Title	A Case Study : Meyer's Formulation of a Specification and Theorem Proving with an SMT Solver Z3 (Model theoretic aspects of the notion of independence and dimension)
Author(s)	Okamoto, Keishi
Citation	数理解析研究所講究録 (2016), 2002: 9-18
Issue Date	2016-07
URL	http://hdl.handle.net/2433/231465
Right	
Type	Departmental Bulletin Paper
Textversion	publisher

A Case Study: Meyer's Formulation of a Specification and Theorem Proving with an SMT Solver Z3

Keishi Okamoto

Abstract

An SMT solver is a program to solve satisfiability problems described in restricted first-order formulas. Recently, SMT solvers are becoming powerful and applied to solve concrete problems in many research areas. But a concrete problem requires a complex first-order formula, then the satisfiability of the resulting formula often cannot be solved with an SMT solver. In these cases, we must use some model-theoretic techniques (Skolemization, quantifier-elimination, etc.) to reduce the complexity of a given first-order formula.

Our future goal is to propose a simple formal specification language and a validation method, which is based on a model-theoretic method, to fill a gap in natural languages and standard formal languages. In this paper, we show a case study. In the case study, we formalize a specification and its properties with Meyer's formulation, and then we prove some theorems for specifications with an SMT solver Z3.

1 Introduction

Formal verification of programs with SAT/SMT solvers are active research topic, since SAT/SMT solvers are becoming powerful for actual applications[1]. But formalization of a program and its properties are required for verification of programs. In [2], Meyer introduce semantics of programs and specifications based on sets. Meyer's formalization is a unified formalization framework for programs and specifications. Moreover, the formalization can be described in a language of the first-order logic.

In this paper, we show a case study. In the case study, we formalize a program and a specification as a first-order logic formula with [2]. Then we prove some theorems on programs and specifications with an SMT solver Z3[4], i.e., we solve satisfiability problems of first-order formulas.

In this chapter, we also introduce formalization of a program and a specification in [2] and an SMT solver Z3. In chapter 2, we formalize a program and a specification, then we prove basic theorems with Z3. In chapter 3, we formalize operations on programs and specifications, then we prove the feasibility preservation theorem with Z3. In chapter 4, we make concluding remarks.

1.1 Formalization of a Program and a Specification

A program is a function from the set of (input) states to the set of (output) states where a state is the tuple of values of variables in the program. For example, for a given state (an input) ($x = 1$), a program $x := x + 1$ returns a state (a output) $x = 2$. On the other hand, in a software development, a developer make a program from a specification document which consists of specifications. Then a specifications can be considered as a abstract program and a program can be considered as a refined specifications. Thus we can formulate a specification as a relation on the set of (input) states and the set of (output) states. For example, a specification of a calculator of the great common divisor can be formulated as a relation $R(x, y, z)$ which represents that a output z is the great common divisor of inputs x and y. A developer develops a program, which meets the given specification, that calculates the output from given inputs. In this paper we consider a program is a specification as in [2].

In the "Design by Contract" paradigm[3], a developer define a contract between a caller program and a called program. A contract is defined with a *precondition* and a *postcondition* of a program. A precondition is a condition that must always be true just prior to the execution of some section of a program. A postcondition is a condition or a predicate that must always be true just after the execution of some section of a program. Then a program is defined as the pair of a precondition and a postcondition. A developer of a called program is expected to develop the called program which returns an output satisfying the postcondition of the called program whenever a caller program gives the called program an input satisfying the precondition of the called program.

In [2], Meyer formalize programs in the Design by Contract paradigm. In [2], a program is considered as a concrete specification and is defined as a triple of a set of possible states, a precondition and a postcondition. We introduce the formal definition of a program in [2]. Let S be a set of states (of programs). And let $Program$ be the triple $\langle Set, Pre, Post \rangle$ where $Set \subseteq S$, which is a set of states, $Pre \subseteq S$, which is a set of preconditions and $Post \in Pow(S \times S)$, which is a set of postconditions. The triple $\langle \{(x, y, z) \mid x, y, z \in \mathbf{Z}\}, \{(x, y, z) \mid x, y > 0\}, \{(x, y, z, x', y', z') \mid z' = GCD(x, y), x' = x, y' = y\} \rangle$ is an example of a program which returns the greatest common divisor z' of given inputs x and y.

