TẠP CHÍ KHOA HỌC & CÔNG NGHỆ ĐẠI HỌC DUY TÂN
DTU Journal of Science and Technology 5(48) (2021) 19-30

TRƯỜNG ĐẠI HỌC DUY TÂN
DUYTAN UNIVERSITY

A verification framework for specification centered developments

Một khung làm việc kiểm chứng cho quy trình phát triển phần mềm lấy đặc tả làm trung tâm

Vũ Diệu Hương*
Vu Dieu Huong*

Information Technology Department, Economic Information System and E-Commerce Faculty, Thuong Mai University, 79 Ho Tung Mau, Hanoi, Vietnam
Văn phòng Công nghệ Thông tin, Khoa Thương mại Điền tử và hệ thống Thông tin Kinh tế, Trường Đại học Thương mại, 79 Hồ Tùng Mậu, Hà Nội, Việt Nam

(Ngày nhận bài: 18/6/2021, ngày phản biện xong: 19/8/2021, ngày chấp nhận đăng: 17/10/2021)

Abstract

Requirement specifications are initial inputs of every software development process. Referring to the specification is frequently needed through the software development phases. Because the specification has such a tremendously important role in the software process, its quality should be ensured before it is used to drive life – cycle activities. Once the quality of the specification is ensured, it is reliable input to verify the other artifacts outputted from various development activities like design and implementation. In this paper, we propose a verification framework for specification centered developments. In this framework, we firstly focus on improve the quality of specification then use such high-quality specification to drive the verification of the design and the implementation. This framework could be applied in domain of reactive systems with high automation, adaptation, and practicality.

Keywords: software verification; formal verification; specification centered development; reactive systems.

Tóm tắt

Đặc tả yêu cầu là đầu vào khởi đầu của mọi tiến trình phát triển phần mềm. Tham chiếu tới đặc tả là cần thiết ở mọi pha trong tiến trình phát triển phần mềm. Vì đặc tả có vai trò đặc biệt quan trọng như vậy, chất lượng của đặc tả nên được đảm bảo trước khi nó được sử dụng để dẫn xuất mọi hoạt động của vòng đời phát triển phần mềm. Một khi chất lượng của đặc tả đã được đảm bảo, nó sẽ là một đầu vào tin cậy để kiểm chứng các chế tác được cung cấp bởi các hoạt động khác nhau trong vòng đời phát triển phần mềm, ví dụ như mô hình thiết kế, chương trình. Trong bài báo này, chúng tôi đề xuất một khung làm việc kiểm chứng cho tiến trình phát triển phần mềm lấy đặc tả làm trung tâm. Trong khung làm việc này, đầu tiên chúng ta sẽ tập trung vào cải tiến chất lượng của đặc tả rồi sử dụng đặc tả có chất lượng được đảm bảo này để dẫn xuất cho hoạt động kiểm chứng thiết kế và chương trình. Khung làm việc này có thể được áp dụng trong miền các hệ thống phản ứng với tính tự động hóa cao, tính thích nghi tốt và tính thực hành cao.

Từ khóa: kiểm chứng phần mềm; kiểm chứng hình thức; phát triển phần mềm lấy đặc tả làm trung tâm; các hệ thống phản ứng.

*Corresponding Author: Vu Dieu Huong; Information Technology Department, Economic Information System and E-Commerce Faculty, Thuong Mai University, 79 Ho Tung Mau, Hanoi, Vietnam
Email: huongvd@tmu.edu.vn, vudhuong@gmail.com

1. Introduction

Requirement Specifications [15] (Specifications for short) describe expected (external) behaviors of the software. They also describe nonfunctional requirements, design constraints and other factors necessary to provide a complete and comprehensive description of the requirements for the software.

Specifications are initial inputs of every software development process. Moreover, they are involved in every phase of the development processes. Referring to the specification is frequently needed through the software development phases. In other words, various development activities are based on specifications: design, implementation, testing, maintaining. Specifications are used as the basis for providing and ensuring the quality of designs and implementations.

Because the specification has such a tremendously important role in the software process, its quality should be ensured before it is used to drive life – cycle activities. Once the quality of the specification is ensured, it is reliable input to verify the other artifacts outputted from various development activities like design and implementation.

Reactive system [16] (RS for short) is a system that continuously interact with its environment by responding to external stimuli. For example, vending machines, elevator, and operating systems are reactive systems. The environment of a vending machine includes customers who want to buy products restocked in that vending machine; and, the environment of an operating system includes software applications running on that operating system. Several reactive systems are considered as safety-critical (e.g., operating systems for mobile vehicles). Such kind of systems should be seriously verified because their errors cause highly destructive effects on the human life and the assets.

