Finite Model Generation and Formal Specification Development

Jian Zhang
Laboratory of Computer Science
Institute of Software, Chinese Academy of Sciences
P.O.Box 8718, Beijing 100080, P.R. China
Email: zj@ox1.ios.ac.cn

Abstract

Efficient supporting tools are very important to the adoption of formal methods. Conventionally such tools are mainly automated theorem provers. In this paper, we show that model generators can also be very helpful in obtaining formal specifications of software components. These tools help us to know more about certain properties of specifications, such as completeness and consistency.

1: Introduction

Specifications are often used in various stages of software development. In most cases, they are informal or semi-formal. But for decades, some researchers have been working on formal methods (which are based on certain mathematical theories). In the ideal situation, a user starts from a high-level formal specification, and then derives an executable program by correctness-preserving transformations; or the user writes a program and then proves it is correct with respect to the specification.

Although many people think that the above paradigm is impractical for developing large software systems, certain forms of formal specifications are still very useful. For example, programmers often write assertions. Run-time checking of assertions may help to discover potential bugs. They also make a program easier to understand. The assertion mechanism is supported by most C/C++ compilers, and it is even included in the definitions of some object-oriented programming languages like Eiffel [4] (see also `http://www.eiffel.com`).

One benefit of the object-oriented paradigm is reusability. Programmers rely heavily on class libraries. They also write their own definitions of classes and provide them to other programmers. In such cases, it is quite important to know under what conditions a software component can be used. [5] describes software construction as a sequence of documented contracting decisions. A software engineer divides his task into a set of subtasks, fulfilling some subtasks by himself, and contracting others out. The main motivation of this paper is how to get the contract right, or how to create good specifications of components.

More often than not, it is difficult to write good formal specifications. First version of a specification can be incorrect or incomplete. Thus supporting tools are necessary. Conventionally theorem provers are used to validate the specifications. In this paper, we study a different kind of tool, i.e., model generators for first-order logic. They can be used to check the satisfiability of logical formulas. With a model generator, we can test a specification written in the first-order logic.

The paper is organized as follows. In the next section, we give a brief introduction to first-order logic and show that many properties of objects can be formally specified by logical formulas. Then in §3, we discuss the satisfiability problem in first-order logic, and describe how to satisfy a set of formulas in finite domains (or, how to find finite models). In §4, we illustrate the usefulness of model finding tools by several examples. Finally, we discuss the limitations of our work and directions for further research.

2: Formal Specifications and First-order Logic

There are several different formal specification methods [7]. Here we are more interested in the so-called property-oriented methods. With such a method, one specifies the behavior of objects indirectly by stating a set of axioms. In this paper, we assume that the axioms are formulas in many-sorted first-order predicate logic.

First let us briefly review some basic concepts of mathematical logic. In a first-order language, we have a set of *variables*, a set of *function symbols* (including *constants*) and a set of *predicate symbols.* A *term* is a variable or a constant, or a function symbol applied to some terms. In the many-sorted case, each term is of some type or sort. Applying a predicate symbol to an appropriate number of terms, we can get a *formula.* Formulas can be combined by logical connectives like $\vee, \wedge, \neg, \rightarrow, \ldots$ They can also be preceded by the quantifier $\forall x : T$ (for any x of type T) or $\exists x : T$ (there exists some x of type T), where x is a variable. For a simple example, the following is a formula (and we denote it by φ):

$$\forall x, y : Person\ [\ Father(x,y)\ \rightarrow\ Male(x)\ ].$$

Here $Person$ is a type or sort, $Father$ and $Male$ are predicate symbols. (Intuitively, $Father(x,y)$ means x is y's father and $Male(x)$ means x is male.) Some function/predicate symbols have special meanings, like $+$ and $\geq$. We can also write a term in infix notation, e.g., $a+b$ instead of $+(a,b)$.

First-order logic has been chosen as the formal basis for the Object-oriented Systems Analysis (OSA) model [1]. It can also be used to specify certain properties of classes in object-oriented programs.

