数理解析研究所講究録
第 1938 巻 2015 年 46-53

Verifying Correctness of Formal Specifications with an SMT Solver

Keishi Okamoto

Sendai National College of Technology,
okamoto@senda-nct.ac.jp

Abstract. Issues in software development is often due to requirements specifications. In this paper, we introduce a verification method for formal specifications with an SMT solver to assure quality of software in design process. In particular we introduce a simple case study in which we translate a specification to a verification condition formula in the language of an SMT solver Z3 and then we verify the formula with Z3.

1 Introduction

As software is becoming complicated, complex and omnipresent, there are many software failure cases. Therefore improving reliability of software must be required. Moreover, problems in software development is often due to requirements specifications. Indeed, Union of Japanese Scientists and Engineers, Software Quality Profession reports that problems due to requirements specifications accounts for 40% in the life cycle of development.

Formal methods are promising approaches to tackle the issues[1], in particular, SMT solvers have draws many researchers' attention[2]. SMT solvers are used for verifying correctness conditions for source codes. These conditions are called verification conditions. Our future goal is to develop a verification condition generator for specifications.

In this paper, we introduce a verification method for formal specifications with an SMT solver to assure quality of software in design process. In particular we introduce a simple case study in which we translate a formal specification with (an invariant,) a pre-condition and a post-condition to a verification condition formula in the language of an SMT solver Z3[3] and then we solve the satisfiability problem of the formula with Z3.

2 Formal Specifications

In this section, we introduce formal specifications and give definitions of correctness of formal specifications.

Formal methods are software development methods based on mathematical logic and computer science. Formal methods are applied to many systems, including software, to improve their reliability. *Formal specification* and *formal* *verification* are two major parts of formal methods. In formal specification, we describe specifications with a formal language based on mathematics and computer science[4, 5]. Then the resulting formal specifications are less ambiguous than traditional specifications and then we can verify many properties of formal specifications. In formal verification, we verify a model of a system, which we want to verify, with mathematical methods, for instance model checking[6, 7] and theorem proving[8, 9]. With model checking, we can automatically check a model, which is often a state transition system, of a system. With theorem proving, we can automatically or interactively prove a model of a system with mathematical formal proof.

In formal methods, we use mathematical objects (natural numbers, sets, tuples, functions, λ-terms etc.) to understand properties of source codes and specifications[1]. A mathematical object to understand a source code (a specification) is called a `model` of the source code (respectively the specification).

From the view point of *State-based Specifications*, we define a state in a transition system as a tuple of values of variables in a given specification, and we consider a specification as a *state transition system* which is the pair $\langle \texttt{State}, \texttt{Trans} \rangle$ of a set `State` of states and a transition relation `Trans` between states. In a state transition system, a state of a program is modeled as a tuple of values of global variables in the program and then an execution of a program is modeled as a relation between states. Therefore, an element in `Trans` represents an execution of an operation in a specification. For an element (s_1, s_2) of `Trans`, we call a *pre-state* for the first element s_1 and a *post-state* for the second element s_2.

We also introduce some terminologies. *Pre-Condition* is a condition that must always be true just prior to the execution of some section of a code or a specification. In other words, pre-condition is a condition at which a pre-state must satisfies. *Post-Condition* is a condition that must always be true just after the execution of some section of a code or a specification. In other words, post-condition is a condition at which a post-state must satisfies.

We show examples of a state-based specification, a corresponding formal specification and a source code:

– State-based Specification: set the value of the variable x to be incremented by 1 (Action) when the value of x is less than 0 (Condition),
– Formal Specification: $\mathtt{x} < 0 \rightarrow \mathtt{x}' = \mathtt{x} + 1$ where $\mathtt{x}'$ is a variable representing the value of x after an execution of an operation,
– Source Code: **if** $\mathtt{x} < 0$ **then** $\mathtt{x} := \mathtt{x} + 1$ **fi**.

Correctness of Formal Specifications There are some (generic) correctness conditions of formal specifications[4, 5]. *Consistency* is a correctness condition that specifications do not contradict each other. This correctness condition is a condition for a set of requirements. For instance, the set of two requirements "if SW = on then SW := off" and "if SW = on then SW := on" are contradict each other. *Completeness* is a correctness condition that there are no underspecification which means that certain required features are omitted.

