Testing First-Order Logic Axioms in Program Verification

Ki Yung Ahn[1][2] and Ewen Denney[3]

[1] Portland State University, Portland, OR 97207-0751, USA
[2] * Mission Critical Technologies, Inc. / NASA Ames Research Center,
Moffett Field, CA 94035, USA
[3] Stinger Ghaffarian Technologies, Inc. / NASA Ames Research Center,
Moffett Field, CA 94035, USA
kya@cs.pdx.edu, Ewen.W.Denney@nasa.gov

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 nontrivial 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.

Key words: model-based testing, program verification, automated theorem proving, property-based testing, constraint solving

1 Introduction

1.1 Background

Program verification systems based on automated theorem provers rely on user-provided axioms in order to verify domain-specific properties of code. AutoCert [1] 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

* MCT/NASA Ames internship program

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) [2] 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 inferring 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 properties[4] – 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 [3]. 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.

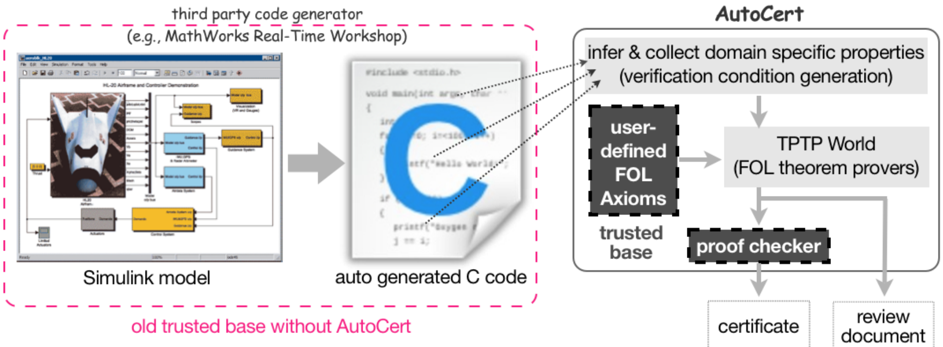

third party code generator
(e.g., MathWorks Real-Time Workshop)
Simulink model
auto generated C code
old trusted base without AutoCert
AutoCert
infer & collect domain specific properties
(verification condition generation)
user-defined FOL Axioms
TPTP World
(FOL theorem provers)
trusted base
proof checker
certificate
review document

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

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

However, 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

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 the 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 §2 we develop a sequence of simple axioms for rotation matrices, and use these to motivate this work by showing some of the pitfalls in developing axioms. Then §3 shows some examples of axioms used by AutoCert and the issues that arise in testing them. §4 describes the implementation and evaluation of our testing framework. We conclude with a discussion of related work (§5) and thoughts for future work (§6).

2 Axioms for Program Verification

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) [4, 5]. 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).

2.1 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.,

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

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`.

```
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 [2], 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.

2.2 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:

C code	Verification Condition

void init(float a[], float t)
{
 a[0]= cos(t); a[1]= sin(t);
 a[2]=-sin(t); a[3]= cos(t);
}
	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))).

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);
}
	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))).

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);
}
	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))).

void initX(float a[], float t)
{
 a[0]=-cos(t); a[1]= sin(t);
 a[2]=-sin(t); a[3]= cos(t);
}
	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))).

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

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

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

```
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.

2.3 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 conjectures.[5] 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 (§2.1) 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 (see §3.2).

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.

[5] 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 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.

3 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 §4.

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 $\lambda(x,y).\, x+y = y+x$ 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 $\lambda(x,y).\, x+y = x$ 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 chose 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 [6] and custom data generators (so-called "smart generators").

In the following subsections, we will give a high-level view of how we test the axioms with the example axioms from §2 and also some from AutoCert.

