International Journal of Computational Science
1992-6669 (Print) 1992-6677 (Online) www.gip.hk/ijcs

2009, Vol. 3, No. 5, 554-567.

Design and Implementation of Executable UML Platform for Assertion-based Dynamic Verification

Masahito Sugai, Akira Teruya, Eiichiro Iwata,
Noriko Matsumoto, Norihiko Yoshida*

Department of Information and Computer Sciences, Saitama University,
255 Shimo-Ohkubo, Saitama 338-8570, Japan
{masahito, akira, eiichiro, noriko, yoshida}@ss.ics.saitama-u.ac.jp

Abstract. Executable UML (xUML) is executable, thus verifiable on top of rigorous semantics. This paper proposes a platform of dynamic property verification, or assertion-based dynamic verification for xUML specifications. The assertion-based verification improves efficiency and reliability of the design process, for instance, of embedded systems. The design and implementation of the platform, and some simple verification examples are presented.

Keywords: Assertion-based verification, dynamic verification, executable UML.

1 Introduction

Embedded systems have grown incredibly large and complex, and their verification is becoming significantly important. The man-hour requirement of verification is also increasing. Hence efficient verification techniques are strongly required.

Recently, in the embedded systems design paradigm, Executable UML (xUML) [1] is attracting attentions, as its model can be executed and verified on a simulator. It is a crucial issue to improve the efficiency of the xUML verification.

One of the most effective verification techniques is assertion-based verification [2], which improves verification efficiency and reliability of the systems. Assertion-based verification uses "asser-

* Corresponding Author. Email: yoshida@mail.saitama-u.ac.jp.

tions" which are declarations of required properties for a target system, and checks the target for assertion violations. Assertion-based verification checks assertions only, whereas conventional verification checks all the statements in the whole code. Therefore it improves verification efficiency. This verification technique is also divided into the same two types as conventional verification technique: assertion-based static verification and assertion-based dynamic verification. The former checks all the possibilities for assertion violations exhaustively. The latter checks target codes for assertion violations in simulation.

Assertion-based verification is considered also effective to improve the verification efficiency on xUML [4]. Xie et al. proposed assertion-based static verification for xUML [5]. They defined new property specification language, xPSL, for declaring assertions. xPSL is an extended language to PSL that supports declarations such as the variables, events, state, and so on. Both an xUML model and xPSL assertions for it are translated into a formal language, which is an input format to the existing tool. Although this research focuses on the assertion-based static verification, the assertion-based dynamic verification is also necessary for the verification on xUML because the latter executed in simulation is more feasible and practical than the former.

Based on the above-mentioned background, we propose the assertion-based dynamic verification for xUML in this paper. There are no other researches on this topic than ours as far as we know. In particular, this paper presents how the assertion-based dynamic verification is realized in xUML. This research will contribute to improve verification efficiency, and design process as a whole in industry.

The rest of this paper is organized as follows: Section 2 summarizes xUML, and Section 3 summarizes assertion-based verification. In Section 4, we define a manner to specify assertions declared in xUML. In Section 5, we devise a manner to realize the assertion-based dynamic verification for xUML. In Section 6, we confirm the validity of our proposal through some experiments. In Section 7, we discuss the applicability of our proposal to every xUML tool and differences among PSL, xPSL, and the assertion we defined. Section 8 concludes this paper with future work.

2 Executable UML (xUML)

xUML is an extended language of UML to be able to execute its models designed by UML diagrams. Therefore the system designed in xUML can be verified dynamically.

