Specifying a Global Optimization Solver in Z

Angel F. Garcia Contreras and Yoonsik Cheon

TR #16-100
December 2016

Keywords: constraint solver, formal specification, global optimization, interval computation, specification-based testing, NumConSol, Z.

1998 CR Categories: D.2.4 [*Software Engineering*] Software/Program Verification — Class invariants, correctness proofs, formal methods; D.2.5 [*Software Engineering*] Testing and Debugging — testing tools (e.g., data generators, coverage testing); F.3.1 [*Logics and Meanings of Programs*] Specifying and Verifying and Reasoning about Programs — invariants, pre- and post-conditions, specification techniques; G.1.0 [*Mathematics of Computing*] General — interval arithmetic; G.1.6 [*Mathematics of Computing*] Optimization — constrained optimization, global optimization, unconstrained optimization.

Department of Computer Science
The University of Texas at El Paso
500 West University Avenue
El Paso, Texas 79968-0518, U.S.A.

Specifying a Global Optimization Solver in Z

Angel F. Garcia Contreras and Yoonsik Cheon
Department of Computer Science
The University of Texas at El Paso
El Paso, Texas, U.S.A.
Email: afgarciacontreras@miners.utep.edu, ycheon@utep.edu

***Abstract*—NumConSol is an interval-based numerical constraint and optimization solver to find a global optimum of a function. It is written in Python. In this document, we specify the NumConSol solver in Z, a formal specification language based on sets and predicates. The aim is to provide a solid foundation for restructuring and refactoring the current implementation of the NumConSol solver as well as facilitating its future improvements. The formal specification also allows us to design more effective testing for the solver, e.g., generating test cases from the specification.**

I. Introduction

Many real-life situations require to make a decision based on the best values for a set of parameters—e.g., when scientists try to find a fit between two sets of data or among a set of observations to a model. These situations in which we seek a maximum or minimum fit can be modeled as optimization problems. There are multiple flavors of optimization problems. It can be unconstrained, in which we only have a function whose parameters we seek to optimize. It can also be constrained, with a minimum contained within the search space delimited by a set of constraints.

We have developed an interval-based continuous optimization solver named NumConSol [1]. It implements a global optimization algorithm, a search algorithm that focuses on finding optimal values for a set of parameters in a problem, with a guarantee that there are no other values that can result in a better fit. The work on the NumConSol solver began in 2012. Its development was unstructured in that new program modules were introduced impromptu on a need basis to support new features and functionalities without following a specific set of development guidelines. This type of development is prone to design and programming errors. Even after significant refactoring of code in early 2016, many problems still keep appearing unpredictably. Some faults were detected only after manual examinations of long lists of execution logs. It is time consuming and requires a lot of manual efforts to uncover faults and identify their causes. The process isn't repeatable.

The NumConSol solver is designed on a strong mathematical foundation. Thus, we belive that its behavior can be nicely captured and specified in a formal specification language like Z [2] [3] to provide a solid foundation for improving its implementation. Based on its formal specification, for example, the solver may be redesigned and its source code may be restructured or refactored. It is also possible to generate a suite of test cases from the specification. In fact, such a test suite—one derived from a formal specification—is said to be more effective in detecting faults than manually created one. We think that the Z notation is a good fit for specifying the solver because it is based on the standard mathematical notation used in axiomatic set theory, lambda calculus, and first-order predicate logic. The Z notation is very concise and flexible.

In Section II below we first provide the background knowledge needed to understand the NumConSol solver, including interval computations and global optimization algorithms. In Section III we develop Z specifications of the NumConSol solver, focusing on those program modules that showed most faults and bugs recently. In Section IV we describe our approach for systematically generating test cases from the specifications written in Section III. In Section V we conclude this document with a concluding remark.

II. Background

An *unconstrained global optimization problem* is defined as: $\min_{x \in X} f(x)$, where $f : \mathbb{R}^n \to \mathbb{R}$ is an objective function. To solve an optimization problem, we use *search algorithms*. There are two types of search algorithms. The first, *local search*, starts on an initial guess of the parameter values. This search type attempts to improve the guessed value by iteratively generating new point guesses until it cannot find an improvement. On the right circumstances, these algorithms converge quickly to a solution, but can only guarantee that their solution is the best solution in the neighborhood of the initial guess. In contrast, a *global search* algorithm considers the entire search space instead of just a single point. The strategy is to iteratively reduce the search space until it generates a sub-space with a size smaller than a certain threshold. This narrow sub-space contains the solution candidate. Global search is exhaustive, as it covers the entire search space. The advantage of this search type is that these algorithms can guarantee that their solutions are the global optimums: there is no better value in the entire search space. This guarantee comes at a cost in performance, as the operations that account for the entire domain are more computationally-intensive.

One of the most well-known exhaustive search algorithms is a *branch-and- bound* (*B&B*) algorithm [4]. This algorithm implements a divide-and-conquer approach. It iteratively divides the search space into smaller sub-spaces, evaluating the likelihood of each sub-space to contain the global solution. Many B&B solvers, such as GlobSol [5] and IbexOpt [6] use

a class of B&B algorithms known as *interval branch-bound* (*IB&B*) algorithms. These algorithm use *interval computations* to model the search space and to compute the bounds on the solution candidates as well (see below for interval computations). Through interval computations, IB&B algorithms can guarantee the global optimality of their solutions.

A. Interval Computations

An *interval* $\boldsymbol{x_i} = \left[\underline{x_i}, \overline{x_i}\right]$ is the set of all real numbers x such that $\underline{x_i} \leq x \leq \overline{x_i}$ [7]. We denote the set of all such intervals as $\mathbb{I}$; it's the universe of intervals. A *box* $\boldsymbol{X}$ is a Cartesian product of intervals, e.g., $\boldsymbol{X} = \boldsymbol{x_1} \times \boldsymbol{x_2} \times \ldots \times \boldsymbol{x_n} \in \mathbb{I}^n$. In interval computations, real arithmetic operations are extended to work with intervals, and each arithmetic operator $\bowtie \in \{+, -, \times, \div\}$ satisfies the following property.

$$\{x \bowtie y \mid x \in \boldsymbol{x}, y \in \boldsymbol{y}\} \subseteq \boldsymbol{x} \bowtie \boldsymbol{y}$$

Likewise, it is possible to extend a real-valued function f to an interval function $\boldsymbol{f}$. The most basic extension is the natural extension, achieved by systematically extending each of the operators of their symbolic expressions to interval arithmetic. In general, interval extensions of real-valued functions ensure that:

$$\forall \boldsymbol{X} \in \mathbb{I}^n \bullet \{f(x) \mid x \in \boldsymbol{X}\} \subseteq \boldsymbol{f}(\boldsymbol{X})$$

Figure 1 is a graphical representation of the interval evaluation of a function. More details about interval analysis can be found in [8].

