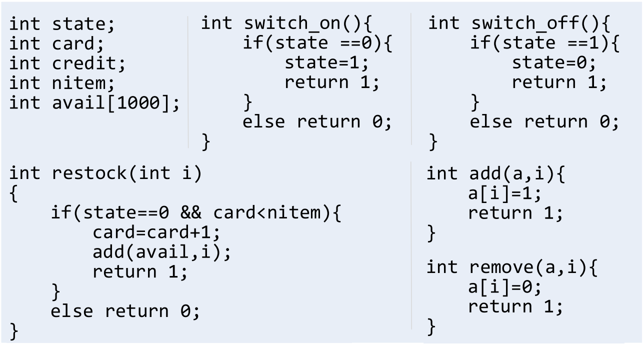


```
Variables  Variables
state   state
card    card
cred    credit
avail   avail;

Values  Values
a   M1
b   M2
c   T1
d   B1
e   T2
c3  NICKEL
c2  DIME
c1 QUARTER
off 0
on 1


Enabeld events  Invocations
restock(a)  restock(M1)
restock(b)  restock(M2)
restock(c)  restock(T1)
restock(d)  restock(B1)
restock(e)  restock(T2)
insert(c1)  insert(QUARTER)
insert(c2)  insert(DIME)
dispense(c) dispense(T1)
dispense(a) dispense(M1)
```

Fig. 10 Mappings used in verification of vending machines ("mapp.txt").

Table 1 From Event-B to C++.

Event-B	**C++**
Variable	Variable
Enumerated types	Enumerated types
Initialization	Function namely `init()`
Events	Functions
Event parameters	Arguments of function
Guards	Conditional structures
Substitutions	Assignment statements
Arithmetic operators	Arithmetic operators
Set operators (e.g., $\cup$, $\setminus$)	Library functions (e.g., `add`, `remove`)



```
int state;
int card;
int credit;
int nitem;
int avail[1000];
```

```
int switch_on(){
    if(state ==0){
        state=1;
        return 1;
    }
    else return 0;
}
```

```
int switch_off(){
    if(state ==1){
        state=0;
        return 1;
    }
    else return 0;
}
```

```
int restock(int i)
{
    if(state==0 && card<nitem){
        card=card+1;
        add(avail,i);
        return 1;
    }
    else return 0;
}
```

```
int add(a,i){
    a[i]=1;
    return 1;
}

int remove(a,i){
    a[i]=0;
    return 1;
}
```

Fig. 11 Behaviors of specification of vending machine are emulated in C++ ("emulator.hpp").

The emulator generator outputs C++ codes in the targeted file with ".hpp" extension, which simulates the behaviors of the specification in the form of functions (Fig. 11). It is in turn included in the source code of the Explorer. The Explorer invokes the functions of the specification to execute the specification and generate the LTS associated to the specification.

4. Case Studies

The purpose of the case studies is to evaluate the generality and the applicability of our framework to verify practical systems. The target systems used in case studies range from simple systems, e.g. vending machines, controlling cars on a bridge, to large or complex systems, e.g. operating systems used in automotive systems. We applied the framework to verify whether the designs of the target systems conform to their specifications. In this paper, we present the experiment results of two case studies.

Vending machine. We have illustrated the specification and the design of the vending machine partially in Figs. 1 and 3. In the framework, bounds are set for the verification by introducing finite ranges of variable values in the Event-B specification. In practical applications of the vending machines, the maximum number of available items is given for each machine. In the specification, variable `card` defines the number of available items; range of values for `card` must be restricted to a finite set such that the highest value of the range must be less than or equal the maximum number to be given. Bounds are introduced to define such restriction. The mappings between elements of the specification and the design are illustrated in Fig. 7. For example, `a` is mapped to `M1` and `cred` is mapped to `credit`.

