Model Checking Object-Z Specification Using SPIN

Lei Xiao, Gongzheng Lu, Shunzhi Zhu

1 School of Computer and Information Engineering of Xiamen University of Technology, Xiamen, 361024, China
2 Department of Computer Engineering, Suzhou Vocational University, Suzhou, 215000, China
Tel.: 15960810542, fax: 0086-0592-6291390
E-mail: lxiao@xmut.edu.cn

Received: 2 May 2013 / Accepted: 19 July 2013 / Published: 31 July 2013

Abstract: Nowadays, model checking is recognized as an efficient technology for verifying system properties. There are many tools such as SPIN and NuSMV support model checking that will greatly enhance software systems applicability. However, it is difficult to establish a tool to directly verify Object-Z, because of its high-level abstraction. This paper puts the Object-Z specification into label transition system (LTS), and then translates the LTS into the input language Promela of Spin. Subsequently, the history invariant in Object-Z is described by Linear Temporal Logic (LTL). So the correctness of Object-Z specification can be verified.

Keywords: Model checking, Object-Z, LTS, SPIN, LTL.

1. Introduction

Model checking [1] is recognized as an efficient technology for verifying system properties. There are many tools such as SPIN supports it which will greatly enhance its applicability. Model checking tool searches the system state space finitely and checks whether exists some states against the specified property. If there is no such state, then it indicates that the system meet the property. If there is such state, the model checking tool gives a counterexample.

Model checking tool has been widely used in the verification of hardware, protocol, concurrent and real-time software. While the Object-Oriented technology is so mature, the model checking methods for Object-Oriented system had been concerned rarely. Let alone developed model checking tools specifically for them.

Object-Z [2] is the Object-Oriented extension of formal specification language Z [3]. It is suitable for describing the property of state-based system. However, lack of automatic verification tool support for Object-Z is the greatest obstacle. If there are tools can be used to check the mistakes or verify the property of Object-Z specification, the potentiality of Object-Z might work out greatly.

This paper proposes a method to model checking Object-Z specification. Firstly, the Object-Z specification is put into label transition system (LTS), and then the LTS is translated into the input language Promela of Spin. Subsequently, the history invariant in Object-Z is described by Linear Temporal Logic (LTL). So the correctness of Object-Z specification can be verified.

We choose LTS for many reasons. First, Object-Z is a high-level abstraction language, there are no model checking tools can be directly applied
for it. So they must be converted to an intermediate representation to use the existing tools. LTS exactly is the intermediate representation can be used to many model checking tools.

Second, Object-Z is the state-based specification language, its specification mechanism is very similar to LTS so that it will make the transition between them relatively simple and direct and maintain the semantics of Object-Z well.

In recent years, model checking Object-Z has been some concerned. [7] and [8] integrate Object-Z with CSP, ASM separately; their work can only be used for special tools, FDR and SMV. [9] translates the Object-Z specification into SMV program, without considering the semantics-preserving in the conversion process. [10] converts the Object-Z into translation system, though it keeps the semantics of Object-Z and can also be used for many model checking tools, but it didn’t consider the combination of operation. [11] proposes a model checking tool PAT, which contains the functionality of model checking TCOZ, the main idea of the tool is similar with [7], the difference is that the semantics of process is defined by LTS, then model checking TCOZ using existing tools.

The rest of this paper is organized as follows. Section 2 introduces Object-Z and LTS. Section 3 gives the translation rules from Object-Z to LTS and from LTS to Promela. Section 4 is the example. Last is the conclusion.

2. Basic Concepts

2.1. Object-Z

Object-Z is the Object-Oriented extension of formal specification language Z. Its main structure is class, which encapsulates state schema and operation schema.

The class of Object-Z is represented by a box with name. The name is the class name. A class can include the declaration of local variables and constants, at most one state schema and initial schema, and zero or more operation schema. The declaration in state schema is called state variables, and the predicates are called invariant of the class. The invariant constrains the possible value of state variables. Initial state schema assigns the initial value to the state variables. An operation is either an operation schema or a schema expression including existing operation and schema operator. Operation is constrained by precondition and post condition.

\[
\text{State}=\text{init} \mid \text{ready} \mid \text{wait} \mid \text{replySuccess} \mid \text{replyFailure} \mid q&a
\]

![Diagram of Booking System](image-url)
The figure above is the Object-Z specification of Booking System class. State is a state variable assigns the possible state of the Booking System class. Ticketnum is a constant. It represents the current number of tickets. INIT defines the initial schema, which initializes the state variable State and the constant ticketnum. The operation Query and Book express that the clients can query and book tickets through travel agency. The operation RequestTicket indicates that travel agency can buy tickets directly from airline. If the ticketnum is greater than 0, then the booking from clients is successful otherwise fail.