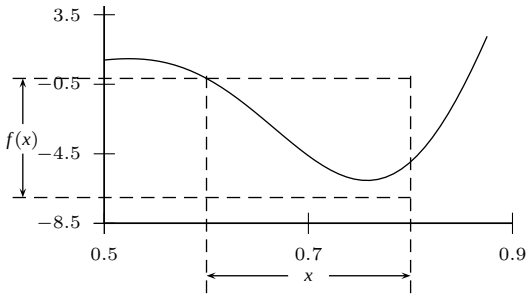

Fig. 1. Interval evaluation of a function

B. Interval Contractors

Constraints limit the search space of a problem. By using interval analysis and the constraints of a problem, it is possible to reduce the search space. An interval contractor attempts to reduce the size of the search space by iterating through the constraints of the problem and discarding those portions of the search space that violate the constraints. The NumConSol solver uses HC4 [9], a popular interval contractor that structures the expressions of the constraints as *attribute trees* in which each non-terminal node is a function or operator, and all leaves are either variables or constants. These attribute trees use inherited and synthesized attributes used for evaluation and contraction. In NumConSol, these attribute trees are represented as abstract syntax trees (see Section III).

C. Branch and Prune

Interval branch-and-bound (IB&B) algorithms use interval computations to model the search space and obtain the bounds on the optimum. IB&B algorithms can be improved by adding a contractor that *prunes* the infeasible regions of each subspace. This new algorithm is known as *branch-and-prune* (*B&P*). The NumConSol solver uses a B&P algorithm with additional constraint heuristics [10].

D. NumConSol Implementation

The NumConSol solver is an interval constraint and optimization solver written in Python (see Section III). Its implementation consists of several program modules, including:

- Branch and prune algorithm
- Speculation with B&P grafting
- Interval arithmetic
- HC4 contractor
- Abstract syntax trees with attributes for HC4
- AST evaluation and differentiation

Its initial code was written impromptu with few prior planning. It had a negative impact on the quality of the code, and several program modules were stricken with numerous faults. The most error-prone modules are the interval arithmetic module, the AST evaluation module, and the AST differentiation module. In particular, many faults were detected in the last two modules; some were hard to find, requiring manual inspections of long lists of execution logs. Once detected, however, many of these faults were simple enough to correct. We believe that the lack of rigorous design and testing causes the prevalence of easy-to-fix but hard-to-find faults and errors.

In the next section, we specify the behavior of the three most fault-stricken program modules to provide a formal basis for their redesign and code refactoring as wells as future feature improvements.

III. Formal Specifications

Software developers use formal methods to specify, design and test software. The use of formal methods increases the quality of the software due to specifications written in mathematic-based notations. In this section, we specify the following program modules of the NumConSol solver.

- Interval arithmetic module
- AST model module
- AST evaluation module
- AST differentiation module

We use the Z notation to write our specifications. The Z notation is a formal specification language based on set theory and mathematical logic [2] [3]. Our Z specifications are abstract in that they abstract away from programming language-specific concepts and constructs such as Python data types, functions, and classes. The key responsiblities of each program module are identified and formulated as operations in an axiomatic style. For example, the following Z axiomatic definition specifies the interval addition ($+$) operation.

$$\_ + \_ : \mathit{Interval} \times \mathit{Interval} \to \mathit{Interval}$$

$$\begin{aligned}&\forall i, j : \mathit{Interval} \bullet \\ &\quad (i + j).\mathit{lb} = i.\mathit{lb} + j.\mathit{lb} \\ &\quad (i + j).\mathit{ub} = i.\mathit{ub} + j.\mathit{ub}\end{aligned}$$

The declaration above the horizontal line states that + is a binary infix operation from a pair of Interval to an Interval. It is a total function. Z uses different arrow symbols to denote different types of functions, e.g., total ($\to$), partial ($\nrightarrow$), injective ($\rightarrowtail$), and relation($\leftrightarrow$). The predicate below the horizontal line expresses the constraint upon the object introduced in the declaration. It states that the lower/upper bounds of the interval $i + j$ is the sum of the lower/upper bounds of i and j. An interval has a lower bound (*lb*) and a upper bound (*ub*) (see Section III-A) for the definition of Interval).

A. Interval Arithmetic Module

The Interval Arithmetic module implements the interval arithmetic described in Section II-A. An interval is modeled as a set of real numbers (*values*) between two bounds (*lb* and *ub*).

$$\begin{aligned}&\mathit{Interval} \\ &\mathit{lb} : \mathbb{R} \\ &\mathit{ub} : \mathbb{R} \\ &\mathit{values} : \mathbb{P}\mathbb{R} \\ &\hline \\ &\mathit{lb} \leq \mathit{ub} \\ &\mathit{values} = \{x : \mathbb{R} \mid \mathit{lb} \leq x \land x \leq \mathit{ub}\}\end{aligned}$$

In Z, an open box called a *schema* defines a composite type, one with a variety of different components or attributes; it can also be used to define an operation. The predicate part of the schema states that the lower bound (*lb*) should be less than or equal to the upper bound (*ub*). Thus, an empty interval can't be modeled. The *values* attribute is a derived attribute in the sense that it is defined by the two bounds. This attribute makes it easy to specify some interval operations (see below).

We also define a helper function to construct an interval. It is a partial function defined on (x, y) pairs, where $x \leq y$. In the definition, the μ-notation, called the *definite description*, denotes a unique object that satisfies the specified constraint.

$$[\_, \_] : \mathbb{R} \times \mathbb{R} \nrightarrow \mathit{Interval}$$

$$\begin{aligned}&\forall x, y : \mathbb{R} \mid x \leq y \bullet \\ &\quad [x, y] = (\mu\ i : \mathit{Interval} \mid i.\mathit{lb} = x \land i.\mathit{ub} = y)\end{aligned}$$

We define operations to support interval computations, including the following.