Operating system. We applied the framework to verify the design of an operating system compliant with OSEK/VDX standard [19] (OSEK OS, for short). OSEK OS is the operating system which is widely used in the automotive systems. External behaviors of OSEK OS are classified into groups of service functions. They are task management, interrupt handling, resource management, and event control. Tasks are the basic building blocks of an application program running on the operating system. The specification of OSEK OS in Event-B [23] describes variables `tasks`, `res`, `evt`, and `inr` which 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. The specification also describes 12 service functions observable from the outside, e.g. `ActivateTask` activates a task. The whole specification is described in 400 code lines in Event-B. The design of OSEK OS is described in about 2800 lines of Promela code, according to the approach in [2]. It first defines data structures such as `task`, `res`, and `ready` which represent an array of tasks, an array of resources, and ready queues, respectively. Following these data structures, a set of functions is defined. For example, `_ActivateTask` and `_TerminateTask` are the functions to perform activation and termination of tasks, respectively. Bounds are set for the verification by introducing finite ranges of values for the variables appearing in the specification. In the experiments, by using various bounds, we can separate the cases that deal with distinct groups of service functions from which check

Table 2 Experiment outputs.

Target System	Bounds: size of ranges					LTS Generation			Model Checking		
Vending Machine	**card**					#State	#Trans	Time (s)	Memory (Mb)	Time (s)	Result
No.1	50					151	252	1.2	129.2	1.0	√
No.2	200					604	1004	50.2	130.4	5.0	√
Operating System	`tasks`	`pri`	`res`	`evt`	`inr`	#State	#Trans	Time (s)	Memory (Mb)	Time (s)	Result
No.3	5	3	0	0	0	32	320	1.2	130.6	4.9	√
No.4	8	3	0	0	0	256	5632	2.1	382.8	99.2	√
No.5	9	3	0	0	0	512	13824	3.0	430.8	362.1	√
No.6	10	3	0	0	0	1024	92160	22			*not completed*
No.7	2	1	1	0	0	8	22	1.1	130.1	7.6	√
No.8	2	1	0	1	0	10	27	1.1	129.2	4.7	√
No.9	3	6	1	0	2	80	520	1.2	129.2	8.3	√
No.10	3	6	1	1	2	152	1036	2.0	132.3	14.1	√
No.11	5	7	0	0	2	128	1536	2.0	133.1	17.5	√
No.12	8	6	1	1	2	1052	93600	26			*not completed*

the relation between different groups. This helps us to avoid the state explosion and keep important behaviors of the target system we want to verify in the cases.

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 2. Here, values in column "Size of Ranges" express bounds of the verification. Column "LTS Generation" shows statistics of the execution sequence generator. Columns "#State", and "#Trans" present the number of distinct states and that of transitions appearing in the execution sequences, each transition corresponds to a function call; column "Time" present the time taken (s) for the generation. Column "Model Checking" presents statistics of the model checker including total actual memory usage, the time taken (s), and the verification result in which √ indicates the verification has been completed.

In verification of the vending machine, case No.1 is conducted with number of available items ranging in [0..50]; this allows to restock 10 slots of products and 5 products in each slot. Case No.2 corresponds 20 slots and 10 products in each slot. These ranges are appropriate in practical applications of the vending machines. When we describe the specification of vending machines, we consider two viewpoints. Firstly, we focus on number of products available in vending machines to be bought; secondly, we focus on type of those products. Reachable states of the LTS are computed by assigning all possible values for variables in the Event-B specification within bounds. Following the first viewpoint, we specify the number of products but not type of products; therefore, "#State" depends on ranges of values for variables: `cred`, `state`, and `card` (number of products). Following the second viewpoint, we additionally specify the type of products, e.g., tea, coffee, and milk. Products of the same type are put in the same slot of vending machines. In this case, "#State" depends on not only ranges of values for variables `cred`, `state`, `card` but also #slot (number of slots). If value of `card` is m and #slot is n then "#State" is around $(m/n)^n$. Our main purpose when we show results of case studies is to demonstrate that our framework could be straightforwardly applied to verify reactive systems; we use the specification of vending machines according to the first viewpoint. In Table 2, we presented results of No.1 where we focus on only the number of products and we use restricted ranges: `cred` $\{0, 2\}$, `state` $\{on, off\}$, and `card` [0..50]. Therefore, "#State" in No.1 represents a combination of values in $\{0, 2\}$, $\{on, off\}$, and [0..50]. Similarly, "#State" in No.2 represents a combination of values in $\{0, 2\}$, $\{on, off\}$, and [0..200].