In design using xUML, the system is divided into some domains, where each domain is an autonomous world composed of some entities. To execute models, the required diagrams are the class diagrams and the statechart diagrams such as Fig. 1 and 2, respectively. `ClassA` has an attribute `att1` and an operation `op1`, and `ClassB` has two attributes `att1`, `att2` and an operation `op2`. The line between the classes means the association, and the association has multiplicity specified on both ends of the line. The multiplicity are restricted to "0..1" (zero or one), "0..*" (zero or more), "1" (ex-actly one), or "1..*" (one or more). The class behaves according to its statechart diagram. In the statechart diagram, each state has a procedure which is composed of action groups. Each action is a unit of a calcu-

lation process such as a loop and an access to data. The state transition occurs when an action generates an event such as event1, event2 and event3 in Fig. 2. The action is specified using the action language. The semantics of action languages conforms to UML Action Semantics [6] defined by OMG. However there is no standard action language.

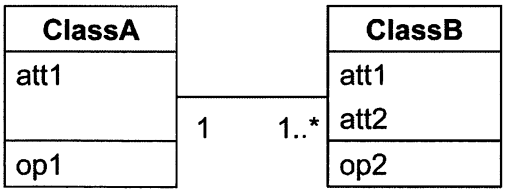

Fig. 1. Example of class diagram

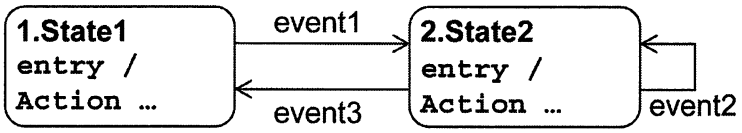

Fig. 2. Example of statechart diagram

As an xUML tool, we use iUML [7] whose action language is ASL(Action Specification Language) [8]. Some examples of the action in ASL are as follows:

a. `instA = find-one ClassA where att1 = 1`

b. `instA.att1 = x`

c.
```
loop
        x = x * x
        i = i + 1
        breakif   i >= 10
   endloop
```

d. `generate CA1:event1() to instA`

The action **a** returns `instA` an instance of `ClassA` that the value of attribute `att1` is 1. The action **b** substitutes the value of variable `x` for the attribute `att1` of instance `instA`. The action **c** is a loop terminated when the value of variable `i` is 10 or more. The action **d** generates the event `event1` destined for instance `instA`.

3 Assertion-based Verification

Assertions are specified using some property specification language. In general, properties are specified based on classical or temporal logic. There are some property specification languages as below:

- PSL (Property Specification Language) [3]

- OVL (Open Verification Library)
- e Temporal Language
- OVA (OpenVera Assertion)
- SVA (SystemVerilog Assertion)

Assertion-based verification reports unsatisfied properties as in error messages. It means that the verification efficiency can be improved by easy identification of error locations.

This verification technique is divided into two types: the assertion-based static verification and the assertion-based dynamic verification. The former, called model checking, checks every state for assertion violations. The verification is always exhaustive in this method, however, it implies possible state explosion. The latter one checks target codes for assertion violations in simulation. Although its verification can be carefully arranged and executed, it can never be exhaustive.

4 Assertion Specification

4.1 Syntax

Fig. 3 shows the syntax of the assertion in BNF. We define the syntax based on PSL. In this figure, boldface words are reserved keywords and operators. "Condition" except for "Event_Specification" in "Component" conforms to the syntax of the conditional statement, namely, "IF" statement of ASL. "Event_Specification" conforms to the syntax of ASL [8].

```
Assert_Statement ::= assert Property
Property ::= Condition
           | ( Property )
           | Property -> Property
           | Property <-> Property
           | always Property
           | eventually! Property
Condition ::= Component
            | Component Binary_Logical_Operator Component
            | ! Component
Component ::= Instance_Handle.Attribute
            | Local_Variable
            | Constant
            | countof { Instance_Handle_Set }
            | countof Class
            | Event_Specification
Binary_Logical_Operator ::= = | != | < | > | <= | >= | & | |
```

Fig. 3. Assertion syntax

4.2 Specification

Now we present how and where to specify the assertions. The place to specify differs according to the type of assertions.