1.2 An SMT Solver Z3

A SAT solver is a solver for solving satisfiability problems of propositional logic formulas. A SAT solver returns *sat* and a witness if the given propositional logic formula is satisfiable, and *unsat* if the formula is not satisfiable. An SMT (Satisfiability Modulo Theories) solver is a solver for solving satisfiability problems of certain first-order logic formulas. An SMT solver returns *sat* and a witness if the given first-order logic formula is satisfiable, and *unsat* if the formula is not satisfiable. For example, An SMT solver returns *sat* and a witness $x = 1, y = 0$ for an input $x = y + 1$ where $x, y \in \mathbf{N}$. In this paper, we use an SMT solver Z3 developed by Microsoft Research[4].

The formula $\forall x \in U.\exists y \in U.R(x,y)$ is valid in a class of models (of size 3) if and only if the formula $\forall x \in U.R(x, f(x))$ is satisfiable in the class where $f\colon U \to U$[5]. (f is called a Skolem function.) Skolemization is important to solve satisfiability problems. Because an SMT solver often returns *unknown* to a formula of the form $\forall x \in U.\exists y \in U.R(x,y)$ while it often returns *sat* or *unsat* to a formula of the form $\forall x \in U.R(x, f(x))$ in those cases.

Formalization of a program and a specification often requires a predicate having a Skolem function as an input. But, in general, higher-order predicates cannot be defined in Z3 except for the array type[4]. For instance, a function $map(f, [x1, x2, x3]) = [f(x1), f(x2), f(x3)]$ cannot be defined in Z3 for a usual function f of a type $((U)\ U)$. An element of a type *Array* X Y, where X is a index set and Y is a value set, can be considered as a function from X to Y. Thus we can define a second-order predicate with array types as follows:

```
; Valid definition of the feasibility relation
(define-fun isfeasible ((p Prog) (f (Array U U))) Bool
 (forall ((x U)) (=> (select (pseudo-pre p) x)
                     (select (post p) x (select f x)))))

; Invalid definition of the feasibility relation
(define-fun isfeasible ((p Prog) (f ((U) U))) Bool
  (forall ((x U)) (=> (select (pseudo-pre p) x)
                      (select (post p) x (select f x)))))
```

2 Specifications and Basic Theorems

In this section, we formalize a specification (a program) as in [2], and prove the refinement theorem and the implementation theorem with Z3.

2.1 Formalization of a Specification in Z3

With the formalization in [2], we formalize a specification and their properties in Z3 as follows:

```
(declare-datatypes () ((U A B C)))        ; Universe $U =\{A, B, C\}$
(define-sort Set () (Array U Bool))       ; $Set \subseteq U$
(define-sort Pre () (Array U Bool))       ; $Pre \subseteq S$
(define-sort Post () (Array U U Bool))     ; $Post : Pow(S\times S)$
(declare-datatypes () ((Prog (mk-prog (set Set) (pre Pre) (post Post)))))
```

We note that the above definition of *Pre* is weaker than that in [2]. We use the intersection of our *Pre* and *Set* for the definition of *Pre* in [2].

We also note that we will show that, in our proofs, theorems have no counter-example of size 3, i.e., the size of U is 3. In a software development, a well-known **Small Scope Hypothesis** tells us that "almost all of bugs have small size counter-examples". With the assumption, almost all of wrong theorems (claims) have small size counter-examples in this paper.

2.2 Refinement Theorem

In this subsection, we prove refinement theorem. We define feasibility of a specification, equality of two specifications and a refinement relation on specifications, and then we prove the refinement theorem with the SMT solver Z3.

We define feasibility of a specification as in [2]. Intuitively, feasibility of a specification means that, for every valid input, there exists a corresponding output.

Definition 1 *A specification p is feasible if $Pre_p \subseteq dom(post_p)$ where Pre_p is the precondition of p and $post_p$ is the postcondition of p.*

Since the formula $x \in dom(post_p)$ is equivalent to the formula $\exists y \in U.(x, y) \in post_p$, the formula $\forall x.(x \in Pre_p \Rightarrow \exists y.\langle x, y\rangle \in post_p)$ represents feasibility of a specification p, which is equivalent to the formula $\exists f.\forall x.(x \in Pre_p \Rightarrow \langle x, f(x)\rangle \in post_p)$. Thus the above definition can be formalized as follows in Z3:

```
(define-fun isfeasible ((p Prog) (f (Array U U))) Bool
 (forall ((x U)) (=> (select (pre p) x) (select (post p) x (select f x)))
```

Since the claim "p is not feasible" is equivalent to the formula $\exists x \in U.(Pre_p(x)\wedge (\forall y \in U \neg post_p(x, y)))$, that claim can be formalized as follows:

```
(define-fun isnotfeasible ((p Prog)(x U)) Bool
(and (select (pre p) x) (forall ((y U)) (not (select (post p) x y)))))
```

We define an equality relation on specifications as in [2].