In verification of OSEK OS, experiments No.3-No.6 are performed to check the task management independently from the other groups of service functions. In these cases, we show ranges for `tasks` and `pri`. Experiments No.7-No.12 are performed to check combination between task management, resource management, event mechanism, and interruption management; therefore, we show ranges for `tasks`, `pri`, `res`, `evt`, and `inr`. In the table, two cases of our verification, No.6 and No.12, have not completed due to out of memory condition. When we extend the bounds, the size of LTS becomes larger. The total number of invocations increases according to the total number of enabled events appearing the LTS, which is indicated in column "#Trans". Using our machine (memory capacity: 8 gigabytes), Spin can use around 430 megabytes for total memory usage, in which around 230 megabytes is used to store states. This allows to store around 400,000 states; therefore, it allows the size of LTS to reach 25,000 transitions. From the statistics of LTS Generation, we can see that the total memory required to store states in cases No.6 and No.12 is over the total memory Spin can use. Therefore, the verification has not completed in these cases. In order to avoid out of memory condition, we could use reasonable range for the variables as shown in No.3-5. Also, we could check small groups of service functions; each of these groups represents an essential behavior of OSEK OS. For example, No.7 checks combination between task management and resource management; No.8 checks combination between task management and event mechanism.

From results of case studies, we found that the framework could be straightforwardly applied to verify various reactive systems where the designs described in Promela and their formal specifications described in Event-B. Even though this framework has a limitation of the model checking; we considered that essential behaviors of the OS could be still verified successfully when we use reasonable bounds. This shows applicability of our framework in verification of reactive systems.

5. Discussion

Generality of the Framework. Our framework was applied to verify the designs of practical systems. The framework directly checks the designs against their formal specifications. Although we show the experiments, when our framework is applied to the vending machine and the operating system, it is not limited to these applications. In the framework, the simulation relation is defined based on semantic of LTS. In models, the states are 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 models is adopted not only for operating systems but also other reactive systems.

In our case studies, Promela is used as a specification language to describe the design and the environment; however, our framework can be applied for the 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.

Notion of Bounds. We introduce a formal definition of the bounds for verifying the simulation relation of the design and its formal specification. The bounds are used to obtain a finite 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 Sect. 3, 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 introduce finite ranges of variable values in the specification. Next, we regard the typical bugs that can be found in the verification with a large value domain. The typical bugs are bugs that could be easily added into the design of practical systems. For finding such bugs of the target system, in addition to restrict the range of values, one can restrict service functions of the target system. The intention of such additional restriction is to exclude transitions not relevant to the bugs and to reduce size of model for which model checking is feasible. In this approach, we intend to lead the verification to focus on partial behaviors instead of all at one. We could distribute partial behaviors in variations of the environment. In our idea, partial behaviors are decided according to the properties and the bugs of the target systems to be checked. For example, one important property of the OS is that “`An extended task in the waiting state must be released to the ready state if at least one event for which the task is waiting has occurred`”. In order to check this property, we need use two tasks, one event, and three service function including `ActivateTask`, `WaitEvent`, and `SetEvent`. We found that one could avoid the state explosion if we use reasonable ranges for data elements and service functions.

Comprehensiveness of Environment. The behaviors of the target systems depend on patterns of function calls from their environments. For the comprehensive verification of reactive systems, we need to use the environments that cover all possible patterns of invocations. Accordingly, an advantage of our framework is that it is able to systematically generate all possible patterns of invocations from the execution sequences of the specification in Event-B. This is essential to generate the environments for the comprehensiveness of verification with respect to the specification.

6. Related Works

Verification of systems using model checking. [7] presents a case study on checking the operating systems compliant with OSEK/VDX. The authors describe the specification in temporal logic formulas. Separately, we describe the specification in Event-B. This improves the consistency of properties extracted from the specification and provides general environments for comprehensive verifications.

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. [10] and [11] present approaches to verify the OS kernels based on theorem proving. Theorem proving can be used to verify the infinite systems; however, it generally requires a lot of interactive proofs. In our framework, we use model checking combining with prover tools of Event-B. Although, ranges are bounded due to the limitation of model checking; however, we are able to improve quality of the properties checked and get completely automatic verification. Therefore, we have a high degree of confidence in the verification results.