3.1 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]: (
      ( 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):

$$\begin{aligned}\lambda(x_1, y_1, x_2, y_2)\ .\ \neg(x_1 + 3y_1 = 3y_2 + x_2 \wedge\ 0 \le x_1 \le 2\ \wedge\ 0 \le y_1 \le 2 \\ \wedge\ 0 \le x_2 \le 2\ \wedge\ 0 \le y_2 \le 2) \\ \vee (x_1 = x_2\ \wedge\ y_1 = y_2)\end{aligned}$$

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 $(x_1 + 3y_1 = 3y_2 + x_2 \wedge 0 \le x_1 \le 2 \wedge 0 \le y_1 \le 2 \wedge 0 \le x_2 \le 2 \wedge 0 \le y_2 \le 2)$. 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.

3.2 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 §2.

```
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 `(leq(lo(A),I) & leq(I,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 4 with 100 randomly generated arrays of size 4 and find that it does indeed pass all tests.

Finally, the axiom `symm_joseph` in Figure 2 is intended to state that $\mathbf{A} + \mathbf{B}(\mathbf{C}\mathbf{D}\mathbf{C}^{\mathrm{T}} + \mathbf{E}\mathbf{F}\mathbf{E}^{\mathrm{T}})\mathbf{B}^{\mathrm{T}}$ 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 `B`, `C`, and `E` are all 3×3 matrices, we get counterexamples such as

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

We can immediately see that something is wrong since `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 `I` and `J`, and the test succeeds for all test inputs under the same assumption that $N = M = 3$ and `B`, `C`, and `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 `B`, `C`, and `E` are not explicitly constrained to make the matrix operations `mmul` and `madd` well defined.

4 Implementation

We have implemented a tool using Template Haskell [7], QuickCheck [8], and Yices [6], as illustrated in Figure 3. An axiom in TPTP syntax is parsed and automatically translated into a lambda term using Template Haskell. Using a metaprogramming language like Template Haskell has the advantage that

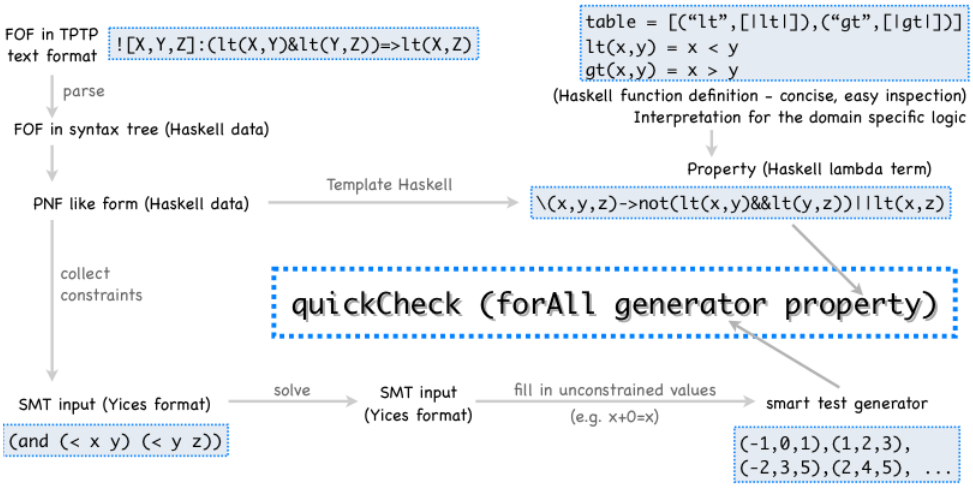
```
fof(symm_joseph, axiom,
 ! [I0, J0, I, J, A, B, C, D, E, F, N, M] : (
    ( leq(0,I0) & leq(I0,N) & leq(0,J0) & leq(J0,N)
    & leq(0, I) & leq(I, M) & leq(0, J) & leq(J, M)
    & select2D(D, I, J) = select2D(D, J, I)
    & select2D(A,I0,J0) = select2D(A,J0,I0)
    & select2D(F,I0,J0) = select2D(F,J0,I0) )
 =>
    select2D(madd(A,mmul(B,mmul(madd(mmul(C,mmul(D,trans(C))),
                                     mmul(E,mmul(F,trans(E)))),
                                trans(B)))),                  I0, J0)
  = select2D(madd(A,mmul(B,mmul(madd(mmul(C,mmul(D,trans(C))),
                                     mmul(E,mmul(F,trans(E)))),
                                trans(B)))),                  J0, I0) ) ).

