

A Framework for Testing First-Order Logic Axioms in Program Verification

Ki Yung Ahn · Ewen Denney

Received: date / Accepted: date ; will be entered by the editor

Abstract Program verification systems based on automated theorem provers rely on user-provided axioms in order to verify domain-specific properties of code. However, formulating axioms correctly (that is, formalizing properties of an intended mathematical interpretation) is non-trivial in practice, and avoiding or even detecting unsoundness can sometimes be difficult to achieve. Moreover, speculating soundness of axioms based on the output of the provers themselves is not easy since they do not typically give counterexamples. We adopt the idea of model-based testing to aid axiom authors in discovering errors in axiomatizations. To test the validity of axioms, users define a computational model of the axiomatized logic by giving interpretations to the function symbols and constants in a simple declarative programming language. We have developed an axiom testing framework that helps automate model definition and test generation using off-the-shelf tools for meta-programming, property-based random testing, and constraint solving. We have experimented with our tool to test the axioms used in AUTOCERT, a program verification system that has been applied to verify aerospace flight code using a first-order axiomatization of navigational concepts, and were able to find counterexamples for a number of axioms.

Keywords model-based testing · program verification · automated theorem proving · property-based testing · constraint solving

1 Introduction

1.1 Background

Model-based development and automated code generation are increasingly used in safety-critical domains (e.g., NASA's Project Constellation uses MathWorks' Real-Time Workshop), not only for simulation and prototyping, but also for actual flight code generation, in

Ki Yung Ahn
Department of Computer Science, Portland State University, Portland, OR 97201, USA
E-mail: kya@cs.pdx.com

Ewen Denney
m/s 269-2, NASA Ames Research Center, Moffett Field, CA 94035, USA
E-mail: Ewen.Denney@nasa.gov

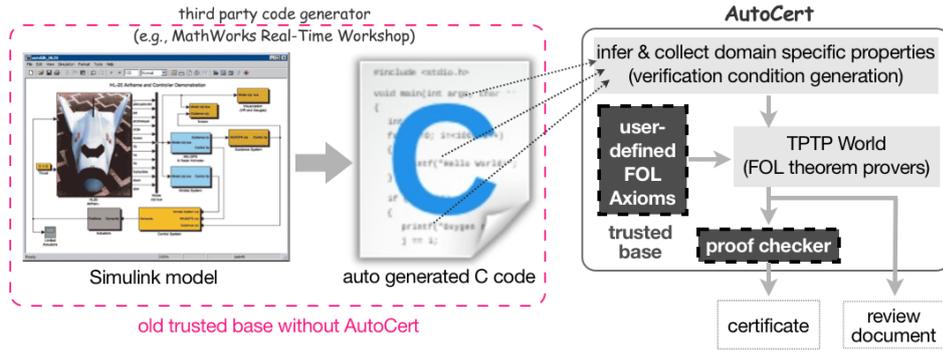


Fig. 1: AUTOCERT narrows down the trusted base by verifying the generated code

particular in the Guidance, Navigation, and Control (GN&C) domain. However, since code generators are typically not qualified, there is no guarantee that their output is correct, and consequently the generated code still needs to be fully tested and certified.

Program verification systems based on automated theorem provers rely on user-provided axioms in order to verify domain-specific properties of code. AUTOCERT (Denney and Trac, 2008) is a source code verification tool for autogenerated code in safety critical domains, such as flight code generated from Simulink models in the guidance, navigation, and control (GN&C) domain using MathWorks’ Real-Time Workshop code generator. AUTOCERT supports certification by formally verifying that the generated code complies with a range of mathematically specified requirements and is free of certain safety violations. AUTOCERT uses Automated Theorem Provers (ATPs) (Sutcliffe, 2000) based on First-Order Logic (FOL) to formally verify safety and functional correctness properties of autogenerated code, as illustrated in Figure 1.

AUTOCERT works by using domain-specific verifiers (Denney and Fischer, 2008) to infer logical annotations on the source code, and then using a verification condition generator (VCG) to check these annotations. This results in a set of first-order verification conditions (VCs) that are then sent to a suite of ATPs. These ATPs try to build proofs based on the user-provided axioms, which can themselves be arbitrary First-Order Formulas (FOFs).

If all the VCs are successfully proven, then it is guaranteed that the code complies with the properties¹—with one important proviso: we need to trust the verification system, itself. The *trusted base* is the collection of components which must be correct for us to conclude that the code itself really is correct. Indeed, one of the main motivations for applying a verification tool like AUTOCERT to autocode is to remove the code generator—a large, complex, black box—from the trusted base.

The annotation inference system is not part of the trusted base, since annotations merely serve as hints (albeit necessary ones) in the verification process—they are ultimately checked via their translation into VCs by the VCG. The logic that is encoded in the VCG does need to be trusted but this is a relatively small and stable part of the system. The ATPs do not need to be trusted since the proofs they generate can (at least, in principle) be sent to a proof checker (Sutcliffe et al, 2005). In fact, it is the domain theory, defined as a set of logical axioms, that is the most crucial part of the trusted base. Moreover, in our experience, it is the most common source of bugs.

¹ The converse is not always true, however: provers can time out or the domain theory might be incomplete.

Formulating axioms correctly (i.e., precisely as the domain expert really intends) is non-trivial in practice. By correct we mean that the axioms formulate properties of an intended mathematical interpretation. The challenges of axiomatization arise from several dimensions. First, the domain knowledge has its own complexity. AUTOCERT has been used to verify mathematical requirements on navigation software that carries out various geometric coordinate transformations involving matrices and quaternions. Axiomatic theories for such constructs are complex enough that mistakes are not uncommon. Second, the axioms frequently need to be modified in order to formulate them in a form suitable for use with ATPs. Such modifications tend to obscure the axioms further. Third, it is easy to accidentally introduce unsound axioms due to implicit, but often incompatible interpretations of the axioms. Fourth, speculating on the validity of axioms from the output of existing ATPs is difficult since theorem provers typically do not give any examples or counterexamples (and some, for that matter, do not even give proofs).

1.2 Overview of the axiom testing framework

Our goal is to design and implement a testing framework for axioms that helps detect problems in the axioms more easily and at an early stage. We adopt the idea of model-based testing to aid axiom authors and domain experts in discovering errors in axiomatization. To test the validity of axioms, users define a computational model of the axiomatized logic by giving interpretations to each of the function symbols and constants as computable functions and data constants in a simple declarative programming language. Then, users can test axioms against the computational model with widely used software testing tools. The advantage of this approach is that users have a concrete intuitive model with which to test validity of the axioms, and can observe counterexamples when the model does not satisfy the axioms.

In summary the testing framework

- is configurable by user-provided interpretations of the logic described in a simple declarative programming language
- and this user-provided interpretation is also valuable as an intuitive and manageable specification of domain knowledge,
- automatically derives a testable property for each axiom,
- automatically derives test data generators for linear constraints in each axiom.

We believe that the interpretation provided for testing also provides a more intuitive and manageable specification of domain knowledge than the axiomatization itself. It is more intuitive since the interpretation is a computational model and program verification is about programs that compute². It is more manageable since it is often much more succinct than the axioms (e.g., consider the array type which is just a given feature or library in programming languages, but it needs several axioms to state several properties of arrays in FOL.) and less likely to change throughout the verification process unlike the axioms (recall Section 1.1). We will discuss why axioms are not the most effective means of specifying domain knowledge in Section 3, which illustrates some of the pitfalls in developing axioms. Then, we show how we test some examples of axioms used by AUTOCERT in a high-level view, and

² In fact, we could even choose to define the interpretation using the actual libraries that will be used in the production implementation, by calling on the Haskell library or using the Foreign Function Interface of Haskell to call external libraries written in other languages like C.

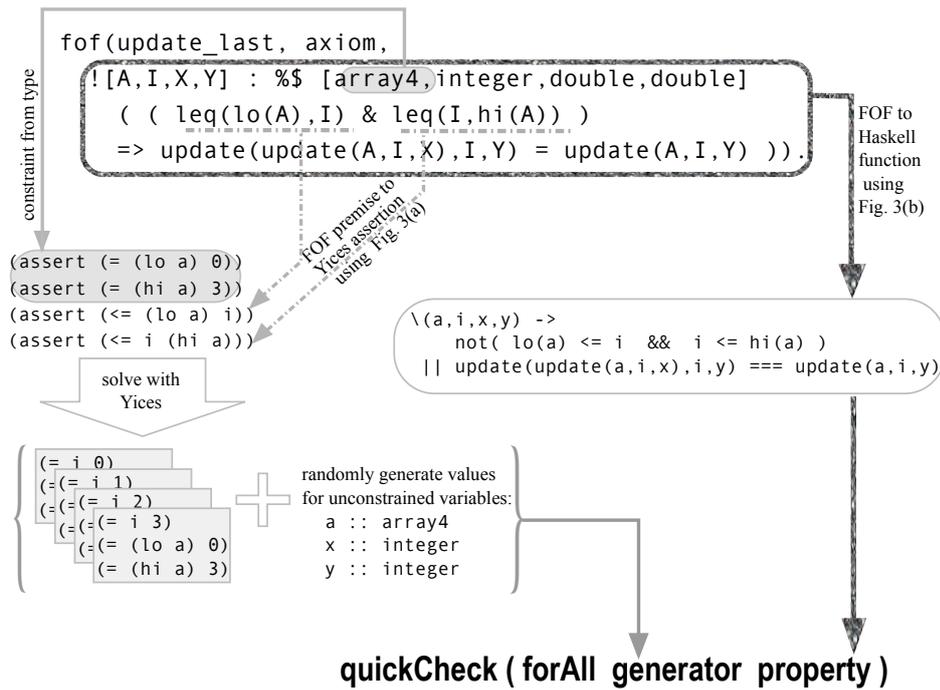


Fig. 2: Overview of the axiom testing framework

also discuss the issues that arise in testing those axioms in Section 4. We describe the algorithms used in our testing framework implementation in Section 5, and elaborate on how to use our testing framework by showing test scripts and running examples for some of the axioms from the Thousands of Problems for Theorem Provers (Sutcliffe, 2009) (TPTP) in Section 6. Section 7 discusses the class of testable axioms. We conclude with a discussion of related work (Section 8) and thoughts for future work (Section 9).

Before going into the details, we first give a brief overview of the testing framework. Figure 2 illustrates the computational flow. The testing framework consists of two major parts: deriving test properties (right side of the figure) and deriving test data generators (left side of the figure). The axiom formula on top of Figure 2 is an input to the axiom testing framework, and the code snippets in Figure 3 are symbol interpretations provided by the user. Given such axiom input and the interpretations of the logical symbols appearing in the axiom, our testing framework automatically derives both the property and test data generator. Then, the generator is plugged with the property to run the test using QuickCheck (Claessen and Hughes, 2000).

The property derived from the axiom is just a lambda expression (or anonymous function) in Haskell,³ which is a program that we can evaluate to true or false when the val-

³ We generate properties as first class values of Haskell, which is different from writing an interpreter or an evaluator over syntax trees of first-order logic formulae. We use Template Haskell (Sheard and Peyton Jones, 2002) to generate these properties as Haskell code at compile time from the axiom formulae input and the user provided interpretation.

```

pred2yicesTable =
  [ ("lt", \ (x,y)-> x < y
    , ("gt", \ (x,y)-> x > y
    , ("leq", \ (x,y)-> x <= y
    , ("geq", \ (x,y)-> x >= y
    ]

term2yicesTable =
  [ ("hi", \ as-> VarE "hi" 'APP' as)
    , ("lo", \ as-> VarE "lo" 'APP' as)
    ]

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; array.yices
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(define hi :: (-> (-> int real) int))
(define lo :: (-> (-> int real) int))

```

(a) Interpretation of logical symbols in Yices

```

pred2hsTable =
  [("lt", [ | uncurry (<) | ])
  , ("gt", [ | uncurry (>) | ])
  , ("leq", [ | uncurry (<=) | ])
  , ("geq", [ | uncurry (>=) | ])
  , ...
  ]

term2hsTable =
  [("lo", [ | lo | ])
  , ("hi", [ | hi | ])
  , ("update", [ | update | ])
  , ...
  ]

lo a = fst (bounds a)
hi a = snd (bounds a)
update(arr,i,c) = arr // [(i,c)]
...

```

(b) Interpretation of logical symbols in Haskell

Fig. 3: Interpretation of logical symbols

ues for function arguments are supplied. There is a natural translation from axioms⁴ to lambda expressions once the interpretations of logical symbols are provided, as can be seen from Figure 2.⁵ The universal quantification $! [A, I, X, Y]$: translates to lambda bindings $\backslash (a, i, x, y) \rightarrow$, logical implication $(P \Rightarrow Q)$ to $(\text{not } p \mid \mid q)$ and logical conjunction $\&$ to $\&\&$ where \backslash is a notation for lambda in Haskell, p and q are translations of P and Q , and not , $\mid \mid$, and $\&\&$ are logical operators defined in Haskell Standard Prelude. Other than the universal quantification and standard logical connectives (e.g., implication, and, or, not), the user must provide an interpretation for the logical symbols appearing in the axioms, as in Figure 3(b), which is also a piece of Haskell code. The interpretation consists of symbol translation tables mapping logical symbol names to Haskell names, along with the definitions of the Haskell names which are not already defined. For example, we only map "leq" to the predefined less-than-equal operator (\leq), but we give definitions for hi , lo and update .⁶ The translation from axioms to properties is possible since the axioms we deal with are limited forms of FOF without any effective existential quantification. (We think that most axioms appearing in program verification would not need existential quantifications.) We describe further details of translation from axioms to properties in Section 5.1.

The test data generator is derived from the type annotation comments⁷ and the premise part of the axiom. This generator only generates meaningful test data that satisfies the con-

⁴ Not all axiom formulae that can be written with TPTP FOF syntax have a natural computational interpretation. In practice, however, most axioms which are actually used in software verification, or as constructive axioms for other domains, do have a computational interpretation. In Section 5, we discuss the class of axioms that our framework can test.

⁵ We will write TPTP syntax (see <http://www.cs.miami.edu/~tptp/TPTP/SyntaxBNF.html>) in typewriter font, e.g., $! [X, Y, Z] : ((X=Y \ \& \ \sim(Y=Z)) \Rightarrow X=Z)$. We use this plain text TPTP syntax when we quote axiom formulas verbatim. When we discuss first order formulae more conceptually or mathematically, such as when we describe algorithms on formulae, we use conventional math symbols for logical connectives, e.g., $\forall [X, Y, Z] : ((X = Y \wedge \neg(Y = Z)) \Rightarrow X = Z)$.

We use italic math font for metavariables of formulae (P, Q, R, A, B, C, D) and terms (T).

⁶ We intentionally used the same Haskell names as the logical symbol names (e.g., mapping "hi" to hi) for the sake of readability, but they do not have to be the same.

⁷ We specify type annotations in TPTP definition comments, which are comments that start with either $\%\$$ (for line comments) or $/*\$$ (for block comments). A type annotation list is simply a comma delimited list of

strains of the type annotations and the premise part of the axiom, avoiding runtime errors (e.g., array bounds error) and vacuously true tests (i.e., true by falsifying the premise). Overall, test data generation takes three steps:

1. extract the constraints from the type annotations and the premise (see the four Yices `assert` commands in Figure 2),
2. solve the constraints with Yices to get a satisfying model
3. randomly generate variables that are not constrained in the model.