Example 1. Stacks. Typically the class "stack" has the following member functions: *push, pop, top, empty.* The results of these functions are closely related. For instance, after an item is pushed on top of a stack, the stack must not be empty. To express this fact, we might write the following statements in a C++ program:

$$\ldots;\ \ s.push(a);\ \ assert(!\ s.empty());\ \ \ldots$$

Alternatively, the relationship may be described by a logical formula:

$$\forall s : stack, \forall x : item,\ \ [\ \neg\ empty(push(x,s))\ ].$$

Here *push* is a binary function symbol, *empty* is a unary predicate symbol, and $\neg$ means "not". We can also write other formulas such as,

$$\forall s : stack, \forall x : item,\ \ [\ \ top(push(x,s)) = x\ ].$$

It should be noted that, for simplicity, we did not consider the case when a stack becomes full.

Appropriate assertions make programs easier to understand and increase their robustness. Syntactically most assertions are just Boolean expressions. Some assertions are placed in the middle of a function's body (as in the above example), while others serve as pre-post conditions of member functions. A precondition says what must hold when the function is called, and a postcondition says what should hold after the function terminates. The function's behavior can be expressed by a logical axiom of the form $PreCond \rightarrow PostCond$. But the value in $PostCond$ should be the result of the function.

Example 2. Consider the *deposit* function in the bank *Account* class. In an Eiffel-like language, we can state the pre-post conditions as follows.

```
deposit(sum:  Integer) is
   require
     sum ≥ 0
```

```
ensure
  bal = old bal + sum
end;
```

Here the keyword **require** introduces preconditions, and **ensure** introduces postconditions. In first-order logic, the above specification may be paraphrased as:

$$\forall x : account, \forall n : integer,$$
$$n \geq 0 \ \rightarrow \ bal(dep(x,n)) = bal(x) + n$$

Here $dep(x,n)$ denotes the result of depositing the amount n to the account x, and $bal(x)$ denotes the balance of x.

In addition to preconditions and postconditions, Eiffel supports *class invariants*. Such properties must be satisfied by all instances of the class. An invariant for the *Account* class might be: $bal \geq 0$. This can be expressed by the formula: $\forall x : account,\ bal(x) \geq 0$.

3: Satisfying Formulas in First-order Logic

3.1: Satisfiability and Models

Satisfiability and models are important concepts in mathematical logic. The reader may find their detailed definitions in standard textbooks. A set of axioms is consistent if it is satisfiable in some domain, or in other words, if it has a *model*. A domain is a non-empty set. It can be finite or infinite. In a model, every constant should be mapped to some element in the set, every function and predicate should be defined, and all the axioms should hold.

The formula $\forall x\,(x = x)$ is satisfied in any domain, so is the formula *true*. For another simple example, consider the formula φ in §2. Suppose the sort *Person* has two elements: { *John*, *Mary* }. Then the following tables define a model of φ:

	John	*Mary*
Male	*T*	*F*

Father	*John*	*Mary*
John	*F*	*T*
Mary	*F*	*F*

Here F and T denote falsity and truth, respectively.

3.2: Satisfiability Testing

The satisfiability problem is: Given a set of formulas, determine whether it is satisfiable or not. In the propositional logic, the problem is known as SAT, which is NP-complete. But for the first-order logic, there is no algorithm which, given an arbitrary formula, decides if it is satisfiable or not. So we have to consider restricted forms of the satisfiability problem in the first-order case. One possibility is to restrict the syntax of formulas. But many decidable subsets of first-order logic are not so expressive. Another way is to adopt some assumptions about the models. For instance, the semantics of *algebraic specifications* [2] is based on initial algebras. In this case, the elements of the domain are exactly those terms which are constructed from certain function symbols (called *constructors*), e.g., s_0, $push(i_0, s_0)$, $push(i_1, push(i_0, s_0))$, ... The related tools try to show consistency by deducing a set of formulas having some desired properties (like the Church-Rosser property).

In this paper, we restrict the models to be finite. This is quite natural for specifying finite sets and arrays. We denote the elements of a finite model by abstract names, like $e_0, e_1, \ldots$ Given a fixed positive integer n, it is always possible to determine if a set of axioms are satisfiable in an n-element domain. The straightforward way is using exhaustive search like the backtracking procedure. This can be enhanced by various techniques from Artificial Intelligence.

Based on a search procedure, we can design some tools which try to produce finite models of formulas. They are called *finite model generators*. Typically, their input consists of a set of first-order formulas, and the size of the domain. (In the many-sorted case, we have to specify the size of each sort.) The output is a finite model, i.e., a set of assignments which define the function and predicate symbols in the given formulas. For example, given the formula φ in §2 and the fact that there are 2 persons, such a tool will produce some 2-element models, one of which is shown in §3.1.