- *Inclusion* ($\in$): membership test
- *Hull* ($\Box$): interval enclosing a set of intervals
- *Intersection* ($\cap$): intersection of two intervals
- *Width* (*wid*): size of an interval
- *Mid point* (*mid*): middle of an interval
- *Magnitude* (*mig*): minimum absolute value of an interval
- *Magnitude* (*mag*): maximum absolute value of an interval
- *Bisection* (*bisect*): two intervals splitting an interval at its middle point

All of these operations are defined as Z functions that either take intervals or returns intervals. The definitions of some of these operations are shown below; the rest can be found in the appendix. Some operations are specified in terms of the bounds of an interval and others in terms of the values.

$$\_ \in \_ : \mathbb{R} \leftrightarrow \mathit{Interval}$$

$$\forall x : \mathbb{R}, i : \mathit{Interval} \bullet x \in i \Leftrightarrow x \in i.\mathit{values}$$

$$\Box\_ : \mathbb{F}_1\ \mathit{Interval} \to \mathit{Interval}$$

$$\begin{aligned}&\forall s : \mathbb{F}_1\ \mathit{Interval} \bullet \\ &\quad \Box s = [\min\{i : s \bullet i.\mathit{lb}\}, \max\{i : s \bullet i.\mathit{ub}\}]\end{aligned}$$

$$\_ \cap \_ : \mathit{Interval} \times \mathit{Interval} \nrightarrow \mathit{Interval}$$

$$\begin{aligned}&\forall i, j : \mathit{Interval} \mid i.\mathit{values} \cap j.\mathit{values} \neq \emptyset \bullet \\ &\quad (i \cap j).\mathit{values} = i.\mathit{values} \cap j.\mathit{values}\end{aligned}$$

$$\mathit{wid} : \mathit{Interval} \to \mathbb{R}$$

$$\forall i : \mathit{Interval} \bullet \mathit{wid}\ i = i.\mathit{ub} - i.\mathit{lb}$$

$$\mathit{bisect} : \mathit{Interval} \to \mathit{Interval} \times \mathit{Interval}$$

$$\begin{aligned}&\forall i : \mathit{Interval} \bullet \\ &\quad \mathit{bisect}\ i = ([i.\mathit{lb}, \mathit{mid}\ i], [\mathit{mid}\ i, i.\mathit{ub}])\end{aligned}$$

There are a few things to note. The intersection ($\cap$) operation is a partial function. It is defined only for pairs of intervals that have a common set of values. This is mainly because an empty interval is not modeled. The *bisect* operation states explicitly that the middle point is included in the two (split) intervals. Being able to be precise about this kind of boundary values is one benefit of using formal specifications.

We also need to promote real arithmetic operations such as addition (+), subtraction (-), multiplication ($\times$), division ($\div$), and power ($\hat{}$) to intervals. For this, we define a higher-order function ($\bar{}$) that takes a real operation and returns an interval version; e.g., $\bar{+}$ denotes the interval addition operation.

$$\_\bar{}\_ : (\mathbb{R} \times \mathbb{R} \nrightarrow \mathbb{R}) \to (\mathit{Interval} \times \mathit{Interval} \nrightarrow \mathit{Interval})$$

$$\begin{aligned}&\forall \diamond : \mathbb{R} \times \mathbb{R} \nrightarrow \mathbb{R}, i, j : \mathit{Interval} \bullet \\ &\quad \mathbf{let}\ s == \{x : i.\mathit{values}, y : j.\mathit{values} \bullet x \diamond y\} \bullet \\ &\qquad i \bar{\diamond} j = [\min s, \max s]\end{aligned}$$

It is also possible to define the interval arithmetic operations explicitly. For example, the addition operation ($\bar{+}$) can be defined as follows.

$$\_\bar{+}\_ : \mathit{Interval} \times \mathit{Interval} \to \mathit{Interval}$$

$$\begin{aligned}&\forall i, j : \mathit{Interval} \bullet \\ &\quad (i \bar{+} j).\mathit{lb} = \min\{x : i.\mathit{values}, y : j.\mathit{values} \bullet x + y\} \\ &\quad (i \bar{+} j).\mathit{ub} = \max\{x : i.\mathit{values}, y : j.\mathit{values} \bullet x + y\}\end{aligned}$$

Similarly, we promote real mathematical functions such

as natural logarithm, natural exponent, sine, cosine, tangent, secant, cosecant, and cotangent to intervals by defining the following higher-order function.

$$\bar{\_} : (\mathbb{R} \nrightarrow \mathbb{R}) \to (\mathit{Interval} \nrightarrow \mathit{Interval})$$

$$\forall f : \mathbb{R} \nrightarrow \mathbb{R}, i : \mathit{Interval} \bullet \\ \quad \mathbf{let}\ s == \{x : i.\mathit{values} \bullet f\ x\} \bullet \\ \quad\quad \bar{f}\ i = [\min s, \max s]$$

Another responsibility of the Interval module is to model the search domains of the problems to be solved. An *interval box* represents the search domain of a problem by specifying a variable's domain represented as an interval (see Section III-B for variables). An interval box is naturally modeled as a mapping from variable names to intervals.

$$[\mathit{Name}]$$
$$\mathit{Box} == \mathit{Name} \nrightarrow \mathit{Interval}$$

We define a few operations to manipulate interval boxes. A box can be updated to add a new variable or change an existing one. The intersection of two boxes can be obtained as another box.