The step 1 of extracting constraint is done once when we derive each of the data generator. Then, when QuickCheck invokes the data generator, it communicates with Yices repeating steps 2 and 3 to generate multiple test data. The user must provide interpretation of logical symbols in Yices (Figure 3(a)) to translate the premise of the axiom into the constraints of Yices `assert` commands, similarly as we gave interpretation in Haskell to derive properties from the axiom. The difference is that we must give definitions in a separate `.yices` file for the Yices names which are not predefined in Yices, because Yices (not Haskell) is to read in and solve these extracted constraints. Note that the constraints that our testing framework can handle are limited by the solver used in the implementation, that is Yices, which can only solve linear constraints of particular domains. But, we think this is still useful since premises of the axioms used in program verification are most likely to be linear properties (e.g., array bounds, comparison on numbers), although their conclusion part may mention non-linear properties. And, even when the constraints are not solvable by Yices, it is still very useful to have automatically derived test properties since we can manually program test generators (see Section 6.2) or supply manually collected test data to those derived properties. We describe further details of translation from axioms to properties in Section 5.2.

2 Methods and Tools

In the previous section, we mentioned several methods and tools used in our axiom testing framework (see Figure 2). In this section, we give a brief introduction to each of them: the TPTP language, Haskell libraries (QuickCheck and Template Haskell) and Yices. We also mention some of the design choices related to the features of the methods and tools introduced in this section.

2.1 The TPTP Language: standardized format for ATP systems

Thousands of Problems for Theorem Provers (TPTP) is a library of test problems for Automated Theorem Proving (ATP) systems. The TPTP Language is a standardized format to write problems, record derivations, and record finite interpretations (or finite models). Through this standardized language, we can share problems and compare the results for the problems among many different ATP systems. The TPTP Language has four sublanguages: CNF (clausal normal form), FOF (first-order form), TFF (typed first-order form), and THF (typed higher-order form). Most ATP systems support CNF and FOF, while fewer ATP systems support THF. TFF is a recently added, syntactically conservative extension to FOF, which can easily be translated into FOF in the absence of interpreted arithmetic. Some ATP

identifiers starting with lowercase alphabet, enclosed in square brackets. The length of a type annotation list must be the same as the list of universally qualified variables that is being annotated.

systems directly support TFF. The complete grammar of the TPTP language including all four sublanguages can be found on the TPTP homepage.⁸

Ideally, we should axiomatize domains involving heavy use of arithmetic in TFF rather than FOF. However, since few ATP systems currently support TFF, in our work we focus on FOF. All our axioms and problems (or verification conditions) are currently written in FOF.

here, we illustrate the FOF syntax using examples. Consider axiomatizing the concepts of graph theory and posing problems conjecturing the properties about the concepts. We can axiomatize the definition of the connected vertices relation as a first-order formula:

$$\forall X, Y. (\text{conn}(X, Y) \Leftrightarrow (X = Y \vee \text{adj}(X, Y) \vee (\exists Z. (\text{adj}(X, Z) \wedge \text{conn}(Z, Y))))))$$

We can write the formula above in FOF syntax as follows:

```
fof(conn_def, axiom,
  ! [X, Y] : (conn(X, Y) <=> (X=Y | adj(X, Y) | (? [Z] : (adj(X, Y) & conn(Z, Y)))))).
```

We can describe a problem that conjectures the transitivity of the connected vertices relation as a first-order formula: $\forall X, Y, Z. (\text{conn}(X, Y) \wedge \text{conn}(Y, Z)) \Rightarrow \text{conn}(X, Z)$. We can write the formula in FOF syntax as follows:

```
fof(conn_trans, conjecture,
  ! [X, Y, Z] : ( (conn(X, Y) & conn(Y, Z)) => conn(X, Z) ) ).
```

Once we write down the axioms and the conjectures in FOF syntax, we can supply them as input to the ATP systems to search for the proofs of the conjectures.

2.2 QuickCheck: a property based testing library

QuickCheck is a property (or specification) based testing library for Haskell. Typical use of the QuickCheck library consists of two parts: describing the testable properties and supplying the test generators. We can write both the properties and the generators in plain Haskell code. QuickCheck provides default random generators for many primitive types (e.g., `Int`, `Double`, and `Char`), and is systematically extensible to datatypes containing those types, which already have default random generators (e.g., pair of integers `(Int, Int)`, list of integers `[Int]`, list of integer pairs `[(Int, Int)]`, and so on). In QuickCheck terminology, we say that the types which have default random generators belong to the `Arbitrary` class.

Testable properties are either expressions of QuickCheck's `Property` datatype, boolean values of type `Bool`, or functions that can return testable properties from the arguments of the `Arbitrary` class. The boolean values are simply unconditional trivial properties that evaluate to either `True` or `False`. Hence, interesting properties are most likely to be specified by functions that return boolean values or `Property` values. Haskell's powerful type class mechanism empowers the QuickCheck library to flexibly handle functions of varying types, such as `Int -> Bool` or `(Int, Int) -> Bool`. All such testable properties eventually boil down to an abstract datatype called `Property`, and the core of the QuickCheck library operates internally on the `Property`.

For example, we can specify the reflexivity property and the transitivity property of the less-than-or-equal relation on integers with the following Haskell code:

⁸ <http://www.tptp.org/>

```
lt_Int_refl :: Int -> Bool
lt_Int_refl x = x <= x
```

```
lt_Int_trans :: (Int,Int,Int) -> Property
lt_Int_trans (x,y,z) = (x <= y && y <= z) ==> y <= z
```

QuickCheck accepts both functions `lt_Int_refl` and `lt_Int_trans` as testable properties, since the arguments to these functions are of `Arbitrary` class and their return types are `Bool` and `Property`. The reason why `lt_Int_trans` returns `Property` is because of the use of QuickCheck's implication combinator `(==>) :: Testable prop -> Bool => prop -> Property`. This combinator has a different semantics in terms of testing, as compared to defining the logical implication combinator simply by translating "P implies Q" into "not P or Q". QuickCheck counts only the meaningful true tests which satisfy the premise as *successful* tests, but discards the vacuously true tests which falsify the premise. In practice, when we demand that QuickCheck run through 100 successful tests for `lt_Int_trans`, it will need to generate about 600 tests discarding about 500 vacuously true tests. The default QuickCheck configuration searches for 100 tests and allows 500 maximum discarded tests. Hence, `lt_Int_trans` we are lucky for QuickCheck to even run through 100 successful tests, but sometimes QuickCheck will give up when running on the default configuration (since the default maximum discarded number of tests is by default is 500, which is less than the average of 600).

For comparison, consider an alternative specification of the transitivity property of the less-than relation on integers, where the implication is translated into its logically equivalent form:

```
lt_Int_trans' :: (Int,Int,Int) -> Bool
lt_Int_trans' (x,y,z) = not (x <= y && y <= z) || y <= z
```

When we ask QuickCheck to run through 100 successful tests for `lt_Int_trans'`, it will only need to generate 100 tests, but 83 of them turn out to be vacuously true test cases falsifying `(x <= y && y <= z)`.

While QuickCheck's implication operator is convenient for filtering out uninteresting tests among randomly generated tests, it has some limitations. Firstly, it becomes intractable to handle the premises satisfying a very small portion of the domain. An extreme example is the symmetry property of the equality relation on integers:

```
eq_Int_symm :: (Int,Int) -> Property
eq_Int_symm (x,y) = x == y ==> y == x
```

QuickCheck will not be able fulfill our demand for 100 successful tests on `eq_Int_symm` in the default configuration, since it will discard too many vacuous tests and give up, reaching the default limit on the number of maximum tests to try. We can barely handle the previous example of `lt_Int_trans` in the default configuration. It is possible to configure QuickCheck parameters to increase the limit, but there is little hope of effectively testing complex properties purely relying on the default random test generators.

A recommended approach is to supply custom smart generators when the default generators are not effective. The QuickCheck library provides the `forAll :: (Testable prop, Show a) => Gen a -> (a->prop) -> Property` combinator that combines a generator (`Gen a`) and a testable function property (`a->prop`) into a `Property`. For example, the property `lt_Int_trans` is equivalent to `(forAll arbitrary lt_Int_trans)`, where `arbitrary` is the default random generator. Instead of supplying the default generator, we can supply custom smart generators, which generate interesting tests only, or at least more often than

the arbitrary generator. For example, we can build a smart generator which returns integer triples of non-decreasing order as follows:

```
orderedTriple :: Gen (Int,Int,Int)
orderedTriple = do (x,y,z) <- arbitrary      -- generate random tuple
                  let [x',y',z'] = sort [x,y,z] -- sort them
                  return (x',y',z')         -- return sorted tuple
```

Then, `(forall orderedTriple lt_Int_trans)` will run through 100 successful tests in exactly 100 generated tests, without generating any vacuous tests. With this smart generator `(forall orderedTriple lt_Int_trans')` effectively behaves the same.

Another limitation of QuickCheck's implication operator is that it cannot be nested on the left. The `(==>)` operator requires the premise to be `Bool` but returns `Property`. This is a reasonable design since `(==>)` has a testing semantics of discarding vacuous tests that falsifies the premise. It is unclear what the testing semantics should be when it is left-nested. Therefore, our testing framework preprocess the axiom formulae of nested implications into a non-nested single level implication. We call these preprocessing steps unrolling and flattening (see Section 5).

Although we can manually write smart generators systematically, it may be tedious to write them every time. Moreover, it would not be practical to require collaborators who are unfamiliar with Haskell to learn the details of QuickCheck, just to run more effective tests on the axioms. Therefore, in this work, we try to automate the process of writing the smart generators by using SMT solvers to generate tests.

2.3 Yices: an SMT solver

Yices is a Satisfiability Modulo Theories (SMT) solver we use to build our axiom testing framework. We can think of SMT solvers as an extension to SAT solvers with decision procedures, which make use of background theories. Linear arithmetic (i.e., equalities, inequalities, addition, subtraction, and multiplication and division by constants) on integers and rational numbers are most commonly supported theories in SMT solvers. Since SMT relies on background theories of specific type of values, SMT problem descriptions should be given in a typed language. Although there exists a standardized language for SMT, the SMT-LIB format, we use Yices format in this work since we are currently only using Yices. The Yices format is, in any case, very close to the SMT-LIB format.

For some simple axioms, we can translate the entire FOF axiom formula into the Yices input format, and check whether it is satisfiable. For example, we only need the SMT solver to validate the transitivity property on integers, since it only involves linear arithmetic. However, most axioms involve constraints, which SMT solvers cannot usually handle. Arithmetic including non-constant multiplication and division is already undecidable in general, and many engineering domains use more complex arithmetic functions such as trigonometry, logarithm and exponentiation on real number bases. We have observed that many axioms we needed to deal with are in the form of premise and conclusion, expressing the idea that under certain constraints some property must hold. Many of these constraints in the premise are linear constraints solvable by SMT solvers, while the conclusion part of the axiom is not linear (e.g., if $t_1 + t_2 = \frac{\pi}{2}$ then $\sin^2 t + \cos^2 t = 1$). Our testing framework automates the generation of tests satisfying such linear constraints (e.g., $t_1 + t_2 = \frac{\pi}{2}$) using the Yices SMT solver. We currently do not have a good way of automation for the axioms with non-linear constraints, but one still can supply manually crafted smart test generators written in Haskell.

2.4 Template Haskell

Template Haskell is a meta-programming library and language extension to Haskell, supported by the GHC compiler. Template Haskell provides a library to build syntax trees for Haskell data structures, and some additional language constructs to reify the syntax tree into Haskell code at compile time. In other words, Template Haskell is a compile time Haskell code generator tightly integrated into Haskell as a language extension.

In our testing framework, we automatically collect the constraints from the premise of the axioms. Then, we need to translate those constraints into Haskell functions testable by QuickCheck. Here, we use Template Haskell to translate the constraints into Haskell code at compile time.

We also use template Haskell to generate some stubs for interfacing with the Yices SMT solver at compile time. In fact, the interaction with SMT does not have to be entirely at compile time, but we choose to implement it this way for simplicity of the implementation.

3 A scenario of axiom development

In vehicle navigation software, *frames of reference* are used to represent different coordinate systems within which the position and orientation of objects are measured. Navigation software frequently needs to translate between different frames of reference, such as between vehicle-based and earth-based frames when communicating between mission control and a spacecraft. A transformation between two different frames can be represented by a so-called direction cosine matrix (DCM) (Kuipers, 1999; Vallado, 2001). Verifying navigation software therefore requires us to check that the code is correctly carrying out these transformations, that is, correctly represents these matrices, quaternions, and the associated transformations. As we will show, however, axiomatizing these definitions and their properties is error-prone.

In the following subsections we will use a simplified running example of a two-dimensional rotation matrix (rather than a 3D transformation matrix).

3.1 Axiomatizing a two-dimensional rotation matrix

The two dimensional rotation matrix for an angle t is given by $\begin{pmatrix} \cos(t) & \sin(t) \\ -\sin(t) & \cos(t) \end{pmatrix}$. Matrices in control code are usually implemented using arrays, and the most obvious way to axiomatize these arrays in FOL is extensionally, i.e.,

$$\text{select}(A,0) = \cos(T) \wedge \text{select}(A,1) = \sin(T) \wedge \dots$$

but this is unlikely to prove useful in practice. Consider the C implementation `init` in Table 1, which is intended to initialize a two dimensional rotation matrix. A VCG will apply the usual array update rules to derive that the output `X` should be replaced by `update(update(update(update(a, 0, cos(t)), 1, sin(t)), 2, uminus(sin(t))), 3, cos(t))`. Unfortunately, provers are generally unable to relate this update term to the extensional definition so, instead, we use the following axiom, written in TPTP first-order formula (FOF) syntax, which defines an array representation of the two-dimensional rotation matrix as a binary relation `rot2D` between an array `A` and angle `T`.

C code	Verification Condition
<pre>void init(float a[], float t) { a[0]= cos(t); a[1]= sin(t); a[2]=-sin(t); a[3]= cos(t); }</pre>	<pre>fof(vc, conjecture, ((lo(a)=0 & hi(a)=3) => rot2D(update(update(update(update(a ,0, cos(t)),1,sin(t)) ,2,uminus(sin(t))),3,cos(t)) ,t))).</pre>
<pre>void init1(float a[], float t) { a[0]= sin(t); a[0]= cos(t); a[1]= sin(t); a[2]=-sin(t); a[3]= cos(t); }</pre>	<pre>fof(vc1, conjecture, ((lo(a)=0 & hi(a)=3) => rot2D(update(update(update(update(update(a ,0, sin(t)) ,0, cos(t)),1,sin(t)) ,2,uminus(sin(t))),3,cos(t)) ,t))).</pre>
<pre>void init2(float a[], float t) { a[0]= sin(t); a[1]= sin(t); a[2]= sin(t); a[3]= sin(t); a[0]= cos(t); a[1]= sin(t); a[2]=-sin(t); a[3]= cos(t); }</pre>	<pre>fof(vc2, conjecture, ((lo(a)=0 & hi(a)=3) => rot2D(update(update(update(update(update(update(update(update(a ,0, sin(t)),1,sin(t)) ,2, sin(t)),3,sin(t)) ,0, cos(t)),1,sin(t)) ,2,uminus(sin(t))),3,cos(t)) ,t))).</pre>
<pre>void initX(float a[], float t) { a[0]=-cos(t); a[1]= sin(t); a[2]=-sin(t); a[3]= cos(t); }</pre>	<pre>fof(vcX, conjecture, ((lo(a)=0 & hi(a)=3) => rot2D(update(update(update(update(a ,0,uminus(cos(t))),1,sin(t)) ,2,uminus(sin(t))),3,cos(t)) ,t))).</pre>

Table 1: 2D rotation matrix code and corresponding verification conditions

VC	axioms	EP (e prover) 1.1	Equinox 4.1
vc	rotation2D_def	Theorem	Theorem
vc1	rotation2D_def rotation2D_def, update_last	CounterSatisfiable Theorem	Timeout Theorem
vc2	rotation2D_def, update_last rotation2D_def, update_last, update_commute	CounterSatisfiable Theorem	Timeout Timeout
vcX	rotation2D_def rotation2D_def, update_last rotation2D_def, update_last, update_commute	CounterSatisfiable CounterSatisfiable Theorem	Timeout Timeout Theorem

Table 2: Results of running EP and Equinox through the SystemOnTPTP website with default settings and a timeout of 60 seconds.

```
fof(rotation2D_def, axiom, ![A,T]:( (lo(A)=0 & hi(A)=3)
=> rot2D(update(update(update(update(A
  , 0,      cos(T) ), 1,sin(T))
  , 2,uminus(sin(T))), 3,cos(T)), T) ) ).
```

The function `init` can be specified with precondition $(lo(a)=0 \& hi(a)=3)$ and postcondition `rot2D(X,t)`, where `X` is the function output. In practice, we also have conditions on

the physical types of variables (e.g., that T is an angle), but omit this here. Using this specification for `init` gives the verification condition `vc` in Table 1. We can prove `vc` from the axiom `rotation2D_def` alone using two provers from SystemOnTPTP (Sutcliffe, 2000), as shown in the first row of Table 2. We chose EP 1.1 and Equinox 4.1 here because these two provers use different strategies. In general, it is necessary to use a combination of provers in order to prove all the VCs arising in practice.

