Translating HOL to Specifications for the Model Checker SMV

Dan Zhou[1] and Paul E. Black[2]

[1] Department of Computer Science and Engineering, Florida Atlantic University
dan@cse.fau.edu
[2] National Institute of Standards and Technology
paul.black@nist.gov

Abstract. HOL and SMV have two radically different formal specification languages which are good at describing different aspects of systems and have very different analysis tools. We would like to integrate them to make full usage of their respective capabilities. One step towards this integration is automatically translating specifications written in HOL into SMV. We argue for the need of such an integration and translation for specification-based testing. We look at the differences between HOL and SMV and specify mechanical translation methods which are appropriate for modeling and testing secure operating systems.

1 Introduction

The unambiguity and rigorousness of formal methods promise the possibility of engineering high-assurance systems—but can we incorporate formal methods into engineering practice in such a way that it benefits instead of hinders system development?

1.1 The Need for Methods Integration

One standard method of assuring the quality of systems is through testing. Specification-based testing generates tests from descriptions of desired system functions and behavior. As a reference for testing the correctness of systems, specifications themselves are subject to analysis and testing.

Formal specification methods, based on mathematical symbols and logic, provide a description of system requirements that is both clear and precise [5]. They add assurance to system development by clearly stating the system requirements (the desired functions and behavior) and design. Facilitated by logic rules, system behavior can be calculated and predicted from formal specifications of computer systems. We can achieve a higher level of assurance by integrating formal specification methods with test-case generation.

There have been studies investigating formal specification-based testing. For example, Ammann, Black, and Majurski studied model checking for test generations [1] and Stocks and Carrington studied testing based on Z specifications [4].

However, no formal method alone is sufficient in addressing the need of large systems [2, 3]. Each formal method is best at modeling and analyzing some aspects of a system. Test cases generated from different formal methods look at different aspects of a system. Therefore, integrating two testing methods provides higher system coverage than each method alone. To our knowledge, integrating testing based on different formal methods has not been widely studied.

1.2 Integrated Automated Test Generation Methodology

A system specification can be divided into dynamic (or behavioral) and static (or functional) aspects, a standard division in object-oriented technology. Some parts of the system are purely *functional or combinatorial*; the output is entirely determined by the input. Other parts of the system have feedback; the result depends on what has come before. This feedback necessarily implies some concept of time: the series of inputs preceded the output. It often is convenient and reasonable to model this with a series of discrete time steps. We call the part of the system best understood with feedback or state *dynamic*, and the part of the system which can be well understood functionally *static*.

For example, in an operating system, a file can change from being readable to not readable, and back. The conditions under which it changes may be very complex. We could model the conditions as a set of static rules, best analyzed with a theorem prover to explore the behaviors of a function with vast input. We could model the dynamic state of the file as a state machine with a few simple rules formally proved to be a sound abstraction of complex rules.

Our integrated test-case generation methodology is based on the division of system models into functional and behavioral descriptions. We use two formal methods for describing systems and generating tests: model checker SMV describes the dynamic behavior of systems using the state machine approach, and theorem prover HOL describes the static behavior of systems by defining data types and functions operating on those types.

Our framework for test-generation is shown in Figure 1. It consists of five steps: (1) We model a system in HOL because HOL's specification language is based on functions and types, which is similar to high-level programming language. (2) We convert the behavioral description part of the system model to SMV through an HOL-to-SMV translator. We make this translation to take advantage of the methods and tools that are available for automated test generation in SMV. (3) Next we generate test cases for the static model (HOL specification) and (4) for the dynamic model (SMV specification). (5) The final step of this methodology is to integrate test results of these two sets. We are in the process of constructing components for this framework.

1.3 Organization of the Paper

In this paper we present a key component in the framework, a specification of HOL to SMV translation. We use a secure operating system as a running example throughout the paper.