We introduce correctness conditions *Invariant* and *Existence of Post-state.* Then we will show examples of verification of these correctness conditions for formal specifications in Section 3 and 4.

Invariant is a correctness condition that must always be true. Invariants must be defined by software engineers depending on specifications. We define the meaning of "must always" inductively as follows:

- an initial state of a state transition system satisfies an invariant,
- reachable states from the initial state satisfy the invariant.

We show two examples of invariants, 1) the value of a variable `WaterTemp` must always satisfies the condition $0 \leq \texttt{WaterTemp} \leq 100$, 2) the value of a variable **x** is must always greater then the value of a variable **y**.

Existence of Post-state is a correctness condition that at a state satisfying a pre-condition, just after the execution of an operation which we want to implement, we can reach at a state satisfying a post-condition. Since a state is a tuple of values of variables, Existence of Post-state is also a correctness condition that for any input satisfying a pre-condition, just after the execution of an operation which we want to implement, we can get a output satisfying a post-condition.

For a given invariant Inv, pre-condition preCon and a post-condition postCon, we show a formal descriptions of Existence of Post-states:

- $\forall s.\exists s'.\text{preCon}(s) \land \text{Inv}(s) \rightarrow \text{postCon}(s') \land \text{Inv}(s')$

where s is a pre-state and s' is a post-state[4]. We notice that a post-condition is often a relation among s, s'. For instance, a post-condition must be a relation $x' \doteq x + 1$ between x and x', for a requirement that if the value of a variable x is less than 0 then inclement the value of x by 1. We also notice that there are specifications which do not require invariants.

3 Verifying Correctness of Programs with SMT Solvers

In this section we explain how to verify a correctness condition of a program by solving a satisfiability problem with an SMT solver. First, we introduce definitions of satisfiability, equisatisfiability and a uninterpreted function of Satisfiability Modulo Theories[2, 10]. Then we introduce two translation techniques, namely *Tsetin Transformation* and *Skolemization*, from a correctness condition of a program to a satisfiability problem.

Definition 1. *A formula φ is satisfiable if there is an assignment under which φ evaluates to true. A formula φ is valid if φ evaluates to true for any assignments. We say that a formula φ is unsatisfiable if it is not satisfiable.*

A model-theoretic approach to solve satisfiability problems is to find an assignment satisfying a given formula, i.e., for a given formula, if we find an assignment and then return "sat" and the assignment, otherwise we return "unsat". There is an assignment $\langle p = true, q = false \rangle$ for a propositional formula $p \lor q$, therefore $p \lor q$ is satisfiable. And there is an assignment $\langle x = 1, y = 2 \rangle$ for a first-order formula $2x = y$, therefore $2x = y$ is satisfiable.

Definition 2. *Formulas φ and ψ are equisatisfiable if they are both satisfiable or they are both unsatisfiable.*

For instance, a formula $p \to q$ and $\neg p \vee q$ are equisatisfiable, (moreover they are logically equivalent) where p and q are Boolean variables. In general, logical equivalence implies equisatisfiability.

Let CNF(φ) be a formula which is conjunctive normal form and logically equivalent to a given formula φ. The satisfiability problem of CNF(φ) is often easy to solve than that of φ. But the complexity of a naive translation of φ to CNF(φ) is exponential. For instance, a formula

$$(x_1 \wedge x_2) \vee (x_3 \wedge x_4) \vee \cdots \vee (x_{2n-1} \wedge x_{2n})$$

is translated to a formula

$$(x_1 \vee x_3 \vee \cdots \vee x_{2n-1}) \wedge (x_1 \vee x_3 \vee \cdots \vee x_{2n}) \wedge \cdots \wedge (x_2 \vee x_4 \vee \cdots \vee x_{2n})$$

which is logically equivalent to the first and the number of its clauses is 2^n. There is an efficient translation method called *Tsetin Transformation*[11]. The complexity of the method is linear but the resulting formula is equisatisfiable to a given formula.