3.2 Adding more axioms

Initialization routines often perform additional operations that do not affect the initialization task. For example, `init1` and `init2` in Table 1 assign some other values to the array elements before initializing them to the values of the rotation matrix elements. Although there are some extra operations, both `init1` and `init2` are, in fact, valid definitions of rotation matrices since they both finally overwrite the array elements to the same values as in `init`. However, we cannot prove the verification conditions generated from these functions from the axiom `rotation2D_def` alone (Table 2), because the theorem provers do not know that *two consecutive updates on the same index are the same as one latter update*.

We can formalize this as the following axiom:

```
fof(update_last, axiom,
  ![A,I,X,Y] : update(update(A,I,X),I,Y) = update(A,I,Y) ).
```

Then, both EP and Equinox can prove `vc1` as a theorem from the two axioms `rotation2D_def` and `update_last`, as shown in Table 2.

The verification condition `vc2` generated from `init2` is not provable even with the `update_last` axiom added. This is because `init2` has more auxiliary array updates before matrix initialization, and `update_last` axiom is not applicable since none of the consecutive updates are on the same index. To prove that `init2` is indeed a valid initialization routine we need the property that *two consecutive independent updates can switch their order*. The following axiom tries to formalize this property.

```
fof(update_commute, axiom,
  ![A,I,J,X,Y] : update(update(A,I,X),J,Y) = update(update(A,J,Y),I,X) ).
```

With this axiom added, EP can prove `vc2` from the three axioms `rotation2D_def`, `update_last`, and `update_commute`, but strangely, Equinox times out. It is true that some theorem provers can quickly find proofs while others are lost, depending on the conjecture. Nevertheless, considering the simplicity of the formulae, the timeout of Equinox seems quite strange and might indicate a problem.

The axiomatization of array operations has been well understood since the work of McCarthy and Painter (1967). The theory of arrays has been reasoned about in ATP systems, and more recently it has been implemented as a built in theory for SMT solvers Bradley et al (2006). Comparing the axioms here with the classical axioms, it should be readily apparent that `update_commute` is problematic. However, we will continue the discussion as if we were creating the array axioms from scratch, to illustrate the problems that can be encountered during axiom development with a minimal example.

3.3 Detecting unsoundness and debugging axioms

It is important to bear in mind when adding new axioms that we are always at risk of introducing unsoundness. One way to detect unsoundness is to try proving obviously invalid con-

jectures.⁹ For example, the verification condition `vcX` for the incorrect initialization routine `initX` is invalid. The function `initX` is an incorrect implementation of the rotation matrix (Section 3.1) because `-cos(t)` is assigned to the element at index 0 instead of `cos(t)`, and hence does not satisfy `rot2D`. However, both EP and Equinox can prove `vcX`. The problem is that we have not thoroughly formalized the property that *independent* updates commute in the axiom `update_commute`.

Note that the theorem provers have not guided us to the suspicious axiom as the source of unsoundness. We decided to examine the axioms based on our own experience and insights, not just because Equinox timed out. Theorem provers may also time out while trying to prove valid conjectures from sound axioms. We should not expect that the most recently added axiom is always the cause of unsoundness. Coming up with an invalid conjecture that can be proven, and thus shows that the axioms are unsound, is usually an iterative process. We used our own intuition to find the cause of the problem, again with no help from the provers. Finally, note that the axiom `rotation2D_def` is already quite different from the natural definition of the matrix given above.

In this section, we have shown that it can be difficult to debug unsoundness of the axioms used in program verification systems even for three simple axioms. In practice, we need to deal with far larger sets of axioms combining multiple theories. In the following section, we will show how our method of testing axioms against a computational model helps us to detect problems in axioms more easily and systematically.

4 Testing axioms

When we have a computational model, we can run tests on logical formulae against that model. Since axioms are nothing more than basic sets of formulae that ought to be true, we can also test axioms against such a model in principle. Before going into the examples, let us briefly describe the principles of testing axioms. More technical details will be given in Section 5.

Given an interpretation for function symbols and constants (i.e., model) of the logic, we can evaluate truth values of the formulae without quantifiers. For example, `plus(zero, zero) = zero` is true and `plus(one, zero) = zero` is false based on the interpretation of `one` as integer 1, `zero` as integer 0, and `plus` as the integer addition function.

We can interpret formulae with quantified variables as functions from the values of the quantified variables to truth values. For example, we can interpret `! [X, Y] : plus(X, Y) = plus(Y, X)` as a function $\lambda(x, y).x + y = y + x$ – in plain text Haskell syntax `\(x, y) -> x + y == y + x` – which takes two integer pairs as input and tests whether $x + y$ is equal to $y + x$. This function will evaluate to true for any given test input (x, y) . When there exist test inputs under which the interpretation evaluates to false, then the original formula is invalid. For example, `! [X, Y] : plus(X, Y) = X` is invalid since its interpretation $\lambda(x, y).x + y = x$ – plain text in Haskell syntax `\(x, y) -> x + y == x` – evaluates to false when applied to the test input $(1, 1)$.

Formulae with implication need additional care when choosing input values for testing. To avoid vacuous satisfactions of the formula we must choose inputs that satisfy the premise. In general, finding inputs satisfying the premise of a given formula requires solving equations, and for this we use a combination of the SMT solver Yices (Dutertre and de Moura, 2006) and custom data generators (so-called “smart generators”).

⁹ It is not enough to just try and prove false since different provers exploit inconsistency in different ways. Moreover, a logic can be consistent yet still be unsound with respect to a model.

In the following subsections, we will give a high-level view of how we test the axioms with the example axioms from Section 3 and also some from AUTOCERT. We elaborate on further details of using our testing framework tool in Section 6, after discussing the internal algorithms of the testing framework implementation in Section 5.

4.1 Testing axioms for numerical arithmetic

Numeric values are one of the basic types in programming languages like C. Although the axioms on numerical arithmetic tend to be simple and small compared to other axioms (e.g., axioms on array operations) used in AUTOCERT, we were still able to identify some unexpected problems by testing. Those problems were commonly due to the untyped first-order logic terms being unintentionally interpreted as overloaded types. Even though the author of the axiom intended to write an axiom on one specific numeric type, say integers, that axiom could possibly apply to another numeric type, say reals.

For example, the following axiom formalizes the idea that the index of an array representing an 3-by-3 matrix uniquely determines the row and the column:

```
fof(uniq_rep_3by3, axiom,
! [X1, Y1, X2, Y2]: %$ [int, int, int, int]
( ( plus(X1,times(3,Y1)) = plus(times(3,Y2),X2)
& leq(0,X1) & leq(X1,2) & leq(0,Y1) & leq(Y1,2)
& leq(0,X2) & leq(X2,2) & leq(0,Y2) & leq(Y2,2) )
=> (X1=X2 & Y1=Y2) ) ).
```

To test the axiom it is first translated into the following function (where we limit ourselves to primitives in the Haskell prelude library):

$$\lambda(x_1, y_1, x_2, y_2) . \neg(x_1 + 3y_1 = 3y_2 + x_2 \wedge 0 \leq x_1 \leq 2 \wedge 0 \leq y_1 \leq 2 \\ \wedge 0 \leq x_2 \leq 2 \wedge 0 \leq y_2 \leq 2) \\ \vee (x_1 = x_2 \wedge y_1 = y_2)$$

or, in plain text Haskell syntax

```
\(x1,y1,x2,y2) ->
not( x1+3*y1==3*y2+x2 && 0<=x1 && x1<=2 && 0<=y1 && y1<=2
&& 0<=x2 && x2<=2 && 0<=y2 && y2<=2 )
|| (x1==x2 && y1==y2)
```

Assuming that this function is defined over integers (i.e., x_1, y_1, x_2, y_2 have integer type), we can generate test inputs of integer quadruples that satisfy the constraint of the premise ($x_1 + 3y_1 = 3y_2 + x_2 \wedge 0 \leq x_1 \leq 2 \wedge 0 \leq y_1 \leq 2 \wedge 0 \leq x_2 \leq 2 \wedge 0 \leq y_2 \leq 2$). Since the constraint is linear, Yices can generate such test inputs automatically, and all tests succeed.

However, nothing in the axiom says that the indices must be interpreted as integers, and the axiom can just as well be interpreted using floating points, and with `plus` and `times` interpreted as the overloaded operators `+` and `*` in C. If we test with this interpretation we find counterexamples such as $(x_1, y_1, x_2, y_2) = (\frac{1}{2}, \frac{1}{2}, 2, 0)$. The existence of such an unintended interpretation can lead to unsoundness. More specifically, the axiom `uniq_rep_3by3` is unsound and may prove some invalid conjectures if there are verification conditions that matches floating point instances of this axiom (e.g., when verification conditions contain floating point expressions of the form $x + 3y$ or $3y + x$).

It is important to note that the type annotation comments we provide for testing framework has nothing to do with ruling out this kind of unsoundness. Type annotation comments are just comments used only by our testing framework. One way to avoid such unsoundness is to introduce explicit typing judgment predicate for all bounded variables as follows:

```
fof(uniq_rep_3by3, axiom,
  ! [X1, Y1, X2, Y2]: (
    ( int(X1), int(Y1), int(X2), int(Y2),
      & plus(X1,times(3,Y1)) = plus(times(3,Y2),X2)
      & leq(0,X1) & leq(X1,2) & leq(0,Y1) & leq(Y1,2)
      & leq(0,X2) & leq(X2,2) & leq(0,Y2) & leq(Y2,2) )
    => (X1=X2 & Y1=Y2) ) ).
```

Some of the axioms in TPTP distribution (e.g., see GE0006+5.ax in TPTP version 5.0.0) follow this style of having explicit typing judgments. Having explicit typing judgment predicates on every variables makes testing easier as well, since extra type annotation comments are no longer needed when all axioms are in this style.

4.2 Testing axioms for arrays

Array bounds errors can cause problems in axioms as well as in programming. For example, recall the axiom `update_last` introduced in Section 3.

```
fof(update_last, axiom,
  ! [A, I, X, Y] : update(update(A, I, X), I, Y) = update(A, I, Y) ).
```

When we give the natural interpretation to `update`, the test routine will abort after a few rounds of test inputs because the index variable `I` will go out of range.

Rather than complicate the model by interpreting the result of `update` to include a special value for out-of-bounds errors, we modify the axiom to constrain the range of the array index variable:

```
fof(update_last_in_range, axiom,
  ! [A, I, X, Y]: ( (leq(lo(A), I) & leq(I, hi(A)))
    => update(update(A, I, X), I, Y) = update(A, I, Y) ) ).
```

Now, all tests on `update_last_in_range` succeed since we only generate test inputs satisfying the premise $(\text{leq}(\text{lo}(A), I) \ \& \ \text{leq}(I, \text{hi}(A)))$.

Similarly, we can also modify the axiom `update_commute` as follows.

```
fof(update_commute_in_range, axiom,
  ! [A, I, J, X, Y]:
  ( (leq(lo(A), I) & leq(I, hi(A)) & leq(lo(A), J) & leq(J, hi(A)))
    => update(update(A, I, X), J, Y) = update(update(A, J, Y), I, X) ) ).
```

Then, we can run the tests on the above axiom without array bounds error, and in fact discover counterexamples where `I` and `J` are the same but `X` and `Y` are different. We can correct this axiom to be valid as follows by adding the additional constraint that `I` and `J` are different (i.e., either `I` is less than `J` or vice versa).

```
fof(update_commute_in_range_fixed, axiom,
  ! [A, I, J, X, Y]:
  ( (leq(lo(A), I) & leq(I, hi(A)) & leq(lo(A), J) & leq(J, hi(A))
    & (lt(I, J) | lt(J, I)) )
    => update(update(A, I, X), J, Y) = update(update(A, J, Y), I, X) ) ).
```

The test for this new axiom succeeds for all test inputs.

As for the axiom `rotation2D_def`, itself, we observed above that it is quite different from the “natural” definition of the matrix. Thus, we test the axiom against the interpretation `rot2D` in Figure 5 with 100 randomly generated arrays of size 4 and find that it does indeed pass all tests.

Finally, the axiom `symm_joseph` in Figure 4 is intended to state that $\mathbf{A} + \mathbf{B}(\mathbf{C}\mathbf{D}\mathbf{C}^T + \mathbf{E}\mathbf{F}\mathbf{E}^T)\mathbf{B}^T$ is a symmetric matrix when \mathbf{A} and \mathbf{F} are $N \times N$ symmetric matrices and \mathbf{D} is an $M \times M$ symmetric matrix. This matrix expression, which is required to be symmetric, arises in the implementation of the Joseph update in Kalman filters. However, when we test this axiom for $N = M = 3$ and assuming \mathbf{B} , \mathbf{C} , and \mathbf{E} are all 3×3 matrices, we get counterexamples such as

$$(I0, J0, I, J, A, B, C, D, E, F, N, M) = \left(1, 0, 0, 0, \begin{pmatrix} 9.39 & 4.0 & -3.53 \\ 4.0 & 0.640 & -0.988 \\ -2.29 & -23.8 & -1.467 \end{pmatrix}, \dots \right).$$

We can immediately see that something is wrong since \mathbf{A} is not symmetric. The problem is that the scope of the quantifiers is incorrect and therefore does not correctly specify that the matrices are symmetric. This is fixed in `symm_joseph_fix` using another level of variable bindings for \mathbf{I} and \mathbf{J} , and the test succeeds for all test inputs under the same assumption that $N = M = 3$ and \mathbf{B} , \mathbf{C} , and \mathbf{E} are all 3×3 matrices. However, `symm_joseph_fix` still shares the same index range problem as `update_last` and `update_commute`. Moreover, nothing in the axiom prevents N and M being negative, and the dimensions for matrices \mathbf{B} , \mathbf{C} , and \mathbf{E} are not explicitly constrained to make the matrix operations `mmul` and `madd` well defined. One way to avoid index range concerns is to explicitly specify the index range for every variable that represents an array or matrices – this is in the same spirit of introducing typing judgment predicates over numeric variables for axioms on numeric operations.

