Softw Syst Model (2011) 10:515–536
DOI 10.1007/s10270-010-0160-1

REGULAR PAPER

An executable object-oriented semantics and its application to firewall verification

Kenro Yatake · Takuya Katayama

Received: 31 July 2008 / Revised: 4 March 2010 / Accepted: 5 March 2010 / Published online: 1 April 2010

Abstract This paper presents a formal executable semantics of object-oriented models. We made it possible to conduct both simulation and theorem proving on the semantics by implementing it within the expressive intersection of the functional programming language ML and the theorem prover HOL. In this paper, we present the definition and implementation of the semantics. We also present a prototype verification tool ObjectLogic which supports simulation and theorem proving on the semantics. As a case study, we show the verification of a practical firewall system.

Keywords Object-Oriented · Theorem proving · Simulation · HOL · ML

1 Introduction

As our society has become more dependent on information systems, it has become more important to ensure the correctness and the validity of those systems. Especially, there is a growing need for the verification on the analysis level of the development. This is because the errors found in the analysis stage are cheaper to correct than those found in the implementation stage. Verification on the analysis level allows early detection of bugs and, as a result, reduces the total cost of development.

Communicated by Dr. María Victoria Cengarle.

K. Yatake (✉) · T. Katayama
Japan Advanced Institute of Science and Technology,
1-1 Asahidai, Nomi, Ishikawa 923-1292, Japan
e-mail: k-yatake@jaist.ac.jp

T. Katayama
e-mail: katayama@jaist.ac.jp

Among many verification methods, we focus on theorem proving. The prominent feature of theorem proving is the induction by which we can prove the correctness of system behavior exhaustively for arbitrary inputs. In order to apply theorem proving to the analysis models such as unified modeling language (UML) [17], we need to implement a formal semantics of object-oriented (OO) models in theorem provers.

We consider that the semantics should be executable. This is because the executable semantics allows us to conduct not only theorem proving but also simulation. The advantage of simulation is that it allows us to identify the result of system execution at a glance. This is especially useful for finding trivial bugs in the early stage of model construction. By conducting simulation in advance of the thorough verification by theorem proving, we can reduce the total cost of verification. In fact, the combination of theorem proving and simulation has been successfully used in ACL2 [10] for the verification of both hardware and software [2,13,18,26].

In this paper, we present an executable semantics of OO models. In order to make it efficiently executable, we implemented the semantics not only in the theorem prover HOL [22], but also in the functional programming language ML [14]. To keep the consistency between the two semantics, we implemented it within the intersection of the expressive powers of HOL and ML. It is known that HOL and ML have similar type systems and there exists an intersection between them. Specifically, they have recursive datatypes and recursive functions (primitive recursion and well-founded recursion) in common [4,20]. By implementing the semantics within this intersection, we made it possible to conduct both simulation and theorem proving in the same semantics. The internal representation of the semantics is a heap memory structure to store objects. In ML, it is used as a runtime environment for

simulation. In HOL, the axiomatic semantics is derived from its definition.

We also present a prototype verification tool ObjectLogic which supports both simulation and theorem proving based on the semantics. It allows us to define models in a Java-like language and automatically generates ML executables for simulation and HOL proof obligations for theorem proving. As a case study, we show the verification of a practical firewall system and discuss the effectiveness.

This paper is organized as follows. Section 2 introduces the preliminaries of HOL. Section 3 explains the definition and implementation of the semantics. Section 4 introduces ObjectLogic. Section 5 presents the verification of a firewall system. Section 6 discusses the effectiveness of the semantics. Section 7 cites related work. Section 8 gives a conclusion. Appendix A lists the list of theorems in the semantics. Appendix B presents the syntax of the modeling language used in ObjectLogic.

2 HOL preliminaries

The HOL system is a theorem prover of higher-order logic which is implemented in Moscow ML. In this section, we briefly explain the notation of HOL used in this paper.

The major types and logical operations are summarized in Table 1. The type `bool` has the constants `T` and `F` for true and false. Natural numbers and integers are represented by `0`, `1`, `2` (negative integers are represented by `~1` and `~2`). Strings are represented with double quotations like `"abc"`. Pairs are represented by parentheses and commas like `(T,5)` and `(1,2,"xyz")` (abbrev. for `(1,(2,"xyz"))`). They are deconstructed by the functions `FST:'a#'b->'a` and `SND:'a#'b->'b`. They return the first and second element in a pair, respectively. The types `'a` and `'b` represent type variables. Lists are constructed by the infix operator `_::_` from the empty list `[]` like `1::2::3::[]` (also represented as `[1;2;3]`). Lists are deconstructed by the functions `HD:'a list->'a` and `TL:'a list->'a list`. The function `HD` returns the first element in the list, while the function `TL` returns the list excluding the first element. As for the proof, an expression of type `bool` can be set to the proof goal. For example, we can set the following expression to the goal:

```
!P Q R. (P \/ Q ==> R) ==> (P ==> R) /\ (Q ==> R)
```

The proof is done interactively using *tactics* which are the commands (ML functions) to simplify the goal. This expression can be proved by the tactic `SIMP_TAC bool_ss []`. The tactic `SIMP_TAC ss L` simplifies the goal with the simplifier set `ss` and the theorems given in the list `L`. This goal is proved only with the simplifier set `bool_ss` (a set of theorems concerned with Boolean) without giving further theorems. HOL has a lot other tactics. If the expression is proved, it becomes a theorem which is denoted with "`|-`" as follows:

```
|- !P Q R. (P \/ Q ==> R) ==> (P ==> R) \/ (Q ==> R)
```

Once a theorem is proved, it can be used for the next proof, e.g., by putting it into the list `L` of `SIMP_TAC`.

Definitions are a kind of theorems. They have the form of equation whose left-hand side is a new constant. For example, the function `MAX`, which returns the bigger value from a pair of natural numbers, is defined by the following theorem:

```
|- !x y. MAX (x,y) = if x >= y then x else y
```

Definitions are directly introduced to HOL without proof. Generally, we should avoid introducing theorems to HOL without proof because it might cause inconsistency. But, definitions are safe because they are always introduced as a form of an equation whose left-hand-side is a new constant.

Table 1 HOL notations

Operators	Meaning	Types	Meaning
`~`	¬	`bool`	Boolean
`/\`	∧	`num`	Natural numbers
`\/`	∨	`int`	Integers
`==>`	⇒	`string`	Strings
`!`	∀	`_#_`	Pairs
`?`	∃	`_ list`	Lists
`\`	λ	`_->_`	Functions

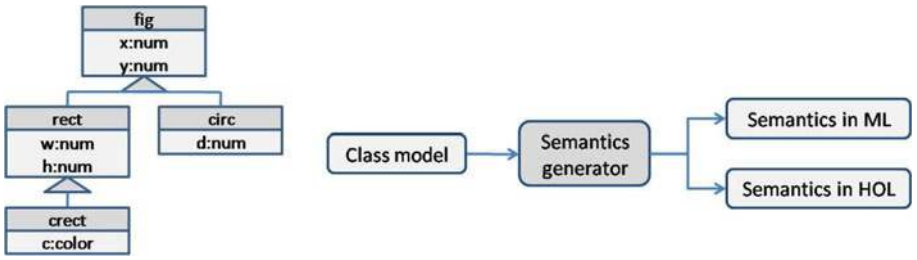
3 The OO semantics

In this section, we present the OO semantics. We first explain the design policy of the semantics. Then, we present the overview of the semantics in HOL and ML with an example. Then, we present the formal definition. Finally, we briefly explain its implementation.

3.1 Design policy

Besides executability, the design policy of the semantics is summarized as follows. First, our semantics is implemented to serve as the semantics of OO analysis models. It supports the concepts such as classes, attributes, single inheritance, object subtyping, and references. So, it can be used as the semantics of, for example, UML class diagrams. The notable feature of our semantics is that it can accommodate arbitrary HOL types (with no type variables) for the types of object attributes. This feature is helpful on the analysis level because

Fig. 1 The semantics generator



we can abstract the model with various types such as list, set and stack, etc. To realize this feature, we made the semantics application-specific. Specifically, we automatically generate the semantics depending on the specific types in the target system. For example, if the target system defines a class with three attributes of type `num`, `bool`, and `string`, we represent the objects by the product type `num#bool#string`. Another option would be to use extensible record types [5, 15], but we kept the implementation simple by using only the product type. The object subtyping and references are realized by putting those products in a heap memory structure.

Second, it is shallowly embedded, i.e., the concepts such as classes, attributes, and inheritance are represented directly by types and constants in HOL. This is because our verification target is each instance of OO models. Shallow embedding facilitates the proof on the instance level compared to deep embedding [16]. It also has the effect of making the semantics simple because all the typing information is directly represented by the type system of HOL, that is, there is no need to additionally include the typing constraints into the semantics.

Finally, it is constructed conservatively by *definitional extension*. It is the standard way of constructing sound theories in HOL where new theories are derived from existing sound theories by only allowing introduction of definition and derivation by sound inference rules. This is in contrast to axiomatical theory construction where axioms are directly introduced in the theory, which often makes the theory inconsistent. We derived the semantics from the definition of the heap memory structure which is constructed by existing theories such as lists and pairs. This guarantees the soundness of the semantics.

3.2 Overview

As we explained, the semantics is defined depending on the specific types in the target system. As shown in Fig. 1, we implemented a *semantics generator* which inputs a *class model* and outputs its semantics both in ML and HOL. The class model defines the static structure of a system such as classes, attributes, and inheritance. All the types are defined in this model. Figure 1 shows an example class model declaring four classes `fig`, `rect`, `crect`, and `circ` which represent figures, rectangles, colored-rectangles, and circles, respectively. It is defined in a text file, but here we show it as a diagram for readability. It is fed to the semantics generator and the semantics is constructed automatically.

3.2.1 The semantics in HOL

In HOL, the semantics is implemented as a *theory* which is a module containing types, constants, operators, and axioms. It is automatically derived by the semantics generator from the underlying heap memory structure of the target system.

The theory elements are introduced corresponding to the class model elements. In the case of the example, the following types are introduced to the theory:

```
store, fig, rect, crect, circ
```

The type `store` plays a central part in the theory. It represents the environment which holds all alive objects in the system and has the constant `emp` representing an empty store. The other types are the types of object references for the four classes. Each of them has a constant representing null reference: `fig_null`, `rect_null`, `crect_null` and `circ_null`, respectively.

Some of the operators introduced in the theory are

```
fig_new : store -> fig # store
fig_ex : fig -> store -> bool
fig_get_x : fig -> store -> num
fig_set_x : fig -> num -> store -> store
rect_cast_fig : rect -> store -> fig
fig_is_rect : fig -> store -> bool
rect_is_rect : rect -> store -> bool
```

The operator `fig_new` is a function to create a new `fig` object in the store. It takes a store as an argument and returns a pair of a newly created object and the store after the creation. The operator `fig_ex` is a predicate to test the existence of a `fig` object. It takes a `fig` object and a store and returns true if the object exists (not null) in the store. The relationship between these operators is specified by the following theorem (derived from two axioms):

```
|- !s. let (f,s') = fig_new s in fig_ex f s'
```

It means "The newly created `fig` object exists in the store after the creation."

The operators `fig_get_x` and `fig_set_x` are functions to read and write the attribute `x` of the class `fig`. The operator `fig_get_x` takes a `fig` object and a store and returns the current value of the attribute `x`. The operator `fig_set_x` takes a `fig` object, a natural number value and a store and returns the store after updating the attribute `x` to the value. The relationship between these operators is specified by the following axiom:

```
|- !f v s. fig_ex f s ==> (fig_get_x f
 (fig_set_x f v s) = v)
```

It means "If the `fig` object exists in the store, the value of the attribute `x` obtained just after updating it to `v` is equal to `v`."