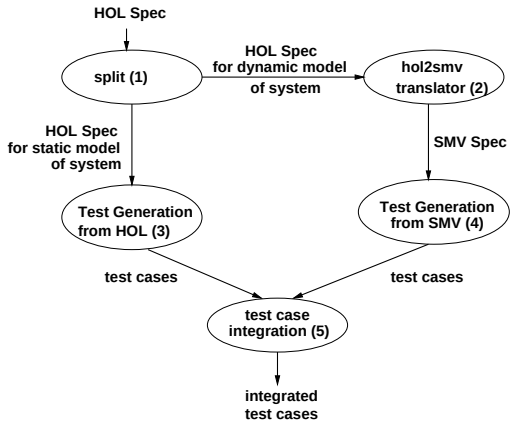

Fig. 1. Specification-Based Test Generation Process

We first describe a simplified view of a secure operating system model and HOL specification of the model in Section 2. In Section 3 we give an overview of HOL-to-SMV translation. We describe in detail the translation of HOL data types and functions in Section 4 and Section 5, respectively.We conclude in Section 6.

2 Secure Operating System Model

Our trusted operating system provides two mechanisms for controlling shared access to information. Discretionary access control (DAC), based on user identity, is the standard UNIX access control mechanism. Mandatory access control (MAC) is where control to information is governed by the nature of the information.

An operating system consists of objects and subjects. Objects such as regular files and processes are the resources to be controlled. Subjects are active entities that seek access, such as *read* or *write*, to objects.

2.1 DAC Attributes

UNIX-based systems provide DAC through assigning ownership to subjects and objects, and permission bits to objects based on the identities of their owners. Users are clustered into groups for easy management. Permission bits on objects indicate the access rights (read, write, and execute) the owner of the object (user), the group that the user belongs to (group), and the rest of the world (other), have to the object. The HOL record type *PBS*

```
PBS = <|r: bool; w: bool; x: bool|>
```

models the permission bits. The HOL record type *DAC*

```
DAC = <| up: PBS; gp: PBS; op: PBS|>
```

represents permission bits for users, group, and the world. In HOL definition, the symbol `<| ... ... |>` represents a record and ";" is a field delimiter.

2.2 MAC Attributes

To enforce MAC, subjects and objects are classified according to clearance and sensitivity level. A subject with a low clearance level cannot read an object with a higher sensitivity level. Subjects and objects are also categorized into compartments to model the *need-to-know* concept. A subject is not authorized to access an object unless it has a need to know the object. Classification and clearance labels are hierarchical, while compartments are not.

MAC Labels Our operating system provides four types of MAC labels: sensitivity label (SL), clearance label (CL), information label (IL), and integrity label (TL). SLs and CLs consist of classification and compartment. The HOL record type *SCLabel* represents both SLs and CLs:

```
SCLabel = <|class: Class; comp: Comp set |>
```

where types *Class* and *Comp* represent classifications and compartments. The construct *set* is a type constructor.

The HOL record types *ILabel* and *TLabel* represent IL and TL in a similar way.

We define a type *Label* to encompass all types of MAC labels:

```
Label = scLabel of SCLabel | iLabel of ILabel | tLabel of TLabel
```

Classifications System administrators can set the values of classifications and compartments. As an example, we define *Class* as an enumeration type with three values—*classTS*, *classS*, *classU*:

```
Class =  classTS | classS | classU
```

Classification *classTS* is higher in hierarchy than *classS*, which in turn is higher than *classU*. In HOL we define two relations on types *Class*: *classGT* and *classGTE*. Predicate *classGT c1 c2* is true if and only if *c2* is higher in hierarchy than *c1*:

```
(classGT classTS classS = T) /\
(classGT classS  classU = T) /\
(classGT classTS classU = T) /\
(classGT _ _ = F)
```

where `classGT _ _ = F` states that the default value is *F*. Predicate *classGTE c1 c2* is true if and only if *c1* is not higher in hierarchy than *c2*:

```
classGTE c1 c2 = classGT c1 c2 \/ (c1 = c2)
```

Compartments We define *Comp* as an enumeration type with four values: *NIST*, *ITL*, *FAU*, and *CSE* for compartments:

```
Comp = NIST | ITL | FAU | CSE
```