Currently there are some finite model generators available. FINDER [6] is one such program which was developed in the early 90's. We also implemented a tool called SEM [9], using some sophisticated techniques. We shall not describe the techniques in detail, since they are not the focus of this paper. SEM is very efficient on many benchmark problems [9]. Both FINDER and SEM are based on many-sorted first-order logic. In the input, a user should declare some sorts/types and some functions, and then give a set of clauses. (Clauses are a special form of formulas). The output consists of one or more models if they exist.

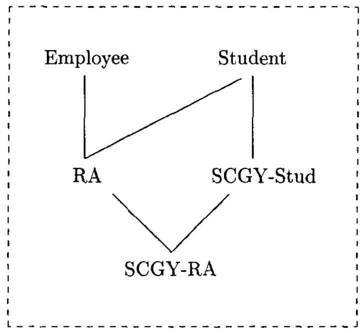
Sometimes there is no model having the given size. Such a result does not imply that the formulas are inconsistent, because there might be a model of a different size. So the exhaustive search method is incomplete, but it is still helpful in many cases, when small models exist.

In some sense, model generation is complementary to theorem proving. A theorem prover tries to show that a specification S has some property P, by proving the validity of the formula $S \rightarrow P$. But when S does not have the property, one may use a model generator to find a counterexample (i.e., a model of the formula $S \wedge \neg P$).

4: Evaluating and Developing Formal Specifications

A good specification should meet some criteria. At least it should be unambiguous and consistent. Ambiguity is avoided if every property is stated in a logic-based language. Consistency may not be so obvious in an object-oriented framework which allows for multiple inheritance, because a class can be derived from several different ancestors. Automated reasoning tools can be useful here. But first we have to flatten ([4], p.346) the classes.

Example 3. Consider research assistants (RAs) in universities. The *RA* class might be a descendent of the *Employee* class and of the *Student* class. Another class called *SCGY-RA* denotes RAs in the Special Class for Gifted Young (SCGY) of the China University of Science and Technology. It is derived from the *SCGY-Stud* class, which is in turn a subclass of *Student.*



Suppose the *Employee* class has an invariant *age* > 20 and the *SCGY-Stud* class has an invariant *age* < 20. Obviously the conditions on the *SCGY-RA* class are inconsistent, and we have to restructure the class hierarchy or reconsider the invariants. Such an *inconsistency* can be detected by a refutation-based theorem prover. On the other hand, the existence of models implies the *consistency* of a set of formulas, so a finite model generator may help us to make sure that the modified specification is consistent.

In addition to being consistent, a specification should also capture the user's requirements as much as possible, although it may be difficult to cover all aspects. Model generators may also be helpful here. In fact, existing tools have assisted mathematicians to get a better understanding of some abstract theories and to formulate interesting conjectures about the mathematical structures they study. See, for example, [8].

Given an initial specification S_0, we can try to generate some of its models, and then see if they are really what we want. Suppose there is one model which does not match our requirements, then we should modify S_0, and get a new specification S_1. This process is repeated, and finally we get a specification which is as good as possible.

Example 4. Specification of a generic sorting function[1]. The objects to be sorted may be integers or strings or ... But there should be some kind of order between them. For simplicity, let us denote it by $<$ (which means "less-than"). The syntax of a sorting function is given by $f : IndS \rightarrow ValS$. Here $IndS$ is a set of indices, say, $\{0, 1, \ldots, n-1\}$, and $ValS$ is a set of n values. One might specify the postcondition of f by the following formula (α):

$$\forall i, j : IndS \; [\; 0 \leq i < j < n \;\rightarrow\; f(i) \leq f(j) \;]$$

Let us test this specification with a finite model generator. Suppose the set $ValS$ consists of 3 values: a, b, c, with $a < b < c$. Then we should have $f(0) = a,\; f(1) = b,\; f(2) = c$. However, if we give the following formulas to the model generator SEM:

$$\begin{gathered} a < b < c \\ f(0) \leq f(1) \\ f(0) \leq f(2) \\ f(1) \leq f(2) \end{gathered}$$

then 10 models will be produced, including the following one:

$$f(0) = f(1) = f(2) = a$$

This does not match our concept of "sorting", and the specification is incomplete. The problem is, the output list should not only be sorted, but also be a permutation of the input. So we add another formula (β) to the postcondition:

$$\forall n : ValS \; \exists i : IndS \; [\; f(i) = n \;]$$

Now a finite model generator will produce only one model:

$$f(0) = a,\;\; f(1) = b,\;\; f(2) = c$$

which is the correct one. If we increase the number of values to 4 or 5, we still get only one model in each case. Thus we are quite confident that the two formulas (α) and (β) adequately specify the result of a sorting function.