The operator `rect_cast_fig` is a function to cast a `fig` object upward from `rect` to `fig`. It takes a `rect` object and a store and returns a `fig` reference to the object. The operator `fig_is_rect` is an "instance-of" operator to test if a `fig` object is an instance of the class `rect`. It takes a `fig` object and a store and returns true if the `fig` object was created from the class `rect`. Likewise, the operator `rect_is_rect` tests if a `rect` object is an instance of the class `rect`. After an object is created, its "apparent" types can be changed by cast operators, but the instance-of operator remembers the "actual" type of the object. The following axiom illustrates this:

```
|- !r s. rect_is_rect r s ==> fig_is_rect
 (rect_cast_fig r s) s
```

It means "If a `rect` object is an instance of the `rect` class, it is still the instance of the `rect` class even if it is cast to the `fig` class."

3.2.2 The semantics in ML

In ML, the semantics is implemented as a *structure* (module unit). Its *signature* (module interface) provides types, constants, and operators of the same names as those in the semantics in HOL like `fig_new`, `fig_set_x` and `fig_get_x`. These operators, in turn, can be actually executed as ML functions. Figure 2 shows their execution in the ML interpreter. First, we apply the operator `fig_new` to the empty store `emp`. This returns a new `fig` object `<fig>` and a new store `<store>` (The internal representation of the store and the object are hidden by the opaque signature constraint). Then, we apply the operator `fig_set_x` to set the value `10` to the attribute `x`. This returns the new store `s`. Finally, we apply the operator `fig_get_x` to get the attribute `x`. This returns the value `10`. In this way, we can execute the operators in the semantics.

Note that the value `10` has an integer type `int`. In ML, the type `int` is used instead of `num` because ML does not have the natural number type. Of course, we could implement the type `num` in ML. But, we avoided doing so because it makes the notation of natural numbers lengthy like `Suc (Suc (Suc Zero))`.

```
- val (f,s) = fig_new emp;
> val f = <fig> : fig
  val s = <store> : store
- val s = fig_set_x f 10 s;
> val s = <store> : store
- val x = fig_get_x f s;
> val x = 10 : int
```

Fig. 2 Execution of the semantics in ML (–:input, >:output)

3.3 Formal definition

Here, we present the formal definition of the semantics. Note that it is formalized meta-theoretically (not itself implemented in HOL).

3.3.1 Class models

The class model is defined as a six tuple:

$$CM = (C, A, \mathcal{A}, \mathcal{I}, \mathcal{T}, \mathcal{V})$$

The sets C and A are the sets of class names and attribute names which appear in the system, respectively. The mapping $\mathcal{A} : C \to 2^A$ relates a class to a set of the attributes defined in the class. The mapping $\mathcal{I} : C \to 2^C$ relates a class to a set of its direct subclasses. We assume single inheritance. The mapping $\mathcal{T} : C \times A \rightharpoonup Type$ relates an attribute to its type (we represent a partial mapping by $\rightharpoonup$. In this case, $\mathcal{T}(c, a)$ is not defined if $a \notin \mathcal{A}(c)$). The set $Type$ is a set of arbitrary concrete types. We define the type of an object reference as the name of the class it belongs to. So, we assume $C \subset Type$. The mapping $\mathcal{V} : C \times A \rightharpoonup Value$ relates an attribute to its default value. The set $Value$ is a set of values of all types in $Type$. The type of $\mathcal{V}(c, a)$ must be $\mathcal{T}(c, a)$. By the symbol $\lhd$, we denote the super-sub relationship of inheritance (inspired by the triangle symbol of inheritance in UML). The expression $c_1 \lhd c_2$ means c_2 is a direct subclass of c_1, which is equivalent to $c_2 \in \mathcal{I}(c_1)$. In addition, $c_1 \lhd^+ c_2$ means c_2 is a descendant class of c_1 and $c_1 \lhd^* c_2$ means $c_1 = c_2$ or $c_1 \lhd^+ c_2$. By $attr(c)$, we denote the attributes and the inherited attributes of the class c, i.e., $attr(c) = \{a | a \in \mathcal{A}(d), d \lhd^* c\}$.

3.3.2 The OO semantics

The semantics for the class model CM is defined axiomatically as a four tuple:

$$Sem_{CM} = (Ty, Con, Op, Ax)$$

The sets Ty, Con, Op and Ax are the sets of types, constants, operators, and axioms, respectively.

The set Ty contains the type $store$ which represents the type of the system environment. It also contains the types $c \in C$ which represent the object references of the class c.

The set Con contains the constant $Emp : store$ which represent the empty store. It also contains the constants $Null^c : c$ which represent the null reference of the class c.

The set Op contains the following operators:

$$Ex^c : c \to store \to bool \quad (c \in C)$$
$$New^c : store \to c \,\#\, store \quad (c \in C)$$
$$Get_a^c : c \to store \to \mathcal{T}(c,a) \quad (c \in C, a \in attr(c))$$
$$Set_a^c : c \to \mathcal{T}(c,a) \to store \to store \quad (c \in C, a \in attr(c))$$
$$Cast_d^c : c \to store \to d \quad (c, d \in C, c \lhd^+ d \text{ or } d \lhd^+ c)$$
$$Is_d^c : c \to store \to bool \quad (c, d \in C, c \lhd^* d)$$

The predicate Ex^c tests if the object of class c exists in the store. The function New^c creates a new instance of class c in the store. The function Get_a^c and Set_a^c reads and updates the attribute a of the object of class c existent in the store, respectively. The function $Cast_d^c$ casts the object type from c to d. The predicate Is_d^c tests if the object of class c is an instance of the class d.

In HOL and ML, the elements in the semantics $store$, c, Emp, $Null^c$, Ex^c, New^c, Get_a^c, Set_a^c, $Cast_d^c$ and Is_d^c are denoted as `store`, `c`, `emp`, `c_null`, `c_ex`, `c_new`, `c_get_a`, `c_set_a`, `c_cast_d` and `c_is_d`, respectively.

The set Ax contains 41 axioms. They specify the behavior of the operators. They are listed in the Appendix A.

3.4 Implementation

The internal representation of the store is a heap memory structure to store objects. We implemented it both in ML and HOL. In ML, it becomes a runtime environment for simulation. In HOL, the axiomatic semantics is derived from its definition.

Figure 3 shows a snapshot of the heap memory for the example model. It consists of four lists corresponding to the four classes: `fig`, `rect`, `crect` and `circ`. Lists are used to store object attributes and the list indices are used for object references. For example, the indices of the list for `fig` are used to represent the `fig` object references `f0`, `f1`, `f2`, ... of type `fig` (`f0` is used for the null reference). Object instances are fragmented into tuples and stored in each list. For example, the tuple in `f1` represents a `fig` instance whose attributes are `x=2` and `y=3`. Two tuples in `f2` and `r1` together represent a `rect` instance whose attributes are `x=12`, `y=5`, `w=5`, and `h=8`. Three tuples in `f3`, `r2`, and `cr1` together represent a `crect` instance whose attributes are `x=1`, `y=4`, `w=6`, `h=6`, and `c=red`. The fragments are linked to each other by storing their references. For example, the two tuples in `f2` and `r1` which compose a `rect` instance are linked to each other by storing the references `r1` and `f2`, respectively.

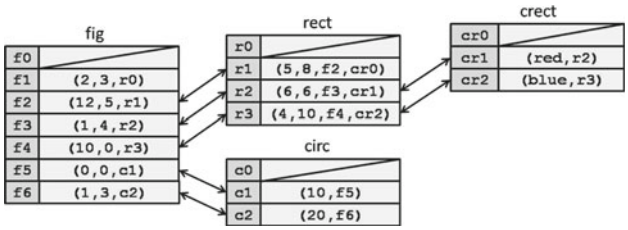


Fig. 3 The heap memory structure

By composing object instances by multiple tuples, we can realize object subtyping. For example, three references `f3`, `r2` and `cr1` all point at the same `crect` instance. This means that the `crect` instance can have three types `fig`, `rect`, and `crect`.

Another option to represent the store is to use a single list to store objects represented by extensible record types. But in this case, the static type checking does not work on references because they are represented by the indices of the single list. In our implementation, we enabled the static type checking by composing the store with multiple lists and giving different types to their indices.

All the operators in the semantics are represented by functions on this heap memory structure. Furthermore, all the axioms are derived from their definitions. The formal implementation is presented in [27].

4 ObjectLogic

To support the verification of OO models on the semantics, we implemented a prototype verification tool called ObjectLogic. It is implemented as a library of HOL (a structure of Moscow ML). It allows us to define models in a Java-like language called ObjectLogic modeling language (OML, see Appendix B) and automatically generates ML code for simulation and HOL code for theorem proving.

Figure 4 shows its architecture. OML is the input language which consists of classes, methods, and assertions (method contracts and class invariants). It is input to the OML parser which extracts the class model and abstract syntax trees for methods and assertions. The class model is input to the semantics generator and the semantics is constructed in ML and HOL. Methods and assertions are input to the code generator. From the methods, it constructs an ML structure and an HOL theory which contains the functions corresponding to the methods (we call them *function structure* and *function theory*, respectively). From the assertions, it constructs an HOL theory which contains the propositions corresponding

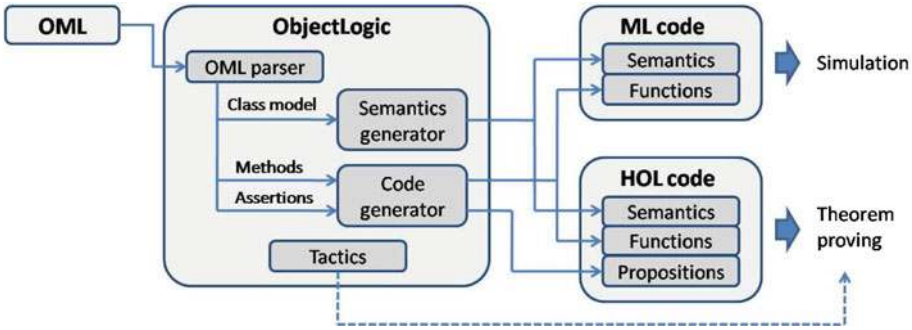


Fig. 4 ObjectLogic

to the assertions (we call this *assertion theory*). We can conduct simulation using the functions in the function structure in ML and theorem proving by proving the propositions in the assertion structure in HOL. ObjectLogic also provides special tactics to support proof on the semantics.

In this section, we first explain OML, then its translation into the semantics, and finally the verification in ML and HOL.

4.1 OML

OML is designed to facilitate the modeling of software applications. It supports classes, attributes, methods, single inheritance. The expressiveness is close to UML sequence diagrams, i.e., it contains conditional branches and finite loops. The finite loop statement `loop` is used like `loop(m(),n)` meaning that the method `m()` is called `n` times. Furthermore, it has iteration commands on object lists. For example, the command `l->select(m())` extracts a list of objects which satisfy the boolean method `m()` from the list `l`, and the command `l->apply(m())` applies method `m()` to each object in the list `l`. These commands are introduced following the tendency that many of the loops in software applications are concerned with manipulating object lists, e.g., "Search a customer which has ID 100 from the list" and "Add interest to all the accounts in the list". The syntax of these commands are inspired by that of object constraint language (OCL) [25]. They allow us to describe the iteration in a single line. Note that the list used here is the same as `'a list` in HOL. So, these commands are guaranteed to terminate.

Another facility of OML is that it can import arbitrary types, constants, and functions from HOL (ML). This facility is convenient because we can abstract the model with various types in the libraries and user-defined types. Furthermore, we can make use of the built-in theorems for those types when performing proofs.