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

we inherit the underlying type system for the interpreted terms. We generate a Haskell function rather than implement an evaluator for the logic. During the translation we transform the logical formula into a “PNF (prenex normal form) like” form moving universally quantified variables to the top level as much as possible.[6] For example, `![X]:(X=0 => (![Y]:(Y=X => Y=0)))` is translated into the logically equivalent `![X,Y]:(X=0 => (Y=X=>Y=0))`. We need to lift all the variables to the top level universal quantification to interpret the axioms as executable functions.

[6] The difference from PNF is that we avoid introducing existential quantification where possible.


FOF in TPTP text format
parse
FOF in syntax tree (Haskell data)
PNF like form (Haskell data)
Template Haskell
collect constraints
(Haskell function definition - concise, easy inspection)
Interpretation for the domain specific logic
Property (Haskell lambda term)
SMT input (Yices format)
solve
SMT input (Yices format)
fill in unconstrained values
(e.g. x+0=x)
smart test generator

```
![X,Y,Z]:(lt(X,Y)&lt(Y,Z))=>lt(X,Z)
```

```
table = [("lt",[|lt|]),("gt",[|gt|])]
lt(x,y) = x < y
gt(x,y) = x > y
```

```
\(x,y,z)->not(lt(x,y)&&lt(y,z))||lt(x,z)
```

```
quickCheck (forAll generator property)
```

```
(and (< x y) (< y z))
```

```
(-1,0,1),(1,2,3),
(-2,3,5),(2,4,5), ...
```

Fig. 3: Testing Framework

```
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) = (a!0) ===    cos t  && (a!1) === sin t
          && (a!2) === (- sin t) && (a!3) === 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. 4: Interpretation for the 2D rotation matrix axiomatization

Given a user-provided interpretation for the constants, the lambda term becomes an executable function which can then be used as a property in QuickCheck, a property-based testing framework for Haskell. The user-provided interpretation should be concise and easy to inspect so that it can serve as a reference model, suitable for inspection by domain experts. We believe that a declarative language like Haskell is suitable because of its conciseness. For example, part of the interpretation for testing the two-dimensional rotation matrix axioms discussed in §2 and §3 is shown in Figure 4. The tables `pred2hsInterpTable` and `term2hsInterpTable` interpret each predicate and term symbol as Haskell values. Here, `("cos",[|cos|])` relates the parsed TPTP symbol `"cos"` with the Haskell library function `cos` wrapped with a Template Haskell bracket. This allows a piece of syntax to be passed around as a meta-programming object and executed later (i.e., when we invoke QuickCheck) without conflicting with the Haskell type system.

The next step is to use a combination of QuickCheck library random generators and Yices to automatically synthesize test generators that generate inputs satisfying the premises of the axiom formula, thus avoiding vacuous tests. In the case where Yices cannot solve the constraints, we use our own smart generators with the help of combinator libraries in QuickCheck.

We sometimes need to patch or fill in unconstrained values that are missing from the results of Yices. For example, among the four variables in the axiom `update_last_in_range`, `X` and `Y` are unconstrained since they do not appear in the premise, so we randomly generate `X` and `Y` independently from Yices. Sometimes, there can be unconstrained variables even if they do appear in the premise because the constraints on those variables are trivial (e.g., solutions for x satisfying $x + 0 = x$).

Then, we can invoke QuickCheck over the property combined with the test generator. The basic idea is to call `quickCheck (forAll generator property)`, where `generator` generates the test data and `property` is the property to test. However, we make a few changes to this basic scheme, which we now discuss while showing how we invoke QuickCheck on some of the axioms in the Haskell interactive environment.

For the axioms on integers with linear constraints such as `uniq_rep_3by3` in §3.1, it is possible to fully automate the test. Recall that we only collect constraints from the premise (i.e., left-hand side of the implication). For example, we can run the test on `uniq_req_3by3` as follows.

