数理解析研究所講究録 第2084巻 2018年 21-26

Formal Proofs of Theorems of Programs with an SMT Solver

Shusei Komatsu, Geoffrey Baudelet, Keishi Okamoto

Abstract

In this paper, we give a formal proof of a theorem in [1] with an SMT solver Z3[2]. We develop a domain specific language, which is an extension of Z3.py[3], to describe proofs based on the formulation of programs in [1]. Moreover, with the domain specific language, we also give verification examples that the refinement relation holds for pairs of programs.

1 Introduction

In this section, first, we introduce formal methods and stepwise refinement. A product developed in the way of stepwise refinement is correct by construction. But verifying correctness of refinement is a difficult task. Next, we briefly introduce SMT which is a satisfiability problem. We use an SMT solver, which is a tool to solve an SMT, to give a formal proof. Finally, we introduce a formalization of programs and specifications introduced in [1].

1.1 Formal Methods and Stepwise Refinement

Formal methods are promising software development methods which are based on mathematics and computer science. Formal methods have strict syntax to describe specifications and programs, and verification methods to show their correctness. In [1], specifications and programs are formalized in a uniform way. Thus, in this paper, we treat a program as a specification and vice versa.

Formal specification is one of formal methods. VDM++[4] and Event-B[5] are formal specification methods. Some descriptions in these formal specification methods are based on first-order logic(FOL).

Event-B adopts a way of software development called stepwise refinement. In stepwise refinement, we first describe an abstract specification and then refine it to more concrete specification step by step.

We show an example of a pair of an abstract specification S1 and a concrete specification S2. An abstract specification S1 is that if given input is positive then all implementation of the S1 returns an output which is a square root of the input. A specification can be a relation which defines pairs of inputs and outputs. Then S1 is a relation $\{(x, \sqrt{x}), (x, -\sqrt{x})\}$. A concrete specification S2 is that if given input is positive then all implementation of the S2 returns

the output which is the positive square root of the input. Then S2 is a relation $\{(x, \sqrt{x})\}$.

At each step, we verify that the concrete specification is a refinement of the abstract specification. We verify refinement relations with theorem proving and other verification methods. But the verification often requires manual tasks with expertise in mathematics.

1.2 SMT Solver

We use an SMT solver to give a formal proof. Satisfiability Modulo Theories (SMT) is a satisfiability problem for a first-order formula with free variables[6]. When we give an input first-order formula with free variables to an SMT solver, the SMT solver returns an output "sat" and a witness if the input is satisfiable, and an output "unsat" otherwise. For instance, if we give an input $x = y + 1$ to an SMT solver then the solver returns and output "sat" and a witness $x = 1, y = 0$.

1.3 Theory of Programs

Many formalization of specifications and programs have been proposed. Meyer proposed a formalization of specifications and programs[1].

In [1], programs and specifications are uniformly formalized based on set theory. A program is formalized as a triple (Set, Precondition, Postcondition) where Set is an underlying set for variables, Precondition is a required condition for inputs and Postcondition is a relation of inputs and outputs. For example, a condition $x \geq 0$ ($x^2 = y$) is a precondition (respectively a postcondition) of a program which returns a square root $y(\in \mathbf{R})$ of a given real number x.

In [1], many properties of programs are manually proved but manual proof may include errors. We will check proofs in [1] by giving formal proofs with an SMT solver. A theorem of the form "for any program p, if p satisfies a condition C_1 then p satisfies a condition C_2" can be formalized as a first-order(FO) formula $\forall p\ C_1(p) \to C_2(p)$. This formula is logically equivalent to an FO formula $\exists p\ C_1(p) \wedge \neg C_2(p)$ that represents a satisfiability problem. Hence we can prove theorems of the form with an SMT solver.

Moreover, in [1], a formalization of refinement relation of a pair of programs are proposed. We refine an abstract program p_i to a more concrete program p_{i+1} again and again in a way of stepwise refinement, i.e., we develop a sequence of programs $p_1, p_2, \ldots, p_i, p_{i+1}, \ldots, p_n$ such that (p_i, p_{i+1}) is a refinement relation. Then the resulting program p_n is correct by construction. Rodin, which is an development tool based on Event-B, helps development based on stepwise refinement. Rodin automatically verify whether a pair of programs is a refinement relation. But size of verification is becoming larger and larger. Thus it is important to develop a powerful tool to verify a refinement relation automatically.

In Section 2, we give a formal proof of a theorem in [1] with an SMT solver

Z3[2]. In Section 3, we give verification examples that the refinement relation holds for pairs of specifications. In Section 4, we conclude this paper.