Figures 5 and 6 show the OML code of the canvas system. The first eight lines in Fig. 5 illustrate how to import libraries from ML and HOL, namely the ML structure `color` and the HOL theory `ColorTheory`. Suppose that they contain the

```
importML color
importHOL ColorTheory
type color  = ML[# color #] HOL[# Color #]
const black : color = ML[# Black #] HOL[# BLACK #]
const red   : color = ML[# Red #]   HOL[# RED #]
const blue  : color = ML[# Blue #]  HOL[# BLUE #]
func isBlack(c:color):bool =
  ML[# (fn c => c = Black) #] HOL[# (\c. c = BLACK) #]

class fig {
  private active : bool = false;
  private x,y : num = 0;
  public fig(a:num,b:num):void { x=a; y=b; }
  public isActive():bool { return active; }
  public move(dx:num,dy:num):void { x=x+dx; y=y+dy; }
  public getPos():num#num { return (x,y); }
}
class rect extends fig {
  private w,h : num = 0;
}
class crect extends rect {
  private c : color = black;
  public move(dx:num,dy:num):void {
    super.move(dx,dy); if (isBlack(c)) c=red;
  }
}
```

Fig. 5 Sample OML code

types `color` and `Color`, respectively, which are declared as follows:

```
datatype color = Black | Red | Blue;       (* ML *)
Hol_datatype Color = BLACK | RED | BLUE;   (* HOL *)
```

To use these types and constants in OML, we have to declare new types and constants in OML and bind them to those in ML and HOL. The example of Fig. 5 declares a new type `color` and binds it to the type `color` in ML and the type `Color` in HOL. It also declares a new constant `black` and binds it to the constants `Black` in ML and `BLACK` in HOL. The other constants `red` and `blue` are declared in the same way. Finally, it declares the function `isBlack()` and binds it to the anonymous functions in ML and HOL. Note that the strings inside `ML[#...#]` and `HOL[#...#]` are not checked by the OML parser and directly embedded to the generated code. It is the responsibility of users (who write OML) to ensure the type consistency of the embedded code.

Fig. 6 Sample OML code (continued)

```
main class canvas {
  private figList : fig List = (Nil:fig List);
  private max : num = 0;
  public canvas(x:num):void { max = x; }
  public addFig(x:num,y:num):void {
    var f : fig = (null:fig);
    f = new fig(x,y);
    if (figList->length()<max) figList->add(f);
  } contracting lengthInc {
    pre : figList->length()<max
    post : figList->length()==figList->length()@pre+1
  }
  public moveTop(dx:num,dy:num):void {
    if (figList!=(Nil:fig List)) (figList->hd()).move(dx,dy);
  }
  public getPosList():num list {
    return figList->collect(f|f.getPos());
  }
  public moveActive(dx:num,dy:num):void {
    figList->select(f|f.isActive())->apply(move(dx,dy));
  }
  invariant lengthMax { figList->length()<=max }
}
```

Classes are defined by a syntax similar to Java. The three classes `fig`, `rect` and `crect` in Fig. 5 are from the previous example. The class `canvas` in Fig. 6 represents a canvas on which the figures are displayed. It is the *main* class of this system. The main class is declared uniquely in a system with the keyword label `main`. It plays the role of the interface of the system with the outer environment. Specifically, the API of the system is defined by the new method and the public methods of the main class. The reason of introducing the main class is to define the *states* of the system, i.e., the initial state is constructed by the new method, and the other states are constructed by applying the public methods repeatedly from the initial state. The induction scheme to prove invariants is based on this interpretation of the system states.

The attributes `figList` and `max` are the list to store figures and the max size of the list, respectively. For the method `addFig()`, which adds a new `fig` object to `figList`, the contract `lengthInc` is defined. It means that if the length of `figList` is below `max`, the length is incremented after applying the method. The methods `getPosList()` and `moveActive()` illustrate the use of the iteration commands. The method `getPosList` returns the list of positions of all the figures. This is realized by the command `collect` which applies the method `getPos()` to each of the `fig` objects in `figList` and returns the list of obtained values. The method `moveActive()` moves all the active figures in the list by `dx` and `dy`. This behavior is realized by two commands `select` and `app`. First, the command `select` extracts from `figList` all the `fig` objects which satisfy the method `isActive()`. Then, the command `apply` applies the method `move(dx,dy)` to all the objects in the obtained list. The class `canvas` also defines the invariant `lengthMax` which means that the length of `figList` never exceeds `max`.

4.2 Translating methods

Here, we explain how the methods are translated into the functions in the semantics using the example. We explain the translation into HOL. The translation into ML is done in the same way.

4.2.1 Methods

Methods are defined using the primitive operations in the semantics. The method `move()` of the class `fig` is translated into the following function:

```
fig_move : fig -> num # num -> store -> store
fig_move this (dx,dy) s =
let s = fig_set_x this (fig_get_x this s + dx) s in
fig_set_y this (fig_set_y this s + dy) s
```

This function takes three arguments: The first argument `this` is the `fig` object to which the method is applied. The second argument `(dx,dy)` is the pair of the method arguments. The third argument `s` is the store where this method is applied. The return value is the store after applying the method. The attribute `x` is accessed by the *Get* operator

fig_get_x and updated by the *Set* operator fig_set_x. The attribute y is accessed and updated in the same way.

If there is an expression a-b (a and b are natural numbers) in OML, it is translated into the expression if a-b<0 then 0 else a-b in ML. As we mentioned, int is used for natural numbers in ML. This translation is for preserving the semantic equivalence between HOL and ML.

4.2.2 Inheritance

The class rect does not define the method move(). So, it inherits the method from the super-class fig. In our semantics, inherited methods must be defined explicitly using the *Cast* operators. The method move() of the class rect is translated into the following function:

```
rect_move : rect -> num # num -> store -> store
rect_move this (dx,dy) s =
let super = rect_cast_fig this s in
fig_move super (dx,dy) s
```

First, the superclass object super is obtained from the rect object this by the *Cast* operator rect_cast_fig. Then, the function fig_move is applied to it.

4.2.3 Overriding

Method overriding is defined in the same way as normal methods. The super-class object super is accessed using the *Cast* operator. The method move() of the class crect, which turns the color from black to red after the move, is translated into the following function:

```
crect_move : crect -> num # num -> store -> store
crect_move this (dx,dy) s =
let super = crect_cast_rect this s
let s = rect_move super (dx,dy) s in
if (\c. c = BLACK) (crect_get_color this s) then
crect_set_color this RED s
else s
```

The object super is defined by applying the *Cast* operator crect_cast_rect to the crect object this. The function isBlack() is replaced by the embedded code (\c. c = BLACK).

4.2.4 Dynamic binding

Dynamic binding is realized by defining a virtual method which switches the method body according to the instance type of the applied object. The instance type is examined by the *Is* operators. The virtual method for the method move() of the class fig is defined as follows:

```
v_fig_move : fig -> num # num -> store -> store
v_fig_move this (dx,dy) s =
if fig_is_fig this s then
fig_move this (dx,dy) s
else if fig_is_rect this s then
rect_move (fig_cast_rect this s) (dx,dy) s
else
crect_move (fig_cast_crect this s) (dx,dy) s
```

It determines which class the fig object this is instance of by the *Is* operators fig_is_fig and fig_is_rect and calls the corresponding function.

This function is used where the method move() is called to a fig object. For example, the method moveTop() of the class canvas, which calls the method move() to the fig object figList->hd(), is defined as follows:

```
canvas_moveTop : canvas -> int # int -> store -> store
canvas_moveTop this (dx,dy) s =
let l = canvas_get_figList this s in
if ~(l = []) then v_fig_move (HD l) (dx,dy) s
else s
```

4.2.5 Iteration

The iteration commands on lists are defined by higher-order functions on lists. The method moveActive() of the class canvas is translated into the following function:

```
canvas_moveActive : canvas -> num # num ->
 store -> store
canvas_moveActive this (dx,dy) s =
FOLDL (\s x. v_fig_move x (dx,dy) s) s
(FILTER (\x. fig_isActive x s)
 (canvas_get_figList this s))
```

The commands select and apply are translated into the higher-order functions FILTER and FOLDL, respectively. The HOL library of lists contains a lot of theorems for these functions. We can make use of them when reasoning about this method.

4.2.6 Constructors

Constructors are defined in the same way as normal methods. The constructor of the class fig is translated into the following function:

```
fig_fig : fig -> num # num -> store -> store
fig_fig f (x,y) s = fig_set_y f y (fig_set_x f x s)
```

Along with this function, the function to create a fig object is also defined:

```
new_fig : num # num -> store -> fig # store
new_fig (x,y) s =
let (new,s) = fig_new s in
let s = fig_fig new (x,y) s in
(new,s)
```

It first creates the fig object by the *New* operator fig_new. Then, it applies the constructor to the new object. It returns the pair of the new object and the store.

This function is used where a new fig object is created. For example, the method addFig() of the class canvas, is defined as follows:

```
canvas_addFig this (x,y) s =
let (f,s) = new_fig (x,y) s in
let l = canvas_get_figList this s in
if (LENGTH l < canvas_get_max this s) then
canvas_set_figList this (f::l) s
else s
```

4.2.7 Visibilities

In HOL, visibilities of methods such as `public` and `private` are not translated into the semantics. It is only checked by the OML compiler. In ML, visibilities are reflected in the signature of the function structure. Specifically, only the functions corresponding to the new method and the public methods of the main class are included in the signature.

4.3 Translating assertions

ObjectLogic translates assertions into propositions in HOL. There are two kinds of assertions: contracts and invariants. Contracts are the pre- and post-conditions which are assumed and ensured before and after the application of methods. Invariants are the properties which hold on all the states of the system.

4.3.1 Contracts

The contract `lengthInc` of the public method `addFig()` is translated into the following proposition:

```
canvas_addFig_lengthInc =
!(this:canvas) (x:num) (y:num) (s:store).
let s' = canvas_addFig this (x,y) s in
canvas_addFig_lengthInc_pre this (x,y) s /\
 canvas_ex this s
==> canvas_addFig_lengthInc_post this (x,y) s s'
```

where

```
canvas_addFig_lengthInc_pre this (x,y) s =
(LENGTH (canvas_get_figList this s)
 < canvas_get_max this s)
canvas_addFig_lengthInc_post this (x,y) s s' =
(LENGTH (canvas_get_figList this s') =
LENGTH (canvas_get_figList this s) + 1)
```

It means that if the pre-condition `canvas_addFig_lengthInc_pre`holds in the store `s`, the post-condition `canvas_addFig_lengthInc_post` holds in the store `s'`. The store `s'` is constructed by applying the function `canvas_add`-Fig to the previous store `s`. The additional pre-condition `canvas_ex this s` is included to ensure that the main class object `this` is not null.

The pre- and post-conditions take the arguments `x` and `y` of the method `addFig()`. Although they are not used in the conditions, they are included mechanically because contracts are essentially dependent on method arguments.

4.3.2 Invariants

Before we explain the translation of invariants, we need to clarify the definition of the system states and the induction scheme to prove invariants. In terms of the semantics, the initial state of the system is defined by the store which is constructed by applying the new function of the main class (`new_canvas` in the example) to the store `emp`. The succeeding states are defined by the stores which are constructed by applying the functions corresponding to the public methods of the main class repeatedly to the initial state. To observe this definition, OML has a constraint that the attribute of the main class must be private and the main class must not be created inside the system. To prove an invariant, we have to use induction on the system states, i.e., as a base step, we prove that it is satisfied in the initial state, and as induction steps, we prove that each public method of the main class maintains the invariant.

The invariant `lengthMax` is translated into the following proposition:

```
canvas_lengthMax (this:canvas)
 (s:store) =
(LENGTH (canvas_get_figList this s)
 <= canvas_get_max this s)
```

ObjectLogic generates the propositions corresponding to all the steps of the induction. The following proposition is the induction step for the public method `addFig()`:

```
canvas_addFig_lengthMax =
!this (x,y) s. let s' = canvas_addFig this (x,y) s in
canvas_invariants this s /\ canvas_ex this s
==> canvas_lengthMax this s'
```

It means that if the invariant `canvas_lengthMax this s` (which is contained in the predicate `canvas_invariants this s`) holds in the previous store `s`, it also holds in the succeeding store `s'`. The pre-condition `canvas_invariants this s` contains all the invariants of the system. This means that invariants can be always used as pre-conditions of induction steps because they hold on all the states of the system. Again, the pre-condition `canvas_ex this s` is included to ensure that the main object `this` is not null.