We show the classical assertions first. Based on its property, we define that the place is in a state procedure of a statechart diagram. This means we checks it when the process reaches the point where it is described. In order to keep its readability, we additionally use the comment line which is easy to distinguish between the primary processes and the assertions. The next two examples show how to specify the classical assertions:

```
- # assert this.val1 > 0 & this.val2 = 0
- # assert !x _> y
```

where "#" is a comment line in ASL language.

Next is the temporal assertion, we describe it in two files, namely "class_keyletter.ast" and "domain.ast". The term "keyletter" here is the abbreviated name of the target class, and we use "ast-files" as the term to refer to these files. The "class_keyletter.ast" file describes the assertions which relates to the class. The "domain.ast" describe the assertions which relates to the domain. In this way, we can specify what kind of instance is maintained in "Instance_Handle" or "Instance_Handle_Set" used in the property of the assertions. The next two examples show how to specify temporal assertions:

```
- assert always (this.flag1 | !this.flag2)
- assert eventually! this.val1 = 0
```

5 Implementation

5.1 Basic Policy

As the basic policy, we use a conditional judgment to implement the function which checks each state or whole state with the required property and generates error messages if it does not satisfy the property. That is, we implement the function as it generates the conditional judgment from the assertion and then inserts the generated conditional judgment into the appropriate place in the code file.

In order to confirm feasibility of our basic policy, we construct an experimental implementation on iUML. To generate an execution file, iUML performs "Write" processing and "Build" processing. As a result of "Write", we can get some files which are necessary to generate the execution file. We call these files "Write-files". "Write-files" are the files which inform us of the class, the procedure of the state described by ASL (al-file), and so on. On the contrary, "Build" is executed after "Write". Based on the Write-files, "Build" generates files which are the code in the programming language C (C-files), after that, it generates execution files from C-files. Therefore, we implement the assertion-based dynamic verification following Fig. 4. This implementation follows the below procedures:

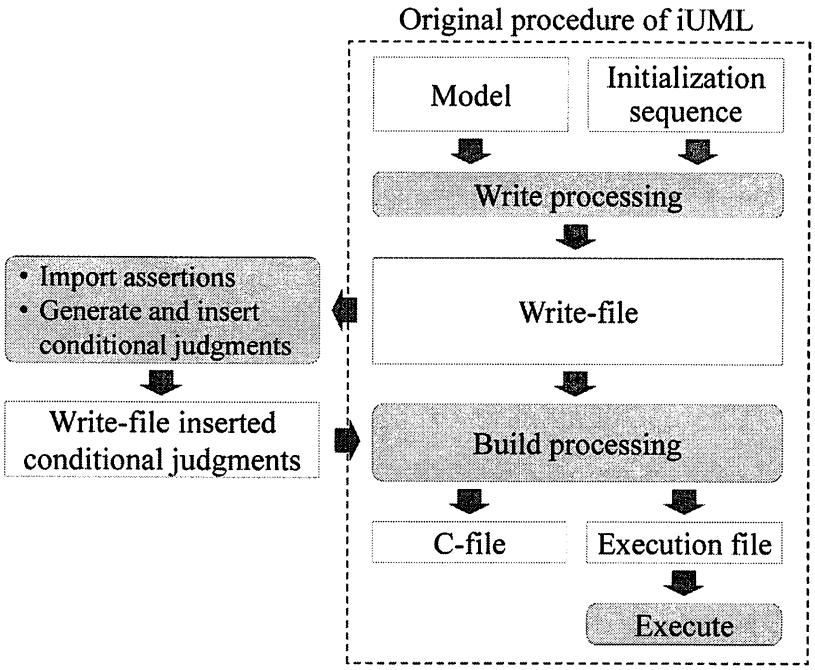

Fig. 4. Implementation flow

1. Import the assertions from al-files and ast-files.
2. Generate conditional judgments following ASL syntax from the imported assertions.
3. Insert the generated conditional judgments into al-files.
4. Generate the execution file by "Build" processing.

To realize the above procedure, we have made a program, which is a precompiler for "Build", using the programming language Ruby [9].