Relation among MAC Labels MAC decisions are made based on the relationship among MAC labels. A label *l1* dominates label *l2* if the classification of *l1* is at least as high as that of *l2*, and if the compartment set of *l1* is a superset of that of *l2*. In the case that one of the labels is a TL, only the classifications of labels are compared. HOL function *RLLDOM* describes this relation:

```
(RLLDOM (tLabel tl1) l3 =
     classGTE tl1.class (classOL l3)) /\
(RLLDOM l4 (tLabel tl2) =
     classGTE (classOL l4) tl2.class) /\
(RLLDOM l5 l6 =
     (classGTE (classOL l5) (classOL l6)) /\
     (compOL l6) SUBSET (compOL l5))
```

where *compOL* and *classOL* are functions that retrieve the compartment and the classification from a label, respectively. Function *RSLSL* defines the dominance relation between two SLs:

```
RSLSLDOM sl1 sl2 = RLLDOM (scLabel sl1) (scLabel sl2)
```

3 Overview of HOL to SMV Translation

We write specifications of a system in HOL and then translate the dynamic behavior part of the specification to SMV. Specifying dynamic behavior directly in SMV can be tedious because the level of SMV description is rather low. HOL, on the other hand, provides a higher level of language constructs for describing systems. HOL descriptions are closer to the level of reasoning that engineers are familiar with, though much more rigorous. In this section we give an overview of SMV and then sketch the HOL-to-SMV translator.

3.1 Overview of SMV

SMV is a symbolic model checker for the temporal logic Computational Tree Logic (CTL), and that the systems to be verified are finite state transition systems described in SMV's own input language. An input consists of two parts. One part is a state machine defined in terms of variables, initial values for the variables, and a description of the conditions under which variables may change value and what values they may take. The other part is a set of temporal logic clauses, in CTL, expressing invariant conditions and temporal logic constraints on possible execution paths. Conceptually, SMV visits all reachable states and verifies that the invariants and constraints are always satisfied.

A basic unit of definition in SMV is a module. A module is a named chip unit that has input pins and output pins, which are parameters or external variables to the unit. A module also maintains internal memory through internal variables. Variables can change values either instantly or in synchronization with a clock, through a next-state statement.

SMV modules can define both data types and functions. For example, module `ADD` sets `c` to the sum of its inputs:

```
MODULE ADD (a, b, c)
DEFINE
  c := a + b;
```

Module `PBS` defines a data structure for UNIX file permission bits:

```
MODULE PBS
VAR
  r: boolean;
  w: boolean;
  x: boolean;
```

3.2 Overview of Translation

There are limitations in what we can translate because of the limited expressive power of SMV compared to HOL.

HOL and SMV provide rather different language constructs and support. HOL provides much more support than SMV, hence a translator needs to implement the support that is available in HOL but not in SMV. Two such supports are equivalence checking of objects and object copying functions.

In SMV, parameters are passed to modules by reference. Modules can only be instantiated as separate entities. They cannot appear in other types of expressions. Only variables with simple, predefined values are allowed in expressions other than simple instantiation. This characteristic of SMV makes it necessary to use internal variables for module instantiation. It is a major limitation to specification of complex systems in SMV. In essence we need to flat out a whole system in terms of simple, predefined types. One immediate effect is that SMV does not support recursive definitions of data types or functions. A more serious result is that SMV does not scale up well.

SMV is not a strongly-typed language in that an instantiation of a module would accept actual parameters of different types in place of a formal parameter as long as an actual parameter has all the pins referred to in the definition of the module. On the contrary, HOL is a strongly typed language. In translating HOL to SMV, we take advantage of SMV's lack of strong type checking to represent polymorphic functions.

At this point, we translate only a subset of HOL specifications to SMV. This subset is described in the next two sections.

As a practical implementation of the translation of HOL specifications to SMV, we add translation capability to HOL parser to take advantage of the

parsing done by HOL. The enhanced HOL parser translates HOL phrases to SMV after it parses the phrase. Currently this translation is automatically done when an appropriate HOL phrase is parsed. We envision that later the translation can be turned on and off using a flag.

4 Translation of Data Types

Every HOL type is translated into an SMV module. We define some standard functions in SMV for each type translated from HOL. The functions that we define include equivalence checking of types and type constructions in SMV. This is necessary because SMV does not support modules in expressions other than simple instantiation. In this section, we first look at type definition in HOL, then describe how to translate HOL types to SMV.

4.1 Type Definitions in HOL

The basic specification elements in HOL are type and function definitions. HOL provides a wide range of ways to construct new types.