Definition 3. *An uninterpreted function symbol is a function symbol which is not interpreted in models.*

We assume that well-known function symbols, addition $+$, multiplication $\times$ etc., are interpreted as usual. We use function symbols $F, G, \cdots$ for uninterpreted function symbols.

We assume that every uninterpreted function F satisfies *Congruence Axiom*:

$$\forall t_1, \cdots, t_n, u_1, \cdots, u_n. \left(\bigwedge_{i=1}^{n} t_i = u_i \to F(t_1, \cdots, t_n) = F(u_1, \cdots, u_n) \right).$$

There are no other axiom assumed to uninterpreted functions.

Let φ be a formula and UF(φ) be a formula which is replacing some interpreted functions in φ with uninterpreted functions. Then it is clear that φ is valid whenever UF(φ) is valid. Moreover, validity checking of UF(φ) is often easier than that of φ.

We introduce a translation method called Skolemization. It is difficult to solve satisfiability problems of certain formulas, for instance $\forall\exists$-formulas. It is useful to translate such a formula to an equisatisfiable formula which is easy to solve its satisfiability problem with an SMT solver. Skolemization is one of such translation methods for $\forall\exists$-formulas.

The following formulas are equisatisfiable. We call the second formula *Skolemization* of the first formula.

1. $\forall x \in D\, \exists y \in D.\, P(x, y)$

2. $\forall x \in D. P(x, F(x))$ where $F: D \to D$

As we have already mentioned, the first formula is of the form of a correctness condition Existence of Post-state. In other words, the formula represents a condition that for any input x, there is an output y satisfying a condition $P(x, y)$.

The first formula does not require that the relation P is a function, namely there is one and only one y for any x. If the second formula is satisfiable then we have an implementation, which is a function, of the requirement expressed by the first formula, i.e., an assignment of F. SMT solvers can check satisfiability problems with uninterpreted functions. We will show that the first formula can be verified by solving a satisfiability problem of a formula having uninterpreted functions.

4 A Case Study: Verification with an SMT Solver Z3

In this section we introduce a simple case study in which we translate a formal specification with (an invariant,) a pre-condition and a post-condition to a verification condition formula in the language of Z3[3] and then we solve the satisfiability problem of the formula with Z3. An input of Z3 is a formula and a output is "sat" with an assignment if the input is satisfiable, or "unsat" if the input is unsatisfiable.

There is a client of Z3, called "Boogie" that is a program verification tool[12]. An input of the client is a source code with assertions and an output of the client is a verification condition formula which can be solved with SMT solvers. Thus we can verify source codes with Boogie and Z3. Our future goal is to develop a verification condition generator for specifications.

Let SW be a variable whose value is on and off. Consider a requirement that turn the value of SW to another if the value of SW is on. This requirement can be described as a pair of a pre-condition $\mathtt{SW} = \mathrm{on}$ and a post-condition $\mathtt{SW} \neq \mathtt{SW}'$. In other words, an implementation of the requirement must change the value of SW when an input value of SW is on. Thus a requirement is considered as a relation, in particular a function, between inputs and outputs.

One of correctnesses conditions of a specification is Existence of Post-state that, for any input satisfying a pre-condition, there is an output satisfying a post-condition after an execution of the program. Thus the correctness of the above requirement is described as a satisfiability problem of the formula

$$\forall \mathtt{SW}\, \exists \mathtt{SW}' \,(\mathtt{SW} = \mathrm{on} \to \mathtt{SW}' \neq \mathtt{SW}).$$

A main feature of Z3 is a quantifier, i.e., Z3 model finder is more effective if the input formula does not contain nested quantifiers. Therefore quantifier elimination technique is important. In this case, we translate the above verification condition formula to the following Skolemized verification condition formula:

$$\forall \mathtt{SW}\, (\mathtt{SW} = \mathrm{on} \to \mathtt{F(SW)} \neq \mathtt{SW})$$

where F:{on, off}→{on, off}.