Quality improvement for RS in a unified verification process is still a great challenge. What we need for such verification process includes: (i) a high-quality specification which is consistent and validated against user requirements; (ii) this specification is used as a confident input to formally verify both of the design and the implementation. We know that there exists a gap between the specification and the design. The specification defines abstract data structures, whereas the design defines implementable data structure. Also, the specification defines results of operations, while the design defines details of how to make the results. There are two main approaches for verification of every software including reactive systems. In the first approach, we can describe the specification in an appropriate specification language like Z [7], VDM (Vienna Development Method) [6], Event-B [5]; then, we derive the behaviors of the design from the higher-level specification as adopted in [5]. The problem of this approach is that in order to represent highly optimized behaviors of reactive systems, we need to use various complex data structures and control structures. Therefore, directly deriving these behaviors from the abstract specification is generally very hard. In the second approach, we can use imperative specification languages like Promela [8] to describe the design. This facilitates describing the highly optimized behavior by using various data structures and control structures. In this approach, the specification or the properties we want to check can be described in temporal logic. However, we consider that temporal logic formulae, which allows us to describe properties about invariants on some variables and the relative order of event calls, but are not adequate for describing the important properties of reactive

system concern with the pre-condition and the post-condition of each event invoked from their environment. Moreover, we could not ensure the consistency of these properties in the temporal logic formulae. Also, such formulae are not appropriate inputs for verification of the implementation.

Our idea is using the rich notions (e.g. sets and relations) in the formal specification languages like Event-B, we could easily describe properties we want to verify. Therefore, we intendedly use Event-B to facilitate describing properties of the reactive system. In Event-B, one can describe the system as a set of events and the behavior of each event can be specified as pre-conditions and post-conditions using the rich notions. It also provides a facility to verify the consistency and the correctness of the properties. This results high-quality specifications. Furthermore, such specifications are appropriate inputs to automated verify the implementation.

For verification of the design, our idea is to describe the design in imperative specification languages like Promela, which is easy to represent the design. We think that dealing with the specification and the design based on the different specification languages is appropriate for systems in which there exists a big gap between the specification and the design like the reactive systems. Then, we verify the design against the specification. In this case, we must deal with the difference between the language used to describe the specification and that for the design.

In this paper, we propose a verification framework for Specification Centered Developments in which Specifications are verified first; Specifications are used as the inputs for ensuring the quality of designs and implementations. In our framework, we intend to use different specification languages for the specification and the design to facilitate the description of them. That is, Event-B is adopted for the specification and Promela for the design. We deal with the difference between specification languages for the specification and the design by common semantics, label transition systems (LTSs), and correspondences between state transitions given by mappings from syntactic elements in the former to those in the latter. This approach is also applied to deal with the gap between the specification and the implementation. Therefore, our framework accepts any programming language used in the implementation. In other words, the specification described in Event-B is also appropriate input to verify the implementation. This makes it possible to systematically ensure the quality of reactive systems based on the confident specification with high automation, adaptation, and practicality.

2. Related works

Using formal specifications as inputs for software verification is presented in [1] and [14]. [1] presents an approach to verify the OS kernel by applying theorem proving. Theorem proving can be used to verify the infinite systems; but, it generally requires a lot of interactive proofs. [14] applied the model checking technique to verify the correctness properties of safety and liveness of target systems. In our workflow, we use model checking combing 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. Animating Event-B models is implemented in ProB [2] and Eboc [3]. 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 Rodin prover of Event-B to ensure the internal consistency and use our animator to validate. We use our own tool to generate the LTS of the Event-B specification. Formal Specification Centered Development Process is presented in [4]. However, this work applies model checking technique to give test suite from the formal specification. Our framework uses the formal specification for not only verification of the design but also that of the implementation.

3. Verification framework

Our framework is illustrated in Fig.1. Initial input of the framework is a statement of user requirements. That is an informal description of user requirements. Originally, the user requirements are usually described in natural languages. Due to the ambiguity of natural languages, such informal descriptions could lack consistency. Moreover, they are not proper inputs for the formal verification. Therefore, we need formal specifications replacing for the informal ones to drive life - cycle activities including verification of the designs and the implementations. The quality of formal specifications should be ensured before we use them to drive life - cycle activities.