5 Design and implementation of the axiom testing framework

We limit the class of first-order formulas which we test in order to make the implementation of the testing framework feasible. In our experience, these restrictions do not pose a practical problem since most verification related axioms fall into this category. Specifically, we assume that axioms are expressed in a restricted subset of TPTP FOF for which the following conditions hold:

- (1) there are no existential quantifications (that is, all quantifications are universal),
- (2) universal quantifications are nested to at most two levels (after lifting quantifiers in strictly positive position to the top level – see below),
- (3) inner universal quantifications can only appear in either strictly positive position or directly negative position in strictly positive position, and
- (4) universally quantified formulas in negative position (i.e., the left-hand side of an implication) have a finite number of satisfying valuations for the quantified variables when given a proper interpretation (and, given any required additional constraints for unrolling – see below).

We now explain these restrictions in greater detail.

```

fof(symm_joseph, axiom,
! [IO, JO, I, J, A, B, C, D, E, F, N, M] : (
  ( leq(0,IO) & leq(IO,N) & leq(0,JO) & leq(JO,N)
    & leq(0, I) & leq(I, M) & leq(0, J) & leq(J, M)
    & select2D(D, I, J) = select2D(D, J, I)
    & select2D(A,IO,JO) = select2D(A,JO,IO)
    & select2D(F,IO,JO) = select2D(F,JO,IO) )
=>
  select2D(madd(A,mmul(B,mmul(madd(mmul(C,mmul(D,trans(C))),
                                mmul(E,mmul(F,trans(E))))),
                                trans(B))))),
                                IO, JO)
= select2D(madd(A,mmul(B,mmul(madd(mmul(C,mmul(D,trans(C))),
                                mmul(E,mmul(F,trans(E))))),
                                trans(B))))),
                                JO, IO) ) ).

fof(symm_joseph_fix, axiom,
! [A, B, C, D, E, F, N, M] : (
  ( ( ! [I, J] : ( (leq(0,I) & leq(I,M) & leq(0,J) & leq(J,M))
                  => select2D(D,I,J) = select2D(D,J,I) ) )
    & ( ! [I, J] : ( (leq(0,I) & leq(I,N) & leq(0,J) & leq(J,N))
                  => select2D(A,I,J) = select2D(A,J,I) ) )
    & ( ! [I, J] : ( (leq(0,I) & leq(I,N) & leq(0,J) & leq(J,N))
                  => select2D(F,I,J) = select2D(F,J,I) ) ) )
=>
  ( ! [I, J] : ( (leq(0,I) & leq(I,N) & leq(0,J) & leq(J,N))
                => select2D(madd(A,mmul(B,mmul(madd(mmul(C,mmul(D,trans(C))),
                                                    mmul(E,mmul(F,trans(E))))),
                                                    trans(B))))),
                                                    I, J)
                = select2D(madd(A,mmul(B,mmul(madd(mmul(C,mmul(D,trans(C))),
                                                    mmul(E,mmul(F,trans(E))))),
                                                    trans(B))))),
                                                    J, I)
                ) ) ) ).

```

Fig. 4: An erroneous axiom on symmetric matrices and the fixed version

Assumption (1): In general, existential quantifications are hard to translate into testable properties. However, existentials appear sparingly in axioms for software verification. For example, in the software verification category (SWV) of the TPTP version 5.0.0 axiom set, only 4 out of 221 axioms use existential quantification.

Assumption (2): Axioms can start with a top-level universal quantification which can contain an inner universal quantification. However the inner quantified formula cannot contain any other universal quantification. For example, the testing framework will accept $! [X] : P$ and $! [X] : ((! [Y] : P) => (! [Z] : Q))$, but not $! [X] : ((! [Y] : ((! [Z] : P) => Q)) => R)$ where P, Q, R are quantifier-free formulas. More formally, we define a function *level* that counts the nesting level of quantifications as follows:

$$\begin{aligned}
\text{level}(\forall [x_1, \dots, x_n] : P) &= 1 + \text{level}(P) \\
\text{level}(\neg P) &= \text{level}(P) \\
\text{level}(P \text{ op } Q) &= \max(\text{level}(P), \text{level}(Q)) \quad \text{where op} \in \{\wedge, \vee, \Rightarrow, \Leftarrow, \Leftrightarrow\} \\
\text{level}(p(T_0, \dots, T_n)) &= 0 \quad \text{where p is a predicate symbol} \\
\text{level}(T_1 = T_2) &= 0
\end{aligned}$$

We only allow formulae of nesting levels less than or equal to two after flattening (Section 5.3.4), that is, after lifting the quantifiers at strictly positive positions to the top level. For

```

pred2hsInterpTable = [ ("rot2D",[|rot2D|]), ("lt",[|lt|]), ("leq",[|leq|]) ]
term2hsInterpTable =
  [ ("lo",[|lo|]), ("hi",[|hi|]), ("update",[|update|])
    , ("uminus",[|uminus|]), ("cos",[|cos|]), ("sin",[|sin|])
    , ("0",[|0|]), ("1",[|1|]), ("2",[|2|]), ("3",[|3|]) ]

rot2D :: (Array Integer Double, Double) -> Bool
rot2D(a,t) = elems a == [ cos t , sin t
                        , - sin t , cos t ]

lo a = fst( bounds a )
hi a = snd( bounds a )

uminus :: Double -> Double
uminus x = -x

update :: (Array Integer Double, Integer, Double) -> Array Integer Double
update(arr,i,c) = arr // [(i,c)]

leq(x,y) = x <= y
let(x,y) = x < y

```

Fig. 5: Interpretation for the 2D rotation matrix axiomatization

example, our testing framework can handle formulae of the form $!\mathbf{[X]} : (!\mathbf{[Y]} : (!\mathbf{[Z]} : Q))$ where Q is a level 0 subformula, since, after lifting, it becomes a flat formula $!\mathbf{[X,Y,Z]} : Q$ whose level is 1.

Assumption (3): A subformula is in strictly positive position when it is either the top-level formula or a subformula reachable by following the right hand side of implications. For example, in $A \Rightarrow (B \Rightarrow C) \Rightarrow D$, D is in strictly positive position, whereas B is positive (since it is negative of negative) but not strictly positive. Both A and C are in negative position, but only A is in what we refer to as “directly negative position”, which is the negative position inside a formula at strictly positive position (Note that the top-level formula is strictly positive). We only allow inner universal quantification in positions like A and D . Universally quantified variables in strictly positive positions disappear after flattening (Section 5.3.4) since they are lifted to the top level. Universally quantified variables in directly negative positions are unrolled (Section 5.3.4).

Logical negation also inverts positiveness and negativeness of the positions. For instance, both P and Q in $(P \Rightarrow \sim Q)$ are in negative position; $\sim Q$ is in positive position, but inside the negation it is inverted so that Q is negative.

We will often use “positive position” to mean “strictly positive position” and “negative position” to mean “directly negative positions in strictly positive positions” when we talk about inner quantifications, since we already assume these restrictions of where inner quantifications can appear in the axiom.

Assumption (4): Inner universal quantifications in negative position are unrolled to give a conjunction of formulas substituted by all possible satisfying valuations. This assumption allows us to conclude that the counterexamples we derive are indeed counterexamples, and is closely related to *Assumption (1)* on existential quantification since $!\mathbf{[X]} : ((!\mathbf{[Y]} : P) \Rightarrow Q)$ is logically equivalent to $!\mathbf{[X]} : ((?\mathbf{[Y]} : (\sim P) | Q))$.¹⁰ Existential quantifications in general

¹⁰ ? is the symbol for existential quantification in TPTP FOF syntax.

Category (1) axioms with no implications and no inner quantifications

```
fof(eq_refl, axiom, ![X] : %$ [integer]
    ( X = X ) ).
```

Category (2) axioms with implications but no inner quantifications

```
fof(gt_trans, axiom, ![X,Y] : %$ [integer,integer]
    ( (gt(X,Y) & gt(Y,Z)) => gt(X,Z) ) ).
```

```
fof(update_last, axiom,
    ![A,I,X,Y] : %$ [array4,integer,double,double]
    ( ( leq(lo(A),I) & leq(I,hi(A)) )
    => update(update(A,I,X),I,Y) = update(A,I,Y) ) ).
```

Category (3) axioms with implication and inner quantifications only in positive positions

```
fof(update_last_nested, axiom,
    ![A,I] : %$ [array4,integer]
    ( ( leq(lo(A),I) & leq(I,hi(A)) )
    => ( ![X,Y] : %$ [double,double]
        update(update(A,I,X),I,Y) = update(A,I,Y) ) ) ).
```

Category (4) axioms with implications and inner quantifications only in negative positions

```
fof(mat_symm_trans_flat, axiom,
    ![A, N, I, J] : %$ [matrix3x3,integer,integer,integer]
    ( ( ( ![I, J] : %$ [integer,integer]
        ( ( leq(0, I) & leq(I, N) & leq(0, J) & leq(J, N) )
        => (selectM(A, I, J) = selectM(A, J, I)) ) )
    & (N=hiRow(A)) & (0=loRow(A)) & (N=hiCol(A)) & (0=loCol(A)) )
    => ( ( leq(0, I) & leq(I, N) & leq(0, J) & leq(J, N) )
        => (selectM(trans(A),I,J) = selectM(trans(A),J,I)) ) ) ).
```

Category (5) axioms with implications and inner quantifications in both positive and negative positions

```
fof(mat_symm_trans, axiom,
    ![A, N] : %$ [matrix3x3,integer]
    ( ( ( ![I, J] : %$ [integer,integer]
        ( ( leq(0, I) & leq(I, N) & leq(0, J) & leq(J, N) )
        => (selectM(A, I, J) = selectM(A, J, I)) ) )
    & (N=hiRow(A)) & (0=loRow(A)) & (N=hiCol(A)) & (0=loCol(A)) )
    => ( ( ![I, J] : %$ [integer,integer]
        ( ( leq(0, I) & leq(I, N) & leq(0, J) & leq(J, N) )
        => (selectM(trans(A),I,J) = selectM(trans(A),J,I)) ) ) ) ).
```

Fig. 6: Examples of axioms in the testable FOL subset

are hard to translate into testable properties. However, when we know that the universally quantified variables at negative positions are bounded, we can eliminate them by enumerating all possible instances. Sometimes, we need to provide additional constraints to make unrolling possible. For example, to unroll $!\lceil N \rceil : (!\lceil I \rceil : ((\text{leq}(0, I) \& \text{leq}(0, N)) \Rightarrow P) \Rightarrow Q)$ by enumerating possible values of I from 0 up to the value of N , we need to supply additional constraints such as $(\text{assert } (= N 3))$.

Figure 6 lists examples of axioms that meet the above criteria in order of complexity from the simplest (`eq_refl`) to the most complex (`mat_symm_trans`). Using these examples, we now explain in detail how our testing framework derives testable properties (Section 5.1) and test data generators (Section 5.2), and then summarize the overall algorithm (Section 5.3). We discuss the class of testable axioms further in Section 7.

5.1 Deriving testable properties

Deriving properties from axioms with one level quantification without implications such as `eq_ref1` in Figure 6 is easy. It translates to $\lambda x \rightarrow x == x$. The universal quantification $! [X] : \dots$ translates to the lambda binding $\lambda x \rightarrow \dots$ in Haskell, and the quantified formula translates to Haskell predicates following the translation rules. As mentioned in Section 1.2, a user may define translation rules in addition to the predefined translation rules for logical connectives and the equality. For `eq_ref1` we only need to refer to a predefined rule mapping the equality to `(==)`.

For axioms with implications, the implication $(P \Rightarrow Q)$ translates to $(\text{not } p \mid\mid q)$ by the predefined translation rule for implication where p and q are translations of P and Q respectively.¹¹ For example, the axiom `gt_trans` translates to $\lambda (x, y, z) \rightarrow \text{not } (x > y \ \&\& \ y > z) \mid\mid \ x > z$. Here, we refer to the user-defined translation rule for logical terms mapping `gt` to `(>)`. Similarly, `update_last` translates to the property shown in Figure 2.

Nested universal quantifications in positive position (i.e., right-hand side of implication) are lifted to the top level before translating them into properties. For example, the axioms `update_last_nested` and `mat_symm_trans` are transformed to logically equivalent formulas `update_last` and `mat_symm_trans_flat` respectively, before translating them to properties. Note that $! [X] : (P \Rightarrow (! [Y] : Q))$ is logically equivalent to $! [X, Y] : (P \Rightarrow Q)$ assuming Y is not free in P .

Nested universal quantifications in negative position (i.e., left-hand side of implication) are unrolled to give conjunctions of formulas substituted by all possible satisfying valuations. Recall *Assumption (4)* that axioms have a finite number of satisfying valuations for their universal quantifications, under a proper interpretation (and sometimes with additional constraints). The testing framework often needs extra information such as type annotation comments or additional constraint specifications, which are not present in the axiom formulas, in order to unroll the universal quantifications in negative position. For example, the testing framework needs to know that the variable A in the axiom `mat_symm_trans_flat` is a 3×3 matrix in order to unroll and eliminate the inner universally quantified variables I and J appearing in `mat_symm_trans_flat`. Note that there are multiple quantifications of I and J in `mat_symm_trans_flat`: the top level one and the inner one. To avoid ambiguity from name conflicts, we alpha rename the inner quantified variables.¹² For instance,

```
! [I ,J] : %$ [integer, integer]
  ( ( leq(0, I) & leq(I, N) & leq(0, J) & leq(J, N) )
    => (selectM(A, I, J) = selectM(A, J, I)) ) )
```

may be alpha renamed to

```
! [I0 ,J0] : %$ [integer, integer]
  ( ( leq(0, I0) & leq(I0, N) & leq(0, J0) & leq(J0, N) )
    => (selectM(A, I0, J0) = selectM(A, J0, I0)) ) )
```

The whole alpha renamed formula is shown in Figure 7. Then, the testing framework can figure out that N is equal to 2 (since `hiRow(A)` and `hiCol(A)` both equal to 2 because A is a 3×3

¹¹ In the actual implementation, we attach some additional code for printing a debugging message when p fails (i.e., evaluates to False for a generated test case while testing) in order to alert the user to vacuously true test cases.

¹² The axiom `mat_symm_trans` happens to have duplicate names only in negative position, but the testing framework alpha renames any inner quantifications (including the ones at positive positions) when there are duplicate variable names.

```

fof(mat_symm_trans_flat_alpha_renamed, axiom,
  ![A, N, I, J] : %$ [matrix3x3,integer,integer,integer]
  ( ( ( ![IO ,JO] : %$ [integer,integer]
    ( ( leq(0, IO) & leq(IO, N) & leq(0, JO) & leq(JO, N) )
      => (selectM(A, IO, JO) = selectM(A, JO, IO)) ) )
    & (N=hiRow(A)) & (0=loRow(A)) & (N=hiCol(A)) & (0=loCol(A)) )
  => ( ( leq(0, I) & leq(I, N) & leq(0, J) & leq(J, N) )
    => (selectM(trans(A),I,J) = selectM(trans(A),J,I)) ) ) ).

```

Fig. 7: Alpha renamed axiom formula

```

fof(mat_symm_trans_flat_unrolled, axiom,
  ![A, N, I, J] : %$ [matrix3x3,integer,integer,integer]
  ( ( ( ( ( leq(0, 0) & leq(0, N) & leq(0, 0) & leq(0, N) )
    => (selectM(A, I, J) = selectM(A, J, I)) )
    & ( ( leq(0, 1) & leq(1, N) & leq(0, 0) & leq(0, N) )
    => (selectM(A, I, J) = selectM(A, J, I)) )
    & ( ( leq(0, 2) & leq(2, N) & leq(0, 0) & leq(0, N) )
    => (selectM(A, I, J) = selectM(A, J, I)) )
    & ( ( leq(0, 0) & leq(0, N) & leq(0, 1) & leq(1, N) )
    => (selectM(A, I, J) = selectM(A, J, I)) )
    & ( ( leq(0, 1) & leq(1, N) & leq(0, 1) & leq(1, N) )
    => (selectM(A, I, J) = selectM(A, J, I)) )
    & ( ( leq(0, 2) & leq(2, N) & leq(0, 1) & leq(1, N) )
    => (selectM(A, I, J) = selectM(A, J, I)) )
    & ( ( leq(0, 0) & leq(0, N) & leq(0, 2) & leq(2, N) )
    => (selectM(A, I, J) = selectM(A, J, I)) )
    & ( ( leq(0, 1) & leq(1, N) & leq(0, 2) & leq(2, N) )
    => (selectM(A, I, J) = selectM(A, J, I)) )
    & ( ( leq(0, 2) & leq(2, N) & leq(0, 2) & leq(2, N) )
    => (selectM(A, I, J) = selectM(A, J, I)) ) )
    & (N=hiRow(A)) & (0=loRow(A)) & (N=hiCol(A)) & (0=loCol(A)) )
  => ( ( leq(0, I) & leq(I, N) & leq(0, J) & leq(J, N) )
    => (selectM(trans(A),I,J) = selectM(trans(A),J,I)) ) ) ).