Simple Enumerate A type can be defined by listing distinct values of the type. For example, the compartment of a sensitivity label is defined as:

```
Comp = NIST | ITL | FAU | CSE
```

These values are referred to as null type constructors for type *Comp* because they build an instance of enumerate type *Comp* from null.

Simple Type You can also define type with non-null type constructors. HOL provides some pre-defined type constructors: *set*, *record*, *list*, etc. These constructors construct a new type from existing types in predefined ways.

For example, *Comp set* constructs a type whose values are sets of objects of type *Comp*. Expression

```
PBS = <|r: boolean; w: boolean; x: boolean|>
```

constructs a record type *PBS*, with fields r, w, and x having boolean values.

You can also construct types using user specified type constructors as follows:

```
ty_new = C1 of ty1 => ty2
```

where *ty_new* is a new type being defined, *ty1* and *ty2* are existing types, and *C1* is a user specified type constructor, which constructs an object of new type *ty_new* from one object of type *ty1* and one object of *ty2*. We call *ty1* and *ty2* the component types of *ty_new*.

Mixed Type We can mix simple enumerate types and simple types to construct new types. For example, in the HOL type definitions for MAC labels

```
SCLabel = <| class : Class; comp  : Comp set |>;
Label   = scLabel of SCLabel | iLabel of ILabel | tLabel of TLabel
```

Types *SCLabel*, *ILabel* and *TLabel* are built through simple type construction. Type *Label* is built through mixed type construction.

Complex Type In HOL we can also take a subset of an existing type and make that into a new type. This is one of the most complex forms of type definition. Another complex way to define a type is through recursive type definition.

4.2 Type Translation

In this study, we translate only a subset of the type definitions to SMV, simple enumeration and some of the simple and mixed type constructions. We ignore the translation of complex type constructions. We also ignore any definition related to a variables type.

Simple Enumerate Type An HOL enumerate type with n null constructors is translated to a module with one internal variable which can have any of the n values. Each null constructor is a possible value. For example, the HOL type *Comp* listed on the previous page is translated to SMV module `Comp`:

```
MODULE Comp
VAR
  Comp: {NIST, ITL, FAU, CSE};
```

Record Type An HOL record type is translated to an SMV module with internal variables. A field *fname* of type *ty* becomes an internal variable `fname` of type *ty*. For example, the HOL type *DAC*

```
DAC = <| up: PBS; gp: PBS; op: PBS|>
```

is translated to the SMV module `DAC`

```
MODULE DAC
VAR
  up:PBS;
  gp:PBS;
  op:PBS;
```

where `PBS` is a module for type *PBS*.

Set Type Some set types are also translated to SMV modules. The HOL type *ty set* is translated to an SMV module with n internal variables, where n is the number of distinct values of type *ty*. These internal variables are boolean variables named after the values of *ty*. A variable with the value *true* indicates the presence of the element in a set, and the value $false$ indicates the absence of the element.

Module `Comp_set` is an example of a set of enumerate type *Comp*:

```
MODULE Comp_set
VAR
  NIST:boolean;
  ITL:boolean;
  FAU:boolean;
  CSE:boolean;
```

Translation of more complex set types are not described because of space limit.

Mixed Type In translating types with mixed type construction, we treat non-null type constructors separately from null type constructs. We group all the null type constructors together and represent them by one internal variable *const*, which can have any value in the group. For non-null type constructor *Cn*, which constructs the new type from a single existing type *ty*, we represent it as an internal variable `Cn` of type `ty`. We then define an internal variable *vartype* as a selector, indicating which variable holds the data for the module. If there are n non-null type constructors, the selector can have $n + 1$ different values.

For example, HOL type *Label* on page 4 is translated to the SMV module `Label`:

```
MODULE Label
VAR
  scLabel: SCLabel;
  iLabel: ILabel;
  tLabel: TLabel;
  vartype: {scLabel_ty, iLabel_ty, tLabel_ty};
```

4.3 Type Construction Functions in SMV

We define modules for object construction and equivalence checking during translation because SMV does not provide them. We look at two types of object constructors. A copy constructor constructs an object that is identical to an existing object. An assembly constructor constructs a new object from component objects. We only discuss the copy constructor here.