The property specified by the history invariant related to Object-Z class can be described by temporal logic formula. The history invariant indicates the history collection of state of class. Here, the history invariant is expressed by linear temporal logic. The history invariant in the figure above states that if the ticketnum is greater than 0, then the booking from clients is successful otherwise fail.

2.2. Label Transition System (LTS)

We use Label Transition System (LTS) to specify the semantics of the class of Object-Z. The semantics of the class of Object-Z is the set of computations that the object of the class can go through. LTS is the compact representation of computations, so it is the suitable mechanism to represent the dynamic semantics of the class of Object-Z. Moreover, there have been many model checkers for verifying the properties of the model specified by LTS. Its definition is given as follows:

**Definition 1** LTS is a quadruple \((S, s_0, L, T)\), where
- **S** is the set of states
- **s_0** is the initial state
- **L=\{EVE, ACT, GUD\}** is the set of labels, where **EVE, ACT, GUD** denote the set of events, actions and guard conditions respectively
- **T⊆S×L×S** is the transition relation

If there is \((s, (e, a, g), s') \in T\), then it can be denoted as \(s \xrightarrow{(e, a, g)} s'\), which is said that when the event is triggered and the guard condition is true then state \(s\) is changed to state \(s'\) and action \(a\) is generated. Such transition is called basic transition. We introduce composite transition in order to ensure the atomicity of the operation in Object-Z. If two transitions \(t_1, t_2\) \(\in T\) and \(\{ source(t_1) \cap source(t_2) \neq \emptyset \}\), then \(t_1\) and \(t_2\) are called composite transition, where \(source(t_j)\) is the source states of \(t_j\). \(t_1\) and \(t_2\) are said coordinated iff \(\{read(t_1) \cap write(t_2) = \emptyset\} \land \{write(t_1) \cap read(t_2) = \emptyset\}\)
where \(read(t_j)\) and \(write(t_j)\) express the data \(t_j\) read from and write to the environment.

2.3. SPIN

Spin is a well-known model checking tool which used to analysis the complicated system to verify their logical consistency and is a generic verification system that supports the design and verification of asynchronous process system [12-14]. SPIN verification models are focused on proving the correctness of process interactions, and they attempt to abstract as much as possible from internal sequential computations. Process interaction can be specified in SPIN with rendezvous primitives, with asynchronous message passing through buffered channels, through access to shared variables, or with any combination of these. In focusing on asynchronous control in software systems, SPIN distinguishes itself from other well-known approaches to model checking, e.g., [15-20].

Nowadays, SPIN is widely used to model checking of all kinds of software systems and verifying communication protocols mainly because it is designed for checking distributed asynchronous systems, a characteristic suitable for a wide range of software systems. The paper [21] proposes a partition-based model checking method, which can be employed to address the state explosion problem in some procedures of verifying complex communication protocols with SPIN. Designing model of a communication protocol into different sub-models by a message-based way, and verifies the design model through validating the sub-models with relatively low resource consumption. The paper [22] compared the two model checking tools: NuSMV and SPIN, and used model checker SPIN on commercial Flight Guidance Systems. The result is that SPIN get a better understanding of the capabilities of explicit versus symbolic model checker in the same domain and can be more flexible than NuSMV in handling large scale systems.

3. Model Checking Object-Z Specification

The specification of Object-Z can be viewed as an abstract state machine. An object of the class produces state sequences through operation. LTS is used to define the compact representation of these state sequences. The definition of Object-Z class includes the following elements:

<table>
<thead>
<tr>
<th>Class name</th>
<th>constant definitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>state schema</td>
<td>operations</td>
</tr>
<tr>
<td>init state schema</td>
<td>history invariant</td>
</tr>
</tbody>
</table>

3.1. From Object-Z Specification to LTS

3.1.1. Translation Rules

**Constant definition:** Constants are defined in the beginning of the class and their values remain
unchanged, thus they are not considered in the definition of dynamic semantics of Object-Z.