```

Fig. 8: Unrolled version of mat_symm_trans_flat

matrix) and I_0 and J_0 are in the range of 0 to 2 (since $\text{leq}(0, I_0)$, $\text{leq}(I_0, N)$, $\text{leq}(0, J_0)$, $\text{leq}(J_0, N)$ should hold), and unroll the inner quantified formula into a conjunction of nine formulas (see Figure 8), which are the instantiations of the quantified formula above, with all satisfying valuations $(0,0)$, $(0,1)$, \dots , $(2,2)$ for the pair of variables I_0 and J_0 . Once the universal quantification in the negative position is eliminated, the translation to a property is basically the same as translating `update_last`.

The nine satisfying valuations are automatically generated by the Yices constraint solver, solving the constraints extracted from the type annotation and the constraints extracted from the premise part (excluding the quantified formula to unroll, of course) of the axiom. We will revisit the details of extracting constraints in the following subsection (Section 5.2) while discussing test data generator derivation.

5.2 Deriving test data generators

We derive test data generators that generate non-trivial test cases to avoid vacuously true test cases. That is, the automatically derived test data generators will only generate test data that

satisfies the premise. The test data generator is actually defined by extracting the constraints from the premise. The constraints extracted from the premise must be the necessary and sufficient condition for satisfying the premise, and they should be solvable by Yices.

The automatically derived test data generator operates in two steps to generate each test case, as illustrated in Figure 2. It first solves the extracted constraints using the Yices constraint solver, and then generates *arbitrary* values for unconstrained variables using QuickCheck’s generator combinator library. For basic types such as integers and floating point numbers, it is predefined how to generate arbitrary values in the QuickCheck library. For the user defined types, the user should provide an instance of the `Arbitrary` class of the QuickCheck library, which enables users to define their own meaning of “arbitrary”.

In order to derive a test data generator for the axiom, the testing framework extracts constraints from the premise part of the axiom and the type annotation comments. Note that the testing framework can derive test data generators for fewer axioms than for which it can derive properties because the category of solvable constraints is limited by the ability of the constraint solver used. In the majority of cases, this limitation is not a problem since many of the constraints from axiom premises are linear (such as array bounds). And, even when we cannot derive solvable constraints for the axiom, it is still helpful to have an auto-derived property since we can manually write the test data generator for the property.

Deriving test data generators from axioms without implications such as `eq_ref1` in Figure 6 is trivial, because there are no constraints to extract. Note that our testing framework only extract constraints from the premise (i.e., left-hand side of implication) of the axiom.

We extract constraints from the axiom with implications; in particular, from the premise. Since Yices is typed, it needs to know the types of the variables appearing in the constraints. Since this type information is available in the type annotation comments, the testing framework can introduce Yices `define` commands for each variable. For example, it introduces two commands (`define x::int`) and (`define y::int`) for the axiom `gt_trans`, since both logic variables `X` and `Y` are annotated as integers.

Some type annotations (e.g., `array4`, `matrix3x3`) contain extra information such as index size constraints, while others (e.g., `integer`, `double`) do not. This additional information introduces further constraints as Yices `assert` commands in addition to the `define` commands. For instance, the variable `A` in the axiom `update_last A` has type annotation `array4`. Thus, it introduces (`define a::(-> int real)`), and it also introduces (`assert (= (lo a) 0)`) and (`assert (= (hi a) 3)`) as shown in Figure 2. The other variables `I`, `X`, and `J` in `update_last` introduce one `define` command each: (`define i::int`), (`define x::real`), and (`define y::real`).¹³

Nested universal quantifications in positive position are lifted to the top level before extracting constraints. This is the same transformation as in the property derivation. For example, the axioms `update_last_nested` and `mat_symm_trans` are transformed to logically equivalent formulas `update_last` and `mat_symm_trans_flat` respectively.

Nested universal quantifications in negative position are unrolled to give a conjunction of formulas substituted by all possible satisfying valuations. This is the same transformation as in the property deriving, but we have not yet discussed the details of how we unroll the universal quantification in negative positions. Here, we discuss more details on how we use Yices to enumerate all possible satisfying solutions of the universally quantified variables at the negative positions with the `mat_symm_trans_flat` example. First, the testing framework alpha renames the inner quantified variables just as in the property derivation – see `mat_symm_trans_flat` in Figure 7. Then, the testing framework extracts constraints from

¹³ `define` commands derived from type annotation comments were omitted from Figure 2 for simplicity.

the top level type annotation and partially extracts constraints from the premise. It extracts the following constraints from the type annotation `matrix3x3` for `A`:

```
(assert (= 2 (hiRow a)))
(assert (= 0 (loRow a)))
(assert (= 2 (hiCol a)))
(assert (= 0 (loCol a)))
```

where `hiRow` and `colRow` would be declared properly in some `.yices` file as

```
(define hiRow :: (-> (-> int (-> int real)) int))
(define loRow :: (-> (-> int (-> int real)) int))
(define hiCol :: (-> (-> int (-> int real)) int))
(define loCol :: (-> (-> int (-> int real)) int))
```

in a similar fashion to `hi` and `lo` declared in Figure 2. It also extract partial constraints from the premise of the axiom excluding the the inner quantified formula, which we are to unroll, That is, from $(N=hiRow(A)) \ \& \ (0=loRow(A)) \ \& \ (N=hiCol(A)) \ \& \ (0=loCol(A))$ it generates following constraints:

```
(assert (= n (hiRow a)))
(assert (= 0 (loRow a)))
(assert (= n (hiCol a)))
(assert (= 0 (loCol a)))
```

To unroll the inner quantified formula

```
![I0 ,J0] : %$ [integer,integer]
  ( ( leq(0, I0) & leq(I0, N) & leq(0, J0) & leq(J0, N) )
    => (selectM(A, I0, J0) = selectM(A, J0, I0)) ) )
```

the testing framework extracts constraints from the premise of this inner quantified formula. That is, from $leq(0, I0) \ \& \ leq(I0, N) \ \& \ leq(0, J0) \ \& \ leq(J0, N)$, it extracts the following constraints:

```
(assert (<= 0 i0))
(assert (<= i0 n))
(assert (<= 0 j0))
(assert (<= j0 n))
```

Finally, the testing framework invokes Yices to solve these 12 constraints (8 from the top premise of the axiom excluding the inner formula, and 4 from the premise of the inner formula). Yices then exhaustively searches for all solutions. To make Yices to search for all solutions, we repeatedly assert the negations of the value assignments (the output following the `(check)` command) for the variables of our interest, which are `i0` and `j0`, until we reach the unsatisfiable state (`(unsat)`) of Yices. After finding all satisfying valuations for `I0` and `J0`, it is trivial to unroll `mat_symm_trans_flat` in Figure 6 into `mat_symm_trans_flat_unrolled` in Figure 8. Once the axiom formula is unrolled and all nested quantifications are flattened, extracting constraints from the unrolled axiom formula is basically the same as for `gt_trans` or `update_last`.

$$\text{formula2prop}_{\text{top}}(\forall[X_1, \dots, X_n] : P) = \lambda(x_1, \dots, x_n) \rightarrow \text{formula2prop}(P) \quad (1)$$

$$\text{formula2prop}(\forall[X_1, \dots, X_n] : P) = \text{error}(\text{"cannot happen"}) \quad (2)$$

$$\text{formula2prop}(P \Rightarrow Q) = \text{not}(\text{formula2prop}(P)) \text{ || } \text{formula2prop}(Q) \quad (3)$$

$$\text{formula2prop}(Q \Leftarrow P) = \text{formula2prop}(P \Rightarrow Q) \quad (4)$$

$$\text{formula2prop}(P \Leftrightarrow Q) = \text{formula2prop}(P \Rightarrow Q) \ \&\& \ \text{formula2prop}(P \Leftarrow Q) \quad (5)$$

$$\text{formula2prop}(P \wedge Q) = \text{formula2prop}(P) \ \&\& \ \text{formula2prop}(Q) \quad (6)$$

$$\text{formula2prop}(P \vee Q) = \text{formula2prop}(P) \ \text{||} \ \text{formula2prop}(Q) \quad (7)$$

$$\text{formula2prop}(\neg P) = \text{not}(\text{formula2prop}(P)) \quad (8)$$

$$\text{formula2prop}(T_1 = T_2) = \text{term2exp}(T_1) \text{ === } \text{term2exp}(T_2) \quad (9)$$

$$\text{formula2prop}(p(T_1, \dots, T_n)) = f_p(\text{term2exp}(T_1), \dots, \text{term2exp}(T_n)) \quad (10)$$

where f_p is the Haskell interpretation of p

$$\text{term2exp}(t(T_1, \dots, T_n)) = f_t(\text{term2exp}(T_1), \dots, \text{term2exp}(T_n)) \quad (11)$$

where f_t is the Haskell interpretation of t

Fig. 9: Property derivation algorithm

5.3 Algorithms

We summarize the algorithms for the informally explained steps of property derivation and constraint derivation in the previous subsections. While describing these algorithms, we assume that the formula input to the algorithm are all alpha renamed and free from name conflict issues. We describe the property derivation algorithm in Section 5.3.1 and the constraint derivation algorithm in Section 5.3.2. The unrolling algorithm for unrolling inner quantification at negative positions are described in Section 5.3.3, and the flattening algorithm for lifting inner quantified variables at positive positions to top level in Section 5.3.4.

Note that the testing framework applies the algorithms in the reverse order of our presentation order in this section, since flattening and unrolling are pre-processing steps prior to constraint derivation or property derivation.

5.3.1 Property Derivation Algorithm

Figure 9 gives the property derivation algorithm. The function $\text{formula2prop}_{\text{top}}$, Equation (1) in Figure 9, applies to the axiom formula. It translates universally quantified variables X_1, \dots, X_n into Haskell lambda bindings x_1, \dots, x_n , and applies formula2prop to the quantified formula P . The time complexity of the formula2prop algorithm is linear in the size of the input formula.

Since we assume the inner quantified formulas are flattened and unrolled before property derivation, Equation (2) is not used. Equation (3) translates logical implication based on the fact that $P \Rightarrow Q$ is logically equivalent to $\neg P \vee Q$. Equations (4) and (5) for other implication related connectives are implemented via Equation (3). Equations (Equation (6)), (Equation (7)), and (Equation (8)) for translating logical conjunction, disjunction, and negation are self-explanatory. Equation (9) translates equality. Equality in First-Order Logic is a special

$$\text{premise2yices}(\forall[X_1, \dots, X_n] : P) = \text{error}(\text{"cannot happen"}) \quad (12)$$

$$\text{premise2yices}(P \Rightarrow Q) = (\text{or } (\text{not } p) \ q) \text{ where } p = \text{premise2yices}(P) \\ q = \text{premise2yices}(Q) \quad (13)$$

$$\text{premise2yices}(Q \Leftarrow P) = \text{premise2yices}(P \Rightarrow Q) \quad (14)$$

$$\text{premise2yices}(P \Leftrightarrow Q) = (= \ p \ q) \text{ where } p = \text{premise2yices}(P) \\ q = \text{premise2yices}(Q) \quad (15)$$

$$\text{premise2yices}(P \wedge Q) = (\text{and } p \ q) \text{ where } p = \text{premise2yices}(P) \\ q = \text{premise2yices}(Q) \quad (16)$$

$$\text{premise2yices}(P \vee Q) = (\text{or } p \ q) \text{ where } p = \text{premise2yices}(P) \\ q = \text{premise2yices}(Q) \quad (17)$$

$$\text{premise2yices}(\neg P) = (\text{not } p) \text{ where } p = \text{premise2yices}(P) \quad (18)$$

$$\text{premise2yices}(T_1 = T_2) = (= \ e_1 \ e_2) \text{ where } e_1 = \text{term2yices}(T_1) \\ e_2 = \text{term2yices}(T_2) \quad (19)$$

$$\text{premise2yices}(p(T_1, \dots, T_n)) = f_p(\text{term2yices}(T_1), \dots, \text{term2yices}(T_n)) \\ \text{where } f_p \text{ is the interpretation of } p \\ \text{which is a meta-function} \\ \text{that generates Yices assertions} \quad (20)$$

$$\text{term2yices}(c) = e \text{ where } e \text{ is the Yices interpretation of } c \quad (21)$$

$$\text{term2yices}(f(T_1, \dots, T_n)) = (e \ e_1 \ \dots \ e_n) \\ \text{where } e \text{ is the Yices interpretation of } f \\ e_i = \text{term2yices}(T_i) \text{ for } i = 1 \dots n \quad (22)$$

Fig. 10: Constraint derivation algorithm

binary predicate which we always translate as an overloaded infix operator (`===`) in Haskell, which user can define their own instance to give customized definition for the meaning of equality, applied to the translations of the right-hand side and left-hand side arguments. Equation (10) translates other n -ary predicates. Translations of these predicates depend on the interpretation provided by the user. It maps the predicate symbol p to f_p a function that takes n Yices expressions, which are the translations of the argument terms of the predicate.

The function `term2exp`, Equation (11) in Figure 9, takes an n -ary term and translates it into a Haskell expression according to the user provided interpretation. It maps the function symbol τ to f_τ , a Haskell function whose type is a function that takes n arguments, applied to the translations of the argument terms.

5.3.2 Constraint derivation algorithm

The function `premise2yices` in Figure 10 applies to the premise part of the axiom. That is, when we have an axiom $\forall[X_1, \dots, X_n] : (P \Rightarrow Q)$, we collect the constraints of the axiom by evaluating `premise2yices(P)`. The time complexity of the `premise2yices` algorithm is linear in the size of the input premise formula.

The function `premise2yices` takes a formula as argument and returns a Yices expression of type `bool`. Since we assume the inner quantified formulas are flattened and unrolled

before property derivation, Equation (12) is not used. Equation (13) translates logical implication based on the fact that $P \Rightarrow Q$ is logically equivalent to $\neg P \vee Q$. Equations (14) and (15) for other implication related connectives are implemented via Equation (13). Equations (16), (17), and (18) for translating logical conjunction, disjunction, negation, are self-explanatory. Equation (19) translates equality. Equality in First-Order Logic is a special binary predicate which we always translate as Yices equality applied to the translations of the right-hand side and left-hand side arguments. Equation (20) translates other n -ary predicates. Translations of these predicates depend on the interpretation provided by the user. It maps the predicate symbol p to f_p a function that takes n Yices expressions, which are the translations of the argument terms of the predicate.

The function `term2yices` in Figure 10 takes a term and translates it into a Yices expression. Equation (21) translates constant symbols (i.e., 0-ary terms) into Yices expressions according to the user provided interpretation. Equation (22) translates n -ary terms where $n > 0$ into Yices expressions. It maps the function symbol f to e , a Yices expression whose type is a function that takes n Yices expression arguments, applied to $e_1 \cdots e_n$, the translations of the argument terms of the function symbol.