2 Formal Proof of a Theorem P11

In this section, we give a formal proof of the theorem P11 in [1]. First, we introduce the theorem P11. Second, we introduce definitions used in P11. Then we give a formal proof of P11. But the original P11 does not hold, thus we propose a modification of P11.

Theorem 1 ([1]) *P11* $q;(p1 \cup p2) = (q;p1) \cup (q;p2)$ *where* $p1, p2, q$ *are programs.*

We prepare definitions and notations to give a formal proof of P11.

Definition 2 ([1]) *For a given set* S, a program p *is a triple* $(Set_p, Pre_p, Post_p)$ *where*

- *State Set:* $Set_p \subseteq S$ *(*S *is a universe set)*
- *Precondition:* $Pre_p \subseteq Set_p$
- *Postcondition:* $Post_p \in Set_p \times Set_p$

Definition 3 ([1]) *Programs* $p1$ *and* $p2$ *are* equivalent *if*

- $Pre_{p1} = Pre_{p2}$ *and*
- $post_{p1}/Pre_{p1} = post_{p2}/Pre_{p2}$.

We write $p1 = p2$ *when* $p1$ *and* $p2$ *are equivalent.*

We define a choice operator $\cup$ and a composition operator ; for programs.

Definition 4 ([1]) *For programs* $p1$ *and* $p2$, *the choice* $p1 \cup p2$ *and the composition* $p1;p2$ *of* $p1$ *and* $p2$ *are defined as follows:*

- $p1 \cup p2 := (Set_{p1} \cup Set_{p2}, Pre_{p1} \cup Pre_{p2}, Post_{p1} \cup Post_{p2})$,
- $p1;p2 := (Set_{p1} \cup Set_{p2},\ Pre_{p1} \cap Post^{-1}(Pre_{p2}),\ Post_{p2} \circ (Post_{p1} \setminus Pre_{p2}))$.

Intended meaning of $p1 \cup p2$ is that either $p1$ or $p2$ is executed. And intended meaning of $p1;p2$ is that $p2$ is executed after $p1$ is executed.

We prove theorems in [1] with an SMT solver Z3. First, we express a theorem in [1] as a FO-formula. Theorem is of the form that, for any program p, if p satisfies a premise then p satisfies a conclusion. Then the negation of a theorem is a satisfiability problem. Thus, if the resulting satisfiability problem is unsatisfiable, then the theorem holds. On the other hand, if the resulting satisfiability problem is satisfiable, then there is a counter-example of the theorem, namely the theorem does not hold.

We use an SMT solver Z3[3] and its Python API Z3.py[4]. Since Z3.py has only basic constructs, the size of description of a theorem tends to long, and it is difficult to give a correct description of a theorem. Thus we define a domain-specific language based on Z3.py. List 1 is a description of theorem P11 in the domain-specific languages.

```
# Generate an instance of a solver
s = Solver()
title = "P11 q;(p1 ∪ p2) = (q;p1) ∪ (q;p2)"
# Generate instances of programs
q, p1, p2 = progs(s, 'q p1 p2')
# The statement of Theorem P11
theorem = q ^ (p1 | p2) == (q ^ p1) | (q ^ p2)
# Verify the theorem
conclude(s, theorem, title)
```

List 1: Theorem P11

Z3 generates an output "sat" and a counter-example of the theorem P11. We try to correct the theorem P11 according to the counter-example. Analysis of the counter-example shows that $q;(p1 \cup p2)$ admits an unintended execution whose input is of $p1$ and output is of $p2$.

Since intended meaning of $q;(p1 \cup p2)$ is $(q;p1) \cup (q;p2)$, we modify the definition of the union operator as follows:

$$Post_{p1 \cup p2} = (Post_{p1}/Pre_{p1}) \cup (Post_{p2}/Pre_{p2}).$$

We verify the theorem P11 with this modified definition. Then the result shows that this modified definition of the union operator does not admit the above unintended execution, namely modified P11 holds.

3 Examples: Verifying Refinement Relations

In this section, we show verification examples of refinement relations in the domain-specific language. Naive formalization, in which variables ranges over integer domain, results in "unknown" by limitation of Z3. Thus we adopt a strong assumption that variables ranges over a finite domain, and then we succeed to verify that the refinement relation holds for the case.

We prepare definitions in a concept of a contracted program.

Definition 5 ([1]) *A program p is* feasible *if $Pre_p \subseteq dom(Post_p)$. A program $p2$ is a* contracted program *of $p1$ if $p2$ is a refinement of $p1$ ($p2 \subseteq p1$) and $p2$ is feasible.*

Thus, if a program $p2$ is a contracted program of $p1$ then $p2$ is a refinement of $p1$.