$$\mathit{update} : \mathit{Box} \times \mathit{Name} \times \mathit{Interval} \to \mathit{Box}$$

$$\forall b : \mathit{Box}, n : \mathit{Name}, i : \mathit{Interval} \bullet \\ \quad \mathit{updat}(b, n, i) = b \oplus \{n \mapsto i\}$$

$$\_ \cap_b \_ : \mathit{Box} \times \mathit{Box} \nrightarrow \mathit{Box}$$

$$\forall b, c : \mathit{Box} \mid b \cap_? c \bullet \\ \quad b \cap c = \{n : \operatorname{dom} b \cap \operatorname{dom} c \bullet n \mapsto b\ n \cap c\ n\}$$

$$\_ \cap_? \_ : \mathit{Box} \leftrightarrow \mathit{Box}$$

$$\forall b, c : \mathit{Box} \bullet \\ \quad b \cap_? c \Leftrightarrow (\exists n : \mathit{Name} \bullet n \in \operatorname{dom} b \cap \operatorname{dom} c) \\ \quad\quad \land\ (\forall n : \operatorname{dom} b \cap \operatorname{dom} c \bullet b\ n \cap_? c\ n)$$

In the definition of the *update* function, the $\oplus$ symbol denotes relational overriding; e.g., $f \oplus g$ is a relation that agrees with f everywhere outside the domain of g but agrees with g where g is defined. The box intersection ($\cap_b$) is a partial function. It's defined only when the two boxes have common variables and all such variables' interval intersections are defined; see the appendix for $\cap_?$ defined on a pair of intervals.

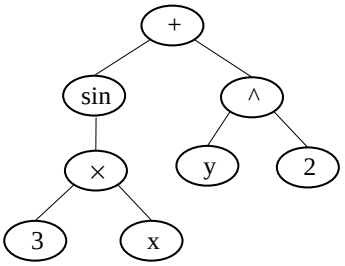

Fig. 2. Abstract syntax tree for $\mathit{sin}(3x) + y^2$

B. Abstract Syntax Tree (AST) Module

The Abstract Syntax Trees (AST) module is to represent the problems to be solved—mathematical expressions to be evaluated to obtain the interval range of the objective. A mathematical expression is modeled as an *abstract syntax tree* (*AST*), a tree representation of the abstract syntactic structure of program source code (see Figure 2). An expression may consists of different kinds of subexpressions represented as different node types, including:

- *constant*: leaf node containing a number.
- *variable*: leaf node containing a variable.
- *binary operation*: none-leaf node denoting a binary arithmetic operation such as addition and multiplication.
- *unary operation*: none-leaf node denoting a unary arithmetic operation such as prefix minus.
- *function*: none-leaf node denoting a unary mathematical function such as $\log$ and $\sin$.

Since an AST has a recursive structure, we model it using Z free types. A free type introduce a new data type by defining constants and constructor functions.

$$\mathit{OPR} ::= \mathit{plus} \mid \mathit{minus} \mid \mathit{times} \mid \mathit{divide} \mid \mathit{power}$$

$$\mathit{FUN} ::= \mathit{ln} \mid \mathit{exp} \mid \mathit{sin} \mid \mathit{cos} \mid \mathit{tan} \mid \mathit{sec} \mid \mathit{csc} \mid \mathit{cot}$$

$$\begin{aligned} \mathit{AST} ::= & \ \mathit{const}\langle\langle \mathbb{R} \rangle\rangle \\ \mid & \ \mathit{var}\langle\langle \mathit{Name} \rangle\rangle \\ \mid & \ \mathit{unary}\langle\langle \mathit{OPR} \times \mathit{AST} \rangle\rangle \\ \mid & \ \mathit{binary}\langle\langle \mathit{OPR} \times \mathit{AST} \times \mathit{AST} \rangle\rangle \\ \mid & \ \mathit{fun}\langle\langle \mathit{FUN} \times \mathit{AST} \rangle\rangle \end{aligned}$$

The first two free type definitions introduce two sets (*OPR* and *FUN*) and distinct constants for all the arithmetic operations and mathematical functions supported by the NumConSol solver. The last free type definition introduces a new set *AST* by defining a set of *constructor functions*. Each constructor function like *const* is an injective function whose target is the set *AST*. It would be instructive to represent the expression of Figure 2, $\mathit{sin}(3x) + y^2$, in Z.

$$\begin{aligned} & \mathit{binary}(\mathit{plus}, \\ & \quad \mathit{fun}(\mathit{sin}, \mathit{binary}(\mathit{times}, \mathit{const}\ 3, \mathit{var}\ x)), \\ & \quad \mathit{binary}(\mathit{power}, \mathit{var}\ y, \mathit{const}\ 2)) \end{aligned}$$

C. AST Evaluation Module

The AST Evaluation module implements an algorithm for evaluating an expression represented as an AST. We specify the algorithm as a recursive function, $\mathcal{E}$, that takes an AST and an interval box and returns an interval. Recall that an interval box is a mapping from variable names to intervals, specifying variables' search domains (see Section III-A). Two basis cases for leaf nodes are specified below.

$$\mathcal{E} : \mathit{AST} \times \mathit{Box} \nrightarrow \mathit{Interval}$$

$$\forall x : \mathbb{R}, b : \mathit{Box} \bullet \\ \quad \mathcal{E}(\mathit{const}\ x, b) = [x, x] \\ \forall n : \mathit{Name}, b : \mathit{Box} \mid n \in \operatorname{dom} b \bullet \\ \quad \mathcal{E}(\mathit{var}\ n, b) = b\ n$$

When the AST is a constant, the result is an interval consisting of only the constant. The lower bound of the interval is equal to its upper bound. For a variable, the result is the variable's interval taken from the interval box. It's a partial function, and the box is sort of an evaluation environment to provide a value for a variable.

When the AST is a unary minus operation, the result is calculated recursively by first evaluating the operand and then applying the interval version of the minus operation ($\bar{-}$). Recall that $\bar{-}$ denotes a promotion of the real minus ($-$) operation to intervals (see Section III-A).

$$\forall a : AST,\ b : Box \bullet \\ \mathcal{E}(unary(minus, a), b) = \bar{-}\mathcal{E}(a, b)$$

The evaluations of binary operations and mathematical functions are defined recursively in a similar fashion.