5.3.3 Unrolling algorithm

When nested universal quantifications appear at negative positions, (i.e., the premise part of the axioms), our testing framework tries to unroll them. Recall, we assumed that the quantified formula to unroll is in the form of implication and have finite satisfying valuations for the quantified variables (see the fourth assumption in the beginning of Section 5).

Some inner quantified formulas have enough information to unroll them. For example, we can unroll the quantified formula `! [I] : ((1eq(0, I) & 1eq(I, 2)) => select(A, I) = 0)` since we know that that satisfying valuations for `I` are 0, 1, and 2 from the premise part of the quantified formula alone, without any additional information. Other inner quantified formulas, such as `mat_symm_trans_flat` which we discussed in Section 5.1, need extra constraints from other parts of the axiom to unroll the quantified formula. We call such extra constraints from other parts of the axiom *outer constraints*.

We can calculate the outer constraint for any subformula (except the subformulas inside the inner quantified formula to unroll) appearing in the premise part of the axiom, which is not necessarily a quantified formula. The function `outerConstraint` in Figure 11 gives the algorithm for tagging outer constraints to the subformulas appearing in the premise part of the axiom. The function `outerConstraint` takes two arguments of an outer constraint and a formula, and returns a tagged version of the formula where each subformula is tagged with its outer constraint.¹⁴ The symbols P' and Q' appearing in the definition of `outerConstraint` are the tagged version of the plain formulas P and Q .

Equations (28) and (29) are the core of `outerConstraint`. Equation (28) tags the outer constraints of a logical conjunction. Consider the premise $R \wedge (P \wedge Q)$. Then the outer constraint of $P \wedge Q$ would be the constraint derived from R , say c , since R is everything outside the subformula $P \wedge Q$. The outer constraint of P is $(\text{and } c \ c_2)$ where c_2 is the constraint derived from Q . We need to collect both constraints c_2 , as well as c , as the outer constraint of P , since Q should hold as well as P in order to make the conjunction $P \wedge Q$ be true. Similarly, the outer constraint of Q is $(\text{and } c \ c_1)$ where c_1 is the constraint derived from P .

¹⁴ We express both plain formula and the tagged formula with a common parametrized data structure using the well known functional programming idiom called *indirect composite* (see http://haskell.org/haskellwiki/Indirect_composite).

$$\text{outerConstraint}(c, \forall [x_1, \dots, x_n] : P) = (c, \forall [x_1, \dots, x_n] : P') \quad (23)$$

$$\text{outerConstraint}(c, P \Rightarrow Q) = (c, \widehat{P} \Rightarrow \widehat{Q}) \quad (24)$$

$$\text{outerConstraint}(c, Q \Leftarrow P) = (c, \widehat{Q} \Leftarrow \widehat{P}) \quad (25)$$

$$\text{outerConstraint}(c, P \Leftrightarrow Q) = (c, \widehat{P} \Leftrightarrow \widehat{Q}) \quad (26)$$

$$\text{outerConstraint}(c, P \wedge Q) = (c, P' \wedge Q') \quad (27)$$

$$\text{where } c_1 = \text{premise2yices}'(P)$$

$$c_2 = \text{premise2yices}'(Q)$$

$$P' = \text{outerConstraint}(\text{and } c_2, P)$$

$$Q' = \text{outerConstraint}(\text{and } c_1, Q) \quad (28)$$

$$\text{outerConstraint}(c, P \vee Q) = (c, P' \vee Q')$$

$$\text{where } P' = \text{outerConstraint}(c, P)$$

$$Q' = \text{outerConstraint}(c, Q) \quad (29)$$

$$\text{outerConstraint}(c, \neg P) = (c, \neg \widehat{P}) \quad (30)$$

$$\text{outerConstraint}(c, T_1 = T_2) = (c, T_1 = T_2) \quad (31)$$

$$\text{outerConstraint}(c, \text{p}(T_1, \dots, T_n)) = (c, \text{p}(T_1, \dots, T_n)) \quad (32)$$

where \widehat{P} and \widehat{Q} above are lifted versions of P and Q with all their nodes tagged with `true`, and

$$\text{premise2yices}'(\forall [x_1, \dots, x_n] : P) = \text{true} \quad (33)$$

$$\text{premise2yices}'(P \Rightarrow Q) = (\text{or } (\text{not } p) q) \text{ where } p = \text{premise2yices}'(P)$$

$$q = \text{premise2yices}'(Q) \quad (34)$$

$$\text{premise2yices}'(Q \Leftarrow P) = \text{premise2yices}'(P \Rightarrow Q) \quad (35)$$

$$\text{premise2yices}'(P \Leftrightarrow Q) = (= p q) \text{ where } p = \text{premise2yices}'(P)$$

$$q = \text{premise2yices}'(Q) \quad (36)$$

$$\text{premise2yices}'(P \wedge Q) = (\text{and } p q) \text{ where } p = \text{premise2yices}'(P)$$

$$q = \text{premise2yices}'(Q) \quad (37)$$

$$\text{premise2yices}'(P \vee Q) = (\text{or } p q) \text{ where } p = \text{premise2yices}'(P)$$

$$q = \text{premise2yices}'(Q) \quad (38)$$

$$\text{premise2yices}'(\neg P) = (\text{not } p) \text{ where } p = \text{premise2yices}'(P) \quad (39)$$

$$\text{premise2yices}'(T_1 = T_2) = (= e_1 e_2) \text{ where } e_1 = \text{term2yices}(T_1)$$

$$e_2 = \text{term2yices}(T_2) \quad (40)$$

$$\text{premise2yices}'(\text{p}(T_1, \dots, T_n)) = f_p(\text{term2yices}(T_1), \dots, \text{term2yices}(T_n)) \quad (41)$$

where f_p is the interpretation of `p`

which is a meta-function

that generates Yices assertions (42)

Fig. 11: tagging outer constraints

Equation (29) tags the outer constraints of a logical disjunction. Consider the premise $R \wedge (P \vee Q)$. Then the outer constraint of $P \vee Q$ would be the constraint derived from R , let us call it c . The outer constraint of P is also just c . Here, we need not propagate the constraints from Q since the disjunction can hold by satisfying P alone without satisfying Q . For similar reason, the outer constraint of Q is also just c .

We end the recursion and tag the formula with the accumulated constraint c when we see an atomic formula in Equations (31) and (32). We also stop recursion on Equations (23), (24), (25), (26), and (30) since inner quantification cannot appear in another inner quantification, implication inside premise, or negation.

Note that `outerConstraint` applies to the premise part of the axiom, and so is `premise2yices'`. The function `premise2yices'`, which calculates the constraint of each subformulas, is same as `premise2yices` in Figure 10 except that it returns `true` for the quantified formula instead of an error (see Equation (23)). The reason that `premise2yices'` returns `true` for the quantified formula is because we cannot not yet look into the inner quantified formula to collect constraints since we treat it as a hole to fill in by unrolling (`true` means no constraint).

Once we have tagged the premise with outer constraints, unrolling the inner quantified formula is straightforward. Let c be the outer constraint of $\forall[x_1, \dots, x_n] : (P \Rightarrow Q)$, and c' be the constraint derived from P . We only need to search for all possible valuations for x_1, \dots, x_n that satisfy $(\text{and } c \text{ } c')$ using `Yices`.

The time complexity of `outerConstraint` is linear, but the time complexity of the unrolling algorithm also depends on that of the decision procedure of `Yices` to solve the constraint $(\text{and } c \text{ } c')$. Since we are searching for all possible valuations for $(\text{and } c \text{ } c')$, and not just one, the complexity will depend on the number of possible valuations as well. Therefore, we cannot give a definitive bound on the time complexity of the unrolling algorithm in terms of the size of the input. Note that unrolling is a heuristic which assumes that the effective existentially quantified variable appearing in the premise has a finite solution.

5.3.4 Flattening algorithm

Figure 12 gives the flattening algorithm. The function `flattentop`, Equation (43) in Figure 12, applies to the axiom formula. It applies the function `flatten` to the quantified formula P to collect the variables to lift: Y_1, \dots, Y_n are the variables lifted from the positive positions of P , and P' is the resulting formula after lifting P . The time complexity of the flattening algorithm is linear in the size of the input formula.

Equation (44) is where the lifting happens. We collect the quantified variables and leave the quantified formula P detaching the universal quantification. We need not recurse on P since we assume only two levels of quantification including the top level.

Equations (45), (46), and (47) are for lifting formulas of implication related connectives. The latter two equations (46) and (47) call on the former one (45). In Equation (45), the function `flatten` recurses on the positive position Q and the variables xs lifted from Q onto its result.

Equations (48) and (49) for logical conjunction and disjunction simply recurse over the structure of the formulas and concatenate the lifted variables from P and Q .

We do not do anything inside the negation in Equation (50), since lifting out universally quantified variables introduces existentials. Similarly, equations (51) and (52) for equality and other predicates have no effect.

$$\begin{aligned} \text{flatten}_{\text{top}}(\forall[X_1, \dots, X_n] : P) &= \forall[X_1, \dots, X_n, Y_1, \dots, Y_m] : P \\ &\text{where } ([Y_1, \dots, Y_m], P') = \text{flatten}(P) \end{aligned} \quad (43)$$

$$\text{flatten}(\forall[X_1, \dots, X_n] : P) = ([X_1, \dots, X_n], P) \quad (44)$$

$$\text{flatten}(P \Rightarrow Q) = (xs, P \Rightarrow Q') \text{ where } (xs, Q') = \text{flatten}(Q) \quad (45)$$

$$\text{flatten}(Q \Leftarrow P) = \text{flatten}(P \Rightarrow Q) \quad (46)$$

$$\begin{aligned} \text{flatten}(P \Leftrightarrow Q) &= \text{flatten}(P \Rightarrow Q) \wedge \text{flatten}(P_\alpha \Leftarrow Q) \\ &\text{where } P_\alpha \text{ is an alpha renamed } P \text{ with fresh variables} \end{aligned} \quad (47)$$

$$\begin{aligned} \text{flatten}(P \wedge Q) &= (xs ++ ys, \text{flatten}(P) \wedge \text{flatten}(Q)) \\ &\text{where } (xs, P') = \text{flatten}(P) \\ &\quad (ys, Q') = \text{flatten}(Q) \end{aligned} \quad (48)$$

$$\begin{aligned} \text{flatten}(P \vee Q) &= (xs ++ ys, \text{flatten}(P) \vee \text{flatten}(Q)) \\ &\text{where } (xs, P') = \text{flatten}(P) \\ &\quad (ys, Q') = \text{flatten}(Q) \end{aligned} \quad (49)$$

$$\text{flatten}(ys, \neg P) = (ys, \neg P) \quad (50)$$

$$\text{flatten}(T_1 = T_2) = ([], T_1 = T_2) \quad (51)$$

$$\text{flatten}(p(T_1, \dots, T_n)) = ([], p(T_1, \dots, T_n)) \quad (52)$$

Fig. 12: flattening algorithm

6 Testing axioms from the TPTP axiom sets

In addition to axioms developed in the AUTOCERT project, we have also tested axioms from the TPTP distribution. The TPTP axiom corpus contains a wide variety of axiomatized theories, and we have tested constructive axioms from the algebra (ALG) and geometry (GEO) axiom set categories. Axioms from the software creation (SWC) and software verification (SWV) categories could also be tested though some would require more effort in order to develop models for the appropriate domain knowledge (e.g., concurrency and security models).

In this section we describe axiom testing in full detail, including the Haskell scripts containing interpretations and how to invoke the tests. We test axioms from ALG and GEO in their original form, only adding type annotation comments to specify the types of quantified variables.

6.1 Median algebra axioms

There is only one axiom file (ALG002+0.ax) in the ALG category of TPTP version 5.0.0, which consists of four simple axioms shown below:

```
f of (majority, axiom, ( ! [X,Y] : /*$ [integer, integer] */ f(X,X,Y) = X ) ).
```

```
f of (permute1, axiom, (
  ! [X,Y,Z] : /*$ [integer, integer, integer] */ f(X,Y,Z) = f(Z,X,Y) ) ).
```

```

fof(permute2,axiom,(
! [X,Y,Z] : /*$ [integer,integer,integer] */ f(X,Y,Z) = f(X,Z,Y) )).

fof(associativity,axiom,(
! [W,X,Y,Z] : %$ [integer,integer,integer,integer]
f(f(X,W,Y),W,Z) = f(X,W,f(Y,W,Z)) )).

```

The axioms above are quoted verbatim from the axiom file in the TPTP distribution, only adding type annotation comments. These axioms axiomatize requirements of how median functions should behave. The function symbol `f` represents a median function that returns a median value among its three arguments. For example, if `f` were to be defined on integers $f(1,3,2) = 2$.

To test the axioms we need to write three Haskell script files in Figure 13: one for parsing in axioms (`ALGLoad.hs`), another for defining interpretation (`ALGInterp.hs`), and the other for the main test script (`ALGMain.hs`). Figure 13 contains the full source code of these three files.

In `ALGLoad.hs`, we parse and read in the list of axioms using the `parseAxiomsFile2` function defined in the `CommonUtils` module. We import the `CommonUtil` module which contains common utilities such as `parseAxiomsFile2`. We name each of the parsed in axioms as `majority`, `permute1`, `permute2`, `associativity`.¹⁵ It would not be difficult to automatically generate scripts like `ALGLoad.hs` for loading axioms and binding them to suitable Haskell names. If we had automated this part, we would only need to write the other two files for testing an axiom set.

In `ALGInterp.hs`, we define interpretation of predicate symbols and function symbols. Since there are no predicate symbols to interpret in the median algebra axioms, we leave the `pred2hsTable` empty. Since `f` is the only uninterpreted symbol appearing in the median algebra axioms, we only need give an interpretation of `f`. Since `f` is a function symbol, in `term2hsTable`, we map `f` to Haskell function `median`, which is a straightforward implementation of a median function over integer triples. We leave both `pred2yicesTable` and `term2hsTable` empty, since we do not need to solve any constraints when testing these axioms. Note, constraints arise from the premise of the implication, but there are no logical implication forms appearing in the median algebra axioms.

We also define an interpretation for the types used in the type annotation comments in `ty2tyTable`. We omitted the discussion of type interpretations in the previous sections since it is similar but simpler than interpretations for predicate and function symbols. We map the type annotation `integer` to a correspondingly named Haskell integer value `integer` defined in the `CommonTypes` module, which also contains definitions for other type values such as `double` for the Haskell type `Double`, and also some array and matrix type values (`array4`, `matrix2x2`, `matrix3x3`). Finally, we collect all this information to create a record `alginterp`.

In `ALGMain.hs`, we import the parsed axiom formulae (`ALGLoad`) and the interpretation `ALGInterp` to define runnable tests. The `MACROS.h` we include provides easy and simple to use macro functions such as `DERIVE_PROP_WITH_GEN`, which wraps some of the “clutter” that comes from using Template Haskell.¹⁶ The macro functions in `MACROS.h` require

¹⁵ Here, we have chosen the names to coincide with the axioms names specified in the axiom file for clarity, but this is not necessary.