Simple Enumerate A copy constructor makes a copy of an object of simple enumerate type by copying the value of the only internal variable from the existing object. Module `Class_constr` is a copy constructor for type *Class*:

```
MODULE Class_constr (x, z)
DEFINE
  z.Class := x.Class;
```

Record Type A copy constructor for record type simply makes copies of all the internal variables. Copy constructor module `DAC` for type *DAC* is an example:

```
MODUEL DAC_constr (x, z)
VAR
  y1: PBS_constr (x.up, z.up);
  y2: PBS_constr (x.gp, z.gp);
  y3: PBS_constr (x.op, z.op);
```

where `PBS_constr` is a copy constructor for *PBS*.

Set Type Copy constructors of sets are similar to what we discussed so far: simply copy all the internal variables from an old object to a new object (which are of type *boolean*).

Mixed Type A copy constructor for a mixed type simply copies the values of all the interval variables from an existing object. Module `Label_constr` is a copy constructor for the mixed type *Label*:

```
MODULE Label_constr (x, z)
VAR
  y1: SCLabel_constr (x.scLabel, z.scLabel);
  y2: ILabel_constr  (x.iLabel,  z.iLabel);
  y3: TLabel_constr  (x.tLabel,  z.tLabel);
DEFINE
  z.vartype := x.vartype;
```

4.4 Equivalence Testing Functions in SMV

SMV has no built-in support for equivalence testing of user-defined objects. We create modules to check the equality of objects for every user-defined type by iteratively testing its component types objects.

Simple Enumerate, Record and Set Types To test the equivalence of two variables of types simple enumerate, record, or set, we write a module to test the equivalence of all the fields. If the fields of a record type are of user-defined types, we employ the equivalence testing modules defined for these user-defined types. Module `DAC_eq` is an equivalence test for record type *DAC*:

```
MODULE DAC_eq (x1, x2, z)
VAR
  y1: PBS_eq (x1.up, x2.up, z1);
  y2: PBS_eq (x1.gp, x2.gp, z2);
  y3: PBS_eq (x1.op, x2.op, z3);
DEFINE
  z := z1 /\ z2 /\ z3;
```

Mixed Type When checking the equivalence of two objects of a type constructed through mixed type construction, we find out which internal variable holds the value for the objects and compare the two corresponding internal variables.

For example, module `Label_eq` defined equivalence checking for type *Label*:

```
MODULE Label_eq (x1, x2, z)
VAR
  y1: SCLabel_eq(x1.scLabel, x2.scLabel, z1);
  y2: ILabel_eq (x1.iLabel, x2.iLabel, z2);
  y3: TLabel_eq (x1.tLabel, x2.tLabel, z3);
DEFINE
  z :=  x1.vartype=x2.vartype &
        case
        x1.vartype=scLabel_ty : z1;
        x1.vartype=iLabel_ty  : z2;
        x1.vartype=tLabel_ty  : z3;
        esac;
```

where SCLabel_eq, ILabel_eq, and TLabel_eq are equivalence checking functions for types *SCLabel*, *ILabel*, and *TLabel* respectively. They are defined similarly as module `DAC_eq` on page 10.

4.5 Set Operations

We define modules for four set operations in addition to what we defined earlier for set types: equivalence check, subset operation, union of two sets, and set element checking. Set is treated specially because our HOL model of PitBull uses it extensively.

Set $x1$ is a **subset** of set $x2$ if every element of $x1$ is also an element of $x2$. As an example in SMV, we define module `Comp_set_SUBSET`:

```
MODULE Comp_set_SUBSET (x1, x2, z)
DEFINE
  z :=  (x1.NIST -> x2.NIST) &
        (x1.ITL  -> x2.ITL)  &
        (x1.FAU  -> x2.FAU)  &
        (x1.CSE  -> x2.CSE);
```

A **union** of the two sets $x1$ and $x2$ is such that all the members in $x1$ and $x2$ are also members of the union set. Module `Comp_set_UNION` defines the union of two sets of *Comp*:

```
MODULE Comp_set_UNION (x1, x2, z)
DEFINE
  z.NIST := x1.NIST | x2.NIST;
  z.ITL  := x1.ITL  | x2.ITL;
  z.FAU  := x1.FAU  | x2.FAU;
  z.CSE  := x1.CSE  | x2.CSE;
```