Definition 2 *A specification p_1 is equal to a specification p_2 if $Pre_{p1} = Pre_{p2}$ and $post_p1/Pre_{p1} = post_{p2}/Pre_{p2}$ where $post/Pre = post \cap (Pre \times S)$*

The above definition can be formalized as follows in Z3:

```
(define-fun eqprog ((p1 Prog) (p2 Prog)) Bool
(and (= (pre p1) (pre p2))
     (forall ((xy (Pair U U))) (=> (and (select (pre p1) (first xy)))
     (= (select (post p1) (fst xy) (snd xy))
        (select (post p2) (fst xy) (snd xy))))))
```

We define a refinement relation on specifications as in [2]. A (concrete) specification p_2 is a refinement of a specification p_1 if

- the set of acceptable inputs for p_2 subsumes that of p_1,
- for each input, p_1 and p_2 return the same output.

The equality relation is an equivalence relation and then the refinement relation on specifications modulo the equally relation is an order relation[2].

The informal definition of the refinement relation is formalized in [2] as follows:

Definition 3 *Let p_i be the tripe $\langle S_1, Pre_1, post_i$ $(i = 1, 2)$. A specification p_2 refines a specification p_1 (or p_1 specifies (or abstracts) p_2) if*

- *Extension: $S_2 \supseteq S_1$ (P1 in [2])*
- *Weakening: $Pre_2 \supseteq Pre_1$ (P2)*
- *Strengthening: $post_2 \cap (Pre_1 \times S) \subseteq post_1$ (P3)*

Since the condition P3 can be formalized as the formula $\forall x, y.[(x, y) \in post_2 \wedge x \in Pre_1 \Rightarrow (x, y) \in post_1]$, the above definition of the refinement relation can be formalized as follows in Z3:

```
(define-fun refines ((p1 Prog) (p2 Prog)) Bool ; p2 refines p1.
(and (subseteq (set p1) (set p2))    ; P1: S2  S1 - Extension
(and (subseteq (pre p1)) (pre p2))   ; P2: Pre2  Pre1 - Weakening
 (forall ((xy (Pair U U)))
  (=> (and (select (pre p1) (fst xy) (select (post p2) (fst xy) (snd xy)))
      (select (post p1) (fst xy) (snd xy)))))  ; P3
```

For example, a specification $p_2 : \langle \mathbf{N}, x > 0, x' = x + 1\rangle$ is a refinement of a specification $p_1 : \langle \mathbf{N}, x > 0, x' > x\rangle$. We write $p_2 \subseteq p_1$ when p_2 refines p_1.

Now we prove a refinement theorem with Z3.

Theorem 4 (P4) *The refinement relation ($\subseteq$) is an order relation.*

Proof. In Z3, the reflexive law can be formalized as follows:

```
(assert (forall ((p Prog)) (refines p p)))
```

Z3 returns *sat* to the formula. Then the reflexive law for the refinement relation holds.

When the formula, which is the negation of the antisymmetric law, $(p_{42} \subseteq p_{41}) \wedge (p_{41} \subseteq p_{42}) \wedge \neg(p_{41} = p_{42})$ is satisfiable in the class of our models, the antisymmetric law for the refinement relation does not hold for the class. The formula can be formalized as follows:

```
(declare-const p41 Prog)
(declare-const p42 Prog)
(assert (and (refines p41 p42)
       (and (refines p42 p41)
            (not (eqprog p41 p42)))))
```

in Z3. Since Z3 returns *unsat* for the formula, the symmetric law holds.

We note that Z3 returns *unknown* for the following formula which also represents antisymmetric law in a naive way:

```
(declare-const p41 Prog)
(declare-const p42 Prog)
(assert (forall ((p1 Prog)(p2 Prog))
          (=> (and (refines p1 p2) (refines p2 p1)) (eqprog p1 p2)))).
```

The negation of the transitivity law for the refinement relation can be formalized in Z3 as follows:

```
(declare-const p11 Prog)
(declare-const p12 Prog)
(declare-const p13 Prog)
(assert (and (refines p11 p12) (and (refines p12 p13)
        (not (refines p11 p13)))))
```

Since Z3 returns *unsat* for the formula, the transitivity law holds. □

2.3 Implementation Theorem

In this subsection, we prove the implementation theorem. We define implementation relation on specifications, and then we prove implementation theorem in the theorem prover Z3.