OML only allows invariants to be defined in the main class. To define invariants for other classes, they must be defined in the context of the main class. For example, to define `fig` objects' invariant `x<=10`, it must be defined in the main class such as `figList->forall(f|f.x<=10)`.

4.4 Simulation

Simulation is conducted by executing the functions contained in the function structure in ML. The function structure in ML contains the functions corresponding to the public methods

of the main class. It also contains the new function of the main class which is used as the initializer of the system. Currently, it is up to users how to execute the model. Execution is done in the ML interpreter as follows:

```
- val (c,s) = new_canvas 10 emp;
> val c = <canvas> : canvas
val s = <store> : store
- val s = canvas_addFig c (2,3) s;
> val s = <store> : store
- val s = canvas_moveTop c (1,2) s;
> val s = <store> : store
- canvas_getPosList c s;
> val it = [(3,5)] : (num * num) list
```

4.5 Theorem proving

Theorem proving is conducted by proving the propositions contained in the assertion theory in HOL. The proof is done interactively using the tactics provided in ObjectLogic. The main tactic is `OBJ_TAC` which automatically tries to simplify the proof goal using the axioms in the semantics. Additionally, the tactic `SLICE_TAC` is provided. This tactic, as the name implies, applies slicing to the goal, i.e., it makes the goal readable by removing the operators which do not affect the proof. Using these tactics, the proof proceeds basically as follows: (1) expanding definitions, (2) applying `SLICE_TAC`, and (3) applying `OBJ_TAC`. We do not need to remember all of the axioms in the semantics because these tactics automatically apply necessary axioms to the goal.

Let us demonstrate the use of the tactics with the proof of the proposition `canvas_addFig_lengthInc`. First, we expand all the definitions in the goal. This results in the goal shown in Fig. 7, which basically is an equation about the value of `canvas_get_figList`. It looks quite lengthy because it contains many operators which do not affect the value of `canvas_get_figList`, such as `fig_set_y`, `fig_set_x`, and `fig_new`. To remove these irrelevant terms, we then apply `SLICE_TAC`. This results in the goal shown in Fig. 8. (In this step, the condition of the `if`-expression matches the assumption thus only `then`-part remains.) The goal is now quite readable as it contains only the essential operators for the value of `canvas_get_figList`. Now, we can use `OBJ_TAC`. The result is the goal shown in Fig. 9. At this point, all the operators are reduced and only a proposition about lists is left. This can be completed by rewriting with the list theorems, i.e., `SIMP_TAC list_ss []`.

The tactics `SLICE_TAC` and `OBJ_TAC` are implemented so as to automatically apply the necessary axioms to the goal. In the proof, `SLICE_TAC` applied the axioms DiffGetSet and DiffGetNew (see Appendix A). `OBJ_TAC` applied the axiom GetSet. The necessary axioms are determined by examining the proof goal. For example, `OBJ_TAC` selected the axiom GetSet because the goal contained the term

```
canvas_get_figList this (canvas_set_figList this... s)
```

As it matches to the left-hand-side of the equation of the axiom GetSet, this axiom is selected as one of the candidates to simplify the goal. The selected axioms are applied to the goal making use of the built-in tactic `SIMP_TAC`, i.e., they are applied to the goal by being put in the list `L` of `SIMP_TAC bool_ss L`.

5 Verification of a firewall system

We applied ObjectLogic to the verification of a practical firewall system. The specification of the firewall system is based on a real product of a company. In this section, we first present the specification and requirements of the firewall system. Then, we explain the modeling in OML. Finally, we show the verification.

5.1 Firewall specification and requirements

The firewall works on the network layer of the OSI reference model. It is a stateful packet filter, i.e., it decides to pass or drop packets based not only on the filtering rules but also on the connection states. It also conducts NAT (Network Address Translation) which is a mechanism to share a single IP address among multiple hosts. It translates between a global IP address of the firewall and multiple private IP addresses of local hosts. This mechanism has the effects of saving global IP addresses and hiding private addresses of the local network.

We explain the behavior of the firewall by an example. Figure 10 shows the procedure of outbound packet filtering. Now, a packet is going from the inside network to the outside network. Its source address, destination address, and protocol are `(30,1100)`, `(250,80)` and TCP, respectively. (We define an "address" as a pair of IP address and a port number which are represented by natural numbers.) (1) First, the firewall checks that the packet satisfies the filter rule. The filter rule defines permissible packet header values. As all of the header values of this packet are defined in the filter rule, it is allowed to pass the firewall. When a packet belongs to an already existing connection, this check is omitted. (2) Second, it adds a new connection to the connection table. The connection table is a table which stores active connections. In this case, the connection between the local address `(30,1100)` and the global address `(250,80)` is created. Connections are deleted by timeout in a specific amount of time after the last communication. The value `300` is the initial value of the timer. When it cannot create a new connection due to lack of memory capacity, it drops the packet.

Fig. 7 Proof goal (1)

```
!this s x y.
  LENGTH (canvas_get_figList this s) < canvas_get_max this s /\
  canvas_ex this s ==>
  (LENGTH
    (canvas_get_figList this
      (if
       LENGTH
       (canvas_get_figList this
         (fig_set_y (FST (fig_new s)) y
           (fig_set_x (FST (fig_new s)) x (SND (fig_new s))))) <
       canvas_get_max this
         (fig_set_y (FST (fig_new s)) y
           (fig_set_x (FST (fig_new s)) x (SND (fig_new s))))
      then
        canvas_set_figList this
          (FST (fig_new s)::
           canvas_get_figList this
             (fig_set_y (FST (fig_new s)) y
               (fig_set_x (FST (fig_new s)) x
                 (SND (fig_new s)))))
          (fig_set_y (FST (fig_new s)) y
            (fig_set_x (FST (fig_new s)) x (SND (fig_new s))))
      else
        fig_set_y (FST (fig_new s)) y
          (fig_set_x (FST (fig_new s)) x (SND (fig_new s)))) =
   LENGTH (canvas_get_figList this s) + 1)
```

Fig. 8 Proof goal (2)

```
!this s.
  LENGTH (canvas_get_figList this s) < canvas_get_max this s /\
  canvas_ex this s ==>
  (LENGTH
     (canvas_get_figList this
       (canvas_set_figList this
         (FST (fig_new s)::canvas_get_figList this s) s)) =
   LENGTH (canvas_get_figList this s) + 1)
```

Fig. 9 Proof goal (3)

```
!this s.
  LENGTH (canvas_get_figList this s) < canvas_get_max this s /\
  canvas_ex this s ==>
  (LENGTH (FST (fig_new s)::canvas_get_figList this s) =
   LENGTH (canvas_get_figList this s) + 1)
```

(3) Third, it creates a new NAT rule and adds it to the NAT table. The NAT table is a table which stores active NAT rules. In this case, the rule which translates the private address `(30,1100)` into the public address `(200,1220)` is created. The public IP address `200` is the IP address of the firewall and the public port `1220` is the port dynamically selected from the open ports of the firewall. NAT rules are deleted by timeout in a specific amount of time after the last use. The value `500` is the initial value of the timer. (4) Finally, it translates the source address of the packet using the new NAT rule and sends it to the outside network.

As for requirements, we expect this firewall to satisfy the properties such as "The outbound packets which do not meet the filter rules are always dropped unless they belong to existing connections" and "The source IP address of the outbound packet is always updated by the public IP address of the firewall". Both of them are crucial for the firewall security. The first property ensures that a local host never connects to illegal hosts in the outside network. The second property ensures that the private IP addresses of the local network never leak to the outside network. In the following, we focus on the second property which we call "NAT correctness".

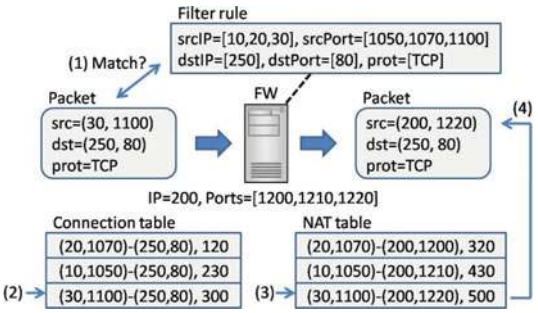


Fig. 10 Outbound filtering

5.2 Modeling in OML

To construct the model, we abstracted the datatypes that appear in the system. IP addresses (and port numbers), which are actually 32 bit data like 192.168.1.10, are abstracted by natural numbers. Generally, representing 32 bit data by natural numbers makes it impossible to realize overflows. But it does not matter because the system never increments and decrements those values and uses them only for comparison. The range of private address values, which actually consists of three classes A, B, and C, is abstracted by the range from 0 to some positive constant. This is because the firewall behavior does not depend on the private address classes. The filter rules are abstracted by lists of permissible values like `[10,20,30]`. Although the filter rules of practical firewalls are defined in a much more complex manner, we kept its datatype as simple as possible because the filter rule itself is not our verification target ([8] verifies inside filter rules). We excluded irrelevant operations from the model such as the checksum recalculation after application of the NAT rules and the audit log recording.

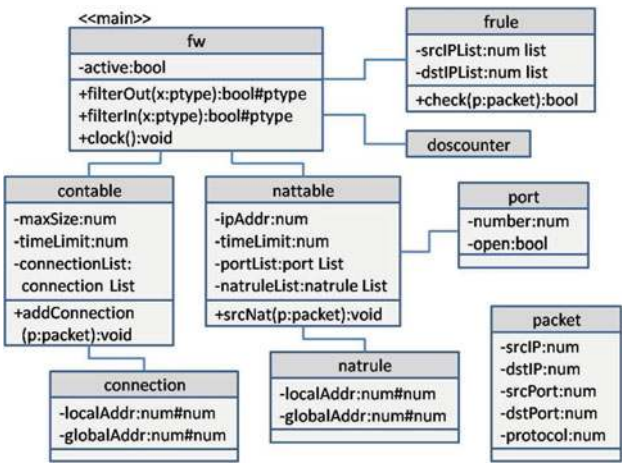


Fig. 11 Class diagram of the firewall system

Figure 11 shows the overall structure of the model as a diagram. It consists of nine classes and the number of attributes and methods are 37 and 105, respectively. The main class is the `fw` class. It has the method `filterOut()` and `filterIn()` to perform outbound and inbound packet filtering, respectively. They input a packet data of type `ptype`. The type `ptype` is declared in OML to be the type `(num#num)#(num#num)#num` which is a tuple of the source address, the destination address and the protocol. If the packet is permitted, they return `true` with the packet data after applying the NAT rule. The `fw` class also has the method `clock()` to proceed time in the system. This is our way of representing time in the model. We assume that `clock()` is called periodically from outer environment. In this method, the timers of the connections and the NAT rules are decremented.

Figure 12 shows the method `filterOut()` in OML. The NAT correctness property is defined as the contract `natCorrectness`. It means that if the input packet is permitted (the first value of `result` is `true`), then the source IP address of the output packet data (obtained by applying the function `srcip(x:ptype):num` to the packet data) is equal to the public IP address of the firewall (obtained by the method `getIP()`). The overall code length of the system is about 1,200 lines.

5.3 Verification

Figure 13 shows the simulation of the firewall system. We first created a `fw` object and set the configuration values such as the public IP address, the port numbers and the filter rules. Then, we applied the filter to a correct packet and an illegal packet and identified if they were correctly handled.

For theorem proving, we proved the contract `natCorrectness` in HOL. The proof took 8 h for one person. We proved 21 lemmas and applied a total of 249 tactics. The detail of the applied tactics is shown in Table 2. We conducted most of the proof with the technique explained in 4.5, i.e., expanding definitions, applying `SLICE_TAC`, and applying `OBJ_TAC`. This routine amounted about 60% of all the tactic applications. Table 3 shows the axioms applied by the two tactics. They automatically applied a total of 110 axioms. Along the proof, we encountered ten case splits. They were caused by the built-in simplifier (`RW_TAC`)when reducing the `if`-expressions. We applied an induction to prove a lemma about lists. We applied a first-order reasoner twice to prove lemmas about booleans. Other tactics include the simplification by theorems (such as booleans and lists), and minor manipulations on the goal (such as quantifier stripping).