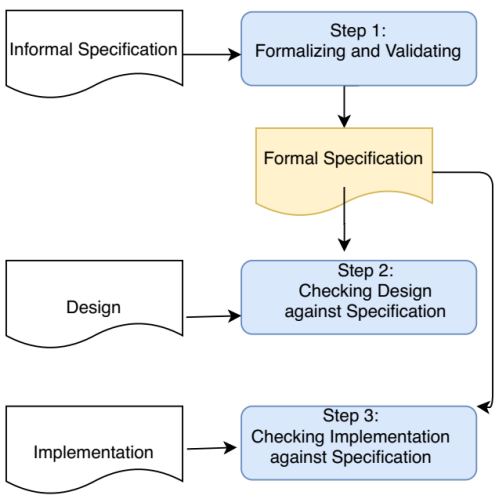

Fig.1: Verification Framework (steps)

Three verification activities are shown in the Fig.1. The first activity is *formalizing and validating the requirements.* This step provides the high-quality formal specification of requirements. The second activity is *checking the design against the formal specification*. This step uses the formal specification as a reliable input to verify the design by applying model checking technique [8]. The last activity is *checking the implementation against the formal specification*. Also, this step uses the formal specification as a reliable input to verify the implementation by applying testing technique [13]. As mentioned earlier, this framework adapts to overall software development processes such water fall, evolution model, agile methods. The verification is an ongoing process and the verification activities are integrated in the development activities. Once, both of the design and the implementation conform to the specification, we are confident that the implementation also conforms to the design. Actually, we could show the conformance between the implementation and the design by applying the same approach.

4. Formalizing and validating requirements in Event-B

4.1. *Requirements* of reactive systems

Features or aspects of reactive systems are commonly mentioned as (i) external behavior or functional requirements, (ii) properties or non-functional requirements, (iii) communication or links between controllers and physical entities like sensors and actuator [9], and (iv) communication between reactive systems and their environments [10].

Our framework focuses on verifying quality of Specification, Design and Implementation for reactive system. This section aims at how to provide a high-quality specification. In our point of view, a high-quality specification is

one that (1) it has sufficient requirements for user's needs and (2) it is well structured, unambiguous, consistent, readable, and easy to review and validate.

The requirements are generally divided into functional requirements and non-functional requirements (i.e. constraints). In the specification of reactive systems, we focus on the behavioral aspect of target systems in communication with their environments. Such aspect concerns with **system services** and **constraints on service operation in combination with the environment**. For example, services of Vending Machine (VM) include *switch on, switch off, restock* an item into VM, *collect coins*, and *dispense* an item to the customers. A few of constraints on these services are as below:

- *Pushing a button shall vend an item 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.*

Constraints are divided into two kinds of conditions. The first one is pre-condition, that is, the condition under which a service is activated. Another one is post-condition, that is action or result of service execution. For example, *when the customer inserts a coin, collected coins must activated and the credit is increased by the value of coins.*

4.2. Formalizing requirements using Event-B

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 requirements of reactive systems. Services 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 services.

The Rodin tool supports not only describing but also verifying Event-B models. In particular, it has a capable of checking the internal consistency. This capability is provided by Proof Obligation Animator and Prover [5], which are included in the Rodin tool. The animator generates verification conditions as proof obligations. By discharging such verification conditions, we could show the consistency of the specification.

Fig. 2 demonstrates a specification of the vending machine in Event-B. Variable itemList defines a set of items that are currently available to be dispensed. It has an abstract data type namely ITEMS. The variable numCup holds the number of cups in the itemList. Its type is an arbitrary set of non-negative integer. The variable amount defines the total of money deposited so far and available to make a purchase. Its type is an arbitrary set of non-negative float.

External behaviors are specified in terms of events in Event-B namely switch-on, switch-off, 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.

The specification could be also described in any model-based languages 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.

