International Journal of Information, Control and Computer Sciences
ISSN: 2517-9942
Vol:4, No:11, 2010

Model checking consistency of UML diagrams using Alloy

Akie NIMIYA†, Tomoyuki YOKOGAWA†, Hisashi MIYAZAKI†‡,
Sousuke AMASAKI†, Yoichiro SATO†, and Michiyoshi HAYASE†

Abstract—In this paper, we proposed a method for detecting consistency violation between UML state machine diagrams and communication diagrams using Alloy. Using input language of Alloy, the proposed method expresses system behaviors described by state machine diagrams, message sequences described by communication diagrams, and a consistency property. As a result of application for an example system, we confirmed that consistency violation could be detected using Alloy correctly.

Keywords—model checking, UML, state machine diagrams, communicatiuon diagram, Alloy.

I. Introduction

Unified Modeling Language (UML)[1] is a formal language used to describe structure and behavior of a software system and is widely used in software development. In software development using UML, a design described by UML diagrams may contain inconsistency even if each diagram has no error. Because it is difficult to detect inconsistency with human review, an automatic detection is expected. We have developed a method for verifying consistency of UML diagrams using symbolic model checker SMV[2], [3].

In the case of verifying large systems, however, size and complexity of an input of SMV become increased. In addition, when a requirement is not fulfilled, SMV could produces only one counterexample which is a trace of a system execution violating that requirement.

Alloy [4], [5] is a simple structural modelling language supported by Alloy analyzer. The Alloy language is used to express complex structural constraints and behaviour. It is a constraint solver with full automatic simulation and verification. The Alloy language is based on the first-order logic that allows a user to model a system by abstracting key characteristics of that system. The Alloy analyzer generates all instances showing whether their properties are satisfied or not.

Some methods were proposed for verifying UML diagrams using Alloy [6], [7]. In [6], Simons et al. proposed a method to convert a subset of UML used in the discovery method into an abstract syntax input to Alloy. In [7], Zito et al. formalized and analyzed package merge concept in UML 2.0 using Alloy. These studies, however, do not focus on consistency of UML diagrams.

In this paper, we proposed a method for verifying consistency of UML diagrams using Alloy. The proposed method converts state machine diagrams and a communication diagram into a model for Alloy. It also provided a way for detecting inconsistency of these diagrams using Alloy analyzer.

†Graduate School of Systems Engineering, Okayama Prefectural University, Kuboki 111, Soja-shi, Okayama, 719–1197 Japan (e-mail: nimiya, miyazaki@circuit.cse.oka-pu.ac.jp, {t-yokoga, amasaki, sato, hayase}@cse.oka-pu.ac.jp).
‡Kawasaki University of Medical Welfare, 288 Matsushima, Kurashiki-shi, Okayama 701-0193, Japan (e-mail: miyazaki@me.kawasaki-m.ac.jp).
Manuscript received September, 2010; revised October 31, 2010.

II. Consistency verification with Alloy

A. Representation of a state machine diagram

Figure 1 shows a state machine diagram with tree transitions t_1, t_2 and t_3. At the start of the execution,the state s_1 is active. t_1 is executed in the case that the state s_1 is active, the guard condition "g is true" evaluates true and the event e is activated, and then the state s_2 becomes active in place of s_1 and the action "activate event a" is executed. t_2 is executed in the case that s_2 is active and g is false, and then the action "increment the variable c" is executed. The state machine reaches to a final state by t_3 when s_2 is active and e is activated. The initial values of the variables g and c are $true$ and 0, respetcively.

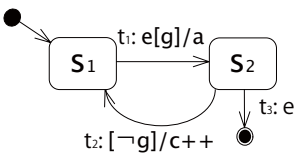


Fig. 1. An example of a state machine diagram

The behaviour of a state machine diagram is defined as a transition system with an ordered set of elements which consists of states, events and variables. We call this element *step* and define a type `Step` for it. `Step` of the state machine diagram in Figure 1 is defined as follows:

```
enum States{s1,s2,fin}
enum Events{e,a}
open util/ordering[Step]
sig Step{
  st:some States,
  ev:set Events,
  g:Int,
  c:Int
}
{ 0<=g && g<=1 }
```

`States` is a set of state names and `Events` is a set of event names. `util/ordering[Step]` provides order relation for

Step. st, ev, g and c are members of Step. st and ev represent active states and activated events, respectively. The boolean variable g is defined as an integer variable g with a condition $0 \leq g \leq 1$.