Generation of LTS from Event-B model. [12] presents the ProB tool which supports interactively animating B models. Using ProB, users can visualize the current state and the enabled operations in each state. Users also can set an upper limit on the number of ways that the same operation can be executed. However, ProB requires some interactions with the users. In our work, we firstly set finite ranges for types; for complex systems like operating systems, we may restrict the functionalites. Then, we explore all possible sequences of state transitions within defined ranges. Our work does not support visualizing the LTS; however, the generation of the LTS is completely automatic. [4] defines the semantic of Event-B model as labeled transition systems to reason about behavioral aspects of specifications in Event-B. We formally define the framework from scratch. We precisely define finite ranges of variable values in Event-B specification as bounds of our verification; then, we generate all possible behaviors from Event-B specification within defined ranges.

Construction of the environment of the operating system. In previous works, we verified the OSEK OS by constructing a general model of the environment from scratch. The environment model is firstly described using UML [25] then translated into Promela. In the current work, we generate the environment from the specification in Event-B. Since the correctness of the specification is guaranteed by tools of Event-B, the quality of the environment is improved as well.

Combination of Event-B model and model checking. For combination of Event-B and model checking, tools such as ProB [12] and Eboc [14] work as model checkers for Event-B. In these approaches, the target models are described in Event-B. ProB and Eboc directly check the target models against the internal consistency. We use tools of Event-B to ensure the internal consistency. We use our own tool to generate the LTS of the Event-B specification. As another approach, [17] translates Event-B model into Promela model and use Spin to check the model. We have not directly translated Event-B code into Promela but translate LTS of the Event-B specification and assertions into Promela. Then, we use Spin to check the simulation between the design model and LTS of the specification in Promela.

7. Conclusion

We proposed an approach to verify designs against their formal specifications which are described in different specification languages respectively. Two main contributions included in this approach: a new combination between Event-B and Promela/Spin for verification of reactive systems; and filling the gap between the specification and the design so that we can check the conformance between them systematically. An advantage of the approach is to make it possible to describe the specification and the design in appropriate languages for a verification of the design. Formal specification languages are intended to facilitate describing the specifications. Promela is intended to analyze the designs. Our approach follows these intentions faithfully. In fact, as mentioned in Sect. 1, it is natural for reactive systems like operating systems to describe the designs in the imperative specification languages. On the other hand, describing their detailed properties in temporal logic is hard. It is easy to imagine that the temporal logic formulas representing the specification shown in the case studies become very complex and prone to mistakes. Instead of the temporal logic, we provide a way to represent the specification in a formal specification language Event-B and check the design against it with the Spin model checker. Event-B is appropriate to represent the specification because it has rich notions such as sets and relations. In addition, Event-B allows us to ensure the consistency and the correctness of the specification by its verification facilities such as discharging proof obligations and refinement. That is, we can check the design against such consistent and correct specification. This would drastically improve the reliability of model checking results because the specification is reliable. We plan to extend the verification framework to accept the additional choice of the specification languages. There is a possibility that our approach is applicable not only for Event-B and Promela but also the other specification languages. The framework could accept the other languages for the specification such as Z, VDM, as long as it is possible to generate LTS from the description of the specification in those languages. In addition, languages for the design must deal with collection of functions.