$$\forall a : AST,\ b : Box \bullet \\ \mathcal{E}(binary(plus, a_1, a_2), b) = \mathcal{E}(a_1, b) \mathbin{\bar{+}} \mathcal{E}(a_2, b) \\ \mathcal{E}(binary(minus, a_1, a_2), b) = \mathcal{E}(a_1, b) \mathbin{\bar{-}} \mathcal{E}(a_2, b) \\ \mathcal{E}(binary(times, a_1, a_2), b) = \mathcal{E}(a_1, b) \mathbin{\bar{\times}} \mathcal{E}(a_2, b) \\ \mathcal{E}(binary(divide, a_1, a_2), b) = \mathcal{E}(a_1, b) \mathbin{\bar{\div}} \mathcal{E}(a_2, b) \\ \mathcal{E}(binary(power, a_1, a_2), b) = \mathcal{E}(a_1, b) \mathbin{\bar{\wedge}} \mathcal{E}(a_2, b)$$

$$\forall a : AST,\ b : Box \bullet \\ \mathcal{E}(fun(ln, a), b) = \bar{\ln}\,(\mathcal{E}(a, b)) \\ \mathcal{E}(fun(exp, a), b) = \bar{\exp}\,(\mathcal{E}(a, b)) \\ \mathcal{E}(fun(sin, a), b) = \bar{\sin}\,(\mathcal{E}(a, b)) \\ \mathcal{E}(fun(cos, a), b) = \bar{\cos}\,(\mathcal{E}(a, b)) \\ \mathcal{E}(fun(tan, a), b) = \bar{\tan}\,(\mathcal{E}(a, b)) \\ \mathcal{E}(fun(sec, a), b) = \bar{\sec}\,(\mathcal{E}(a, b)) \\ \mathcal{E}(fun(csc, a), b) = \bar{\csc}\,(\mathcal{E}(a, b)) \\ \mathcal{E}(fun(cot, a), b) = \bar{\cot}\,(\mathcal{E}(a, b))$$

D. AST Differentiation Module

This module is responsible for generating the partial derivative of an expression. Partial derivatives are used in various modules of the NumConSol solver, such as variable selection for node bisection and certain types of interval contractors. We specify the behavior of this module as a function, $\mathcal{D}$, that takes an AST and a variable name and returns an AST. The function is defined recursively based on the structure of the argument AST. The bases are of course the leaf node types: constant and variable.

$$\mathcal{D} : AST \times Name \to AST$$

$$\forall x : \mathbb{R}, n : Name \bullet \mathcal{D}(const\ x, n) = const\ 0 \\ \forall m, n : Name \bullet \\ m = n \Rightarrow \mathcal{D}(var\ m, n) = const\ 1 \\ m \neq n \Rightarrow \mathcal{D}(var\ m, n) = const\ 0$$

For all none-leaf node types, the function is defined recursively by first calculating the partial derivatives of the operands and then combining them. Below we show one representative definition for each node type; the complete definition of the function is found in the appendix.

$$\forall a : AST,\ n : Name \bullet \\ \mathcal{D}(unary(minus, a), n) = unary(minus, \mathcal{D}(a, n))$$

$$\forall a_1, a_2 : AST,\ n : Name \bullet \\ \mathcal{D}(binary(plus, a_1, a_2), n) = \\ binary(plus, \mathcal{D}(a_1, n), \mathcal{D}(a_2, n))$$

$$\mathcal{D} : AST \times Name \to AST$$

$$\forall a : AST,\ n : Name \bullet \\ \mathcal{D}(fun(sin, a), n) = \\ binary(times, fun(cos, a), \mathcal{D}(a, n))$$

IV. Using Formal Specifications

A formal specification describes system properties, from which one can systematically specify, develop, and verify systems [11]. The precision of a formal specification language like Z forces one to express requirements and design decisions in an unambiguous way, thus exposing unclear points in a specification and avoiding misconceptions. A formal specification is useful not only in the early phases of software development but also in the later development phases such as testing and maintenance. In fact, one motivation of our work is to provide a formal basis for refactoring and restructuring of the current NumConSol implementation as wells as its future improvements. Formalizing the NumConSol solver allowed us to uncover hidden assumptions (e.g., nonempty intervals and partial operations) and fine boundaries (e.g., split intervals), which caused subtle and hard-to-find bugs in our code. They were documented explicitly and concisely in our specifications. The use of Z also let us to be explicit about the signature (parameters and return types) of operations, which is another common source of subtle bugs in a dynamically-typed language like Python.

Another motivation of our work is to provide a solid foundation for automatically testing the NumConSol solver. The current, manual testing is not only tedious and time consuming but also error-prone. It's simply not effective for regression testing; it isn't able to provide a safety net for frequent code refactoring and improvements of the solver. The ultimate goal is to automate tests significantly from test case generation to test execution. However, a practical, short-term goal is to automate test execution using a popular testing framework like PHPUnit [12]. The first step toward this goal is to generate test cases systematically from formal specifications written in Z [13] [14] [15]. It is said that test cases generated from formal specifications are more effective that manually generated ones, and formal specifications can also be turned into test oracles to determine test outcomes [16] [17]. We created a test plan to test the NumConSol modules. Our test plan uses several black-box testing techniques, including equivalence partitioning, boundary value analysis, and grammar-based techniques [18] [19] [20]. Below we describe several sample test cases to illustrate our approach for generating test cases from Z specifications.

1) Equivalence partitioning: This technique, also known as *equivalence classes technique*, divides the input data of a program into partitions of equivalent data (equivalence classes) from which test cases can be derived. Test cases are created in such a way to cover each partition at least once. This technique works for all our program modules. To partition the input data into equivalence classes, we look at the specification of an operation. In principle, if two different input values produce the same output value, they belong to the same equivalence class; otherwise, they belong to different equivalence classes. For example, the inclusion ($\in$) operation of the Interval Arithmetic module has a specification, $x \in i \Leftrightarrow x \in i.values$, where $x : \mathbb{R}$ and i : *Interval* (see Section III-A). Thus, we can define two equivalence classes: $\{(x,i) \mid x \in i\}$ and $\{(x,i) \mid x \notin i\}$. A sample test suite covering all equivalence classes is $\{(5,[1,10]) \mapsto \mathit{true}, (0,[1,10]) \mapsto \mathit{false}\}$, where the last element of a test case is the expected output.