We define an implementation relation on specifications as in [2].

Definition 5 *An implementation of a specification p is a feasible refinement of p. There is a non-feasible refinement of a specification p if post(p) is empty.*

This implementation relation can be formalized in Z3 as follows:

```
(define-fun IsImple ((p1 Prog) (p2 Prog) (f2 (Array U U))) Bool
 (and (isfeasible p2 f2) (refines p2 p1)))
```

which represents that a specification p_2 is an implementation of a specification p_1 where the array f_2 is a Skolem function required for the relation "isfeasible".

Now we prove an implementation theorem [2].

Theorem 6 (P5) *A specification (p_1) having an implementation (p_2) is feasible.*

Proof. If there are specifications p_1 and p_2 such that "p_2 refines p_1" and "p_2 is feasible" and "p_1 is not feasible" (in a class of models), then the implementation theorem does not hold (for the class). Thus the implementation theorem can be proved by solving a satisfiability problem. We formalized the implementation theorem as follows:

```
(declare-const p1 Prog)
(declare-const p2 Prog)
(declare-const f2 (Array U U))
(declare-const x1 U)
(assert (and (refines p1 p2)
        (and (isfeasible p2 f2) (isnotfeasible p1 x1))))
```

Z3 returns *unsat* for the above formula, thus the implementation theorem holds. □

3 Feasibility Preservation Theorems

In this section, we define three operations on specifications, i.e., choice, composition and restriction. Then we prove feasibility preservation theorems for these three operators.

We give informal and formal definitions of choice, composition and restriction operators. 1) Intuitively the the choice $p_1 \cup p_2$ of specifications p_1 and p_2 performs like p_1 or like p_2. For the choice $p_1 \cup p_2$, the postcondition is defined as $post_1 \cup post_2$ and the precondition is defined as $Pre_1 \cup Pre_2$. 2) Intuitively the the composition $p_1;p_2$ of specifications p_1 and p_2 performs first like p_1 then like p_2. For the composition $p_1;p_2$, the postcondition is defined as $(post_1 \setminus Pre_2);post_2$ and the precondition is defined as $Pre_1 \cap post_1^{-1}(Pre_2)$. 3) Intuitively the the restriction $C : p$ of a specification p to a set C performs like p on C. For the composition $p_1;p_2$, the postcondition is defined as $post_p/C$, where $post_p/C = post_p \cap (C \times S)$, and the precondition is defined as Pre_p

3.1 Feasibility Preservation Theorem (Choice)

Theorem 7 (P6) *If specifications p_1 and p_2 are feasible then the choice specification $p_1 \cup p_2$ is feasible.*

Proof. The feasibility preservation theorem for the choice operator ($\cup$) can be formulated as the claim that there are no specifications p_{61} and p_{62} such that p_{61} and p_{62} are feasible and the union $p_{61} \cup p_{62}$ is not feasible.

We defined a relation "isfeasible" for a specification p. Then p is feasible if $isfeasible(p)$ is satisfiable in the class of models. Now we define a ternary relation "isnotfeasibleunion". Essentially the relation is binary, but technically it is ternary in which the third argument is required for a witness of satisfiability problem.

For given specifications p_1 and p_2, the choice $p_1 \cup p_2$ is feasible (in the class of models) if the formula $\forall x.[(x \in Pre_{p1} \cup Pre_{p2}) \Rightarrow \exists y.(\langle x, y\rangle \in post_{p1} \cup post_{p2})]$ valid (respectively for the class). Then $p_1 \cup p_2$ is not feasible (in the class of models) if the formula $x \in Pre_{p1} \cup Pre_{p2} \wedge \forall y.\neg(\langle x, y\rangle \in post_{p1} \cup post_{p2})$ is satisfiable (respectively in the class). Thus the "isnotfeasibleunion" can be formalized in Z3 as follows:

```
(define-fun isnotfeasibleunion ((p1 Prog) (p2 Prog) (x U)) Bool
 (and (or (select (pre p1) x) (select (pre p2) x))
  (forall ((y U)) (not (or (select (post p1) x y) (select (post p2) x y)))
```