Since the step changes by a transition, the order relation of Step is represented by a disjunction of pre- and post-conditions of all transitions. The transitions in Figure 1 are represented as follows:

```
fact{
  all s:Step, s':s.next{
       s1 in s.st && e in s.ev && s.g=1
    && s'.st=s.st-s1+s2 && s'.ev=s.ev-e+a
    && s'.g=s.g && s'.c=s.c
    || s2 in s.st && s.g=0
    && s'.st=s.st-s2+s1 && s'.ev=s.ev
    && s'.g=s.g && s'.c=s.c+1
    || s2 in s.st && e in s.ev
    && s'.st=s.st-s2+fin && s'.ev=s.ev
    && s'.g=s.g && s'.c=s.c
  }
}
```

The order relation is defined by fact notation. s is a variable of type Step and s' represents the subsequent element of s. The pre-condition of t_1 is represented as a boolean formula which means state s1 is active, event e is activated, and guard condition g=1 evaluates true in s. The post-condition of t_1 is represented as a boolean formula which means state s2 becomes active in place of s1, event e is disactivated and event a is activated in step s'. Boolean formulas representing t_2 and t_3 are obtained as in the case of t_1. The order relation of the state machine diagram is obtained as a disjunction of the boolean formulas representing transitions.

Initial states of a state machine diagram is also defined by fact notation. The initial states in Figure 1 is represented as follows:

```
fact{
  first.st=s1
  #first.ev=0
  first.g=1
  first.c=0
}
```

first represents the first element of the ordered set Step and # represents the number of elements in the set. This means that state s1 is active, no event is activated, and values of g and c are 1 and 0 in the initial state.

B. Representation of a communication diagram

Figure 2(a) shows a communication diagram which has two message communications. First Obj1 sends message e to Obj2, and then sends message a to Obj2. In this diagram, the first message is a synchronous message (denoted by the solid arrowhead) completed with an implicit return message and the second message is an asynchronous message (denoted by line arrowhead).

First, labels snd and rcv with sequence number are attached at both ends of message communications as shown in Figure 2(b). The completed communication is defined as a set of the labels. The behaviour of a communication diagram is represented as changes of the set by the transition of state machine diagrams. These labels are defined as the following set Label:

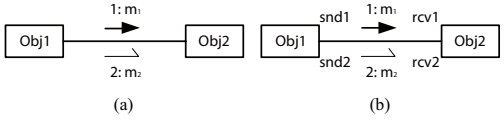


Fig. 2. Examples of communication diagrams

```
enum Label{snd1,rcv1,snd2,rcv2}
```

Next, we add a variable cp of type Label to Step. cp represents a set of completed communications.

```
sig Step{
  st:some States,
  ev:set Events,
  g:Int,
  c:Int,
  cp:set Label;
}
```

In the initial state, cp is empty.

```
fact{
  #first.cp=0
}
```

A label *lbl* is added to cp in the case that *lbl* is not in cp, the precedent labels of *lbl* are in cp and the message communication is executed. Since snd1 has no precedent label, the condition to add snd1 to cp is that snd1 is not in cp and m1 becomes active. The change of cp by sending the first message m_1 is represented as the following boolean formula:

```
!(snd1 in s.cp)
&& !(m1 in s.ev) && m1 in s'.ev
&& s'.cp=s.cp+snd1
```

The precedent label of rcv1 is snd1, which is the sender of m_1. The condition to add rcv1 to cp is that snd is in cp, rcv1 is not in cp and m1 becomes inactive. The change of cp by receiving m_1 is represented as the following boolean formula:

```
snd1 in s.cp && !(rcv1 in s.cp)
&& m1 in s.ev && !(m1 in s'.ev)
&& s'.cp=s.cp+rcv1
```

As in the case of the first message, the change of cp by sending the second message m_2 is represented as the following boolean formula:

```
(snd1 + rcv1) in s.cp && !(snd2 in s.cp)
&& !(m2 in s.ev) && m2 in s'.ev
&& s'.cp=s.cp+snd2
```

Since the precedent message communication of m_2 at Obj1 is synchronous, the precedent labels of snd2 are snd1 and

rcv1. The change of cp by receiving m_2 is represented as the following boolean formula:

```
(snd2 + rcv1) in s.cp
&& m2 in s.ev && !(m2 in s'.ev)
&& s'.cp=s.cp+rcv2
```

The precedent labels of rcv2 are snd2, which is the sender of the message, and rcv1, which is the precedent message communication at Obj2.

