Applying Model Checking to Concurrent Object-Oriented Software *

Seung Mo Cho, Doo Hwan Bae, Sung Deok Cha, Young Gon Kim
Department of Computer Science, KAIST
373-1, Kusong-dong, Yusong-gu, Taejon, Korea
{seung,bae,cha,ygkim}@salmosa.kaist.ac.kr

Byung Kyu Yoo, Sang Taek Kim
Multimedia Research Lab
Korea Telecom

Abstract

Model checking is a formal verification technique which checks the consistency between a requirement specification and a behavior model of the system by explorating the state space of the model. We apply model checking to formal verification of concurrent object-oriented systems, using an existing model checker SPIN which has been successful in verifying parallel systems. First, we propose an Actor-based modeling language, called APromela, by extending a modeling language Promela which is a modeling language supported in SPIN. APromela supports not only all the primitives of Promela, but additional primitives needed to model concurrent object-oriented systems, such as class definition, object instantiation, message send, and synchronization. Second, we provide translation rules for mapping APromela's such modeling primitives to Promela's. By giving an example of specification, translation, and verification, we also demonstrate the applicability of our proposed approach, and discuss the limitations and further research issues.

1. Introduction

When building the concurrent object-oriented systems, we must be able to verify or validate their correctness. Although testing is still a major technique used to validate the software, modeling and formal verification has long been studied and gained increasing popularity. The theoretical progress in the verification techniques and performance improvement in hardware have made it a feasible solution.

Model checking[4, 5, 6] is one of such formal verification techniques. The designer of software or hardware system makes a (generally abstract) model of the expected behavior of the system. Then the verification tool, a model checker, verifies whether the model satisfies the desired properties. Errors found can be used as valuable information when debugging.

SPIN[8, 9] is a model checker whose main design goal is to verify and validate asynchronous process protocols. It provides a system modeling language, called *Promela*, a notation for expressing requirements, and a methodology for establishing the consistency between them. It is one of the most studied and used model checker that implements the enumerative state space exploration.

Promela supports only the process abstraction for computation. The processes communicate via asynchronous channels or shared variables. The process abstraction is a powerful tool to model various systems, but is not sufficient to model concurrent object-oriented systems. When we need other abstractions such as *objects*, there are two choices possible : to rebuild the model using only the process abstraction, or to extend the modeling language Promela to support such abstractions.

Our work is to modify Promela and to extend the tool with the purpose of verifying concurrent object-oriented systems. The concurrent object-oriented model of a system can be *directly* specified in this language, not indirectly with processes. We used the Actor model[2, 3] as a basic computation model in designing the extension. We chose the model because of its theoretical elegance and popularity in concurrent object community.

We built a number of primitives to allow the designer to model systems in object-oriented manner with Actor-based concurrency model. The model can be constructed in several ways. During the development of a system, the modeling can be based on the requirement analysis, or the behavior design. After the development is completed, the model may be extracted from the resulting source code of the system. The model, written in APromela (Actor-based Promela), can be automatically translated to a Promela specification, and simulated and verified using SPIN.

2. Research Background

2.1. Actor Model

Actors are autonomous components of a system which operate asynchronously. They encapsulate data, procedures to manipulate the data, and a reactive process which triggers local procedures in response to messages received. Because actors are conceptually concurrent and distributed, the simplest form of message passing between them is asynchronous. In response to a message, an actor may *create* new actors, *send* messages to actors, and *change its state* with which it responds to the next message.

2.2. Promela and SPIN

Promela is a language used to model communication protocols, and other kinds of distributed systems, at an abstract level. A program in Promela consists of processes that communicate either asynchronously over FIFO channels or by binary rendezvous between processes.

Associated with the language is the tool called SPIN. It is possible to perform an exhaustive simulation by generating the complete state space of the system. Properties like deadlock and livelock can be detected automatically by checking the generated state-space. Even more interesting is that other more specific properties can be expressed in linear time temporal logic, and be verified by exhaustive simulation.

*This research was sponsored in part by the Multimedia Research Lab., Korea Telecom.

3. APromela Language

APromela(Actor-based Promela), an extension of Promela, is a specification language based on Actor model. APromela allows a designer to create directly a validation model of the concurrent object-oriented system.

The basic syntax of Promela can be found in [8]. We extend it with the primitives for actor creation and communication (Figure 1).