The proof failed several times due to the lack of invariants in the pre-condition. The following two invariants were necessary for the `fw` class:

Fig. 12 The method `filterOut()` in OML

```
public filterOut(x:ptype):bool#ptype {
  var sa,da,sa2 : num#num;
  var p : packet = null:packet;
  p = new packet(x);
  if (!isActive()) {
    drop(p); return (false,x);
  } else if (connectionExists(p)) {
    contable.updConnection(p); nattable.srcNat(p);
    return (true,p.getData());
  } else if (contable.isFull()||
             nattable.srcNatRuleExists(p)&&nattable.isFull()) {
    drop(p); return (false,x);
  } else if (isValid(p)) {
    contable.addConnection(p); nattable.srcNat(p);
    return (true,p.getData());
  } else {
    drop(p); return (false,x);
  }
} contracting natCorrectness {
  post: fst(result) implies srcip(snd(result))==this.getIP()
}
```

Fig. 13 Simulation of the firewall system

```
- val (fw,s) = new_fw store_emp;@(* Initialization *)
(* Set configuration *)
- val (_,s) = fw_setIpAddr fw 200 s;
- val (_,s) = fw_setPorts fw [1200,1210,1220] s;
- val (_,s) = fw_setFilterRules fw SRCADDR [10,20,30] s;
...
(* Apply filtering to a correct packet *)
- val (r,s) = fw_filterOut fw ((20,1070),(250,80),TCP) s; (* TCP=0 *)
> val r = (true, ((20, 1200), (250, 80), 0)) :
    bool * ((num * num) * (num * num) * num)
  val s = <store> : store
(* Apply filtering to an illegal packet *)
- val (r,s) = fw_filterOut fw ((20,1070),(300,80),TCP) s;
> val r = (false, ((20, 1070), (300, 80), 0)) :
    bool * ((num * num) * (num * num) * num)
  val s = <store> : store
```

```
nattable!=(null:nattable)
Forall l g. connectionExists(l,g)
 implies srcnatRuleExists(l)
```

The first invariant means that the `nattable` object linked with the `fw` object must not be `null`. This is the characteristic of the graph structure of objects. The `contable` object is created only once and linked to the `fw` object in the constructor of the `fw` class and none of the public methods of the `fw` class changes this link. So, the first invariant obviously holds. The second invariant is the property about consistency between the connection table and the NAT table. It means that if a connection exists in the connection table, a NAT rule corresponding to its local address always exists in the NAT table. As shown in Fig. 10, if a connection `(20,1070)-(250,80)` exists, the NAT rule `(20,1070)-(200,1200)` exists corresponding to the local address `(20,1070)`. This invariant is necessary in the case that an outbound packet belongs to an existing connection. In this case, no new NAT rules are created. So, in order to apply NAT to the packet, we must use the NAT rule which was created when the connection was first created.

6 Discussion

6.1 Effectiveness

The advantage of ObjectLogic is that it enables us to perform theorem proving on the level of objects which is close to our intuition. This makes the proof easy to read compared with

Table 2 Tactics used in the proof

Tactics	Times	Ratio (%)
Definition expansion	101	40.6
SLICE_TAC	46	18.5
OBJ_TAC	8	3.2
Case split	10	4.0
Induction	1	0.4
First-order reasoning	2	0.8
Lemma application	25	10.0
Others	56	22.5
Total	249	100.0

Table 3 Axioms applied by SLICE_TAC and OBJ_TAC

Axioms (SLICE_TAC)	Times	Axioms (OBJ_TAC)	Times
ExSet	22	GetSet	5
DiffGetSet	60	GetSetNew	1
DiffGetNew	11	NotExNull	3
DiffFstNewSet	1		
DiffSetSet	6		
DiffSetNew	1		
Total	101	Total	9

the proof on the level of primitive theories such as pairs and lists. It also makes it easy for us to debug the model, i.e., as the abstract level is the same as that of the model, we can easily identify which part of the model was wrong when a proof failed.

Furthermore, we can conduct proofs using two tactics `SLICE_TAC` and `OBJ_TAC` without having to remember each of the axioms. Actually in the proof, they took care of all the reasoning about objects. This helped us concentrate on the proof of the essential property of the firewall system. Furthermore, as Table 3 shows, the tactics used the same axioms many times. This means that the proof steps to derive the axioms from the definitions of the heap memory model were saved many times. This is a clear evidence that our axiomatization of the semantics actually simplified the verification.

The combination of simulation and theorem proving also worked well to construct the correct model. In simulation, we were able to find many trivial bugs. For example, we found the lack of method call to add a connection by seeing the result that the connection table remained unchanged. We also found that `then`- and `else`-parts of the if-statement were reversed by the result that an obviously correct packet was dropped. By conducting simulation and excluding trivial bugs in advance, we were able to avoid the tediousness to find them in theorem proving. To make simulation more effective, we need to implement a test case generator which can improve the test coverage with respect to various aspects such as branches, statements, and conditions.

In theorem proving, we were able to find a subtle bug through the discovery of invariants. In the proof of the NAT correctness, we found an invariant about consistency between the connection table and the NAT table, i.e., a NAT rule must exist while the corresponding connection exists. This invariant contributed to finding another invariant that the initial timer value of a NAT rule must not be lower than that of a connection. In OML, it is defined as the following invariant:

```
contable.timeLimit <= nattable.timeLimit
```

This invariant is necessary because the NAT rule must not be deleted earlier than the connection. In order to maintain this invariant, we must pay attention to the two methods `setConnectionTime()` and `setNatruleTime()` which are the methods of the `fw` class to set the initial value of the timer for connections and NAT rules, respectively. At first, we naively designed these methods to be able to set arbitrary values to the attributes. But, by the discovery of the invariant, we noticed our mistake and were able to fix it so that the methods preserve the constraint between the two values. We consider that this kind of constraints on internal values are difficult to find in simulation.

6.2 Possible improvements

Although the high abstractness of the semantics was effective, we still took as much as 8 h to complete the proof. There are mainly three reasons for the inefficiency. First, we had to consume a lot of time to find invariants when the proof failed. Second, we had to redo the proof many times to understand the proof structure which was made complex by many nested branches. Third, we had to make many lemmas to arrange the proof.

The first and second cases are unavoidable because they come from the nature of the firewall system. But, the third case can be improved a lot by enhancing `SLICE_TAC`. In fact, as many as 15 of 21 lemmas were concerned with slicing. For example, we proved the following lemma:

```
|- !(fw:fw) (ct:contable) (c:connection) (s:store).
fw_getIP fw (contable_addConnection ct c s)
 = fw_getIP fw s
```

This means that the `fw` class' method `getIP()` is not affected by the `contable` class' method `addConnection()`. This is because the attributes accessed by these two methods do not overlap. Even though this theorem was easily proved with the definition expansion and `SLICE_TAC`, we had to prove it separately as a lemma because the definition expansion made the proof goal quite large and made the application of `SLICE_TAC` quite slow.

This problem can be solved by implementing SLICE_TAC to work directly on the method level. Specifically, it should be implemented so that it can automatically prove the theorem inside the tactic and apply it to the goal. This allows us to apply slicing on the method level without expanding definitions. If it is used in the proof, we can reduce a total of 126 tactic applications (50.6% of all) required to prove the extra 15 lemmas. As a result, we can concentrate more on the essential part of the proof.

6.3 Exception handling

Currently, we do not implement the exception handling mechanism explicitly in the semantics. The exceptional cases such as the null accessing and the illegal down-casting are handled differently in ML and HOL. (In this sense, the equivalence of the semantics is broken when exceptional cases occur.) In ML, we simply raise the exception of ML. In HOL, we return a constant representing an undefined value, which is summarized as the following axioms:

```
|- !s. fig_get_x fig_null s = undef
|- !x s. fig_set_x fig_null x s = s
|- !f s. fig_is_fig f s ==> (fig_cast_rect f s = undef)
```

The first and the second axioms are the cases when an attribute of a null object is read and set, respectively. The third axiom is the case when an illegal down-casting occurs. For the first and the third cases, we return the constant `undef` of type `'a`. When this constant comes up in the proof goal, we can notice the occurrence of the exception. (Of course, we do not always notice this, for example, when an equation is accidentally evaluated to `undef=undef` and reduced to `T`.) For the second case, we just "skip" the exception by leaving the store `s` unchanged.

The reason to skip the exception for the second case is related to the efficiency of SLICE_TAC. Specifically, if we do not skip the exception and return `undef`, we have to check if the object is non-null every time an attribute setting occurs. Actually, if we return `undef`, the axiom DiffGetSet

```
|- !i j x s. fig_get_y i (fig_set_x j x s)
 = fig_get_y i s
```

must be changed to

```
|- !i j x s. fig_ex j s ==>
(fig_get_y i (fig_set_x j x s) = fig_get_y i s)
```

This means that we cannot remove `fig_set_x` without proving the existence of the object `j`. As shown in Table 3, DiffGetSet is an important axiom for SLICE_TAC. If this implication is imposed, SLICE_TAC becomes quite inefficient because it cannot always perform the simplification by the equation.

For this reason, we adopted the axiom to skip the exception. We consider that this is practically reasonable because we can considerably improve the efficiency of the proof by SLICE_TAC with only sacrificing the check for the attribute setting. Furthermore, the lack of the check can be compensated by the simulation which supports all the exceptions.

7 Related work

7.1 Combining a programming language and a theorem prover

ACL2 is well known as the tool to combine a programming language and a theorem prover. It is based on an applicative subset of Common Lisp. It is often used as the semantics for both simulation and theorem proving. For example, the work by Moore [13] implements the operational semantics of JVM for testing and proving about Java methods. The work by Al Sammane [2] implements a tool TheoSim which supports simulation and theorem proving of VHDL designs. The work by Wilding et al. [26] defines a formal model of a microprocessor to integrate simulation and formal analysis.

We followed the approach of combining a programming language and a theorem prover using ML and HOL. The reason for the use of HOL is to construct the OO semantics conservatively by definitional extension. To derive our semantics from the definitions, it requires the expressiveness of the higher-order logic. Specifically, we need the induction theorem on lists to derive the properties about the heap memory operations. We also need predicate variables to construct the type `store` from the heap memory type. It is an advantage of HOL to be able to construct sound theories using such powerful expressiveness. As the semantics itself is expressible by the first-order logic, it is possible to export it to ACL2 and conduct verification using its powerful reasoner.

7.2 Embedding OO semantics

There has been a lot of works on implementing OO semantics in theorem provers especially for Java. For deep embedding, the work by von Oheimb et al. [24] implements a semantics of Java for both source language level and bytecode level in Isabelle/HOL. The work by Barthe et al. [3] implements a semantics of JavaCard platform (virtual machine and bytecode verifier) in Coq. Both of them adopt deep embedding because their verification target is on the meta-language level such as type safety, soundness of Hoare logic, and correctness of the bytecode verifier. We adopted a shallow embedding because our verification target is on the instance level such as method contracts and class invariants. Shallow embedding makes the proof on the instance level easier and the theory itself simpler than deep embedding.

For shallow embedding, the work by Brucker et al. [5] implements an object data model close to ours. The type of the object data is made extensible by representing it by the

product type and the sum type. This enables the reuse of proof by structural subtyping of objects. Our model, however, does not realize the reuse of proof because object subtyping is realized explicitly by cast functions. Instead of using extensible types, we represented objects by linked-tuples. This enables static type checking on object references. Furthermore, we made our data model executable by embedding it within the expressivity of ML.