5.2 Classical Logic

In classical properties, the state should be checked when the execution reaches the point where the assertion is described. Therefore we insert the conditional judgment just after the assertion.

Fig. 5 indicates an example of an assertion and the code generated from the assertion. The conditional judgment is shown in "Insert code", as from the second line to the sixth. The condition sentence is basically logical negation of the property, so that the error message is generated when the state satisfies the negation. However, if the property includes an event specification as shown in Fig. 5, the event specification is replaced with a flag which corresponds to the event generation. This flag is initialized to FALSE in the model initialization sequence, and also this flag is changed to TRUE just after an action generating the event, and changed to FALSE just after the conditional

Assertion

```
1  # assert SC1:event1 & x
```

⇓ Generate code

Insert code

```
1  ef = find-only EventFlag
2  if !(ef.flag1 & x) then
3  $INLINE
4  fprintf(stderr, "Assertion error: \"SC1:event1 & x\" failed
   at domain: %s, file: %s, line: 1\n", CGEN_DOMAINNAME,
   CGEN_SOURCENAME);
5  $ENDINLINE
6  endif
7  ef.flag1 = FALSE
```

Fig. 5. Example of generating insert code

judgment. In addition, ASL does not support some logical operators such as logical implication "–>" and logical equivalence "<–>". Therefore we transform them to equivalent formula: for examples "x –> y" is transformed to "!x|y", and "x <–> y" is transformed to "(x & y)|(!x & !y)", where "!", "|" and "&" indicates logical negation, logical disjunction and logical conjunction, respectively.

ASL has a facility of "in-line statements" which can be inserted into C-files as is. As ASL does not support the print-out functionality, we use the "fprintf" function of the language C for the output of error messages as inline statements. In Fig. 5, CGEN_DOMAINNAME and CGEN_SOURCENAME indicates the abbreviated domain name and the ast-file name, respectively, and the name of the ast-file indicates the class and the state. Therefore, from the error message, we can identify which domain, class, and state do not satisfy the property.

5.3 Temporal Logic

In this subsection, we discuss simple temporal properties appending only one temporal operator to a classical logic expression, and how to implement it.

5.3.1 Always

How can we implement the function to check the simple temporal property, "a classical logic expression is always satisfied"? Fig. 6 shows an example of an assertion described in an ast-file and a code generated from it.

In order to verify that a certain classical logic expression is always satisfied, we must check it in every state, however we need not check it after once any state does not satisfy it. Therefore, we use a flag initialized to TRUE, and insert a conditional judgment using it into every state. The judgment condition is expressed as the logical conjunction of the flag and logical negation of the

classical logic expression. When negation of the logical expression satisfies, the flag is changed to FALSE. In addition, when the ast-file describes an assertion about a certain class, "Code inserted into every state" in Fig. 6 is inserted into every state of the class. On the contrary, when the ast-file describe an assertion about a certain domain, the code is inserted into every state of every class related to the domain.

ast-file (class_SC.ast)

```
1  assert always this.val != 0
```

⇓ Generate code

Code inserted into every state

```
1  af = find-only AssertionFlag
2  if af.ok1 & !(this.val != 0) then
3  $INLINE
4  fprintf(stderr, "Assertion error: \"always this.val != 0\"
   (domain: SD, file: class_SC.ast, line: 1) failed at domain: %s,
   file: %s\n", CGEN_DOMAINNAME, CGEN_SOURCENAME);
5  $ENDINLINE
6  af.ok1 = FALSE
7  endif
```

Fig. 6. Example of generating insert code (always operator)