In addition, cp does not change if all events are unchanged, cp includes all labels or no condition to add labels is satisfied. This is represented as the following formula:

```
s'.cp=s.cp
&& (s'.ev=s.ev
    || s'.cp=snd1+rcv1+snd2+rcv2
    || !((!(snd1 in s.cp)
          && !(m1 in s.ev) && m1 in s'.ev)
       ||(snd1 in s.cp && !(rcv1 in s.cp)
          && m1 in s.ev && !(m1 in s'.ev))
       || ... ) )
```

The behaviour of a communication diagram is a disjunction of boolean formulas of all message communications. The message communications in Figure 2 are represented as follows:

```
fact{
  all a:Step, s':s.next{
    !(snd1 in s.cp)
    && !(m1 in s.ev) && m1 in s'.ev
    && s'.cp=s.cp+snd1
    || snd1 in s.cp && !(rcv1 in s.cp)
    && m1 in s.ev && !(m1 in s'.ev)
    && s'.cp=s.cp+rcv1
    ...
    || s'.cp=s.cp
    && (s'.ev=s.ev
        || s'.cp=snd1+rcv1+snd2+rcv2
        || ... )
  }
}
```

C. Representation of consistency

We considered consistency as correspondence between behaviour of state machine diagrams and a communication diagram. A model which satisfies these diagrams can be obtained as a conjunction of boolean formulas representing these diagrams. If cp will eventually include all of the labels, this model is consistent. This property is represented as follows:

```
last.cp = snd1+rcv1+snd2+rcv2
```

last represents the last element of the ordered set Step.

D. Finding model

The Alloy analyzer has two types of executions: simulation and checking. In simulation execution the Alloy analyzer finds instances which satisfy the model and given specification. In checking execution the analyzer finds counterexamples to an assertion.

Our method first carries out a simulation execution to verify whether the given UML diagrams can reach the final states. The reachability to the final state is represented by the following predicate stable:

```
pred stable(){
  fin in last.st
}
```

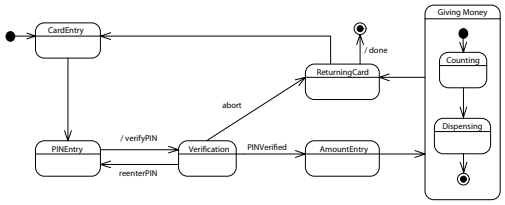
If the reachability is satisfiable, the analyzer finds the model instance which indicates the trace to the final state. Then a checking execution is carried out to verify the consistency of the diagrams. In order to generate counterexamples, an assertion representing inconsistency is provided as follows:

```
assert inconsistent{
  !(last.cp = snd1+rcv1+snd2+rcv2)
}
```

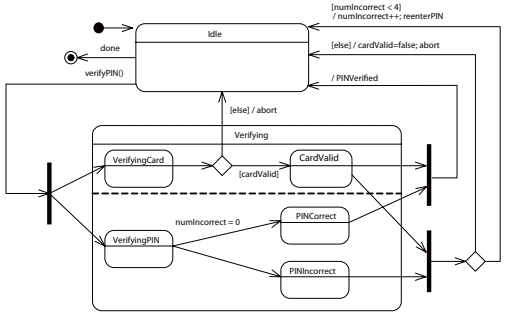
An assertion is defined by assert. If the assertion is violated (that is, consistency is satisfied), the instance which shows the consistency as a counterexample by the Alloy analyzer.

III. Application Result

We applied the proposed method to the ATM system[8] described by state machine diagrams in Figure 3 and communication diagrams in Figure 4. In this verification, the number of Step is set to 25.



(a) ATM



(b) Bank

Fig. 3. State machine diagrams

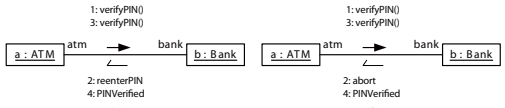


Fig. 4. Communication diagrams

We first carried out the simulation execution to the model obtained from the state machine diagrams in Figure 3 as follows.

```
pred stable(){
  fin_ATM+fin_BANK in last.st
}
run stable for 25
```

The result of the simulation execution is shown in Figure 5(a). Figure 5(a) indicates that a model obtained from those diagrams has an instance. Then the checking execution was carried out to verify the consistency of the diagrams as follows.

```
assert inconsistent{
  !(last.cp=S1+R1+S2+R2+S3+R3+S4+R4)
}
check inconsist for 25
```

The result of the checking execution is shown in Figure 5(b). Figure 5(b) indicates that this model had counterexample which violated the assertion, that is, this model is consistent. A counterexample generated by Alloy is shown in Figure 6.