```
> mQuickCheck( $(interpQints uniqr3by3) (genCtrs uniqr3by3) )

(0 tests)
non-trivial case
...
(99 tests)
non-trivial case
+++ OK, passed 100 tests.
```

The first change to the basic scheme is that `mQuickCheck` has a wrapper over the library function `quickCheck` which allows its argument to be of an IO monad type. This allows test generators to perform the side effect of communicating with the Yices process in order to solve the constraints. `interpQints` generates an interpretation for the axiom from its argument `uniqr3by3`, and `uniqr3by3` is the syntax tree for the axiom `uniq_req_3by3`. The `genCtrs` function generates constraints from the premises of the axiom.

Second, for the axioms on arrays and on types other than integers, we may need type annotations and other constraints to narrow the search space. For example, to test `update_commute_in_range` we call:

```
> mQuickCheck(
    $(interpQ updatecommr
       [| (listArray(0,3)[1.0..]::Array Integer Double,
           0::Integer, 0::Integer, 0.0::Double, 0.0::Double) |] )
    (ASSERT(Y.VarE "_I":>=LitI 0):ASSERT(Y.VarE "_J":<=LitI 3):
     ASSERT(Y.VarE "_J":>=LitI 0):ASSERT(Y.VarE "_I":<=LitI 3):
     (genCtrs updatecommr))