5.3.2 Eventually!

Next, how can we implement the function to check the temporal property, "a classical logic expression is eventually satisfied"? Fig. 7 shows an example of two code blocks generated from an assertion which is described in an ast-file. In order to verify that a certain classical logic expression is eventually satisfied, we must check it in every state. When it never satisfies the condition from the beginning of the execution to the end of the procedure of the last transition state, we must get an error message. Therefore we use a flag initialized to FALSE, and insert a conditional judgment using it into every state and the last state of the state transition. The judgment condition inserted into the former, every state, is expressed as the logical conjunction of logical negation of the flag and the classical logic expression. When this logical expression satisfies, the flag is changed to TRUE. On the contrary, the conditional judgment inserted into the latter one, the last state of the state transition, is expressed as logical negation of the flag. When this logical expression satisfies, the error message is generated.

ast-file (class_SC.ast)

```
1  assert eventually! this.val = 1
```

⇓ Generate code

Code inserted into every state

```
1  af = find-only AssertionFlag
2  if !af.ok1 & (this.val = 1) then
3  af.ok1 = TRUE
4  endif
```

Code inserted into the last state of the transition

```
1  af = find-only AssertionFlag
2  if !af.ok1 then
3  $INLINE
4  fprintf(stderr, "Assertion error: \"eventually! this.val = 1\"
   (domain: SD, file: class_SC.ast, line: 1) failed at domain: %s,
   file: %s\n", CGEN_DOMAINNAME, CGEN_SOURCENAME);
5  $ENDINLINE
6  endif
```

Fig. 7. Example of generating insert code (eventually! operator)

6 Experiments

In order to confirm our implementation, we constructed an oven model and a vending machine model using iUML.

6.1 Oven Model

Fig. 8 shows the class diagram of the oven. Fig. 9 shows the statechart diagram of the ControlDevice class, and in this statechart diagram, we intentionally commented out one processing of state **4**, which is shown in Fig. 10. This intention is to violate an assertion as mentioned below:

- `# assert this.heater_on = FALSE`

which is inserted in the beginning of the state **1** of the ControlDevice class sequence.

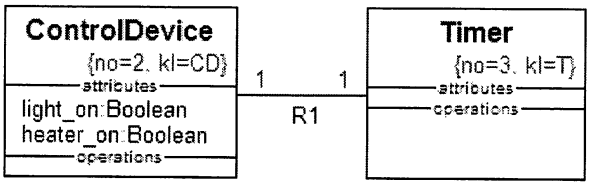

Fig. 8. Class diagram of oven model

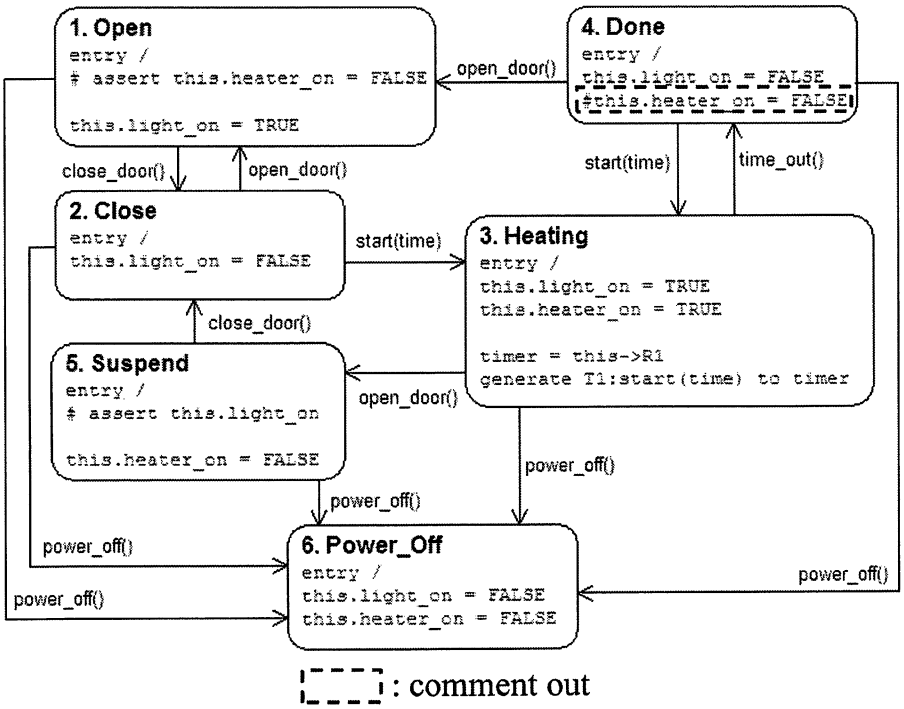

Fig. 9. Statechart diagram of ControlDevice class

```
this.heater_on = FALSE
```