Poetzsch-Heffter and Müller [19] define a store model for the logical foundation of Java verification. The operators defined on the store are similar to ours such as object creation, attribute get and set. But it does not have the operators concerning subtyping such as the cast and instance-of operators. They are defined on the level of Hoare-logic. The work by van den Berg et al. [23] and Marché and Paulin-Mohring [11] implements memory models of Java semantics for reasoning about Java programs annotated with JML specifications in Isabelle/HOL and Why, respectively. We defined a similar memory model, but it differs from them in that it allows arbitrary types for object attributes, which is effective in the verification on the analysis level.

7.3 Interactive verification tools for OO specification

The KeY tool [1] is a software development system based on UML. It supports construction of the specification in OCL and the implementation in Java Card. It also supports verification of both of them by generating proof obligations in dynamic logic, an extension of Hoare logic. The HOL-OCL tool [6] is an interactive proof environment for UML/OCL. The semantics of OCL is embedded conservatively in Isabelle/HOL faithfully to its three-valued logic. It also provides its own proof procedure for reasoning about the specification. The Why tool [7] is a verification system for Java and C. It inputs JML-annotated Java programs and annotated C programs from the front end tools Krakatoa and Caduceus, respectively, and outputs verification conditions to various theorem provers. The Jive tool [12] and the LOOP tool [9] are also verification systems for JML-annotated Java. In Jive, the proof is done on the source code level using tactics which are implemented based on Hoare logic. In LOOP, the proof is done inside PVS based on the weakest precondition calculus. The work by Schirmer [21] implements a verification environment for a general imperative programming language model. It can generate verification conditions based on Hoare logic whose completeness and soundness is proved in Isabelle/HOL.

Compared with these tools which support verification of standard and general languages, ObjectLogic is directed at a verification tool for a specific domain. Our future vision is to seek for the fully automated proof in the data management domain and automatically generate implementation code from the language.

8 Conclusion and future work

In this paper, we presented an executable semantics of OO models for the foundation of both simulation and theorem proving. The semantics is implemented in two languages: HOL and ML. We preserved the semantics equivalence by implementing the underlying heap memory structure within the intersection of their expressiveness. We mainly presented the formal definition of the semantics and the prototype verification tool ObjectLogic which supports simulation and theorem proving on the semantics. As a case study, we showed the verification of a practical firewall system and discussed the effectiveness and possible improvements of ObjectLogic. Future work is to improve the degree of automation of the tool by enhancing the tactics. We are also considering to apply it to the verification of more complex systems which include inheritance.

Appendix A: Axioms

The full list of axioms (with 3 theorems) is presented here. They are divided into two groups depending on how they are derived. One is the axioms which can be derived simply by expanding the definitions of operators. The other is the axioms which can be derived as invariants of the heap memory structure. The latter ones are marked with "*". Furthermore, the axioms used by `SLICE_TAC` are marked with "†". The others are applied by `OBJ_TAC`. The correspondence between axioms and operations are presented in Table 4.

1. **NotExEmp**
   $\vdash \forall o.\ \neg(Ex^c\ o\ Emp)$
   No objects exist the empty store.
2. **NotExNull**
   $\vdash \forall s.\ \neg(Ex^c\ Null^c\ s)$
   No null objects exist in the store.
3. **ExIs***
   $\vdash \forall o\ s.\ Ex^c\ o\ s = Is^c_{d_1}\ o\ s \vee \cdots \vee Is^c_{d_n}\ o\ s (\{d_1, \ldots, d_n\} = \{d \mid c \lhd^* d\})$
   The c object o which exists in the store is an instance of either the class c or one of the descendants of c.
4. **NotExFstNew**
   $\vdash \forall s.\ let\ (o, s') = New^c\ s\ in\ \neg(Ex^c\ o\ s)$
   The newly created object does not exist in the previous store. This axiom implies that the new object is distinct from all the previous objects.
5. **NotExFstNewCast**
   $\vdash \forall s.\ let\ (o, s') = New^c\ s\ in\ \neg(Ex^c\ (Cast^c_d\ o\ s')\ s)$
   This axiom is similar to the previous one. The object obtained by casting the newly created object does not exist in the store before creation.

Table 4 Correspondence between axioms and operators

	New	Ex	Get	Set	Cast	Is	Null	Emp
1. NotExEmp		○						○
2. NotExNull		○					○	
3. ExIs*		○				○		
4. NotExFstNew	○	○						
5. NotExFstNewCast	○	○			○			
6. IsImpNotIs*						○		
7. IsCast*					○	○		
8. IsNew	○					○		
9. IsNewCast	○				○	○		
10. DiffIsNew†	○					○		
11. IsSet†				○		○		
12. DownNull					○	○	○	
13. NotExCast		○			○		○	
14. Up11*		○			○			
15. Down11*					○	○		
16. UpDown*		○			○			
17. DownUp*		○			○			
18. CastCast					○			
19. CastSet†				○	○			
20. ExCastNew	○	○			○			
21. DiffCastNew	○				○			
22. NotExGet		○	○					
23. NotExSet		○		○				
24. SprGet			○		○			
25. SprSet				○	○			
26. GetSet		○	○	○				
27. DiffObjGetSet			○	○				
28. DiffGetSet†			○	○				
29. GetNew	○		○					
30. GetNewCast	○		○		○			
31. ExGetNew	○	○	○					
32. DiffGetNew†	○		○					
33. SetSet†				○				
34. DiffObjSetSet				○				
35. DiffSetSet†				○				
36. ExSetNew	○	○		○				
37. DiffSetNew†	○			○				
38. DiffNewNew†	○							
39. SetGet†			○	○				
40. FstNewSet†	○			○				
41. DiffFstNewNew†	○							
42. ExNew	○	○						
43. ExSet†		○		○				
44. GetSetNew	○		○	○				

6. **IsImpNotIs***
$\vdash \forall o\ s.\ Is_d^c\ o\ s \Rightarrow \neg(Is_e^c\ o\ s)\ \ (d \neq e)$
If the c object o is an instance of the class d, it is not an instance of the different class e.
7. **IsCast***
$\vdash \forall o\ s.\ Is_e^c\ o\ s \Rightarrow Is_e^d\ (Cast_d^c\ o\ s)\ s\ \ (d \lhd^+ e)$
If the c object o is an instance of the class e, the object obtained by casting to the ancestor-class d is also the instance of e, i.e., the instance type is invariable.
8. **IsNew**
$\vdash \forall o_1\ s.\ let\ (o_2, s') = New^c\ s\ in$
$Is_c^c\ o_1\ s' = (o_1 = o_2) \vee Is_c^c\ o_1\ s$
The c object o_1 is an instance of the class c in the store after creating a new instance of the class c iff o_1 is either the newly created object o_2 or the object which was already an instance of c before the creation.
9. **IsNewCast**
$\vdash \forall o_1\ s.\ let\ (o_2, s') = New^d\ s\ in$
$Is_d^c\ o_1\ s' = (o_1 = Cast_c^d\ o_2\ s') \vee Is_d^c\ o_1\ s$
This axiom is similar to the previous one. The c object o_1 is an instance of the class d in the store after creating a new instance of the class d iff o_1 is either the object obtained by casting the new object o_2 to c or the object which was already an instance of d before the creation.
10. **DiffIsNew**†
$\vdash \forall o\ s.\ Is_d^c\ o\ (Snd\ (New^e\ s)) = Is_d^c\ o\ s\ \ (d \neq e)$
Whether the c object is an instance of the class d or not is not affected by the object creation of the class e which is different from d.
11. **IsSet**†
$\vdash \forall o_1\ o_2\ x\ s.\ Is_d^c\ o_1\ (Set_a^e\ o_2\ x\ s) = Is_d^c\ o_1\ s$
Is operations are independent of Set operations.
12. **DownNull**
$\vdash \forall o\ s.\ Is_d^c\ o\ s \Rightarrow (Cast_e^c\ o\ s = Undef)\ \ (e \not\lhd^* d)$
Casting the d instance to the non-ancestor class e results in the undefined value.
13. **NotExCast**
$\vdash \forall o\ s.\ \neg(Ex^c\ o\ s) \Rightarrow (Cast_d^c\ o\ s = Null^d)$
Casting the non-existent object results in the null object of the destination class.
14. **Up11***
$\vdash \forall o_1\ o_2\ s.\ Ex^d\ o_1\ s \wedge Ex^d\ o_2\ s \Rightarrow$
$\neg(o_1 = o_2) \Rightarrow \neg(Cast_c^d\ o_1\ s = Cast_c^d\ o_2\ s)\ (c \lhd^+ d)$
If two c objects o_1 and o_2 are different and exist in the store, the two object obtained by up-casting to the class c are also different, i.e., $Cast$ operators are injective.
15. **Down11***
$\vdash \forall o_1\ o_2\ s.\ Is_e^c\ o_1\ s \wedge Is_e^c\ o_2\ s \Rightarrow$
$\neg(o_1 = o_2) \Rightarrow \neg(Cast_d^c\ o_1\ s = Cast_d^c\ o_2\ s)\ \ (c \lhd^+ d$ and $d \lhd^* e)$
If two c objects o_1 and o_2 (which are both instance of the class e) are different, the two objects obtained by down-casting to the class d (which is equal to or an ancestor of e) are also different.
16. **UpDown***
$\vdash \forall o\ s.\ Ex^d\ o\ s \Rightarrow (Cast_d^c\ (Cast_c^d\ o\ s)\ s = o)\ \ (c \lhd^+ d)$
If the d object o exists in the store, the object obtained by up-casting to c and then down-casting to d is equal to o itself.
17. **DownUp***
$\vdash \forall o\ s.\ Ex^d\ (Cast_d^c\ o\ s)\ s \Rightarrow (Cast_c^d\ (Cast_d^c\ o\ s)\ s = o)\ \ (c \lhd^+ d)$
If the c object o is down-castable to the class d (i.e., the down-cast object exists in the store), the object obtained by down-casting to d and then up-casting to c is equal to o itself.
18. **CastCast**
$\vdash \forall o\ s.\ Cast_e^d\ (Cast_d^c\ o\ s)\ s = Cast_e^c\ o\ s$
$((c \lhd^+ d$ and $d \lhd^+ e)$ or $(e \lhd^+ d$ and $d \lhd^+ c))$
Two transitive casts from c to d and from d to e are concatenated to a single cast from c to e.
19. **CastSet**†
$\vdash \forall o_1\ o_2.\ Cast_d^c\ o_1\ (Set_a^e\ o_2\ x\ s) = Cast_d^c\ o_1\ s$
$Cast$ operations are independent of Set operations.
20. **ExCastNew**
$\vdash \forall o\ s.\ Ex^c\ o\ s \Rightarrow\ (Cast_d^c\ o\ (Snd\ (New^e\ s)) = Cast_d^c\ o\ s)$
$(c \lhd^* e$ and $d \lhd^* e)$
If the c object o exists in the store, the value of the object reference obtained by casting o to d is not affected by the object creation of the class e. This axiom holds when c and d are equal to or ancestors of e. The other case is stated in the next axiom.
21. **DiffCastNew**
$\vdash \forall o\ s.\ Cast_d^c\ o\ (Snd\ (New^e\ s)) = Cast_d^c\ o\ s\ \ (c \not\lhd^* e$ or $d \not\lhd^* e)$
The value of the object reference obtained by casting the c object o to d is not affected by the object creation of the class e if either c or d is not equal to or an ancestor of e.
22. **NotExGet**
$\vdash \forall o\ s.\ \neg(Ex^c\ o\ s) \Rightarrow (Get_a^c\ o\ s = Undef)$
Getting the attribute of a non-existent object results in the undefined value.
23. **NotExSet**
$\vdash \forall o\ x\ s.\ \neg(Ex^c\ o\ s) \Rightarrow (Set_a^c\ o\ x\ s = s)$
Setting the attribute of a non-existent object causes nothing to the store.
24. **SprGet**
$\vdash \forall o\ s.\ Get_a^d\ o\ s = Get_a^c\ (Cast_c^d\ o\ s)\ s\ \ (c \lhd^+ d$ and $a \in \mathcal{M}_{attr}(c))$
For the d object o, getting the attribute a defined in the super-class c is equivalent to getting a after up-casting it to c.