If an operation is partial, we also need to introduce an equivalence class for invalid data to perform negative testing—testing unexpected behavior. In the Interval Arithmetic module, there are several such operations, including interval construction ($[\_,\_]$) and interval intersection ($\cap$). For example, the intersection operation has a constraint $i.values \cap j.values \neq \emptyset$ in its definition, as shown below.

$$\forall i, j : Interval \mid i.values \cap j.values \neq \emptyset \bullet (i \cap j).values = i.values \cap j.values$$

Thus, a negative equivalence class for the intersection operation is $\{(i,j) \mid i.values \cap j.values = \emptyset\}$, where i and j are intervals. And a sample test case to cover this equivalence class is $([1,10],[20,30]) \mapsto \perp$, where $\perp$ denotes an implementation-specific way of indicating an error situation, e.g., returning a special error value or throwing an exception.

It isn't often straightforward to determine whether an operation is total or partial. As an example, consider the hull ($\square$) operation of the Interval Arithmetic module, whose definition is copied below.

$$\square\_ : \mathbb{F}_1\ Interval \to Interval$$
$$\forall s : \mathbb{F}_1\ Interval \bullet \square s = [\min\{i : s \bullet i.lb\}, \max\{i : s \bullet i.ub\}]$$

According to its signature, it's a total function ($\to$). However, note that its domain is $\mathbb{F}_1$, a non-empty finite powerset. $\mathbb{F}_1$ is a short-hand notation for $\mathbb{F}$ (finite powerset) with an implicit constraint of being not empty. Thus, the above definition can be re-written or *normalized* to:

$$\square\_ : \mathbb{F}\ Interval \pfun Interval$$
$$\forall s : \mathbb{F}\ Interval \mid \#s > 0 \bullet \square s = [\min\{i : s \bullet i.lb\}, \max\{i : s \bullet i.ub\}]$$

It is now clear that the operation is partial ($\pfun$). Its definition has an explicitly-written constraint, $\#s > 0$, in the quantifier.

2) Boundary value analysis: One way to improve the effectiveness of equivalence partitioning is to perform boundary value analysis, in which test cases are designed to include representatives of boundary values in a range. This technique works very well for testing the Interval Arithmetic module because every interval has clearly defined boundaries, i.e., lower and upper bounds. For the interval inclusion operation, $x \in i$, where x is a real number and i is an interval, for example, we can identify six boundary values for x: $i.lb - \delta$, $i.lb$, $i.lb + \delta$, $i.ub - \delta$, $i.ub$, and $i.ub + \delta$, where δ is a threshold value specifying the accuracy of real numbers. We can also define the boundary values of the intersection of two intervals, say for testing the interval intersection ($\cup$) operation.

3) Grammar-based techniques: The use of AST in the NumConSol solver and its formalization using the Z free type poses an interesting opportunity for generating test cases systematically for program modules such as the AST Evaluation module and the AST Differentiation module that manipulate ASTs. Our test plan for testing these modules is grammar-based testing. A grammar-based technique generates test cases based on grammar rules [17] [20]. A valid test case is a sentence derived from the grammar rules. In our case, the grammar rules are the Z free type definitions, and sentences (test cases) are ASTs. Since grammar-based test-data generation allows one to explore the grammar rules and grammatical patterns more systematically, it will be generally more effective than manual test-data generation.

V. Conclusion

We specified in Z the behavior of several NumConSol program modules, those modules stricken with faults and bugs. NumConSol is an interval-based numerical constraint and optimization solver to find a global optimum of a function. It is written in Python. By writing formal specifications we were able to uncover many hidden assumptions and fine boundary cases, which oftentimes lead to subtle faults and bugs that are hard to detect. The use of Z also allows us to document explicitly the signature of operations (parameter types and return types). We also created a test plan to test the NumConSol program modules. The key of our test plan is to generate test cases systematically from the formal specifications of the modules using a combination of several black box testing techniques such as equivalence partitioning, boundary value analysis, and grammar-based techniques. We expect that our formal specifications be useful in redesigning and refactoring the NumConSol solver as well as improving its functionalities. Our future work is to carry out such tasks by utilizing the formal specifications.

References

[1] A. F. Garcia C., "Contributions to global optimization using interval methods and speculation," M.S. Thesis, The University of Texas at El Paso, Dec. 2014.

[2] J. M. Spivey, *Understanding Z: a Specification Language and its Formal Semantics*. NY: Cambridge University Press, 1988.

[3] ——, *The Z Notation: A Reference Manual*, ser. International Series in Computer Science. NY: Prentice-Hall, 1989.

[4] A. H. Land and A. G. Doig, "An automatic method of solving discrete programming problems," *Econometrica*, vol. 28, no. 3, pp. 497–520, 1960.

[5] R. B. Kearfott, "Globsol user guide," *Optimization Methods Software*, vol. 24, no. 4-5, pp. 687–708, Aug. 2009.

[6] G. Trombettoni, I. Araya, B. Neveu, and G. Chabert, "IbexOpt : un module d'optimisation globale sous contraintes fiable," in *13e congrès annuel de la Société française de Recherche Opérationnelle et d'Aide à la Décision*, France, 2012.

[7] R. E. Moore, *Interval Analysis*. Prentice-Hall, 1996.

[8] R. E. Moore, R. B. Kearfott, and M. J. Cloud, *Introduction to interval analysis*. Siam, 2009.

[9] F. Benhamou, F. Goualard, L. Granvilliers, and J. Puget, "Revising hull and box consistency," in *Proc. of the 1999 Int. Conf. on Logic Programming*. Cambridge, MA, USA: MIT, 1999, pp. 230–244.

[10] A. F. Garcia C. and M. Ceberio, "Comparison of strategies for solving global optimization problems using speculation and interval computations," in *Annual meeting of the North American Fuzzy Information Processing Society*, 2016.

[11] J. M. Wing, "A specifier's introduction to formal methods," *Computer*, vol. 23, no. 9, pp. 8–23, Sep. 1990.

[12] "Phpunit," https://phpunit.de, accessed: December 20, 2016.

[13] S. Helke, T. Neustupny, and T. Santen, "Automating test case generation from Z specifications with Isabelle," in *Proceedings of the 10th International Conference of Z Users on The Z Formal Specification Notation*, ser. ZUM '97. London, UK: Springer-Verlag, 1997, pp. 52–71.

[14] R. M. Hierons, "Testing from Z specification," *Journal of Software: Testing, Verification and Reliability*, vol. 7, pp. 19–33, 1997.