4.3. Validating formal requirements

We develop an animator for Event-B specification to give a visualization of the specified system's behavior. This assists the analyzers, users and stake-holders to validate formal specifications against informal requirements by demonstration of state transitions and stimulus – response behaviors of the target systems in the combination with their environments. While animating the specified system's behavior, the users give stimulus as they are the target system's environment.

```
CONSTANTS ITEMS, AMOUNTS, MAXCUP,
          PRICE, COIN

VARIABLES itemList, numCup, amount
INVARIANT
    itemList ⊆ ITEMS,
    numCup ∈ itemList -> N,
    amount ∈ N
INITIALIZATION
    amount := 0
    itemList := ITEMS
    numCup := 0
EVENT
  Restock = WHEN x ∈ itemList
    AND numCup(x) < MAXCUP
    THEN numCup(x) := numCup(x) + 1

  Dispense = WHEN x ∈ itemList
    AND amount ≥ PRICE AND
    numCup(x) ≥ 1
    THEN numCup(x) := numCup(x) - 1
    AND
    amount = amount - PRICE AND

  Insert = WHEN c ∈ COIN
    THEN amount := amount + c

  Return = WHEN c ∈ COIN
    THEN amount := amount - c
```

Fig.2: Specification of Vending Machine

Communication between the animator and the users is described in the following steps:

1. Starting at the initialization, the animator 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.
2. The animator prompts the users to enter a stimulus
3. The animator 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 animator computes the effect of the event based on substitution of that event. This computation produces a new state of the system.
4. The users check the actual results which is produced by the animator with respect to expected results.
5. Steps 2, 3 and 4 are repeated until the users want to stop the animation.

Fig.3 illustrates the output of animating the specification of VM in Fig.2. States are

presented by values of variable itemList, numCup and amount. For example, the initial state of the VM is ({a,b}, (0,0),0). This means the VM provides two kinds of products namely a and b. In the initial state, the number of product a is 0 and that of product b is also 0. Next, the users enter restock a as a stimulus sent from the environment. Then, the animator provides a new state ({a},(1,0),0). In this state, the number of product a is 1.

```
1  ({a,b},(0,0),0):
2  -->stimulus: restock a
3  ({a,b},(1,0),0):
4  -->stimulus: restock b
5  ({a,b},(1,1),0):
6  -->stimulus: restock b
7  ({a,b},(1,2),0):
8  -->stimulus: insert 1
9  ({a,b},(1,2),1D):
10 -->stimulus: dispense a
11 ({a,b},(0,2),0):
12 ...
```

Fig.3: Animation of VM's behaviors

By discharging proof obligations with support of Rodin prover and validating the behaviors with support of animator, we could show the consistency and sufficiency of the formal specification.

5. Checking design against formal specification

Specifications of reactive systems 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 as mentioned in section 4. Contrarily, the designs present internal behaviors of reactive systems. They show details how to implement the system services.

Designs must be close to the implementation. The mathematical data structures must be replaced by the data structures implementable on a computer and under-specified design decisions must be introduced. Generally, designs of reactive systems describe implementation of functions which realize the observable behaviors appearing in the specifications. In this framework, we present the designs of reactive systems described in appropriate specification languages - Promela. To verify the design of reactive systems, environment models are important; they describe possible entities and behaviors in communication with the target system. The environment model of reactive systems is also presented here. We use a simple example, a vending machine, to demonstrate the specifications, the designs, and the environments of reactive systems.

5.1. Describing design in Promela

We assume that the design only defines a set of service functions, it cannot operate by itself. To operate it, we need an environment which calls functions of the reactive system. Therefore, the design needs to be verified in the combination with their environments. We also present the environment model and the combination model.

```
#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.4: Design in Promela

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. Fig. 4 demonstrates a detailed design of the vending machine. In the example, variables having abstract types in the specification, e.g. itemList are replaced by variables having concrete types in the design, 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 on 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 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.

5.2. Communication of system and environment

Fig. 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.

```
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.5. Design and Enviroment in Promela

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. A combination of a design and an environment describes the execution of the design according to the environment.

5.3. Approach

The verification technique used in the framework is model checking. We check the conformance of two models based on the simulation relation between them. In particular, we check whether the design simulates the specification. As demonstrated in Fig. 2, 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. 5 (left) 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. In this framework, we apply model checking to verify

the simulation between two LTSs. 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. For more details we refer the reader to [11].

6. Verifying implementation against formal specification

Specification, Design and Implementation is three main artifacts of every software development. The specification is the highest abstract level description of software. It generally includes desirable properties and external behaviors of the system. The design is more detailed than the specification. It includes internal behaviors. It is a model of the implementation, that is, it describes a solution to realize the observable behaviors appearing in the specification. The implementation is the most concrete. It is built by realizing the design in concrete programming languages. In section 4, we already present an approach to ensure that the design conforms to the specification. This section presents an approach to verify the implementation against the specification by applying the model-based testing technique.