The syntax of the following parts are equal to the original one : lexical conventions, comments, identifiers, constants.

```
program ::= { actor-def } * MAIN sequence

actor-def ::= ACTORTYPE NAME
              '{' [ acq_vars ] [ init ] { method } * '}'

acq_vars ::= VAR decl_lst

decl_lst ::= one_decl { ';' one_decl } *

one_decl ::= [ TYPE ivar { ',' ivar } * ]

ivar ::= var_dcl | var_dcl ASGN expr

var_dcl ::= NAME [ '[' CONST ']' ]

var_ref ::= NAME [ '[' expr ']' ]

init ::= INIT '(' [ decl_lst ] ')' sequence

method :: = METHOD NAME '(' [ decl_lst ] ')'
            [ RESTRAIN syn_cond ]  sequence

syn_cond :: = '(' syn_cond ')'
       | syn_cond binop syn_cond
       | unop syn_cond
       | var_ref
       | CONST

sequence ::= step { ';' step } *

step ::= [ decl_lst ] stmnt

stmnt ::= var_ref ASGN expr
        | PRINT '(' STRING { ',' expr } * ')'
        | ASSERT expr
        | GOTO NAME
        | NAME ':' stmnt
        | IF options FI
        | DO options OD
        | BREAK
        | var_ref '<-' NAME '(' [ arg_lst] ')'

options ::= { SEP sequence } +

binop ::= '+' | '-' | ...

unop ::= '~' | '-' | SND

expr ::= '(' expr ')'
       | expr binop expr
       | unop expr
       | CREATE NAME '(' [ arg_lst] ')'
       | var_ref
       | CONST
       | var_ref '.' var_ref
       | var_ref ':' NAME

arg_lst ::= expr { ',' expr } *
```

Figure 1. Syntax of the APromela Language

3.1. Overview of APromela

An APromela specification consists of a main program and a number of actor type definitions. The main program creates some initial actors and initiates the execution. The actor type definition corresponds to the class definition in ordinary object-oriented languages.

Actor type definition

An actor type definition is composed of *acquaintance* (or instance) variables and a set of method definitions. The values assigned to the acquaintance variables comprise an actor's state. A definition may have an optional `init` method. An `init` method is hidden from outside and executed exactly once when an actor is created.

Actor creation

Actors are created and initialized dynamically using the `create` primitive operator. The operator takes an `actortype` name and a set of arguments to the `init` method. It allocates a unique mail address to the newly created actor and returns this address.

Message send

The asynchronous message send, which is the basic communication mechanism in Actor model, is represented as `<-` operator. The method name accompanies the parameters. The mail address of the current actor is referenced as `this` in a method body.

Local Synchronization Constraints

The method bodies may contain a `restrain` part. It indicates when the method can be invoked. If the condition is not true, the message execution is deferred and the next message is processed first.

3.2. Example

We present an example specification written in APromela language. It is a classical producer-consumer example. The system consists of three actors : a producer, a consumer, and a bounded buffer. The synchronization is done by the local synchronization constraints of the buffer.

```
#define SIZE 3

actortype buffer {
   var
      bit data[SIZE]; byte count = 0; byte front = 0

   method put ( bit a ) restrain (count < SIZE)
      data[(front + count) % SIZE] = a; count ++

   method get ( actor ret ) restrain (count > 0)
      ret <- retrieved (data[front]);
      front = (front + 1) % SIZE; count --;
}

actortype producer  {
   var
      bit next; actor buf

   init ( actor a )
      buf = a; this <- putting ()

   method putting ()
      buf <- put (next); next = ! next; this <- putting ()
```

```
}

actortype consumer  {
   var
      actor buf;

   init ( actor a )
      buf = a; this <- getting ()

   method getting ()
      buf <- get (this)

   method retrieved ( bit b )
      printf("read num = %d\n", b); this <- getting ()
}

main  {
   actor buf;
   actor proc;
   actor cons;

   buf = create buffer ();
   proc = create producer (buf);
   cons = create consumer (buf);
}
```

4. Translation from APromela to Promela

Instead of designing a simulator/verifier from scratch we propose to perform a mapping of APromela constructs into corresponding ones in Promela. The basic idea of translation is to use processes and channels in Promela for actors and mail queues in APromela, respectively.

4.1. Mail queue

A natural idea for implementing the mail queue is to use a channel for each mail queue of an actor. The channel must transfer all kinds of the messages to the actor to which the channel belongs. Thus the type of a channel should be a union of the parameters of the methods of an actor.

For the actor definition template in Figure 2, the channel definition should be as follows.

```
chan  this = [ LENGTH ] of { byte, method1_parameters,
             method2_parameters, ... }
```

in which `LENGTH` is the size of channel which is a user-defined constant. The first field of type `byte` holds the method name.

```
actortype actor_name {
   var
      var_declarations
   init ( init_parameters )
      init_method_body
   method method1_name ( method1_parameters )
      method1_body
   method method2_name ( method2_parameters )
      method2_body
   :
   :
}
```

Figure 2. Template for actor type definition

Once we determined the representation of the mail queues of actors, the message send is easily implemented using send-statements to the channels. The fields of the message that are irrelevant of the method are filled with dummy values.