If an SMT solver returns *sat* then we can implement the requirement. We show the requirement encoded to Z3 format in Figure 1 and the result of the satisfiability problem in Figure 2.

```
(declare-datatypes () ((S on off)))
(declare-fun F (S) S)
(assert (forall ((SW S)) (=> (= SW on) (not (= (F SW) SW)))))

(check-sat)
(get-model)
```

Fig. 1. A verification condition formula in Z3 language

```
sat
(model
  (define-fun F ((x!1 S)) S
    off)
)
```

Fig. 2. The result of the satisfiability problem

Suppose that we add an additional requirement that the value of SW must be on after an operation. Then the new verification condition formula can be formalized as follows:

$$\forall \mathtt{SW}\, \exists \mathtt{SW}' (\mathtt{SW} = \mathtt{on} \rightarrow \mathtt{SW}' \neq \mathtt{SW} \land \mathtt{SW}' = \mathtt{on}).$$

Then a corresponding verification condition formula in Z3 language is the following.

```
(declare-datatypes () ((S on off)))
(declare-fun F (S) S)
(assert (forall ((SW S))
  (=> (= SW on) (and (not (= (F SW) SW)) (= (F SW) on)))))

(check-sat)
```

Fig. 3. A verification condition formula in Z3 language

The satisfiability problem for the verification condition formula in Figure 3 is unsatisfiable. This result tells us that these two requirements contradict to each other. Thus we cannot implement the requirements.

In some cases, Skolemization is not enough for solving a satisfiability problem of a formula with Z3. We show an example of this insufficiency. Consider a formula $\forall x$::Int $\exists y$::Int $x < y$. We have the resulting formula $\forall x$::Int $x < F(x)$, where F::Int $\to$ Int, by Skolemization. Then the Skolemized formula described as an input to Z3:

```
(declare-fun F (Int) Int)
(assert (forall ((x Int)) (< x (F x))))
```

But this cannot be solved with Z3. Of course, the following trivial satisfiability problem can be solved.

```
(declare-fun F (Int) Int)
(assert (forall ((x Int)) (= (+ x 1) (F x))))
```

Thus more efficient translation is required to solve satisfiability problems with Z3.

5 Summary

In this paper, we introduced a verification method for formal specifications with an SMT solver to assure quality of software in design process. In particular, we introduced some definitions of Satisfiability Modulo Theories and translation techniques. Then we show a case study in which we formalize a requirement to a logical formula and translate it to a Skolemized formula. Then we solved its correctness Existence of Post-state with an SMT solver Z3. But the satisfiability problem of complex formula cannot be solved with Z3, therefore more efficient equisatisfiable translation is required.

References

1. 中島震, 形式手法入門, オーム社, (2012)
2. Handbook of Satisfiability (Frontiers in Artificial Intelligence and Applications), Armin Biere, Marijn Heule, Hans Van Maaren, Toby Walsh (Eds.), Ios Press Inc, (2009)
3. Z3, http://z3.codeplex.com/
4. John Fitzgerald, Peter Gorm Larsen, Modelling Systems, Practical Tools and Techniques in Software Development, Second Edition, Cambridge University Press (2009)
5. Jean-Raymond Abrial, Modeling in Event-B, System and Software Engineering, Cambridge University Press (2010)
6. Gerard J. Holzmann, The SPIN Model Checker: Primer and Reference Manual, Addison-Wesley Professional; 1 edition, (2003)
7. Model Checking, Edmund M. Clarke Jr., Orna Grumberg, Doron Peled, The MIT Press, (1999)

8. The Coq Proof Assistant, https://coq.inria.fr/
9. Agda, http://wiki.portal.chalmers.se/agda/pmwiki.php
10. Daniel Kroening, Ofer Strichman, Decision Procedures: An Algorithmic Point of View (Texts in Theoretical Computer Science. An EATCS Series, Springer Berlin Heidelberg, (2008)
11. G.S. Tseitin, On the complexity of derivation in propositional calculus, Slisenko, A.O. (ed.) Structures in Constructive Mathematics and Mathematical Logic, Part II, Seminars in Mathematics (translated from Russian), pp.115-125. Steklov Mathematical Institute (1968)
12. Boogie: An Intermediate Verification Language, http://research.microsoft.com/en-us/projects/boogie/