[15] H. Singh, M. Conrad, and S. Sadeghipour, "Test case design based on Z and the classification-tree method," in *First IEEE International Conference on Formal Engineering Methods*. IEEE Computer Society, 1997, pp. 81–90.

[16] Y. Cheon and G. T. Leavens, "A simple and practical approach to unit testing: The JML and JUnit way," in *ECOOP 2002 — Object-Oriented Programming, 16th European Conference, Máalaga, Spain, Proceedings*, ser. Lecture Notes in Computer Science, B. Magnusson, Ed., vol. 2374. Berlin: Springer-Verlag, Jun. 2002, pp. 231–255.

[17] I. Enderlin, A. Giorgetti, F. Dadeau, and F. Bouquet, "Grammar-based testing using realistic domains in PHP," in *IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST 2012)*. IEEE Computer Society, 2012, pp. 509–518.

[18] R. M. Hierons, K. Bogdanov, J. P. Bowen, R. Cleaveland, J. Derrick, J. Dick, M. Gheorghe, M. Harman, K. Kapoor, P. Krause, G. Lüttgen, A. J. H. Simons, S. Vilkomir, M. R. Woodward, and H. Zedan, "Using formal specifications to support testing," *ACM Comput. Surv.*, vol. 41, no. 2, pp. 9:1–9:76, Feb. 2009.

[19] G. J. Myers and C. Sandler, *The Art of Software Testing*. John Wiley & Sons, 2004.

[20] H.-F. Guo and Z. Qiu, "Automatic grammar-based test generation," in *Testing Software and Systems: 25th IFIP WG 6.1 International Conference, ICTSS 2013, Istanbul, Turkey, November 13-15, 2013, Proceedings*. Springer, 2013, pp. 17–32.

APPENDIX

A. Interval Arithmetic

Interval

$lb : \mathbb{R}$
$ub : \mathbb{R}$
$values : \mathbb{P}\mathbb{R}$

$lb \leq ub$
$values = \{x : \mathbb{R} \mid lb \leq x \land x \leq ub\}$

$[\_,\_] : \mathbb{R} \times \mathbb{R} \pfun Interval$

$\forall x, y : \mathbb{R} \mid x \leq y \bullet$
$[x,y] = (\mu\ i : Interval \mid i.lb = x \land i.ub = y)$

$\_ \in \_ : \mathbb{R} \rightarrow Interval$

$\forall x : \mathbb{R}, i : Interval \bullet x \in i \Leftrightarrow x \in i.values$

$\Box\_ : \mathbb{F}_1\ Interval \rightarrow Interval$

$\forall s : \mathbb{F}_1\ Interval \bullet$
$\Box s = [\min\{i : s \bullet i.lb\}, \max\{i : s \bullet i.ub\}]$

$\_ \cap \_ : Interval \times Interval \pfun Interval$

$\forall i, j : Interval \mid i \cap_? j \bullet$
$(i \cap j).values = i.values \cap j.values$

$\_ \cap_? \_ : Interval \rightarrow Interval$

$\forall i, j : Interval \bullet$
$i \cap_? j \Leftrightarrow i.values \cap j.values \neq \emptyset$

$wid : Interval \rightarrow \mathbb{R}$

$\forall i : Interval \bullet wid\ i = i.ub - i.lb$

$mid : Interval \rightarrow \mathbb{R}$

$\forall i : Interval \bullet mid\ i = (i.ub + i.lb)/2$

$mig : Interval \rightarrow \mathbb{R}$

$\forall i : Interval \bullet mig\ i = \min\{\forall x : i.values \bullet | x |\}$

$mag : Interval \rightarrow \mathbb{R}$

$\forall i : Interval \bullet mag\ i = \max\{\forall x : i.values \bullet | x |\}$

$bisect : Interval \rightarrow Interval \times Interval$

$\forall i : Interval \bullet$
$bisect\ i = ([i.lb, mid\ i], [mid\ i, i.ub])$

$\_\bar{}\_ : (\mathbb{R} \times \mathbb{R} \pfun \mathbb{R}) \rightarrow (Interval \times Interval \pfun Interval)$

$\forall \diamond : \mathbb{R} \times \mathbb{R} \pfun \mathbb{R}, i, j : Interval \bullet$
$\textbf{let}\ s == \{x : i.values, y : j.values \bullet x \diamond y\} \bullet$
$i \bar{\diamond} j = [\min s, \max s]$

$\bar{}\_ : (\mathbb{R} \pfun \mathbb{R}) \rightarrow (Interval \pfun Interval)$

$\forall f : \mathbb{R} \pfun \mathbb{R}, i : Interval \bullet$
$\textbf{let}\ s == \{x : i.values \bullet f\ x\} \bullet$
$\bar{f}\ i = [\min s, \max s]$

$[Name]$
$Box == Name \pfun Interval$

$update : Box \times Name \times Interval \rightarrow Box$

$\forall b : Box, n : Name, i : Interval \bullet$
$update(b, n, i) = b \oplus \{n \mapsto i\}$

$\_ \cap_b \_ : Box \times Box \pfun Box$

$\forall b, c : Box \mid b \cap_? c \bullet$
$b \cap c = \{n : \mathrm{dom}\, b \cap \mathrm{dom}\, c \bullet n \mapsto b\ n \cap c\ n\}$

$$\_ \cap_? \_ : Box \leftrightarrow Box$$

$$\begin{array}{l}
\forall\, b, c : Box \bullet \\
\quad b \cap_? c \Leftrightarrow (\exists\, n : Name \bullet n \in \operatorname{dom} b \cap \operatorname{dom} c) \\
\qquad \land\ (\forall\, n : \operatorname{dom} b \cap \operatorname{dom} c \bullet b\ n \cap_? c\ n)
\end{array}$$

B. Abstract Syntax Tree (AST)

$$OPR ::= plus \mid minus \mid times \mid divide \mid power$$

$$FUN ::= ln \mid exp \mid sin \mid cos \mid tan \mid sec \mid csc \mid cot$$

$$\begin{array}{rcl}
AST & ::= & const\langle\langle \mathbb{R} \rangle\rangle \\
& \mid & var\langle\langle Name \rangle\rangle \\
& \mid & unary\langle\langle OPR \times AST \rangle\rangle \\
& \mid & binary\langle\langle OPR \times AST \times AST \rangle\rangle \\
& \mid & fun\langle\langle FUN \times AST \rangle\rangle
\end{array}$$