Fig. 10. Action commented out in state **4**

Fig. 11 shows the first execution result. This execution performed the state transition in order of the states **2**, **3**, **4** and **1** in an instance of the ControlDevice class. In Fig. 11, an error message of violations is reported against the above assertion. In the state **3** the value of `this.heater_on` is set to TRUE, and in the state **4** it is not set to FALSE because of the intentional comennting out. Therefore in the state **1**, the value of `this.heater_on` is TRUE and the above assertion is not satisfied.

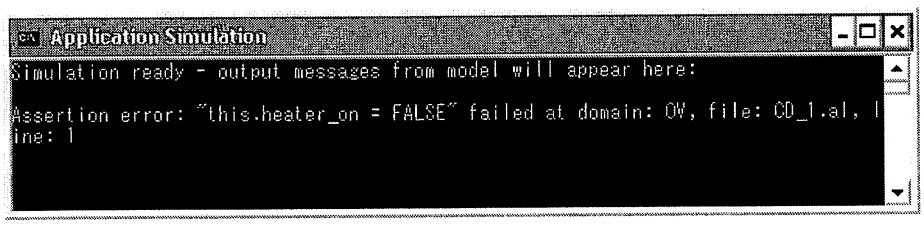

Fig. 11. Execution result

Secondly, we executed the same, without commenting out the statement in Fig. 10, and we confirm that it did not report the error messages. This result indicates that the above assertion is satisfied as we expected. In the state **4** the value of `this.heater_on` is set to FALSE. Therefore in the state **1** the value of `this.heater_on` is FALSE, and the above assertion is satisfied.

6.2 Vending Machine Model

Fig. 12 shows the class diagram of the vending machine. Fig. 13 shows the statechart diagram of the Can_Slot class, and in this statechart diagram, we intentionally commented out a few processing of state **2** and **3**, which are shown in Fig. 14 and Fig. 15, respectively. This intention is to violate assertions as mentioned below:

a. `# assert this.purchasability_lamp`
b. `assert always (this.stock=0 -> CS5:sellout)`
c. `assert eventually! this.purchasability_lamp`

We inserted assertion **a** in the beginning of the state **3** of the Can_Slot class sequence, and described assertion **b**, **c** in the ast-file of Can_Slot class (class_CS.ast).