We show an example in which a program $p2$ is a contracted program of a program $p1$.

```
title = "double1"
p1, p2 = progs(s, "p1 p2")
s.add(+p1, +p2) # Assume that p1, p2 are feasible
s.add(p1.set() == p2.set(),
      p1.pre() == p2.pre())
x,y = Ints('x y')
s.add(ForAll([x, y], p1.post(x, y) ==
      (2 * x == y)))
s.add(ForAll([x, y], p2.post(x, y) ==
      Or(And(x==1, y==2), And(x==2, y==4))))
conclude(s, contracts(p2, p1), title)
```

List 2: Verification Example (Integer Domain)

In this example, $p1$ is an abstract program and $p2$ is a concrete program whose input and output range over integers. For an input x and an output y, $p1$ requires that $\forall x \forall y\ 2 * x = y$, and $p2$ requires that $\forall x \forall y[(x = 1 \wedge y = 2) \vee (x = 2 \wedge y = 4)]$. Then the second requirement logically implies the first requirement.

Z3 returns "unknown" for the satisfiability problem in this example. The quantifiers in $p1$ and $p2$ may cause a problem since Z3 checks the satisfiability problem over integers. But the satisfiability problem should be "unsat" since $p2$ is a contracted program of $p1$.

We modify the requirement for $p2$ as follows:

```
s.add(ForAll([x, y], p2.post(x, y) ==
      Or(And(x==1, y==3), And(x==2, y==4))))
```

to detect the case that $p2$ is not a contracted program of $p1$. Since $2 * x = y$ does not hold for $x = 1$ and $y = 3$, the modified $p2$ is not a contracted program of $p1$. Then Z3 returns "sat" which means that there is a counter-example to that $p2$ is a contracted program of $p1$.

Next, we show another example in which a program $p4$ is a contracted program of a program $p3$. In this example, the range of variables are restricted to a finite domain $\{n1, n2, n3, n4, n5, n6, n7, n8\}$. In the following list, the intended meaning of an element ni is the integer i and a "double" function returns $2 * i$ for a given input i.

```
title = "double2"
p3, p4 = progs(s, "p3 p4")
s.add(eq_set(p3,p4),eq_pre(p3,p4),+p3,+p4)
x,y = consts('x y', U)
double = Function('double', U, U)
s.add([double(i)==j for i, j in
    [(n1, n2), (n2, n4), (n3, n6), (n4, n8)]])
s.add(ForAll([x, y], p3.post(x, y) == (y == double(x))))
s.add(ForAll([x, y], p4.post(x, y) == Or(
[And(x == i, y == j) for i, j in [(n1, n2), (n2, n4)]])))
conclude(s, contracts(p4, p3), title)
```

List 3: Verification Example (Finite Domain)

$p3$ is a program which returns $2 * i$ for a given input i where $i = n1, n2, \ldots, n8$. Then $p3$ is a restriction of $p1$ to a finite domain $\{n1, n2, \ldots, n8\}$. $p4$ is a program which returns $n2$ for the input $n1$ and $n4$ for the input $n2$. Then $p4$ is the same program as $p2$.

The result of the satisfiability problem in List 3 is "unsat" which means that $p4$ is a contracted program of $p3$.

We modify the requirement for $p4$ in List 3 as follows:

```
s.add(ForAll([x, y], p4.post(x, y) == Or(
[And(x == i, y == j) for i, j in [(n1, n2), (n2, n5)]])))
```

Since $y = double(x)$, whose intended meaning is that $y = 2 * x$, does not hold for $x = n2$ and $y = n5$, the modified $p4$ is not a contracted program of $p3$. Then Z3 returns "sat" which means that there is a counter-example to that $p4$ is a contracted program of $p3$.

4 Concluding Remarks

In this paper, we give a formal proof of the theorem P11 in [1]. Moreover, we give formal proofs of other 44 theorems in [1]. Besides, we also try to give a formal proof of refinement relations. Naive formalization, in which variables range over integers, implies that Z3 returns "unknown". Then, we adopt an assumption that variables range over a finite domain. With this assumption, Z3 returns "unsat". But more feasible assumption is preferable.

References

[1] Theory of Programs, Bertrand Meyer, 2015

[2] Z3 Prover, https://github.com/Z3Prover/z3

[3] Z3API in Python, URL:www.cs.tau.ac.il/ msagiv/courses/asv/z3py/guide-examples.htm

[4] Validated Designs for Object-Oriented Systems, John Fitzgerald et al., Springer-Verlag, 2005

[5] Modeling in Event-B system and software Engineering, Jean Raymond Abrial, CAMBRIDGE, 2010

[6] Handbook of Satisfiability, Armin Biere, Marijn Heule, Hans Van Maaren, Toby Walsh (Ed.), IOS Press, 2009