C. AST Evaluation

$$\mathcal{E} : AST \times Box \nrightarrow Interval$$

$$\begin{array}{l}
\forall\, x : \mathbb{R},\ b : Box \bullet \\
\quad \mathcal{E}(const\ x, b) = [x, x] \\
\forall\, n : Name,\ b : Box \mid n \in \operatorname{dom} b \bullet \\
\quad \mathcal{E}(var\ n, b) = b\ n \\
\forall\, a : AST,\ b : Box \bullet \\
\quad \mathcal{E}(unary(minus, a), b) = \bar{-}\mathcal{E}(a, b) \\
\forall\, a : AST,\ b : Box \bullet \\
\quad \mathcal{E}(binary(plus, a_1, a_2), b) = \mathcal{E}(a_1, b) \mathbin{\bar{+}} \mathcal{E}(a_2, b) \\
\quad \mathcal{E}(binary(minus, a_1, a_2), b) = \mathcal{E}(a_1, b) \mathbin{\bar{-}} \mathcal{E}(a_2, b) \\
\quad \mathcal{E}(binary(times, a_1, a_2), b) = \mathcal{E}(a_1, b) \mathbin{\bar{\times}} \mathcal{E}(a_2, b) \\
\quad \mathcal{E}(binary(divide, a_1, a_2), b) = \mathcal{E}(a_1, b) \mathbin{\bar{\div}} \mathcal{E}(a_2, b) \\
\quad \mathcal{E}(binary(power, a_1, a_2), b) = \mathcal{E}(a_1, b) \mathbin{\bar{\wedge}} \mathcal{E}(a_2, b) \\
\forall\, a : AST,\ b : Box \bullet \\
\quad \mathcal{E}(fun(ln, a), b) = \bar{\ln}\,(\mathcal{E}(a, b)) \\
\quad \mathcal{E}(fun(exp, a), b) = \bar{\exp}\,(\mathcal{E}(a, b)) \\
\quad \mathcal{E}(fun(sin, a), b) = \bar{\sin}\,(\mathcal{E}(a, b)) \\
\quad \mathcal{E}(fun(cos, a), b) = \bar{\cos}\,(\mathcal{E}(a, b)) \\
\quad \mathcal{E}(fun(tan, a), b) = \bar{\tan}\,(\mathcal{E}(a, b)) \\
\quad \mathcal{E}(fun(sec, a), b) = \bar{\sec}\,(\mathcal{E}(a, b)) \\
\quad \mathcal{E}(fun(csc, a), b) = \bar{\csc}\,(\mathcal{E}(a, b)) \\
\quad \mathcal{E}(fun(cot, a), b) = \bar{\cot}\,(\mathcal{E}(a, b))
\end{array}$$

D. AST Differentiation

$$\mathcal{D} : AST \times Name \to AST$$

$$\begin{array}{l}
\forall\, x : \mathbb{R}, n : Name \bullet \mathcal{D}(const\ x, n) = const\ 0 \\
\forall\, m, n : Name \bullet \\
\quad m = n \Rightarrow \mathcal{D}(var\ m, n) = const\ 1 \\
\quad m \neq n \Rightarrow \mathcal{D}(var\ m, n) = const\ 0 \\
\forall\, a : AST, n : Name \bullet \\
\quad \mathcal{D}(unary(minus, a), n) = \\
\qquad unary(minus, \mathcal{D}(a, n)) \\
\forall\, a_1, a_2 : AST, n : Name \bullet \\
\quad \mathcal{D}(binary(plus, a_1, a_2), n) = \\
\qquad binary(plus, \mathcal{D}(a_1, n), \mathcal{D}(a_2, n)) \\
\quad \mathcal{D}(binary(minus, a_1, a_2), n) = \\
\qquad binary(minus, \mathcal{D}(a_1, n), \mathcal{D}(a_2, n)) \\
\quad \mathcal{D}(binary(times, a_1, a_2), n) = \\
\qquad binary(plus, \\
\qquad\quad binary(times, a_1, \mathcal{D}(a_2, n)), \\
\qquad\quad binary(times, a_2, \mathcal{D}(a_1, n))) \\
\forall\, x : \mathbb{R}, n : Name \bullet \\
\quad \mathcal{D}(binary(divide, a_1, a_2), n) = \\
\qquad binary(minus, \\
\qquad\quad binary(divide, \mathcal{D}(a_1, n), a_2), \\
\qquad\quad binary(divide, \\
\qquad\qquad binary(times, a_1, \mathcal{D}(a_2, n)), \\
\qquad\qquad binary(power, a_2, const\ 2))) \\
\quad \mathcal{D}(binary(power, a_1, a_2), n) = \\
\qquad binary(plus, \\
\qquad\quad binary(times, \\
\qquad\qquad binary(times, a_2, \\
\qquad\qquad\quad binary(power, a_1, \\
\qquad\qquad\qquad binary(minus, a_2, const\ 1))), \\
\qquad\qquad \mathcal{D}(a_1, n)), \\
\qquad\quad binary(times, \\
\qquad\qquad binary(times, \\
\qquad\qquad\quad fun(ln, a_1), \\
\qquad\qquad\quad binary(power, a_1, a_2)), \\
\qquad\qquad \mathcal{D}(a_2, n))) \\
\forall\, a : AST, n : Name \bullet \\
\quad \mathcal{D}(fun(ln, a), n) = binary(divide, \mathcal{D}(a, n), a) \\
\quad \mathcal{D}(fun(exp, a), n) = binary(times, a, \mathcal{D}(a, n)) \\
\quad \mathcal{D}(fun(sin, a), n) = binary(times, fun(cos, a), \mathcal{D}(a, n)) \\
\quad \mathcal{D}(fun(cos, a), n) = \\
\qquad unary(minus, binary(times, fun(sin, a), \mathcal{D}(a, n))) \\
\quad \mathcal{D}(fun(tan, a), n) = \\
\qquad binary(times, \\
\qquad\quad binary(power, fun(sec, a), const\ 2), \\
\qquad\quad \mathcal{D}(a, n)))
\end{array}$$