As mentioned earlier, reactive systems do not operate by themselves but in combination with their environments. The stimulus or events from the environment trigger the target system's functions or services. Each event may have additional parameters and may trigger additional events. Each event usually changes the run-time states of the system. Therefore, to guarantee some level of correctness of a reactive system, we need to test it with all many as possible the number of sequences of events (test scenarios) together with the possible values of event parameters. However, the number of test cases grows exponentially by the length of the events and the size of the parameters' domains, so testing reactive systems is challenging. We need an efficient and scalable approach to reduce the number of test cases.

For critical reactive systems, their requirements are usually specified using formal notation, e.g. Event-B specifications. In our framework, we propose a model-based testing approach [13] where models are Event-B specifications. Our approach provides system developers a skeleton that can generate test scenarios which contain both input values and expected results. The skeleton acts as a test environment and a test driver that exercises the system under test (SUT) and reports bugs if the actual results are different from the expected ones.

Of course, there are other techniques such as model checking or formal verification to guarantee some correctness properties of reactive systems. However, testing is still a widely applied in practice, because the formal approaches are only applicable for certain classes of programs and correctness properties.

In particular, our problem statements and approach are summarized as follows. Given an Event-B specification as shown in Fig.2 and an implementation of a reactive system, also called system under test (SUT) as shown in Fig.6, check if all *representative behaviors* of the specification are correctly implemented in the SUT. Here the representative behaviors are the set of different sequences of events with their representative input values.

We know, Event-B model may include abstract types and infinite range of values. To generate sequences of events, 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. This step is to restrict the original Event-B specification to make a so called *specification for test* (SfT). The technique that we use is based on representative values from equivalence class partitioning, a popular black-boxed testing technique [13], which is effective and suitable in our context as we will explain in more details in the next section. Secondly, we build a labeled transition system (LTS) of the state spaces from the restricted specification. Finally, the paths of the LTS are used to build test cases, which contain both test inputs and expected results for assertions. For more details we refer the reader to [12].

7. Experiments

We have illustrated the specification, the design and the implementation of the vending machine partially in Fig. 2, 4 and 5, respectively. 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. Supposing that a domain knowledge database of vending machines shows 3 kinds of items available for customers to be bought such as water, tea, coffee, and 200 for each. Such kind of vending machines should accept quarter, dollar as input before selecting any item. Based on this domain knowledge database, we restrict sets as below:

ITEM in {w(water),t(tea),c(coffee)}

PRICE in {$1}

COIN in {Q(QUARTER),D(DOLLAR)}

AMOUNT in {$0.25, $0.5, $1}

MAXCUP in [0..200]

Table 1: Results outputted by Spin

System	Bounds: size of ranges	LTS Generation			Model Checking		
VM	**numCup**	#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	√

These ranges of values are appropriate in practical applications of vending machines. All experiments are conducted on Intel (R) Core (TM) i5 Processor at 2.67GHz running Windows 10. In verification of the design, verification results outputted by Spin are shown in Table 1. 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 $\surd$ indicates the verification has been completed.

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.

Experiment results of checking the implementation against the formal specification of VM are shown in Table 2. In all cases of our experiment, memory usage is minimal (M). Traces of the test show that about 800 invocations are tested per second and every state and invocation are covered in the test. The test is executed infinitely if no error is detected. This is due to unlimited nature of reactive system, which is caused by unbounded length of the test sequences. The test is interrupted immediately after the first error is detected and the test returns a violation assertion.

Table 2: Results outputted by Test Driver

#	Systems	Mem(Mb)	State Cover.	Arc Cover.	Result
1	VM-Ver1	M	N/A	N/A	1 error
2	VM-Ver2	M	100%	100%	0 error

The experiment #1 are executed infinitely. The experiment #2 returns a violation assertion. Following the trace shown with the error, we could easily detect bugs in implementation of service select item for Vending Machines. Memory usage and performance are very good to decrease cost of the test.

We asked the engineers to intentionally add some bugs into versions of Vending Machine. Then, we test these versions using the same test code. All bugs are detected in a short time. Types of bugs could be detected by our test code as below:

- Conditions enabling services and result of individual services of SUT is not corrected with respect to its specification. For example, one version of VM allows the customers to select an item when the credit is smaller than the price of the item to be bought.
- Combination of services does not operate correctly with respect to its specification. For example, the results of insert money and select item to be bought are not appropriate inputs for dispense item.