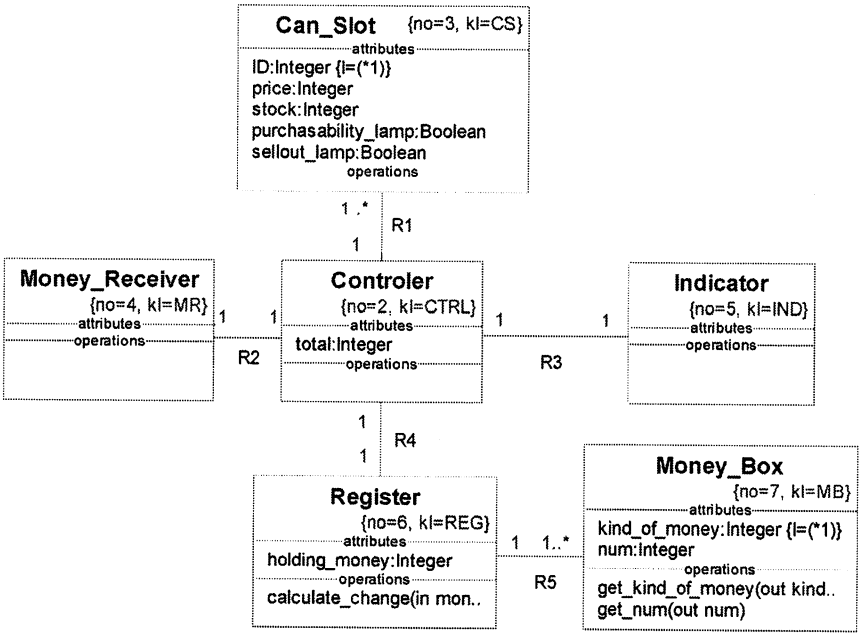

Fig. 12. Class diagram of vending machine model

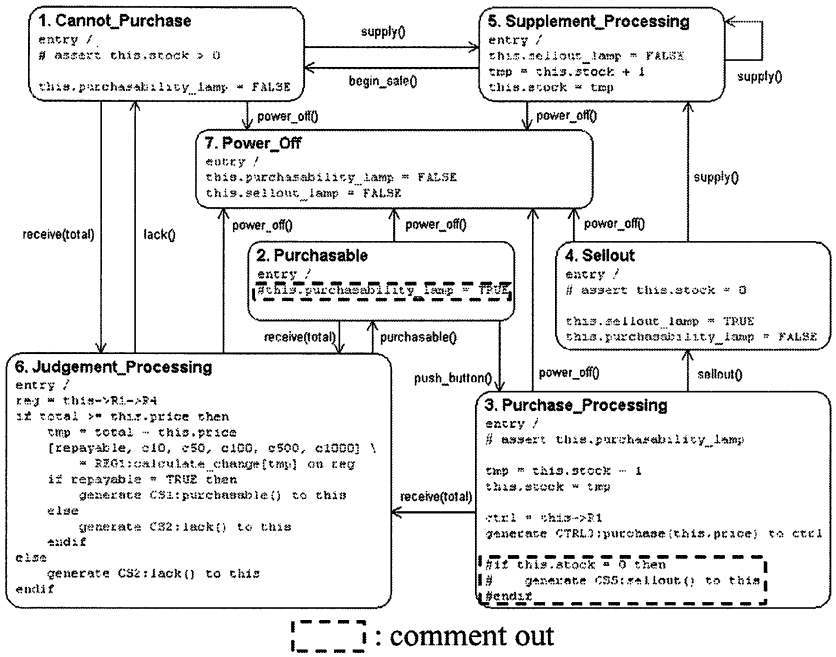

Fig. 13. Statechart diagram of Can_Slot class

```
this.purchasability_lamp = TRUE
```

Fig. 14. Action commented out in state **2**

```
if this.stock = 0 then
    generate CS5:sellout() to this
endif
```

Fig. 15. Action commented out in state **3**

Fig. 16 shows the first execution result. This execution performed the state transition in order of the states **1**, **6**, **2**, **3** and **7** in an instance of the Can_Slot class. **A**, **B** and **C** are the error messages of violations against above assertion **a**, **b** and **c**, respectively. In the state **1** the value of `this.purchasability_lamp` is set to FALSE, and in the state **2** it is not set to TRUE because of the intentional comennting out. Therefore in the state **3**, the value of `this.purchasability_lamp` is FALSE and the assertion **a** is not satisfied. In the state **3**, although the value of `this.stock` is 0, the event `CS5:sellout` is not generated becouse of the intentional comennting out. Therefore in the state **3** the assertion **b** is not satisfied. In every state the value of `this.purchasability_lamp` is never set to TRUE because of the intentional comennting out in the state **2**. Therefore the assertion **c** is not satisfied.

Secondly, we executed the same, without commenting out the statement in Fig. 14, and we confirm that it did not report the error messages (**A** and **C**). This result indicates that the assertion **a** and **c** satisfied its property as we expected. In the state **2** the value of `this.purchasability_lamp` is TRUE, and the assertion **c** is satisfied. In addition, in the state **3** the value of `this.purchasability_lamp` is TRUE, and the assertion **a** is satisfied.