Next we carried out the checking execution to the model obtained from the state machine diagrams and the communication diagram in Figure 4(b). The result of the checking execution is shown in Figure 5(c). Figure 5(c) shows that an obtained model has no counterexample, that is, this model is inconsistent. To make this counterexample more visible, we added the following relations to the program.

```
fact {
  trs = CardEntry->PINEntry
      + PINEntry->Verification
      + Verification->AmountEntry
      + Verification->ReturnCard
      + ...

  sub = GivingMoney->Counting
      + GivingMoney->Dispensing
      + GivingMoney->fin_GivingMoney
      + ...

  odr = S1->R1 + S2->R2 + S3->R3 + S4->R4
      + S1->R2 + R2->S3 + S3->R4 + R1->S2
      + S2->R3 + R3->S4
}
```

`trs` represents that there exists a transition between the states and `sub` represents that the latter state is a substate of the former. `odr` represents that there exists an order relation between the labels.

IV. Conclusion

In this paper, we proposed the method for verifying consistency of UML diagrams using Alloy. We provided representations of state machine diagrams and a communication diagram using the Alloy language. We also showed that our method could detect inconsistency of state machine diagrams and a communication diagram.

```
Executing "Run stable for 25"
  Solver=sat4j Bitwidth=4 MaxSeq=7 SkolemDepth=1 Symmetry=20
  21399 vars. 1693 primary vars. 72073 clauses. 1031ms.
  Instance found. Predicate is consistent. 1844ms.
```

(a) Simulation execution of Fig3(a),(b) and Fig4(a)

```
Executing "Check inconsistent for 25"
  Solver=sat4j Bitwidth=4 MaxSeq=7 SkolemDepth=1 Symmetry=20
  21399 vars. 1693 primary vars. 72079 clauses. 1094ms.
  Counterexample found. Assertion is invalid. 4156ms.
```

(b) Checking execution of Fig3(a),(b) and Fig4(b)

```
Executing "Check inconsistent for 25"
  Solver=sat4j Bitwidth=4 MaxSeq=7 SkolemDepth=1 Symmetry=20
  21399 vars. 1693 primary vars. 72079 clauses. 1141ms.
  No counterexample found. Assertion may be valid. 4188ms.
```

(c) Checking execution of Fig3(a),(b) and Fig4(b)

Fig. 5. Verification results

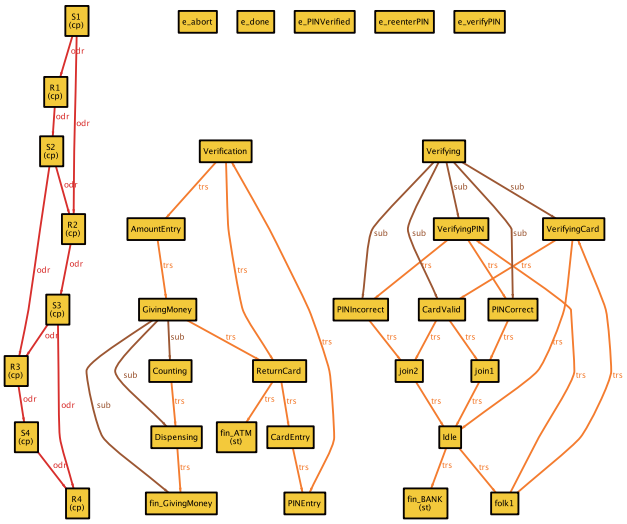


Fig. 6. A counterexample generated by Alloy

A future work is to develop the method for correcting erroneous diagrams using counterexamples. In addition, it is important to generate helpful counterexamples to detect errors of diagrams efficiently.

References

[1] O. M. Group, *Unified Modeling Language*. Object Management Group, 2001, http://www.uml.org.
[2] K. McMillan, *Symbolic Model Checking*. Kluwer Academic, 1993.
[3] S. Harada, T. Yokogawa, H. Miyazaki, Y. Sato, and M. Hayase, "A tool support for verifying consistency between UML diagrams by SMV," in *ITC-CSCC*, 2009, pp. 897–900.
[4] *Alloy*, http://alloy.mit.edu/.
[5] D. Jackson, *Software Abstractions: Logic, Language,and Analysis*. The MIT Press, 2006.
[6] A. J. H. Simons and C. A. F. y Fernández, "Using alloy to model-check visual design notations," in *ENC*. IEEE Computer Society, 2005, pp. 121–128.
[7] A. Zito and J. Dingel, "Modeling UML2 package merge with alloy," in *1st Alloy Workshop (Alloy '06)*, ser. Lecture Notes in Computer Science, O. Nierstrasz, J. Whittle, D. Harel, and G. Reggio, Eds., vol. 4199. Springer, 2006, pp. 86–95.
[8] T. Schäfer, A. Knapp, and S. Merz, "Model checking UML state machines and collaborations," *Electronic Notes in Theoretical Computer Science*, vol. 55, no. 3, pp. 357–369, 2001.