With these definitions, the feasibility preservation theorem for the choice operator can be formalized in Z3 as follows:

```
(declare-const p61, p62 Prog)
(declare-const f61, f62 (Array U U))
(declare-const x63 U)
(assert (and (isfeasible p61 f61) (and (isfeasible p62 f62)
 (isnotfeasibleunion p61 p62 x63))))
```

Since Z3 returns *unsat* for the above formula, the theorem holds. □

3.2 Feasibility Preservation Theorem (Composition)

Theorem 8 (P6) *If* p1 *and* p2 *are feasible then the composition specification* p1; p2 *is feasible.*

Proof. The feasibility preservation theorem for the composition operator (;) can be formulated as the claim that there are no specifications p_{621} and p_{622} such that p_{621} and p_{622} are feasible and their composition $p_{621}; p_{622}$ is not feasible.

We defined the relation "isfeasible" for a specification p in Section 2.2. We define a relation "IsNotFeasibleComp" for specifications p_1 and p_2 such that $p_1; p_2$ is not feasible (in a class of models) if and only if $IsNotFeasibleComp(p_1, p_2)$ is valid (respectively for the class). On the other hand, The composition $p1; p2$ is feasible (in a class of models) if and only if the formula $\forall x.[x \in Pre_{p1;p2} \Rightarrow \exists y.\langle x, y\rangle \in post_{p1;p2}]$ is valid (respectively for the class) where $Pre_{p1;p2}$ ($post_{p1;p2}$) is the precondition (respectively postcondition) of $p_1; p_2$. Then $p_1; p_2$ is feasible (in a class of models) if and only if the formula

$$x \in Pre_1 \wedge x \in post_1^{-1}(Pre_2) \wedge \forall y.\forall u.\neg\{\langle x, u\rangle \in post_1 \cap (U \times Pre_2) \wedge \langle u, y\rangle \in post_2\}$$

is satisfiable (respectively in the class) .

Now we define a relation "IsNotFeasibleCompa" such that $p_1; p_2$ is not feasible (in a class of models) if and only if the formula $IsNotFeasibleComp(p_1, p_2, u_1, u_2)$ is satisfiable (respectively in the class) as follows:

```
(define-fun IsNotFeasibleComp ((p1 Prog)(p2 Prog)(u1 U)(u4 U)) Bool
(and (select (pre p1) u1)
(and (and (select (post p1) u1 u4) (select (pre p2) u4))
(and (forall ((u2u3 (Pair U U))) (not (and (select (post p1) u1 (fst u2u3))
 (and (select (pre p2) (fst u2u3)))) (select (post p2) (fst u2u3) (snd u2u3))

(declare-const p621,p622 Prog)
(declare-const f621,f622 (Array U U))
(declare-const u621,u624 U)
(assert (and (isfeasible p621 f621) (and (isfeasible p622 f622)
        (IsNotFeasibleComp p621 p622 u621 u624))))
```

Since Z3 returns *unsat*, the theorem holds. □

3.3 Feasibility Preservation Theorem (Restriction)

Theorem 9 (P6) *If* p *is feasible then* $C : p$ *is feasible.*

The feasibility preservation theorem for the restriction operator can be formulated as the claim that there are no specification p such that p is feasible and its restriction $C : p$ to C is not feasible. We will formalize the claim that $C : p$ is not feasible in Z3.

By the definition of the restriction operator, the claim $\langle x, y\rangle \in post_{C:p}$ is equivalent to the claim $\langle x, y\rangle \in post_p \cap (C \times U)$, and to the claim $\langle x, y\rangle \in post_p \wedge x \in C$. Then the last claim can be formalized in Z3 as follow:

```
(define-fun restriction_rel ((x U)(y U)(p Prog)(c Set)) Bool
(and (select c x) (select (post p) x y)))
```

The feasibility of the restriction $C : p$ of p to C can be formalized as the formula $\forall x.[x \in Pre_{C:p} \Rightarrow \exists y.\langle x, y\rangle \in post_{C:p}]$. Then $C : p$ is feasible (for a class of models) if and only if the formula is valid (respectively for the class). Thus $C : p$ is not feasible if and only if the formula $x \in Pre_p \wedge \forall y.\neg\langle x, y\rangle \in post_{C:p}$ is satisfiable with respect to x.

```
(define-fun isnotfeasiblerest ((p Prog)(c Set)(x U)) Bool
 (and (select (pre p) x)
      (forall ((y U)) (not (restriction_rel x y p c)))))
```