25. **SprSet**
$\vdash \forall o\ s.\ Set_a^d\ o\ x\ s = Set_a^c\ (Cast_c^d\ o\ s)\ x\ s\ \ (c \lhd^+ d$ and $a \in \mathcal{M}_{attr}(c))$
For the d object o, setting the attribute a defined in the super-class c is equivalent to setting a after up-casting it to c.
26. **GetSet**
$\vdash \forall o\ s.\ Ex^c\ o\ s \Rightarrow (Get_a^c\ o\ (Set_a^c\ o\ x\ s) = x)$
If the object o exists in the store, the attribute a of o obtained just after setting it to x is equal to x.
27. **DiffObjGetSet**
$\vdash \forall o_1\ o_2\ s.\ \neg(o_1 = o_2) \Rightarrow (Get_a^c\ o_1\ (Set_a^c\ o_2\ x\ s) = Get_a^c\ o_1\ s)$
If the two objects o_1 and o_2 are different, getting the attribute a of o_1 is not affected by the setting of the same attribute of o_2.
28. **DiffGetSet**†
$\vdash \forall o_1\ o_2\ s.\ Get_a^c\ o_1\ (Set_b^d\ o_2\ x\ s) = Get_a^c\ o_1\ s$
$((c \not\lhd^* d$ and $d \not\lhd^* c)$ or $a \neq b)$
If the two classes c and d are not in the inheritance relationship or the attribute name a and b are different, getting the attribute `a` of the object o_1 is not affected by the setting of the attribute `b` of the object o_2.
29. **GetNew**
$\vdash \forall s.\ let\ (o, s') = New^c\ s\ in\ Get_a^c\ o\ s' = \mathcal{V}(c, a)$
Getting the attribute of the newly created object results in the default value for the attribute.
30. **GetNewCast**
$\vdash \forall o\ s.\ let\ (o, s') = New^d\ s\ in$
$Get_a^c\ (Cast_c^d\ o\ s')\ s' = \mathcal{V}(c, a)\ \ (c \lhd^+ d$ and $e \lhd^* c)$
This axiom is similar the previous one. Getting the attribute of the object obtained by up-casting the newly created object results in the default value of the attribute.
31. **ExGetNew**
$\vdash \forall o\ s.\ Ex^c\ o\ s \Rightarrow (Get_a^c\ o\ (Snd\ (New^d\ s)) = Get_a^c\ o\ s)\ \ (c \lhd^* d)$
If the c object o exists in the store, getting the attribute of o is not affected by the object creation of the class d. This axiom holds when d is equal to or a descendant of c. The other case is stated in the next axiom.
32. **DiffGetNew**†
$\vdash \forall o\ s.\ Get_a^c\ o\ (Snd\ (New^d\ s)) = Get_a^c\ o\ s\ \ (c \not\lhd^* d)$
The attribute value of the c object is not affected by the object creation of the class d which is neither equal to nor a descendant of c.
33. **SetSet**†
$\vdash \forall o\ x\ y\ s.\ Set_a^c\ o\ x\ (Set_a^c\ o\ y\ s) = Set_a^c\ o\ x\ s$
Two consecutive settings of the same attribute cancel the previous one.
34. **DiffObjSetSet**
$\vdash \forall o_1\ o_2\ x\ y\ s.\ \neg(o_1 = o_2) \Rightarrow$
$(Set_a^c\ o_1\ x\ (Set_a^c\ o_2\ y\ s) = Set_a^c\ o_2\ y\ (Set_a^c\ o_1\ x\ s))$
Two attribute settings of different objects are interchangeable.
35. **DiffSetSet**†
$\vdash \forall o_1\ o_2\ x\ y\ s.\ Set_a^c\ o_1\ x\ (Set_b^d\ o_2\ y\ s) = Set_b^d\ o_2\ y\ (Set_a^c\ o_1\ x\ s)((c \not\lhd^* d$ and $d \not\lhd^* c)$ or $(a \neq b))$
Two settings of different attributes are interchangeable.
36. **ExSetNew**
$\vdash \forall o\ x\ s.\ Ex^c\ o\ s \Rightarrow$
$(Set_a^c\ o\ x\ (Snd\ (New^d\ s)) = Snd\ (New^d\ (Set_a^c\ o\ x\ s)))(c \lhd^* d)$
If the object o exists in the store, the attribute setting of the object and the object creation of the class d are interchangeable. This axiom holds when c is equal to or an ancestor of d. The other case is stated in the next axiom.
37. **DiffSetNew**†
$\vdash \forall o\ x\ s.\ Set_a^c\ o\ x\ (Snd\ (New^d\ s)) = Snd\ (New^d\ (Set_a^c\ o\ x\ s))\ \ (c \not\lhd^* d)$
The attribute setting of the c object and the object creation of the class d (which is neither equal to nor a descendant of c) are interchangeable.
38. **DiffNewNew**†
$\vdash \forall s.\ Snd\ (New^c\ (Snd\ (New^d\ s))) = Snd\ (New^d\ (Snd\ (New^c\ s)))(\text{not}\ relative(c, d))$
Two object creations of different classes c and d are interchangeable as long as they belong to different inheritance tree ($relative(c, d) = \exists r.\ r \lhd^* c \wedge r \lhd^* d$).
39. **SetGet**†
$\vdash \forall o\ s.\ Set_a^c\ o\ (Get_a^c\ o\ s)\ s = s$
Setting the attribute a in the store s to the same attribute in the same store results in the original store.
40. **FstNewSet**†
$\vdash \forall o\ x\ s.\ Fst\ (New^c\ (Set_a^d\ o\ x\ s)) = Fst\ (New^c\ s)$
The value of the object reference obtained by the object creation is not affected by the attribute setting.
41. **DiffFstNewNew**†
$\vdash \forall s.\ Fst\ (New^c\ (Snd\ (New^d\ s))) = Fst\ (New^c\ s)\ \ (c \not\lhd^* d)$
The value of the object reference obtained by the object creation of the class c is not affected by the object creation of the class d which is neither equal to nor a descendant of c.
42. **ExNew**
$\vdash \forall o_1\ s.\ let\ (o_2, s') = New^c\ s\ in$
$Ex^c\ o_1\ s' = (o_1 = o_2) \vee Ex^c\ o_1\ s$
This is a theorem derived from ExIs and IsNew. The c object o_1 exists in the store after creating a new instance of the class c iff o_1 is either the newly created object o_2 or the object which already existed before the creation.
43. **ExSet**†
$\vdash \forall o_1\ o_2\ x\ s.\ Ex^c\ o_1\ (Set_a^e\ o_2\ x\ s) = Ex^c\ o_1\ s$

This is a theorem derived from ExIs and IsSet. Ex operations are independent of Set operations.

44. **GetSetNew**
$\vdash \forall x\, s.\, let\ (o, s') = New^c\ s\ in\ Get_a^c\ o\ (Set_a^c\ o\ x\ s') = x$
This is a theorem derived from ExNew and GetSet. If the object o is a newly created object, the attribute a of o obtained just after setting it to x is equal to x.

Appendix B: OML syntax

The syntax of OML is defined as BNF notation. The notation ϵ is an empty string. The notation A^+ repeats A at least once. The notation $A \mid B$ selects A or B. The notation A^* is equivalent to $A\ A^+$. The notation $A^?$ means A is optional.

OML consists of a header and a set of classes:

$$OML ::= Header\ (Class)^*$$

The header consists of imports of libraries and declaration of new types, constants, and functions:

$$\begin{aligned}
Header &::= (Import)^*\ (Decl)^* \\
Import &::= \texttt{importML}\ Names \mid \texttt{importHOL}\ Names \\
Decl &::= TypeDecl \mid ConstDecl \mid FuncDecl \\
TypeDecl &::= \texttt{type}\ name\ \texttt{==}\ (Type \mid Emb) \\
ConstDecl &::= \texttt{const}\ name\ \texttt{:}\ Type\ \texttt{==}\ (Exp \mid Emb) \\
FuncDecl &::= \texttt{func}\ name\ \texttt{(}\ Args\ \texttt{):}\ Type\ \texttt{==}\ (Exp \mid Emb) \\
Emb &::= \texttt{HOL[\#}\ string\ \texttt{\#]}\ \texttt{ML[\#}\ string\ \texttt{\#]} \\
Names &::= name\ (\texttt{,}\ name)^* \\
Args &::= \epsilon \mid name\ \texttt{:}\ name\ (\texttt{,}\ name\ \texttt{:}\ name)^* \\
name &::= (\texttt{a}|...|\texttt{z}|\texttt{A}|...|\texttt{Z})\ (\texttt{a}|...|\texttt{z}|\texttt{A}|...|\texttt{Z}|\texttt{0}|...|\texttt{9}|)^*
\end{aligned}$$

New types, constants, and functions are defined using the standard types and expressions in OML or the embedded code from ML and HOL.

A class consists of three kinds of members: attributes, methods, and invariants. Contracts can be attached to a method:

$$\begin{aligned}
Class &::= ClassDecl\ \texttt{\{}\ (Member)^*\ \texttt{\}} \\
ClassDecl &::= (\texttt{main})^?\ \texttt{class}\ name\ (\texttt{extends}\ name)^? \\
Member &::= (Attr \mid Meth \mid Inv)^* \\
Attr &::= Acc\ name\ \texttt{:}\ Type\ \texttt{==}\ Exp\ \texttt{;} \\
Meth &::= Acc\ name\ \texttt{(}\ Args\ \texttt{):}\ Type\ \texttt{\{}\ Body\ \texttt{\}}\ (Conts)^? \\
Inv &::= \texttt{invariant}\ name\ \texttt{\{}\ Exp\ \texttt{\}} \\
Conts &::= \texttt{contracting}\ Cont\ (\texttt{and}\ Cont)^* \\
Cont &::= name\ \texttt{\{}\ (Quant)^?\ (\texttt{pre:}\ Exp)^?\ \texttt{post:}\ Exp\ \texttt{\}} \\
Acc &::= \texttt{private} \mid \texttt{public} \mid \texttt{protected} \\
Args &::= \epsilon \mid Arg\ (\texttt{,}\ Arg)^* \\
Arg &::= name\ \texttt{:}\ Type
\end{aligned}$$

The main class must be unique and must not be created inside the system. The invariants are only defined in the main class. The attributes of the main class must be private. The public methods of the main class can neither input nor output objects (The system is closed). It is prohibited to input objects to the public methods of the main class It is also prohibited to call side-effecting methods inside contracts and invariants. Both in the contracts and invariants, visibilities become transparent, i.e., all the members can be publicly accessed (This is for omitting the burden to define additional methods for debugging).

There are following kinds of types:

$$\begin{aligned}
Type ::= &\ name \mid \texttt{void} \mid \texttt{num} \mid \texttt{bool} \mid \texttt{string} \mid Type\ \texttt{\#}\ Type \mid \\
&\ Type\ \texttt{list} \mid name\ \texttt{List} \mid \texttt{(}\ Type\ \texttt{)}
\end{aligned}$$

The type *name* includes the class names for object types and the names for newly declared types. All the types except for objects are simple data and not objects. For example, the strings in Java are treated as objects, but they are treated as values in OML. The type `List` is the type of object lists which is distinguished from the normal list type `list`. Both lists are also treated as values.