¹⁶ We use Template Haskell to plug the interpretations of symbols with Haskell code into the automatically derived properties, which are also first class Haskell functions. This gives us the ability to interactively run tests and evaluate both the derived properties and the interpretation functions in the interactive environment of GHC. However, programming with Template Haskell is more complicated than writing plain Haskell code in

```

module ALGLoad where

import System.IO.Unsafe (unsafePerformIO) -- Haskell base lib
import CommonUtil -- common utility of testing framework

axioms = unsafePerformIO ( parseAxiomsFile2 "tptp5f/ALG002+0.ax" )

[majority, permutel, permute2, associativity] = axioms

```

(a) ALGLoad.hs – parsing axioms from the file

```

{-# LANGUAGE TemplateHaskell #-}
module ALGInterp where

import Data.List (sort) -- Haskell base lib
import CommonUtil -- common utility of testing framework
import CommonTypes -- integer, double, and array types

pred2hsTable = [ ]
term2hsTable = [ ("f", [| median |]) ]
pred2yicesTable = [ ]
term2yicesTable = [ ]
ty2tyTable = [ ("integer", [| integer |]) ]

median :: (Integer, Integer, Integer) -> Integer
median (x,y,z) = sort [x,y,z] !! 1

alginterp = Interp { predInterp = pred2hsTable
                    , termInterp = term2hsTable
                    , includeY = [ ]
                    , predInterpY = pred2yicesTable
                    , termInterpY = term2yicesTable
                    , typeInterp = ty2tyTable }

```

(b) ALGInterp.hs – defining the interpretation

```

{-# LANGUAGE CPP, TemplateHaskell #-}
module ALGMain where

import Codec.TPTP -- TPTP syntax and parser lib
import CommonUtil -- common utility of testing framework
import CommonTypes -- integer, double, and array types
import YicesQuick -- interfacing Yices and QuickCheck
import GenTest -- property and generator derivation
import ALGLoad
import ALGInterp

#define INTERPRETATION alginterp
#include "MACROS.h"

qcMajor = mQuickCheck DERIVE_PROP_WITH_GEN( (formula majority) )
qcPerm1 = mQuickCheck DERIVE_PROP_WITH_GEN( (formula permutel) )
qcPerm2 = mQuickCheck DERIVE_PROP_WITH_GEN( (formula permute2) )
qcAssoc = mQuickCheck DERIVE_PROP_WITH_GEN( (formula associativity) )

```

(c) ALGMain.hs – the main test script

Fig. 13: Haskell scripts for testing median algebra axioms

INTERPRETATION macro to be defined. Here, we set `alginterp` as the definition for the macro `INTERPRETATION`. These macros are expanded by CPP, the C preprocessor, which GHC invokes when the language pragma `CPP` is specified at the top of the file. The `mQuickCheck` function is a monadic wrapper function of the `quickCheck` library function of QuickCheck, which takes an `IO prop` type argument whereas `quickCheck` takes a `prop` type argument. We need this wrapper since the derived test data generator exploits impure `IO` effects for inter-process communication with Yices. When we only derive the property without deriving the generator, we can use `quickCheck` library function directly. The `formula` selects the `formula` field of the axiom record, which has several other records such as the `name` field and an optional `annotation` field (see `logic-TPTP`¹⁷ library for details).

After writing these three Haskell scripts, we can load the test script with GHCi, GHC's interactive environment, as follows:

```
$ ghci ALGMain.hs
GHCi, version 6.12.1: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
[1 of 8] Compiling TyAnnSyntax      ( TyAnnSyntax.hs, interpreted )
[2 of 8] Compiling YicesQuick        ( YicesQuick.hs, interpreted )
[3 of 8] Compiling CommonTypes      ( CommonTypes.hs, interpreted )
[4 of 8] Compiling CommonUtil      ( CommonUtil.hs, interpreted )
[5 of 8] Compiling GenTest          ( GenTest.hs, interpreted )
[6 of 8] Compiling ALGLoad          ( ALGLoad.hs, interpreted )
[7 of 8] Compiling ALGInterp        ( ALGInterp.hs, interpreted )
[8 of 8] Compiling ALGMain          ( ALGMain.hs, interpreted )
...
Ok, modules loaded: ALGMain, CommonUtil, CommonTypes, YicesQuick, GenTest,
ALGLoad, ALGInterp, TyAnnSyntax.
*ALGMain> qcMajor
[(define _X::int),(define _Y::int)]
+++ OK, passed 100 tests.
*ALGMain> qcPerm1
[(define _X::int),(define _Y::int),(define _Z::int)]
+++ OK, passed 100 tests.
*ALGMain> qcPerm2
[(define _X::int),(define _Y::int),(define _Z::int)]
+++ OK, passed 100 tests.
*ALGMain> qcAssoc
[(define _W::int),(define _X::int),(define _Y::int),(define _Z::int)]
+++ OK, passed 100 tests.
```

6.2 Constructive geometry axioms

We have tested some constructive geometry axioms in the `GEO` category, which axiomatizes properties of points and lines—in particular, the `GEO006` and `GEO008` axiom sets. Figure 14 shows the axioms in one of the axiom files.

We choose to model this geometry in the two dimensional Cartesian plane. There are other possible models, including three dimensional Cartesian space, but the geometry of points and lines in the Cartesian plane is the simplest meaningful model. We model points as arrays of size 2 whose first and second elements are x and y coordinate values and lines

several ways because of the stage restriction for compile time. We hide such details by using a preprocessing macro.

¹⁷ <http://hackage.haskell.org/package/logic-TPTP>

```

%---Compatibility of convergence and unorthogonality
fof(occ1,axiom,(
  ! [L,M] : %$ [line, line]
    ( convergent_lines(L,M)
      | unorthogonal_lines(L,M) ) )).

%---Apartness axiom for the conjunction of convergence and unorthogonality
fof(oac1,axiom,(
  ! [L,M,N] : %$ [line, line, line]
    ( ( convergent_lines(L,M)
        & unorthogonal_lines(L,M) )
      => ( ( convergent_lines(L,N)
            & unorthogonal_lines(L,N) )
          | ( convergent_lines(M,N)
              & unorthogonal_lines(M,N) ) ) ) ).

%---Axioms for the orthogonal construction
fof(oo1,axiom,(
  ! [A,L] : %$ [point, line]
    ~ unorthogonal_lines(orthogonal_through_point(L,A),L) ).

fof(oo2,axiom,(
  ! [A,L] : %$ [point, line]
    ~ apart_point_and_line(A,orthogonal_through_point(L,A)) ).

%---Constructive uniqueness axiom for orthogonals
fof(ouo1,axiom,(
  ! [A,L,M,N] : %$ [point, line, line, line]
    ( distinct_lines(L,M)
      => ( apart_point_and_line(A,L)
          | apart_point_and_line(A,M)
          | unorthogonal_lines(L,N)
          | unorthogonal_lines(M,N) ) ) ).

```

Fig. 14: Orthogonality axioms in GE0006+3. ax

also as arrays of size 2 whose first and second elements are slope and y-intersect (i.e., a and b are first and second elements when the equation for lines is $y = ax + b$).

We write three scripts of parsing, interpretation, and the main testing just as we did for testing ALG axioms. The parsing script and the main testing script are pretty much the same as the scripts for ALG axioms. The interpretation script has much more contents since there are much more predicate and function symbols to interpret. We show some snippets of testing GEO006 axioms in Figure 15. We give Haskell interpretations to the predicate symbols like `unorthogonal_lines` and function symbols like `orthogonal_through_point`. We interpret the binary predicate `unorthogonal_lines` to be true when the product of slopes of two arguments is not equal (by the operator `=/=`) to 1, which is the obvious interpretation, and interpret the binary function `orthogonal_through_point` similarly. We also give Yices interpretation of symbols that appear in the premise part of the axioms such as `convergent_lines`. We interpret the binary predicate `convergent_lines` to be true when the slopes of the two arguments differ. Note that we need not give Yices interpretations for all the symbols, unlike for Haskell interpretations, since the testing framework only collects constraints from the premise part of the axioms.

After writing these three Haskell scripts, we can load the test script into GHCi, as in Figure 16. Notice that tests for axioms `oac1` and `ouo1` which consists of logical implication

```

pred2hsTable =
  [ ..., ("unorthogonal_lines", [| unorthogonal_lines |]) , ... ]
term2hsTable =
  [ ..., ("orthogonal_through_point", [| orthogonal_through_point |]), ... ]
pred2yicesTable =
  [ ...
    ,("convergent_lines", \margs-> APP (margs!!0) [LitI 0]
      := APP (margs!!1) [LitI 0])
  ]
term2yicesTable = [ ]
ty2tyTable = [ ("point", [| point |]), ("line", [| line |]) ]

type Point = Array Integer Double
type Line = Array Integer Double

point = listArray(0,1)[0.0..] :: Point -- point type as array of size 2
line = listArray(0,1)[0.0..] :: Line -- line type as array of size 2

slope l = l!0 -- slope is the first element (0th) of the array
yintercept l = l!1 -- yintercept is the second element (1th) of the array

unorthogonal_lines :: (Line, Line) -> Bool
unorthogonal_lines (l1, l2) = slope l1 * slope l2 /= 1

orthogonal_through_point :: (Line, Point) -> Line
orthogonal_through_point (l, p) = listArray(0,1)[m, b]
  where
    m = 1 / slope l
    b = y - m * x
    x = p!0
    y = p!1

geo006interp = Interp { predInterp = pred2hsTable
                       , termInterp = term2hsTable
                       , includeY = ["array.yices"]
                       , predInterpY = pred2yicesTable
                       , termInterpY = term2yicesTable
                       , typeInterp = ty2tyTable }

```

(a) Excerpt of GE0006Interp.hs – defining interpretation

```

#define INTERPRETATION geo006interp
#include "MACROS.h"

qc_occu1 = mQuickCheck $ DERIVE_PROP_WITH_GEN( (formula occu1) )
qc_oac1 = mQuickCheck $ DERIVE_PROP_WITH_GEN( (formula oac1) )
qc_ooc1 = mQuickCheck $ DERIVE_PROP_WITH_GEN( (formula ooc1) ) -- fails
qc_ooc2 = mQuickCheck $ DERIVE_PROP_WITH_GEN( (formula ooc2) ) -- fails
qc_ouo1_ = mQuickCheck $ DERIVE_PROP_WITH_GEN( (formula ouo1) )
genPointLine :: Gen (Point, Line)
genPointLine = liftM2 (,) genPoint genGoodLine
genPoint :: Gen Point
genPoint = liftM (listArray (0,1)) (vector 2)
genGoodLine :: Gen Line
genGoodLine = do a<-arbitrary
                 b<-arbitrary
                 return $ listArray (0,1) [if a==0 then 1 else a, b]
qc_ooc1' = quickCheck $ forAll genPointLine DERIVE_PROP( (formula ooc1) )
qc_ooc2' = quickCheck $ forAll genPointLine DERIVE_PROP( (formula ooc2) )

```

(b) Excerpt of GE0006Main.hs – the main test script

Fig. 15: Haskell scripts for testing median algebra axioms

prints more output than those axioms which do not have implications. Both the tests for `oac1` and `oao1` collect additional constraints from the premise, in addition to the constraint generated from type annotations, and, solves the constraint to generate non-trivial test cases that satisfy the premise. (So, the `non-trivial` outputs are good signs.) The test for axioms `ooc1` and `ooc2` fail and report a counterexample for each of them. By examining the counterexample array value, we can easily see that something goes wrong with zero values in arrays. Here, it is because our model has corner cases where some axioms break down: when the slope of a line is zero (i.e., when $a=0$ for the line $y = ax + b$), the slope value for orthogonal line to this line is infinity. Since the reported counterexamples are just Haskell values and interpretations of logical symbols are just Haskell functions, we can even run the function over the value to confirm what we think is wrong really goes wrong. After realizing the limits of the model, we may either adjust the model (or interpretation) to cover the corner cases or refine the axioms further. But, even before making adjustments or refinements, we can still write tests to examine whether the axiom make sense excluding that corner case we have just discovered. We derive the property only without deriving the generator using the macro function `DERIVE_PROP` instead of `GEN_PROP_WITH_GEN`. We can then supply a custom generator written manually using the `QuickCheck` library combinators. The generator we wrote is `genPointLine`, which generates pairs of an arbitrary point (generated by `genPoint`) and almost arbitrary line but disallowing slope of zero (generated by `genGoodLine`). Then, the new tests excluding the case where slope of the line is zero succeeds.

7 Logical equivalence and testable equivalence

We automatically derive testable properties from the axiom formulae, but this is possible since we restricted the possible formula we can handle. The unspoken assumption is that for those restricted forms of formulae we accept, translating them with certain rewriting rules that preserve logical equivalence does not affect the testable property, that is, the transformation preserves “testable equivalence”. However, this is not exactly true even within the category of formulae we accept. For example, consider the application of the translation rule for the (\Leftrightarrow) connective in the property derivation algorithm (Section 5.1), in combination with unrolling (Section 5.3.3) and flattening (Section 5.3.4). For the formula

$$![x] : ((![y] : P(x, y) \Rightarrow Q(x, y)) \Leftrightarrow R(x))$$

we can unroll $![y] : P(x, y) \Rightarrow Q(x, y)$ into

$$(P(x, 0) \Rightarrow Q(x, 0)) \wedge (P(x, 1) \Rightarrow Q(x, 1)) \wedge (P(x, 2) \Rightarrow Q(x, 2))$$

under some interpretation. Conceptually, we can think of the property derivation process as having two phases: the first transforms the formula into a logically equivalent¹⁸ formula suitable for deriving a property, and the second translates the transformed formula into a property. The problem is that the first phase might have multiple possible transformations, in contrast to the second phase, which is deterministic.

In our current implementation of the testing framework, we first transform this formula into the following logically equivalent formula:

$$![x] : (((![y] : P(x, y) \Rightarrow Q(x, y)) \Rightarrow R(x)) \wedge (R(x) \Rightarrow (![y] : P(x, y) \Rightarrow Q(x, y))))$$

¹⁸ To be more accurate, it is not just the logical equivalence since we also do transformation of unrolling which is only equivalent under the given interpretation.

```

*GEO006Main> -- Tests on the axiom without premise -----
*GEO006Main> qc_occu1
+++ OK, passed 100 tests.
*GEO006Main> -- Tests on axioms with premise -----
*GEO006Main> qc_oac1
(0 tests)
non-trivial case
(1 test)
non-trivial case
(2 tests)
...
non-trivial case
(99 tests)
non-trivial case
+++ OK, passed 100 tests.
*GEO006Main> qc_ouo1
(0 tests)
non-trivial case
(1 test)
non-trivial case
(2 tests)
...
non-trivial case
(99 tests)
non-trivial case
+++ OK, passed 100 tests.
*GEO006Main> -- Failing tests that reports counterexamples -----
*GEO006Main> qc_ooc1
*** Failed! Falsifiable (after 1 test):
(array (0,1) [(0,0.0),(1,0.0)],array (0,1) [(0,0.0),(1,0.0)])
*GEO006Main> qc_ooc2
*** Failed! Falsifiable (after 1 test):
(array (0,1) [(0,0.0),(1,0.0)],array (0,1) [(0,0.0),(1,0.0)])
*GEO006Main> -- Examine the counterexample -----
*GEO006Main> let p = array (0,1) [(0,0.0),(1,0.0)]
*GEO006Main> let l = array (0,1) [(0,0.0),(1,0.0)]
*GEO006Main> orthogonal_through_point(p,l)
array (0,1) [(0,Infinity),(1,NaN)]
*GEO006Main> -- Testing the axioms with manually written generators -----
*GEO006Main> qc_ooc1'
+++ OK, passed 100 tests.
*GEO006Main> qc_ooc2'
+++ OK, passed 100 tests.

```

Fig. 16: Running the tests for axioms in `GHC006+3.ax` in `GHCi`

Then, we unroll the inner quantified formula at negative position and flatten the other inner quantified formula at positive position as follows:

$$\begin{aligned}
 ![x, Y] : & (((P(x, 0) \Rightarrow Q(x, 0)) \wedge (P(x, 1) \Rightarrow Q(x, 1)) \wedge (P(x, 2) \Rightarrow Q(x, 2))) \Rightarrow R(x)) \\
 & \wedge (R(x) \Rightarrow (P(x, Y) \Rightarrow Q(x, Y)))
 \end{aligned}$$

Since the formula is flattened to level 1, we can then derive a testable property.