4.2. Actor Definition

An actor in APromela can be represented as a process in Promela. The process consists of a set of methods that the actor services. Considering the semantics of methods invocation in Actor model, we can make a translation algorithm of incorporating all the methods in `do ... od` construct with appropriate conditions. It translates the `actortype` definition template in Figure 2 into the process definition of Promela as shown in Figure 3.

```
proctype actor_name ( chan temp_chan, init_parameters ) {
   var_declarations
   parameter_declarations /* all parameters for methods */
   initializing
   do
      :: atomic {
         this ? method1_name, parameters
         -> method1_body
         }
      :: atomic {
         this ? method2_name, parameters
         -> method2_body
         }
      :
      :
   do
}
```

Figure 3. Process definition translated from Figure 2

4.3. Creating Actors

Creating an actor is to instantiate a process representing the actor, and then to setup the mail queue (channel). It requires subtle interaction between the creating actor and the created one.

Actor creation expression statement takes the form of Figure 4.

```
actor a;
:
:
a = create actor_name ( init_parameters ) ;
```

Figure 4. The general form of actor creation

Type `actor` is implemented with type `chan`. When creating an actor, one must instantiate two objects : a process and a channel. The only object dynamically instantiatable in Promela is the process. So the newly created channel should be defined in the *initializing* part of the actor definition. It is then returned back to the creating actor for further communication with the created actor. Therefore the creating actor waits for the newly created actor to return the mail address of its own through a temporary channel, `temp_chan`.

```
chan a;
chan temp_chan = [0] of { chan } ;
:
atomic {
   run actor_name ( temp_chan , init_parameters ) ;
   temp_chan ? a ;
}
```

Figure 5. Process initiation translated from Figure. 4

Using this scheme, the above APromela template for actor creation can be translated into the corresponding Promela construct in Figure 5.

And the *initializing* part in Figure 3 becomes as in Figure 6.

```
chan  this = [ LENGTH ] of { byte, method1_parameters,
             method2_parameters, ... }
init_method_body
temp_chan ! this;
```

Figure 6. Code for initializing part

4.4. Local Synchronization Constraints

The local synchronization constraints can be easily implemented by wrapping the method bodies with `if` construct. When a method is invoked, the `if` condition is evaluated using the current state and the passed parameters. If the condition is false, the message is put back into the channel, and processed later.

```
:: atomic  { this ? method_name, parameters ->
   if
    :: condition -> method_body
    :: ! condition -> this ! method_name, parameters;
   fi }
```

Figure 7. Promela code for local synchronization

4.5. Example

Using the rules, we can translate an APromela specification into a Promela specification. We translated the example given in the Section 3.2 and verified it with SPIN. SPIN was able to verify the deadlock freedom of this model with a few seconds of computation when the value of `LENGTH` is set to 3.

5. Conclusion

We designed a new modeling language which is an extension of Promela based on Actor model. This can be used as a modeling tool for concurrent object-oriented systems. The verification of the specification written in the language is possible using the model checker SPIN.

The feasibility of applying model checking to software is of course controversial. However, we want to emphasize that the verified objects are always the *model*, not the *reality*, whatever the target system is. Our contribution is to help the designer build the model in an object-oriented fashion. The feasibility depends on the abstraction which the designer uses in modeling. Moreover, the complexity inherent in software can be managed more easily when the system is modeled in an object-oriented way. Some recent researches also show the applicability of model checking to real software[1, 7].

Besides modeling by hand, APromela can also be an intermediate language for model checking of *source codes*. There are various concurrent object-oriented programming languages based on Actor model. The concurrency semantics of the languages are very similar. Therefore, verifying the concurrency features of the programs written in these languages may be performed by designing a preprocessor which extracts the concurrency features from the programs and translates them to APromela programs. The resulting programs can be used as an input to the translator we suggested, and then undergo the model checking.

References

[1] R. J. Anderson, P. Beame, S. Burns, W. Chan, F. Modugno, D. Notkin, and J. D. Reese, "Model Checking Large Software Specifications," in *Proc. the Forth ACM SIGSOFT Symposium on the Foundation of SE*, p. 156-166, 1996.

[2] G. Agha, S. Frolund, W. Kim, R. Panwar, A. Patterson, and D. Sturman, "Abstraction and Modularity Mechanisms for Concurrent Computing," in *IEEE Parallel and Distributed Technology: Systems and Applications,* 1(2):3-14, May 1993.

[3] G. Agha, "Actors: A Model of Concurrent Computation in Distributed Systems, " MIT Press, 1986.

[4] R. Alur, T.H. Henzinger, and P.H. Ho, "Automatic Symbolic Verification of Embedded Systems," *IEEE Transactions of Software Engineering,* 22(3):181-201, 1996.

[5] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang, "Symbolic model checking : 10^{20} states and beyond," *Information and Computation,* 98(2):142-171, 1992.

[6] E.M. Clarke, E.A. Emerson, and A.P. Sisla, "Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications," *ACM Transactions on Programming Languages and Systems,* 8(2):244-263, 1986.

[7] G. Duval, "Specification and Verification of an Object Request Broker," in *Proc. ICSE 98*, 1998.

[8] G. Holzmann, "Design and Validation of Computer Protocols," New Jersey, 1991, Prentice Hall.

[9] G. Holzmann, "The Model Checker SPIN," *IEEE Transactions on Software Engineering*, Vol. 23, No. 5, pp. 279-295, May 1997.