The method body consists of declaration of local variables and a sequence of statements:

$$\begin{aligned}
Body &::= (Var)^*\ (Stm)^* \\
Var &::= \texttt{var}\ Names\ \texttt{:}\ Type\ \texttt{==}\ Exp\ \texttt{;} \\
Stm &::= Skip \mid Ass \mid Call \mid App \mid New \mid Loop \mid If \mid Return \\
Names &::= name\ (\texttt{,}\ name)^*
\end{aligned}$$

Statements are defined as follows:

$$\begin{aligned}
Skip &::= \texttt{;} \\
Ass &::= (Exp\ \texttt{.})^?\ name\ \texttt{=}\ Exp\ \texttt{;} \\
Call &::= ((Exp\ \texttt{.})^?\ name\ \texttt{=})^?\ (Exp\ \texttt{.})^?\ name\ \texttt{(}\ Exps\ \texttt{);} \\
App &::= Exp\ \texttt{->}\ \texttt{apply}\ \texttt{(}\ name\ \texttt{(}\ Args\ \texttt{))}\ \texttt{;} \\
New &::= (Exp\ \texttt{.})^?\ name\ \texttt{=}\ (\texttt{new} \mid \texttt{newList})\ name\ \texttt{(}\ Exps\ \texttt{);} \\
Loop &::= Exp\ \texttt{.}\ \texttt{loop}\ \texttt{(}\ name\ \texttt{(}\ Args\ \texttt{),}\ Exp\ \texttt{);} \\
If &::= \texttt{if}\ \texttt{(}\ Exp\ \texttt{)}\ Stm\ (Else)^? \mid \texttt{if}\ \texttt{(}\ Exp\ \texttt{)}\ \texttt{\{}\ (Stm)^*\ \texttt{\}}\ (Else)^? \\
Else &::= \texttt{else}\ Stm \mid \texttt{else}\ \texttt{\{}\ (Stm)^*\ \texttt{\}} \\
Return &::= \texttt{return}\ Exp\ \texttt{;} \\
Exps &::= \epsilon \mid Exp\ (\texttt{,}\ Exp)^*
\end{aligned}$$

There are two commands for object creation: `new` and `newList`. The first one is the usual one for creating a single object used like `f = new foo(1);`. The second one is for creating multiple objects used like `L = new`-List foo([1,2,3]). It creates three `cutomer` objects with constructor arguments `1`, `2` and `3`, respectively. The created objects are put in the ist `L` in this order. The command `loop` repeats a method for a finite number of times. For example, the statement `fw.loop(clock(),10);` calls the method `clock()` to the `fw` object for `10` times.

Expressions are defined as follows:

```
Exp     ::= Obj | Num | Bool | String | Pair | List | ObjList |
            Prefix Exp | Exp Infix Exp | Exp Suffix |
            (Exp .)? name | Exp . name ( Exps ) | Fun ( Exps ) |
            Cond | _ | result | Quant Exp | ( Exp )
Obj     ::= this | super | null: Type | Exp : Type
Num     ::= (1|...|9) (0|...|9)*
Bool    ::= true | false
Prefix  ::= !
Infix   ::= + | - | * | / | % | && | || | implies | == | != | < | > | <= | >=
Suffix  ::= @pre
Fun     ::= fst | snd | concat | hd | tl | last | nth | cons | length |
            append | zip | unzip | filter | forall | exists |
            map | member | foldl | foldr | reverse | sum | mklist
String  ::= " string "
Pair    ::= ( Exps )
List    ::= []: Type | [ Exps ]
ObjList ::= Nil: Type | Exp -> add ( Exp ) | Exp -> del ( Exp ) |
            Exp -> append ( Exp ) | Exp -> hd() | Exp -> tl() |
            Exp -> length() | Exp -> contains ( Exp ) |
            Exp -> select ( name | Exp ) | Exp -> reject ( name | Exp ) |
            Exp -> forall ( name | Exp ) | Exp -> exists ( name | Exp ) |
            Exp -> collect ( name | Exp )
Cond    ::= ( Exp )? Exp : Exp
Quant   ::= (Forall (name)+ . | Exists (name)+ . )+
```

It is not allowed to call side-effecting methods in expressions. Quantifiers and the infix `implies` can be used only in the contracts and invariants. The expression `result` and the suffix `@pre` can be used only in the post-condition of contracts. Cast is denoted by `:` like `rectobj:fig`. The application of object list function is denoted by `->`, which makes it easy to connect them sequentially like `L->select(x|x.F())->length()`. The expression `_` is used as the special argument for functions on `list`. The meaning depends on the functions. For example, the expression `filter([1,2,3], _<=2)` evaluates to the list `[1,2]`. In this case, `_` is used to represent each element in the list. The function `mk_list` is used to construct lists. The expression `mklist(n,G(_))` constructs a list whose length is `n` and the `i`-th value is `G(i)`. For example, the expression `mklist(4,_*_)` evaluates to the list `[1,4,9,16]`. In this case, `_` is used to represent the position of the list. By combining `mklist` and `newList`, we can construct various kinds of object lists.

References

1. Ahrendt, W., Baar, T., Beckert, B., Bubel, R., Giese, M., Hahnle, R., Menzel, W., Mostowski, W., Roth, A., Schlager, S., Schmitt, P.H.: The key tool. Softw. Syst. Model. **4**(1), 32–54 (2005)
2. Al Sammane, G., Schmaltz, J., Toma, D., Ostier, P., Borrione, D.: TheoSim: combining symbolic simulation and theorem proving for hardware verification. In: SBCCI '04: Proceedings of the 17th Symposium on Integrated Circuits and System Design, pp. 60–65. ACM, New York, NY, USA (2004)
3. Barthe, G., Dufay, G., Jakubiec, L., Serpette, B.P., de Sousa, S.M.: A formal executable semantics of the JavaCard platform. In: Sands, D. (ed.) Programming Languages and Systems, 10th European Symposium on Programming, ESOP 2001 Held as Part of the Joint European Conference on Theory and Practice of Software, ETAPS 2001 Genova, Italy, 2–6 April. Lecture Notes in Computer Science, vol. 2028, pp. 302–319. Springer, London (2001)
4. Berghofer, S., Nipkow, T.: Executing higher order logic. In: TYPES '00: Selected Papers from the International Workshop on Types for Proofs and Programs, pp. 24–40. Springer, London, UK (2002)
5. Brucker, A.D., Wolff, B.: An extensible encoding of object-oriented data models in HOL. J. Autom. Reason. **41**(3–4), 219–249 (2008)
6. Brucker, A.D., Wolff, B.: HOL-OCL—a formal proof environment for UML/OCL. In: Fiadeiro, J., Inverardi, P. (eds.) Fundamental Approaches to Software Engineering (FASE08). Lecture Notes in Computer Science, vol. 4961, pp. 97–100. Springer, Heidelberg (2008)
7. Filliâtre, J.-C., Marché, C.: The Why/Krakatoa/Caduceus platform for deductive program verification. In: Damm, W., Hermanns, H. (eds.) CAV. Lecture Notes in Computer Science, vol. 4590, pp. 173–177. Springer, Berlin (2007)
8. Gouda, M.G., Liu, X.-Y.A.: Firewall design: consistency, completeness, and compactness. In: International Conference on Distributed Computing Systems, pp. 320–327, 2004
9. Jacobs, B., Poll, E.k: Java program verification at Nijmegen: developments and perspective. In: Futatsugi, K., Mizoguchi, F., Yonezaki, N. (eds.) ISSS. Lecture Notes in Computer Science, vol. 3233, pp. 134–153. Springer, Berlin (2003)
10. Kaufmann, M., Moore, J.S.: ACL2. http://www.cs.utexas.edu/users/moore/acl2/
11. Marché, C., Paulin-Mohring, C.: Reasoning about Java programs with aliasing and frame conditions. In: Hurd, J., Melham, T.F. (eds.) TPHOLs. Lecture Notes in Computer Science, vol. 3603, pp. 179–194. Springer, Berlin (2005)
12. Meyer, J., Poetzsch-Heffter, A.: An architecture for interactive program provers. In: TACAS '00: Proceedings of the 6th International Conference on Tools and Algorithms for Construction and Analysis of Systems, pp. 63–77. Springer, London, UK (2000)
13. Moore, J.S.: Proving theorems about Java and the JVM with ACL2. In: Models, Algebras and Logic of Engineering Software, pp. 227–290. IOS Press, Amsterdam (2003)
14. Moscow, ML.: http://www.dina.dk/sestoft/mosml.html
15. Naraschewski, W., Wenzel, M.: Object-oriented verification based on record subtyping in higher-order logic. In: 11th International Conference on Theorem Proving in Higher Order Logics. LNCS, ANU, vol. 1479, pp. 349–366. Springer, Berlin (1998)
16. Nipkow, T., von Oheimb, D., Pusch, C.: μJava: Embedding a programming language in a theorem prover. In: Bauer, F.L., Steinbrüggen, R. (eds.) Foundations of Secure Computation. NATO Science Series F: Computer and Systems Sciences, vol. 175. IOS Press, Amsterdam (2000)
17. OMG: Unified modeling language. http://www.omg.org/
18. Pike, L., Shields, M., Matthews, J.: A verifying core for a cryptographic language compiler. In: ACL2 '06: Proceedings of the Sixth International Workshop on the ACL2 Theorem Prover and its Applications, pp. 1–10. ACM, New York, NY, USA (2006)
19. Poetzsch-Heffter, A., Müller, P.: A programming logic for sequential java. In: ESOP '99: Proceedings of the 8th European Symposium on Programming Languages and Systems, pp. 162–176. Springer, London, UK (1999)

20. Rajan, S.P.: Executing HOL specifications: towards an evaluation semantics for classical higher order logic. In: HOL'92: Proceedings of the IFIP TC10/WG10.2 Workshop on Higher Order Logic Theorem Proving and its Applications, pp. 527–536. North-Holland/Elsevier, Amsterdam (1993)
21. Schirmer, N.: A verification environment for sequential imperative programs in Isabelle/HOL. In: Logic for Programming, AI, and Reasoning. LNAI, vol. 3452, pp. 398–414. Springer, Berlin (2005)
22. The HOL System.: http://hol.sourceforge.net/
23. van den Berg, J., Huisman, M., Jacobs, B., Poll, E.: A Type-theoretic memory model for verification of sequential java programs. In: WADT '99: Selected Papers from the 14th International Workshop on Recent Trends in Algebraic Development Techniques, pp. 1–21. Springer, London, UK (2000)
24. von Oheimb, D.: Hoare logic for java in Isabelle/HOL. Concurr. Comput. Pract. Exp. **13**(13), 1173–1214 (2001)
25. Warmer, J., Kleppe, A.: The Object Constraint Language: Precise Modeling with UML. Addison-Wesley, Reading (1999)
26. Wilding, M., Greve, D., Hardin, D.: Efficient simulation of formal processor models. Form. Methods Syst. Des. **18**(3), 233–248 (2001)
27. Yatake, K., Aoki, T., Katayama, T.: Implementing application-specific object-oriented theories in HOL. In: Van Hung, D., Wirsing, M. (eds.) ICTAC. Lecture Notes in Computer Science, vol.3722, pp. 501–516. Springer, Berlin (2005)

Author Biographies

Kenro Yatake received B.S. from Tokyo Institute of Technology (2000), M.S. and Ph.D. from Japan Advanced Institute of Science and Technology (2002, 2006). His professional career is a Researcher of Japan Advanced Institute of Science and Technology (2006), and an Assistant Professor of Japan Advanced Institute of Science and Technology (2007, 2008). He is currently the Research Assistant Professor of Japan Advanced Institute of Science and Technology (2009). His research interests are theorem proving of software applications and model checking of embedded systems.

Takuya Katayama received B.E., M.E. and Ph.D. from Tokyo Institute of Technology (1962, 1964, 1971). His professional career is an Associate at Tokyo Institute of Technology (1967), an Associate Professor at Tokyo Institute of Technology (1974), a Professor at Tokyo Institute of Technology (1985), a Professor at Japan Advanced Institute of Science and Technology (1991–2007), a Dean of school of information science at Japan Advanced Institute of Science and Technology (1991–1993, 1997–1999), and a Director of Library at Japan Advanced Institute of Science and Technology (1999–2003). He is currently the president of Japan Advanced Institute of Science and Technology (2008). His research interests are object-oriented methodology, software evolution, fault-tolerant software, and formal methods.