To check if a named **element** is in a set, we only need to know if the value of the named field is *true*:

```
MODULE in_Comp_set (x1, x2, z)
DEFINE
  z := case
       x1.Comp=NIST : x2.NIST;
       x1.Comp=ITL  : x2.ITL;
       x1.Comp=FAU  : x2.FAU;
       x1.Comp=CSE  : x2.CSE;
       esac;
```

5 Translation of Functions

One key factor that influences the HOL-to-SMV translation is that HOL provides some help for user-defined types while SMV does not. For example, if *c1* and *c2* are two variables of type *Class*, in HOL we can write $c1 = c2$, while in SMV we need to write our own module for checking the the equivalence of these two variables. The translator needs to provide equivalence checking for user-defined types.

Another factor that influences the translation is the difference between HOL's and SMV's models on functions. In HOL we can define a function f that takes input x and returns an output y in the format $y = f_{HOL}(x)$. We can compose functions with expression $z = f(g(x))$. In SMV we can only define function f as a module in a format that is essentially $f_{SMV}(x, y)$. To achieve the effect of a composed function, we use variable y to connect two functions $f(x, y)$ and $g(y, z)$. We model functions in SMV through modules. To define a function that calls other functions we declare internal variables as instantiations of modules that define the called functions and then use simple operators to combine outputs from these modules.

5.1 Function Definition in HOL

The simplest form of an HOL function definition is a single clause composed of arithmetic, logic, and user-defined operations on a set of arguments. You can also define functions in HOL based on values of arguments. This type of definition is comprised of multiple clauses conjuncted together, with each clause defining the behavior of the function for some particular values of input arguments. A special case of the multi-clause function definition is when one argument is of a supertype. In this case you can have a definition where each clause defines the behavior of the function for some subtypes. This type of definition in effect defines polymorphic functions.

5.2 Translation to SMV

Single-Clause Definitions We translate the simplest form of HOL function definition to a single SMV module. This module comprises: (1) same number of

input parameters as the HOL function; (2) one output parameter; and (3) internal variables, each of them as an instantiation of a module that is a translation of an operation in the HOL function definition; (4) one SMV DEFINE clause to define the value of the output parameter. Item (3) is necessary because SMV only supports module invocations in simple instantiations.

Function *classGTE* represents the simplest form of HOL function definition:

```
classGTE c1 c2 = classGT c1 c2 \/ (c1 = c2)
```

In function *classGTE*, there are two inputs, *c1* and *c2*, and two operations, *classGT* and the equivalence checking function of two *Class* objects. Module `classGTE` is the translation of HOL function *classGTE*. The module has three external parameters and two internal variables which are instantiations of modules `classGT` and `Class_eq`. The output of the module `classGTE` is the logic *or* (`|` in SMV) of the outputs of `classGT` and `Class_eq`

```
MODULE classGTE (x1, x2, z)
VAR
  y1: classGT (x1, x2, z1);
  y2: Class_eq (x1, x2, z2);
DEFINE
  z:= z1 | z2;
```

Value-Based Multi-Clause Definitions In translating functions with multiple clauses where the selection of clauses is based on input values, we use *case* statements for clause selection and translate each clause as for single-clause definitions.

For example, HOL function *classGT* on page 4 is translated to SMV module `classGT`:

```
MODULE classGT (x1, x2, z)
DEFINE
  z:=   case
        x1.Class=classTS & x2.Class=classS : TRUE;
        x1.Class=classS  & x2.Class=classU : TRUE;
        x1.Class=classTS & x2.Class=classU : TRUE;
        1 : FALSE;
        esac;
```

Type-Based Multi-Clause Definitions HOL is a strong typed language; every function in HOL has a unique type. An SMV module, on the other hand, can have any types of external variables as long as these variables have pins referred to in the definition of the module.

For example, to retrieve the compartment information of a label in HOL, we define a polymorphic function *compOL* based on different component types or sub-types of *Label*:

```
(compOL (scLabel scl) = scl.comp) /\
(compOL (iLabel il) = il.comp)
```

In SMV, module `compOL` models the same operation as a polymorphic function on sensitivity labels and on information labels:

```
MODULE compOL (x, z)
VAR
  y1: Comp_set_constr (x.comp, z);
```

where parameter x represents the input which can be either *SCLabel* or *ILabel*. Variable z gives the output which is a set of compartments. Module `Comp_set_constr` is a copy constructor for type *Comp set*.