In the same way, we executed the same, without commenting out the statement in Fig. 15, and we confirmed that the error message **B** was not reported. This result indicates that the assertion **b** is satisfied as we expected. In the state **3** the value of `this.stock` is 0, and the event `CS5:sellout` is generated. Therefore the assertion **b** is satisfied.

Based on these results, we confirmed our implementation is appropriate.

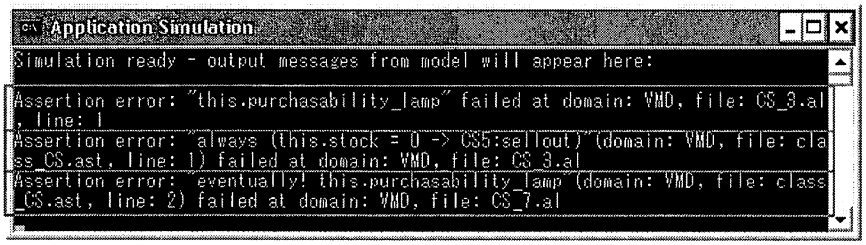

Fig. 16. Execution result

7 Discussions

We used iUML in the above experiments. Our assertion-based dynamic verification in xUML generates the conditional judgments of ASL from the assertions, and inserts them in some files describing the state procedures by ASL. iUML generates C-files to execute, and we implemented our approach in the language C. This means it is possible to apply our approach to another xUML tool if the tool generates codes written in any concrete programming language to execute. In fact, any tool is considered generating files of program codes to execute in some programming language. Hence, we conclude that our proposal is applicable to any other xUML tool as well.

Next, we discuss differences among PSL, xPSL, and the assertions we defined. PSL focuses on hardware description language (HDL), and supports various temporal operations. xPSL is an extended language to PSL to support declarations of variables, events, and so on. xPSL focuses on both xUML and HDL. On the contrary, our assertion focuses on xUML only, especially iUML.

8 Conclusions

In order to realize the assertion-based dynamic verification for xUML, we proposed a specification of assertions and implementation of the property checking, and confirmed the validity of our proposal through some experiments on the classical logic and the simple temporal logic.

The extension for more complicated temporal properties are left as our future work.

Acknowledgments

This research was supported in part by Joint Research Project with Nippon Signal Co. Ltd. The authors are grateful to Ms. Nurul Azma Zakaria for her valuable contribution.

References

1. Stephen J. Mellor, Marc J. Balcer: *Executable UML: A Foundation for Model-Driven Architecture*, Addison-Wesley. (2002)
2. Harry D. Foster, Adam C. Krolnik, David J. Lacey: *Assertion-Based Design*, Kluwer Academic. (2003)
3. Harry Foster, Erich Marschner, et al.: *Property Specification Language Reference Manual Ver. 1.1*, http://www.eda.org/ieee-1850/.
4. Masahito Sugai, Akira Teruya, Eiichiro Iwata, Nurul Azma Zakaria, Noriko Matsumoto, Norihiko Yoshida: Assertion-based Dynamic Verification for Executable UML Specifications, *Proc. 8th Int. Conf. on Applied Computer Science* (2008) 181-186.
5. Fei Xie, Huaiyu Liu: Unified Property Specification for Hardware/Software Co-Verification, *Proc. 31st Annual International Computer Software and Applications Conf.* 1 (2007) 483-490.
6. UML Action Semantics: http://www.omg.org/cgi-bin/doc?ptc/02-01-09.
7. iUML: http://www.kc.com/products/iuml.php.
8. Ian Wilkie, Adrian King, Mike Clarke, et al.: *UML ASL Reference Guide*, http://www.kc.com/download/index.php.
9. David Thomas, Andrew Hunt: *Programming Ruby*, Addison-Wesley. (2000)