References

[1] J.R. Abrial, Modeling in Event-B: System and software engineering, Cambridge University Press, 2010.
[2] T. Aoki, “Model checking multi-task software on real-time operating systems,” 11th IEEE International Symposium on Object Oriented Real-Time Distributed Computing, pp.551–555, 2008.
[3] C. Baier and J.P. Katoen, Principles of Model Checking (Representation and Mind Series), The MIT Press, 2008.
[4] D. Bert, M.L. Potet, and N. Stouls, “Genesyst: A tool to reason about behavioral aspects of B Event specifications. Application to security properties,” CoRR, vol.abs/1004.1472, 2010.
[5] P. Bostrom, “Creating sequential programs from Event-B models,” Proc. 8th International Conference on Integrated Formal Methods, IFM ’10, Berlin, Heidelberg, pp.74–88, 2010.
[6] P.J. Broadfoot and A.W. Roscoe, “Tutorial on FDR and its applications,” Proc. 7th International SPIN Workshop, pp.322–322, 2000.
[7] Y. Choi, “Model checking trampoline OS: A case study on safety analysis for automotive software,” Softw. Test., Verif. Reliab., vol.24, no.1, pp.38–60, 2014.
[8] M.B. Dwyer, G.S. Avrunin, and J.C. Corbett, “Patterns in property specifications for finite-state verification,” Proc. 21st International Conference on Software Engineering, ICSE ’99, pp.411–420, 1999.
[9] G.J. Holzmann, The SPIN Model Checker - primer and reference manual, Addison-Wesley, 2004.
[10] T. In der Rieden and S. Knapp, “An approach to the pervasive formal specification and verification of an automotive system: Status report,” Proc. 10th International Workshop on Formal Methods for Industrial Critical Systems, FMICS ’05, pp.115–124, 2005.
[11] G. Klein, J. Andronick, K. Elphinstone, G. Heiser, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, “seL4: Formal verification of an operating-system kernel,” Commun. ACM, vol.53, no.6, pp.107–115, 2010.
[12] M. Leuschel and M. Butler, “ProB: An automated analysis toolset for the B method,” Int. J. Software Tools for Technology Transfer, vol.10, no.2, pp.185–203, 2008.
[13] N. Lynch and F. Vaandrager, “Forward and backward simulations: Untimed systems,” Inf. Comput., vol.121, no.2, pp.214–233, Sept. 1995.
[14] P. Matos, B. Fischer, and J. Marques-Silva, “A lazy unbounded model checker for Event-B,” Formal Methods and Software Engineering, Lect. Notes Comput. Sci., vol.5885, pp.485–503, 2009.
[15] R. Milner, Communication and concurrency, PHI Series in Computer Science, Prentice Hall, 1989.
[16] A. Muller, “VDM - the vienna development method,” Bachelor thesis in Formal Methods in Software Engineering, Johannes Kepler University Linz, 2009.
[17] T. Muller, “Formal methods, model-cheking and rodin plugin development to link Event-B and Spin,” IEICE Technical Report, SS., vol.109, no.170, pp.43–48, 2009.
[18] G. O’Regan, “Z formal specification language,” in Mathematics in Computing, pp.109–122, Springer London, 2013.
[19] OSEK/VDX Group, “OSEK/VDX operating system specification

2.2.3, http://portal.osek-vdx.org/."

[20] S. Reeves and D. Streader, "Guarded operations, refinement and simulation," Electron. Notes Theor. Comput. Sci., vol.259, pp.177–191, 2009.

[21] J. Rumbaugh, I. Jacobson, and G. Booch, Unified Modeling Language Reference Manual, 2nd ed., Pearson Higher Education, 2004.

[22] M.Y. Vardi and P. Wolper, "An automata-theoretic approach to automatic program verification," Proc. 1st Annual Symposium on Logic in Computer Science (LICS '86), pp.332–344, 1986.

[23] D.H. Vu and T. Aoki, "Faithfully formalizing OSEK/VDX operating system specification," Proc. 3rd Symposium on Information and Communication Technology, pp.13–20, 2012.

[24] D.H. Vu, Y. Chiba, K. Yatake, and T. Aoki, "Checking conformance of a Promela design to its formal specification in Event-B," Preliminary proceeding of the third International Workshop on Formal Techniques for Safety-Critical Systems (FTSCS), pp.188–203, 2014.

[25] K. Yatake and T. Aoki, "Model checking of OSEK/VDX OS design model based on environment modeling," Proc. 9th International Colloquium on Theoretical Aspects of Computing (ICTAC '12), pp.183–197, 2012.

Dieu-Huong Vu received her B.S., M.S. degrees from College of Technology, Vietnam National University, Hanoi (2001, 2004). She is currently a PhD candidate at JAIST (Japan Advanced Institute of Science and Technology). Her research interests include formal methods, formal verification, theorem proving, model checking, object-oriented design/analysis.

Yuki Chiba received his B.S., M.S., and Ph.D. degrees from Tohoku University (2003, 2005, 2008). He is currently an assistant professor at JAIST (Japan Advanced Institute of Science and Technology). His research interests include program transformation, term rewriting system, automated theorem proving, and model checking.

Kenro Yatake received his B.S. degree from Tokyo Institute of Technology (2000), M.S. and Ph.D. degrees from Japan Advanced Institute of Science and Technology (2002, 2006). He is currently an assistant professor at JAIST (Japan Advanced Institute of Science and Technology). His research interests include formal method, theorem proving, model checking.

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