**State schema:** Local variables and state variables are defined in a state schema, and the values of them at some time can be mapped to a state in LTS. Let var and state be the local variables and state variables defined in state schema respectively. Given function \( \Pi: \text{var} \times \text{state} \rightarrow S \), it maps the different values of the pairs of (var, state) to the set of states in LTS.

**Initialization state schema:** Initialization state schema assigns initial values to variables defined in state schema. The initial values of these variables are mapped to the initial state in LTS directly.

**State transition:** State transition can be achieved by changing the values of the variables defined in the \( \Delta \)-list of the operation. For the pair consisting of the variables defined in \( \Delta \)-list, a state can be translated into a new state if one of the variables in the pair changes its value or two variables both change their value. For instance, the value of var is changed to var' and the value of state remains unchanged in

\[(\text{var}, \text{state}) \rightarrow (\text{var}', \text{state}) \in T.\]

**Operation:** An operation is either an operation schema or an operation expression including existing operations and schema operators. Operation is constrained by pre-conditions and post-conditions, and describes the state changes of the object. For the operation schema, it is mapped to an event in LTS, its pre-conditions and post-conditions to the guard condition and action. Given function \( F: \text{Op} \rightarrow \text{Seq} \), it maps the set of operations Op to the sequence of events in LTS. Function \( P: \text{Pre}(\text{Op}) \rightarrow \text{GUD} \) maps the pre-conditions of the operation to the guard condition, and function \( Q: \text{Post}(\text{Op}) \rightarrow \text{ACT} \) maps the post-conditions of Op to an action.

For the operation expressions including schema operators, we discuss them in the following four cases [1]:

1. **Conjunction operation:** \( \text{Op} \triangleq \text{Op1} \land \text{Op2} \), both Op1 and Op2 are executed.
2. **(Non-deterministic) Choice operation:** \( \text{Op} \triangleq \text{Op1} \mid \text{Op2} \), one of Op1 and Op2 is chosen and executed randomly.
3. **Parallel operation:** \( \text{Op} \triangleq \text{Op1} \parallel \text{Op2} \), both Op1 and Op2 are executed with bi-directional communication.
4. **Sequential operation:** \( \text{Op} \triangleq \text{Op1} \mathbin{|} \text{Op2} \), Op1 and Op2 are executed sequentially, and the output variable of Op1 must match with the input variable of Op2.

Let \( LE \in T \) be the set of enable transitions in LTS, and \( t_1 \) has higher priority than \( t_2 \) denoted by \( t_2 \prec t_1 \). Each operation in the composite operation is translated into a transition in LTS directly.

**Conjunction operation:** Two operations can be a conjunction if there is no common input and output variables between them, that is, they are coordinated. The LTS semantics of conjunction operation is as follows:

\[
\begin{array}{l}
\forall i \in \{1 \ldots k\}, t_i \in T, t'_i < t_i, a_i \\
\{ e_1 \bullet \text{revar}(a_i) \bullet \ldots \bullet \text{revar}(a_{i-1}) \bullet e_k \land g_k \} \\
a_1 \bullet o_2[RS] \bullet \ldots \bullet a_j[RS_j] \\
\end{array}
\]

Where \( \text{revar}(a_i) \) denotes that rename the prime variable occurring in \( \text{Opi} \) which has the same name with the nonprime variable occurring in \( \text{Opi}+1 \). If there is no such variable, then \( \text{revar}(a_i) \) is null. \( \text{after}(a_i)/\text{before}(a_i+1) \) represents that the renamed variable substitute for the nonprime variable in the post-conditions of \( \text{Opi}+1 \) after \( \text{Opi} \) and before \( \text{Opi}+1 \). If \( \text{revar}(a_i) \) is null, then \( \text{after}(a_i)/\text{before}(a_i+1) \) is null.

If the pre-conditions of the operation \( \text{Opi} \) are satisfied, then the state will be changed and the action corresponding to the post-conditions of \( \text{Opi} \) will be generated. For the conjunction operation, if the conjunction of the pre-conditions of each operation is true, then each operation will execute sequentially from the start state of the first operation to the final state of the last one, and generate the sequence of actions. In the semantics, \( t_i \) is the transition corresponding to the \( \text{Opi} \). And \( e_i, g_i \) and \( a_i \) are the event, guard condition and action labeled on \( t_i \).