Now we prove the feasibility preservation theorem for the restriction operator. The theorem can be formulated in Z3 as follows:

```
(declare-const p631 Prog)
(declare-const f631 (Array U U))
(declare-const c631 Set)
(declare-const x631 U)
(assert (isfeasible p631 f631))
(assert (isnotfeasiblerest p631 c631 x631))
; additional assumption: C \cup pre(p) is not empty.
(declare-const x632 U)
(assert (and (select c631 x632) (pre p631) x632)))
```

Since Z3 returns *sat* for the formula, the theorem does not hold. But it contradicts to the fact that the theorem indeed holds.

Z3 also shows a counterexample. We analyze the counterexample to find a flaw in original definitions and our formalization. Analysis of the counterexample shows the followings:

- Counter Example
  - Assumption: $U = \{a, b, c\}$
  - $p = \langle U, U, U \times U\rangle$
  - $C = \{a, c\}$
  - $f\colon U \to U$ as $f(u) = a$ for any $u \in U$
- Analysis
  - p is feasible ($\because f(u) = a$ for any $u \in U$)
  - $C \cap Pre_p$ is not empty. ($\because$ witness a)

By the definition ($C\colon p = \langle set_p, Pre_p, post_p \cap (C \times U)\rangle$) of the restriction, $C\colon p = \langle \{a, b, c\}, \{a, b, c\}, \{\langle a, U\rangle, \langle c, U\rangle\}\rangle$. Thus $C\colon p$ is not feasible since $b \in Pre_{C:\, p}$ and the formula $\neg\exists y.\langle b, y\rangle \in post_{C:\, p}$ holds for the counterexample.

We conjecture that the fact $b \in Pre_{C:\, p}$ is a cause of failure of the theorem since b is a witness of the fact that $C\colon p$ is not feasible. Then we re-define $Pre_{C:\, p} := Pre_p \cap C$ (not Pre_p). With this definition, the feasibility preservation theorem holds. Thus our conjecture is true.

4 Concluding Remarks

In this paper, with Meyer's formalization[2], we formalized specifications and their properties (theorems) as first-order logic formulas in the language of the SMT solver Z3. Then we proved theorems, i.e., the refinement theorem, the implementation theorem and the feasibility preservation theorems, with Z3.

Our aim is to show that automatic software verification (and theorem proving in mathematics) with an SMT solver is useful and feasible. We automatically proved theorems while the theorems are proved manually in [2]. Manual proofs are error prone and time consuming. On the other hand, the case study shows an advantage of formalization. Engineers are familiar with automatic theorem proving (verification) while manual proof requires expertise in mathematics. But automatic proof requires formalization of definitions and theorems. Formalization is a time consuming manual task and requires expertise. But manual formalization is less difficult than manual theorem proving. Moreover, the formalization helps to gain a deeper understanding of definitions and theorems since the formalization requires strict statements.

Reuse of function definitions is difficult since Skolemization is not bottom-up. For example, a formula representing that $x \in \mathit{dom}(p)$ cannot be uniformly replaced with a formula representing that $\exists y.(x, y) \in \mathit{post}_p$ (or $\exists f.(x, f(x)) \in \mathit{post}_p$) in general. A formula $\exists x.x \in \mathit{dom}(p)$ is equivalent to a formula $\exists x.\exists y.(x, y) \in \mathit{post}_p$ while a formula $\forall x.x \in \mathit{dom}(p)$ is equivalent to a formula $\exists f.\forall x.(x, f(x)) \in \mathit{post}_p$. Thus a framework for reusable functions definitions will be needed to describe complicate theorems.

A counterexample generated by Z3 supported to identify flaws (typos) of definitions in our case study. This counterexample-guided error correction approach will be applied to software verification in general. On the other hand, we proved the theorems only for models of size 3. Small Scope Hypothesis tells that almost all of bugs have small size counterexamples. But proofs for large sizes models will be required when an SMT solver cannot find any flaws of definitions. Thus we will prove other theorems in [2] to show the feasibility of the approach.

References

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

[2] Theory of Programs, Bertrand Meyer, arXiv (2015)

[3] Object-Oriented Software Construction 2nd ed., Bertrand Meyer, Prentice Hall (2000)

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

[5] Model Theory (Encyclopedia of Mathematics and its Applications), Wilfrid Hodges, Cambridge University Press (1993)