8. Discussion

This is a highly automatic framework for software verification. In this framework, we provide an animator, a generator, and a test driver. Firstly, the animator is used to animate the behaviors of the reactive systems; therefore, it supports validating the behaviors of reactive systems. Secondly, the generator is used to generate the environment and assertions from the formal specification. The environment and assertions are then inputted into Spin model checker for verification of the design. Moreover, the generator is also used to generate test cases and test oracles which are inputs for verification of the implementation. Finally, the test driver is used to execute test cases on the implementation and compare to test oracles.

From results of case studies presented in [12] and [13], we found that the framework could be straightforwardly applied to verify various reactive systems (such as operating systems, bank ATMs, elevators) 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 reactive systems could be still verified successfully when we use reasonable restrictions for the sets. This shows applicability of our framework in verification of reactive systems.

9. Conclusion

We present an approach for developing and ensuring the quality of software based on the formal specification of requirements. In this framework, we combine animation and theorem proving to improve the quality of the formal specification. Therefore, we could easily validate the requirements and ensure the

consistency of the requirements. Based on such confident formal specification, we apply model checking technique to verify the design and apply testing technique to verify the implementation. We provide an animator, a generator and a driver to automate steps in this framework. This framework adapts to overall software development processes such as waterfall, evolution model, agile methods. This makes it possible to systematically ensure the quality of reactive systems based on the confident specification with high automation, adaptation, and practicality.

References

[1] 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 (2010). s*eL4: Formal verification of an operatingsystem kernel,"* Communications of the ACM, vol. 53, no. 6, pp. 107– 115.

[2] M. Leuschel and M. Butler (2008), *"ProB: An automated analysis toolset for the B method,"* International Journal on Software Tools for Technology Transfer, vol.10, no.2, pp.185–203.

[3] P. Matos, B. Fischer, and J. Marques-Silva (2009), "*A lazy unbounded model checker for Event-B," in Formal Methods and Software Engineering,* Lecture Notes in Computer Science, vol.5885, 485-503, 2009.

[4] M.P.E. Heimdahl and D. George (2004). *Test-suite reduction for model based tests: effects on test quality and implications for testing.* Proceedings of the 19th IEEE International Conference on Automated Software Engineering. Linz, Austria

[5] Jean-Raymond Abrial (2010). *Modeling in Event-B: system and software engineering*. Cambridge Univ Press.

[6] Andreas Muller (2009). *VDM the vienna development method*. Bachelor thesis in "Formal Methods in Software Engineering", Johannes Kepler University Linz.

[7] Gerard ORegan (2013). *Z formal specification language*. In Mathematics in Computing, pages 109{122. Springer London.

[8] Gerand J. Holzmann (2004). *The SPIN Model Checker – primer and reference manual*. Addison – Wesley.

[9] R. Venkatesh, U. Shrotri, G. M. Krishna and S. Agrawal (2014), "*EDT: A specification notation for reactive systems". Design, Automation & Test in Europe Conference & Exhibition*, pp. 1-6, doi: 10.7873.

[10] J. Fernandes, J. B. Jørgensen, S. Tjell (2007). *Requirements Engineering for Reactive Systems: Coloured Petri Nets for an Elevator Controller*. The 14th Asia-Pacific Software Engineering Conference (APSEC'07).

[11] Dieu-Huong Vu, Yuki Chiba, Kenro Yatake, Toshiaki Aoki (2015). *A Framework for Verifying the Conformance of Design to Its Formal Specifications*. IEICE Trans. Inf. Syst. Vol.98-D, no.6, pp.1137—1149.

[12] Dieu Huong Vu, Anh Hoang Truong, Yuki Chiba, Toshiaki Aoki (2017). *Automated testing reactive systems from Event-B model. 4th NAFOSTED Conference on Information and Computer Science*, 2017, pp. 207-212, doi: 10.1109/NAFOSTED.2017.8108065.

[13] Frederic Dadeau, Kalou Cabrera Castillos, and Regis Tissot (2012*). Scenario based testing using symbolic animation of B models*. Software Testing, Verification and Reliability, 22(6):407-434.

[14] Nadeem Akhtar, Malik M. Saad Missen (2014). *Contribution to the Formal Specification and Verification of a Multi-Agent Robotic System*. European Journal of Scientific Research ISSN 1450-216X / 1450-202X Vol.117 No.1, pp. 35-55.

[15] Phillip A. Laplante (2017). *Requirements Engineering for Software and Systems*. ISBN 9781138196117 by Auerbach Publications. 400 Pages 95 B/W Illustrations.

[16] Luca A., Anna I., Kim G. L., and Jiri S (2011). *Reactive system modeling specification and verification*. Cambridge University Press. ISBN 978051181410.