...
*** Failed! Falsifiable (after 4 tests):
(array (0,3) [...], 0, 0, 2.8288383471313097, 1.9408590255175935)
```

After a few tests it finds a counterexample such that both the index variables `I` and `J` are 0. The additional annotation is because of the dependencies between variables. The variables `I` and `J` in the axiom `update_commute_in_range` are constrained by the index range of `A`. This means that we can only generate useful test values of `I` and `J` after generating a test value for `A`. The type information including the array index range is specified in `[|...|]` and the `ASSERT`'s specify the constraints for the variables `I` and `J`. Currently we do not automatically infer these dependencies, but this could be done.

Lastly, when we test axioms involving floating point numbers, such as `symm_joseph`, we need to give some tolerance for errors. Otherwise, tests will fail for most of the mathematical properties (e.g., associativity of addition) we expect to hold on real number arithmetic. We define an overloaded comparison operator `(===)` in Haskell which compares integers with the usual equality operator `(==)` but compares floating point numbers with a predefined error tolerance.

4.1 Evaluation

In addition to testing the axioms from AutoCert's domain theory, we have also carried out some simple mutation testing on several of the axioms in order to simulate the most common errors. For example, we replace logical operators (e.g., conjunction with disjunction), change the polarity of premises (adding and removing negation), replace numeric indices (to give off-by-one errors), and switch variables and function symbols (e.g., `I` with `J`, `sin` with `cos`).

There are three possibilities: the premises are satisfiable and the mutated axiom is still valid, the premises become unsatisfiable and the axiom is vacuously true, or the mutant is invalid. We created a range of mutants for the axioms considered in this paper and, after filtering out the provable mutants we were, in each case, able to either derive a counterexample within 10 steps, or conclude that the mutated premises were vacuously true.

5 Related Work

The idea of evaluating propositions with respect to a computational interpretation goes back to early work of Green [9] and Weyhrauch [10]. More recently, there has been some work on the use of testing to validate and debug logical conjectures. Claessen and Svensson [11] 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 generator. Berghofer and Nipkow [12] 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 [13] 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. [14] explore testing and proving in a dependent type setting, but do not automate the synthesis of test generators.

Planware [15] 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 [16] 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 [17] 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.

6 Conclusion

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 [18].

Previously, we had frequently run into 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 [13]), 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).

We are currently extending the framework in two directions: testing verification conditions, and testing functions. As observed by Claessen and Svensson [11], we would like to know when a VC really is invalid, and when it is simply unprovable due to a missing axiom, say. We are extending the framework to be able to also test VCs, and thus provide insight into when the problem lies in a missing axiom, rather than an invalid VC. A related goal is to black-box test library functions which implement the concepts in the axioms, using the same mathematical specifications.

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 testable, but we believe that this can be done in a principled and systematic manner.

References

1. Denney, E., Trac, S.: A software safety certification tool for automatically generated guidance, navigation and control code. In: IEEE Aerospace Conference. (March 2008)
2. Sutcliffe, G.: System description: SystemOn TPTP. In: 17th International Conference on Automated Deduction (CADE 2000). Volume 1831 of Lecture Notes in Computer Science., Springer (2000) 406–410
3. Sutcliffe, G., Denney, E., Fischer, B.: Practical proof checking for program certification. In: Proceedings of the CADE-20 Workshop on Empirically Successful Classical Automated Reasoning (ESCAR '05). (July 2005)
4. Kuipers, J.B.: Quaternions and Rotation Sequences. Princeton University Press (1999)
5. Vallado, D.A.: Fundamentals of Astrodynamics and Applications. Second edn. Space Technology Library. Microcosm Press and Kluwer Academic Publishers (2001)
6. Dutertre, B., de Moura, L.: The YICES SMT solver. Tool paper at `http://yices.csl.sri.com/tool-paper.pdf` (2006)
7. Sheard, T., Peyton Jones, S.: Template metaprogramming for Haskell. In: ACM SIGPLAN Haskell Workshop 02, ACM Press (October 2002) 1–16
8. Claessen, K., Hughes, J.: QuickCheck: a lightweight tool for random testing of Haskell programs. In: Proceedings of the ACM SIGPLAN International Conference on Functional Programming. (September 2000) 268–279
9. Green, C.: The Application of Theorem Proving to Question-Answering Systems. PhD thesis, Stanford University (1969)
10. Weyhrauch, R.: Prolegomena to a theory of mechanized formal reasoning. Artificial Intelligence **13**(1,2) (1980) 133–170
11. Claessen, K., Svensson, H.: Finding counter examples in induction proofs. In: The Second International Conference on Tests and Proofs (TAP 2008). (2008) 48–65
12. Berghofer, S., Nipkow, T.: Random testing in Isabelle/HOL. In: 2nd IEEE International Conference on Software Engineering and Formal Methods (SEFM 2004). (2004) 230–239
13. Carlier, M., Dubois, C.: Functional testing in the Focal environment. In Beckert, B., Hähnle, R., eds.: The Second International Conference on Tests and Proofs (TAP 2008). Volume 4966 of Lecture Notes in Computer Science., Springer (9-11 April 2008) 84–98
14. Dybjer, P., Haiyan, Q., Takeyama, M.: Combining testing and proving in dependent type theory. In: 16th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2003), Springer (2003) 188–203
15. Blaine, L., Gilham, L., Liu, J., Smith, D., Westfold, S.: Planware: Domain-specific synthesis of high-performance schedulers. In: The 13th IEEE International Conference on Automated Software Engineering (ASE '98), Honolulu, Hawaii, USA, IEEE Computer Society (1998) 270–280
16. Becker, M., Smith, D.R.: Model validation in Planware. In: Verification and Validation of Model-Based Planning and Scheduling Systems (VVPS 2005), Monterey, California, USA (6–7 June 2005)
17. Paulson, L., Nipkow, T.: Isabelle: A Generic Theorem Prover. Volume 828 of Lecture Notes in Computer Science. Springer-Verlag (1994)
18. Claessen, K., Sutcliffe, G.: A simple type system for FOF. `http://www.cs.miami.edu/~tptp/TPTP/Proposals/TypedFOF.html` (2009)