However, this is not the only possible transformation. Although our current implementation chooses to unroll only the inner quantified formula in negative positions, unrolling inner quantified formulas in positive positions is just as valid. Therefore we could unroll the

two formulas, in both positive and negative positions as follows:

$$\begin{aligned} ![\mathbf{x}] : & (((P(\mathbf{x}, 0) \Rightarrow Q(\mathbf{x}, 0)) \wedge (P(\mathbf{x}, 1) \Rightarrow Q(\mathbf{x}, 1)) \wedge (P(\mathbf{x}, 2) \Rightarrow Q(\mathbf{x}, 2))) \Rightarrow R(\mathbf{x})) \\ & \wedge (R(\mathbf{x}) \Rightarrow ((P(\mathbf{x}, 0) \Rightarrow Q(\mathbf{x}, 0)) \wedge (P(\mathbf{x}, 1) \Rightarrow Q(\mathbf{x}, 1)) \wedge (P(\mathbf{x}, 2) \Rightarrow Q(\mathbf{x}, 2)))) \end{aligned}$$

Alternatively, we could just unroll the inner formula in the original formula before rewriting \Leftrightarrow in terms of \Rightarrow :

$$![\mathbf{x}] : (((P(\mathbf{x}, 0) \Rightarrow Q(\mathbf{x}, 0)) \wedge (P(\mathbf{x}, 1) \Rightarrow Q(\mathbf{x}, 1)) \wedge (P(\mathbf{x}, 2) \Rightarrow Q(\mathbf{x}, 2))) \Leftrightarrow R(\mathbf{x}))$$

Since the formula is flattened to level 1, we can again derive a testable property. The property we derive here, though, is clearly different from that derived by the previous translation, i.e., the one used in our current implementation, since they take a different number of arguments – one for $[\mathbf{x}]$ and the other for $[X, Y]$. We cannot always say which one is better than another, in general, since it depends on the axiom author’s intention. The more complicated the forms of axiom that we accept (e.g., allowing existentials, inner quantification nested more than two levels, quantification inside negation) for testing, the more diverse the choices will be for translating formulas into the forms for which it is straightforward to derive testable properties.

8 Related work

The idea of evaluating propositions with respect to a computational interpretation goes back to early work of Green (1969) and Weyhrauch (1980). More recently, there has been some work on the use of testing to validate and debug logical conjectures. Claessen and Svensson (2008) use QuickCheck to test FOL conjectures arising in inductive proofs of protocol correctness. Propositions are interpreted as invariants on a particular state transition system. Generating test cases for invariants amounts to generating paths from a random initial state. To test inductive invariants they “adapt” an arbitrarily chosen (possibly non-reachable) state to the proposition-under-test, effectively giving a test data generator. Berghofer and Nipkow (2004) also use QuickCheck, to test theorems in Isabelle/HOL, particularly those involving inductive data types and inductive predicates. They create generators to generate data of arbitrary size for any inductive data type. In both these cases, the authors’ goals are to test conjectures in a logic, rather than the axioms of the underlying logic itself, given a computational model.

Carlier and Dubois (2008) have similar motivation and approach to ours, but in the setting of a typed functional language and a higher-order proof assistant. Since they mostly rely on random testing they generate and discard many test cases before they collect meaningful test cases. In contrast, we try to generate tests data efficiently by automatically synthesizing smart generators. Dybjer et al (2003) explore testing and proving in a dependent type setting, but do not automate the synthesis of test generators. We think dependent types can offer systematic information to help synthesize test generators for size dependent data, such as matrices, which depend on row and column sizes.

Planware (Blaine et al, 1998) is a system for the deductive synthesis of planning and scheduling software. In deductive synthesis, implementations are synthesized from specifications through a sequence of correctness-preserving refinements. Correctness of these steps ultimately rests on a logical axiomatization of the domain theory. In Planware, the axioms are validated (Becker and Smith, 2005) via a theory morphism, that is, by translation into

conjectures in another logic, in this case, set theory, where they are proven as theorems. The target theory thus serves as the intended interpretation.

Theory development in Isabelle (Paulson and Nipkow, 1994) also typically proceeds in such a “definitional” style, where more complex properties are built from a small set-theoretic core. However, we have not adopted this approach since we consider the domain-specific axioms (in contrast to the underlying laws of arithmetic and relational algebra) to be our starting point. These definitions and laws, typically coming from mission documents, are thus tantamount to requirements. Moreover, it would be a lot of work to derive them from first principles, and would provide little benefit to engineers.

The formulas which our system handles are a subset of the Bernays-Shoenfinkel-Ramsey (or effectively propositional, EPR) class of first-order formulas, which is known to be decidable; see Fontaine (2007); Pérez and Voronkov (2007); Piskac et al (2010). To the best of our knowledge, however, the techniques for deciding EPR formulae are not applicable in our setting. Here, we have interpreted function symbols, in particular, arithmetic functions. Although there do exist techniques to decide EPR formulae in the presence of interpreted function symbols, these can only be used under certain conditions: the domain and the range of the functions must be distinct. Even when these conditions are met, algorithmic complexity for the decision procedure may become intractable. Many of the axioms that arise in the engineering domain contain arithmetic functions, and the domain and range coincide for many such functions (e.g., mapping integers to integers, and reals to reals). Therefore, even though our system happens to be handling a subset of the EPR class, we do not think those techniques are directly applicable.

9 Conclusion and future work

We have described our approach to model-based testing of first-order logic axioms used by the verification tool AUTOCERT. We believe that our approach can help to systematically debug axioms, and also help maintain soundness of the logic while actively developing axioms. We have shown that it is quite feasible to derive counterexamples, even when the axioms are difficult to inspect. The computational model serves both as an interpretation against which the axioms can be tested, and as a reference which can be inspected by domain experts, since they remain significantly clearer than the axiomatization, particularly when we optimize the axioms to make the theorem provers search for proofs more efficiently. One clear conclusion we draw is the need for a *typed* logic to reduce unsoundness. Although types can be encoded in an untyped setting, we plan to investigate the recently proposed Typed First-Order Form (Claessen and Sutcliffe, 2009).

Previously, we had frequently run into either unsoundness or inconsistency, and sometimes this was not noticed until quite some time after the erroneous axioms had been added. Using the testing framework we have been able to find counterexamples for some axioms that had been previously known to be suspicious, as well as some previously unsuspected axioms. It also helped us avoid unsoundness arising from implicit but different models of the logic. Testing and proving are therefore complementary aspects to developing a formal verification.

An important aspect of testing is discovering corner cases of idealized models (e.g., overflow in fixed point arithmetic and round-off errors in floating point arithmetic). In our work, we used an arbitrary tolerance for round-off errors, but a more sophisticated notion, depending on input variables, is appropriate.

In terms of developing an infrastructure for the certification of safety-critical software, minimizing the trusted base is important. An important part of testing, and thus qualifying, the axioms will be to develop an appropriate notion of *coverage* (as in (Carlier and Dubois, 2008)), to give some measure of confidence that enough testing has been done. In the case of testing programs, coverage criteria are usually expressed in terms of branches and decisions taken by the software. For axioms, we also aim to cover all branches (that is, all independent ways of satisfying the hypotheses) as well as covering the domain (e.g., by considering all representatives of each frame of a DCM).

Lastly, we have also tested a number of axioms that involve physical units and equations. These axioms need to be modified in order to make them observable and therefore testable, but we believe that this can be done in a principled and systematic manner. We are also loosening the restrictions on formulas. To accept more formulas for testing, we would like to allow control over the two distinct phases of property generation discussed in Section 7. The first phase would be an interactive semi-automated process that aids users in selecting transformations while preserving logical equivalence to the original formula. In the second phase, the tool would derive testable properties from the narrower category of axiom formulae which have straightforward translations into properties.

One inconvenience we observed in our current framework is that the automation works in an all-or-nothing fashion. We simply reject the generation of smart generators when we cannot fully automate all the steps for the given axiom. Then the user needs to craft the smart generator manually from scratch for that particular axiom. Of course, we can always add better preprocessing algorithms to handle a wider class of formulae, and better SMT solvers with more decision procedures that can solve more complex constraints. However, there will always be formulae still outside the scope of full automation. One approach would be to provide users with code templates with holes to instantiate when the automation fails. The code template should contain automatically generated code that calls SMT solvers to partially solve the linear arithmetic fragment (or, any other theories the SMT solver supports) of the constraints.

One limitation of our current framework is that the unrolling and flattening algorithms cannot handle effectively existential constraints and deeply nested left implications (see Section 5 for details). Our goal in designing and implementing this axiom testing framework has been to come up with a tool that works for many axioms. Though it may not be intrinsically difficult to manually analyze and craft smart test generators for one axiom, this quickly becomes time consuming and tedious when there are many of them.

For future work, we aim to extend the framework in several directions:

- Support for existential quantification (at least with manually programmed existential value suppliers),
- Partial automation when the constraints are only partially solvable,
- Using the shrinking feature of QuickCheck to generate minimal counter-examples, and
- Using the coverage feature of QuickCheck to inspect and report which subformulae and terms have been covered, and how many times.

A long term goal is to extend the framework to test verification conditions (VCs) and functions. Since VCs are generally machine generated and more complex than the axioms, this would be possible after having extended our framework to a mature enough stage, including the improvements mentioned in the items listed above. As observed by Claessen and Svensson (Claessen and Svensson, 2008), we would like to know when a VC really is invalid, and when it is simply unprovable due to a missing axiom. When we extend the

framework to be able to also test VCs, then it can help us gain insight into when the problem lies in a missing axiom, rather than an invalid VC. Another related goal is to black-box test library functions which implement the concepts in the axioms, using the same mathematical specifications.

We also think there are some goals to be achieved together among the communities related to theorem proving, constraint solving, and automated testing:

- Promoting TFF support from ATP systems:
Although it is possible to track type information encoding typing judgments as predicates, it tends to discourage axiom authors to keep track of all the type specifications when using the untyped FOF sublanguage, since the axiom formulae become quite verbose. We believe the newly standardized TFF will greatly benefit axiom authors in the engineering domain as more APT systems start to support TFF, since it is a typed language and also supports standard interpretations for arithmetic.
- Syntax for annotating testing semantics of a logical formula:
We believe our constraint collection strategy of analyzing the shape of the formula does work for the majority of the axioms in practice. However, sometimes the automatically collected conditions may not match exactly with what the axiom author intended as the “hypothesis”. The TPTP language allows the specification of hypothesis for problems (or conjectures), but does not support specifying which fragment of an individual axiom formula can be considered as an hypothesis. We are trying to find better ways to annotate hypotheses in order to prove more fine-grained testing semantics (see Section 7) within a formula.

References

- Becker M, Smith DR (2005) Model validation in Planware. In: Verification and Validation of Model-Based Planning and Scheduling Systems (VVPS 2005), Monterey, California, USA
- Berghofer S, Nipkow T (2004) Random testing in Isabelle/HOL. In: 2nd IEEE International Conference on Software Engineering and Formal Methods (SEFM 2004), pp 230–239
- Blaine L, Gilham L, Liu J, Smith D, Westfold S (1998) Planware: Domain-specific synthesis of high-performance schedulers. In: The 13th IEEE International Conference on Automated Software Engineering (ASE '98), IEEE Computer Society, Honolulu, Hawaii, USA, pp 270–280
- Bradley AR, Manna Z, Sipma HB (2006) What’s decidable about arrays? In: Emerson EA, Namjoshi KS (eds) VMCAI, Springer, Lecture Notes in Computer Science, vol 3855, pp 427–442, URL http://dx.doi.org/10.1007/11609773_28
- Carlier M, Dubois C (2008) Functional testing in the Focal environment. In: Beckert B, Hähnle R (eds) The Second International Conference on Tests and Proofs (TAP 2008), Springer, Lecture Notes in Computer Science, vol 4966, pp 84–98, URL http://dx.doi.org/10.1007/978-3-540-79124-9_7
- Claessen K, Hughes J (2000) QuickCheck: a lightweight tool for random testing of Haskell programs. In: Proceedings of the ACM SIGPLAN International Conference on Functional Programming, pp 268–279
- Claessen K, Sutcliffe G (2009) A simple type system for FOF. <http://www.cs.miami.edu/~tptp/TPTP/Proposals/TypedFOF.html>
- Claessen K, Svensson H (2008) Finding counter examples in induction proofs. In: The Second International Conference on Tests and Proofs (TAP 2008), pp 48–65

-
- Denney E, Fischer B (2008) Generating customized verifiers for automatically generated code. In: Proceedings of the Conference on Generative Programming and Component Engineering (GPCE '08), ACM Press, Nashville, TN, pp 77–87
- Denney E, Trac S (2008) A software safety certification tool for automatically generated guidance, navigation and control code. In: IEEE Aerospace Conference
- Dutertre B, de Moura L (2006) The YICES SMT solver. Tool paper at <http://yices.cs1.sri.com/tool-paper.pdf>
- Dybjer P, Haiyan Q, Takeyama M (2003) Combining testing and proving in dependent type theory. In: 16th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2003), Springer, pp 188–203
- Fontaine P (2007) Combinations of theories and the bernays-schönfinkel-ramsey class. In: Beckert B (ed) VERIFY, CEUR-WS.org, CEUR Workshop Proceedings, vol 259, URL <http://ceur-ws.org/Vol-259/paper06.pdf>
- Green C (1969) The application of theorem proving to question-answering systems. PhD thesis, Stanford University
- Kuipers JB (1999) Quaternions and Rotation Sequences. Princeton University Press
- McCarthy J, Painter J (1967) Correctness of a compiler for arithmetic expressions. In: Schwartz JT (ed) Proceedings Symposium in Applied Mathematics, Vol. 19, Mathematical Aspects of Computer Science, American Mathematical Society, Providence, RI, pp 33–41
- Paulson L, Nipkow T (1994) Isabelle: A Generic Theorem Prover, Lecture Notes in Computer Science, vol 828. Springer-Verlag
- Pérez JAN, Voronkov A (2007) Encodings of problems in effectively propositional logic. In: Marques-Silva J, Sakallah KA (eds) SAT, Springer, Lecture Notes in Computer Science, vol 4501, p 3, URL http://dx.doi.org/10.1007/978-3-540-72788-0_2
- Piskac R, de Moura LM, Bjørner N (2010) Deciding effectively propositional logic using DPLL and substitution sets. *J Autom Reasoning* 44(4):401–424, URL <http://dx.doi.org/10.1007/s10817-009-9161-6>
- Sheard T, Peyton Jones S (2002) Template metaprogramming for Haskell. In: ACM SIGPLAN Haskell Workshop 02, ACM Press, pp 1–16
- Sutcliffe G (2000) System description: SystemOn TPTP. In: 17th International Conference on Automated Deduction (CADE 2000), Springer, Lecture Notes in Computer Science, vol 1831, pp 406–410
- Sutcliffe G (2009) The TPTP Problem Library and Associated Infrastructure: The FOF and CNF Parts, v3.5.0. *Journal of Automated Reasoning* 43(4):337–362
- Sutcliffe G, Denney E, Fischer B (2005) Practical proof checking for program certification. In: Proceedings of the CADE-20 Workshop on Empirically Successful Classical Automated Reasoning (ESCAR '05)
- Vallado DA (2001) Fundamentals of Astrodynamics and Applications, 2nd edn. Space Technology Library, Microcosm Press and Kluwer Academic Publishers
- Weyhrauch R (1980) Prolegomena to a theory of mechanized formal reasoning. *Artificial Intelligence* 13(1,2):133–170