Consistency and completeness are good properties of formal specifications in general. Moreover, specifications of object-oriented programs should satisfy some other constraints. As an example, the redefinition rule [5] says, if a subclass redefines a function, then its precondition should be weaker than the original, and its postcondition should be stronger. As explained at the end of §3.2, model generators can be used to produce counterexamples which show the violation of these rules. If such a violation occurs, one should reimplement the function.

5: Concluding Remarks

First-order logic may serve as a formal basis for object-oriented analysis and programming. Traditionally, automated support for this logic comes mainly from theorem provers. In this paper, we discuss a different kind of tools, namely, model generators, and argue that they are quite helpful in

[1]This example has been used in [3]. We assume that the objects are different from each other.

the development of formal specifications. Specifically, they can: (1) show the consistency of "good" specifications; (2) produce counterexamples for "bad" specifications; and (3) produce models which demonstrate interesting properties of an abstract theory. Using several examples, we described how to use model generators to find inaccuracies and incompleteness in some specifications.

Note that our work is based on a very simple model of object-oriented programming. It is not dependent on any particular programming language, although we talked about Eiffel more often than about other languages. Also, some practitioners may not fully agree with the author on certain points. They might say something like this: We do not want to write so many assertions, because of run-time cost. But that is only a matter of choice. Actually, some Eiffel classes in [4] are not adequately specified. For example, the *head* and *tail* routines of the *string* class have the same preconditions and postconditions.

So far, model generation programs [6, 9] have been used mainly for solving logical puzzles and problems from finite mathematics. This paper describes a different kind of application for such tools.

Certainly our approach has some limitations. The restriction to *finite* models is a reasonable assumption in many cases. But for some classes like stacks and queues, the algebraic specification approach may be better. Another shortcoming is that, it is not so convenient to describe some properties using classical logic. In this framework, you have to write many formulas to express some domain knowledge, and you have to define many common concepts and functions. In addition, existing model generators have much difficulty in dealing with arithmetic expressions. (This is also true for some algebraic specification tools which do not accept natural numbers directly. Instead, a user has to write $1 = succ(0)$, $2 = succ(1)$, ...)

As part of our future work, we shall study the integration with other formal methods, test larger examples, and improve our tools so that they can be used more conveniently.

Acknowledgments

This work was partially supported by the National High Technology Program under grant No. 863-306-05-03-3. Part of this paper was written during the author's visit to the Automated Reasoning Project of the Australian National University. We thank Kinh Nguyen for his comments.

References

[1] Clyde, S.W., Embley, D.W. and Woodfield, S.N., "Tunable formalism in Object-oriented System Analysis: Meeting the needs of both theoreticians and practitioners," *Proc. OOPSLA '92*, 452–465.

[2] Ehrig, H. and Mahr, B., *Fundamentals of Algebraic Specifications 1: Equations and Initial Semantics,* Springer, 1985.

[3] Gries, D. *Science of Programming* Springer, New York, 1981, Chapter 6.

[4] Meyer, B., *Object-Oriented Software Construction*, Prentice-Hall, 1988.

[5] Meyer, B., "Design by contract," in: *Advances in Object-Oriented Software Engineering,* (eds.) D. Mandrioli and B. Meyer, Prentice-Hall, 1992.

[6] Slaney, J., *FINDER, Finite domain enumerator: Version 3.0 notes and guide*, Automated Reasoning Project, Australian National University, 1993.

[7] Wing, J.M., "A specifier's introduction to formal methods," *IEEE Computer*, Sept. 1990, 8–24.

[8] Zhang, J., "Constructing finite algebras with FALCON," *J. Automated Reasoning,* 17(1): 1–22, 1996.

[9] Zhang, J. and Zhang, H., "SEM: a System for Enumerating Models," *Proc. IJCAI-95*, 298–303.