**Choice operation:** The LTS semantics of choice operation is as follows:

\[
\begin{array}{l}
\forall i \in \{1 \ldots j\}, t_i \in T, t'_i < t_i, s_i \\
\{ e_i, g_i \} \\
\{ e_j, g_j \} \\
\{ e_k, g_k \} \\
\end{array}
\]

where \( k \in \{i, j\} \). For choice operation, if the pre-conditions of one of the operation are satisfied, then the transition corresponding to the operation will occur.

**Parallel operation:** It is similar to the conjunction operation, the differences between them are that the communication of parallel operation is done by input and output variables.
which are required consistent, and the source state of the operations in the parallel operation must be the same. Let the function input:Op→IV be used to get the input variables of operation Op, and output:Op→OV be used to get the output variables of operation Op. Function constraint: (Op→OV)→CON is used to get the constraints on the output variables of Op. The input and output variables of the operations Opi and Opj are consistent iff (output(Opi) ⊆ input(Opj)) ∧ (output(Opj) ⊆ input(Opi)). If the output variables of Opi is consistent with the input variables of Opj or the output variables of Opj is consistent with the input variables of Opi at the same time, then we assume that Opi has higher priority than Opj. The constraints on the output variable of Opi are combined with the guard condition in the transition which corresponds to Opj. The LTS semantics of parallel operation is as follows:

\[ C : \text{constraint(output}(t_i)) \quad S : \text{output}(t_i)/\text{input}(t_j) \]

Where output(t_i)/input(t_j) is expressed that the output variables of Opi substitute for the input variables of Opj.

**Sequential operation:** Sequential operation is the sequential combination of two operations, which require that the source state of the second operation is equal to the target state of the first one.

Operation Op1 and Op2 can be combined sequentially iff (target(Op1) = source(Op2)) ∧ (output(Op1) = input(Op2)), where target(Op1) represents the target state of Op1. The LTS semantics of sequential operation is as follows:

\[ \]
3.2. From LTS to Promela

**State.** The states in LTS can be declared by `mtype{}` in Promela, the pair `(var,state)` can be defined as a new state variable.

**Input variables and output variables.** The input variables and output variables including in the guard condition and action of the translation are defined by channel `chan`. For the input variables and output variables in the composite translation, they can be defined as synchronous channel.

**Translation.** The guard condition and action of the translation can be defined as the precondition and postcondition in Promela separately. The precondition can be the Boolean expression between `if...fi`, the atomic is used to represent the atomicity of the translation. We take the translation from state ready to wait as an example to explain.

```
state==ready->
if
  :: c?=B->atomic{a!=RT;state=wait;}
fi
```

The entire LTS can be declared by active proctype `processname{}`, where `processname` represents the process name.

![Fig. 1. LTS representation of Booking system Class.](image1)

```
Fig. 1. LTS representation of Booking system Class.
```

![Fig. 2. Another LTS representation of Booking system Class.](image2)

```
Fig. 2. Another LTS representation of Booking system Class.
```

4. Verification Result of Class Booking System

So far, we can verify the property 
\[ \square (client?book \land ticketnum>0) \Rightarrow client!success \]
\[ \lor \square (client?book \land \neg (ticketnum>0)) \Rightarrow client!failure \]
by SPIN. By the verification, we can know that the property is satisfied by the Object-Z specification of Booking System class. Fig. 3 is shows the result of verification:
5. Conclusion

This paper proposes a method used to verify the Object-Z specification in detail. We first translate the Object-Z specification into LTS which is an intermediate representation can be used by many model checking tools. We give the translation rules, and the semantics preserving is concerned in the translation procedure. In order to verify the correctness of Object-Z specification, the translation procedure from LTS to Promela is given. Finally the verification of Booking System class is used to show the feasibility of our work.

Although the translation rules have been provided, the relevant translation algorithm needs to be implemented in future. The history invariant is expressed by linear temporal logic, how to verify the property if it is represented by other forms, it is also a problem we consider future.

Acknowledgements

This work is supported by Natural Science Foundation of China (61070151) and supported by the colleges and universities innovation project of Xiamen science and technology plan (3502Z20123037).

References