When an HOL function is defined based on the type of input variables, we translate the HOL function into several SMV modules that are variants of the same function. Each module represents a distinct clause of the HOL function. For example, HOL function *RLLDOM* on page 5 represents the dominance relation between two DAC labels. This definition breaks the operation into three cases: (1) the first argument is an integrity label; (2) the second argument is n integrity label; and (3) all the other cases. Therefore we define three modules `RLLDOM_TLabel_allo`, `RLLDOM_allo_TLabel`, and `RLLDOM_allo` for the three cases respectively:

```
MODULE RLLDOM_TLabel_allo (x1, x2, z)
VAR
  y1: classGTE (x1.class, x2.class, z);

MODULE RLLDOM_allo_TLabel (x1, x2, z)
VAR
  y1: classGTE (x1.class, x2.class, z);

MODULE RLLDOM_allo (x1, x2, z)
VAR
  y1: Comp_set_SUBSET (x2.comp, x1.comp, z1);
  y2: classGTE (x1.class, x2.class, z2);
DEFINE
  z := z1 & z2;
```

When a polymorphic function is used, the translator decides which actual implementation is called. For example, HOL function *RSLSLDOM* defined on page 5 is translated to module `RSLSLDOM` as follows, after we determine that the two arguments to *RLLDOM* are both of type *SCLabel*:

```
MODULE RSLSLDOM (sl1, sl2, z)
VAR
  y1: RLLDOM_allo (sl1, sl2, z);
```

6 Conclusion

Our research goal is to facilitate the construction of high-assurance secure systems. The objective of this project is to investigate the integration of test generation based on two formal methods, theorem prover HOL and model checker SMV. This integration is desirable because: (1) A system is best modeled from multiple view points: behavior (dynamic) and functional (static) aspects. (2) HOL provides theorem-proving ability to verify the correctness of the functionality of the system. SMV provides model checking ability to assure the correctness of system behavior.

In this paper we propose an integrated formal-specification based testing framework using HOL and SMV. One component in this framework is an HOL-to-SMV translator. We categorize HOL type and function definitions and define specification for the translator based on our categorization. We also provide support for user-defined types in translation because of the difference in language between HOL and SMV. We provide equivalence checking and copy construction of objects and standard set operations.

The translation is generally successful in modeling a secure operating system. However, there are limitations in what we can translate. First, we simply cannot translate a large HOL model into SMV because of the model size increase as it is translated. Second, because of the limited expressibility of SMV, some HOL language constructs are best left in HOL. Recursive data and function definitions in HOL are such examples.

There are also some subsets of HOL that we have not looked into because we have not run into them in our application. For example, variable and function types have not been investigated.

So far we have provided a specification of a translator. Our next step is to implement the translator. To complete the framework, we need further study on how to generate tests from HOL; how to effectively divide HOL specification into functional and behavioral aspects. We also need to experiment with the integration of tests generated from SMV and HOL.

References

1. Paul E. Ammann, Paul E. Black, and William Majurski. Using model checking to generate tests from specifications. Technical Report NISTIR 6166, U.S. National Institute of Standards and Technology, Gaithersburg, MD, USA, november 1998.
2. Edmund Clarke and Jeannette Wing. Formal methods: State of the art and future directions. *Report of the ACM Workshop on Strategic Directions in Computing Research, Formal Methods Subgroup*, August 1996. Available as CMU Computer Science Technical Report CMU-CS-96-178.
3. Klaus Schneider and Dirk W. Hoffmann. A hol conversion for translating linear time temporal logic to ω-automata. In Yves Bertot, Gilles Dowek, Andre Hirschowitz, Christine Paulin, and Laurent Thery, editors, *Theorem Proving in Higher Order Logics – TPHOLs'99*, number 1690 in Lecture Notes in Computer Science. Springer-Verlag, 1999.

4. Phil Stocks and David Carrington. A framework for specification-based testing. *IEEE Transactions on Software Engineering*, 22(11):777–793, November 1996.
5. Jeannette M. Wing. A specifier's introduction to formal methods. *IEEE Computer*, 23(9):8–24, September 1990.