Auditing User-Provided Axioms in Software Verification Conditions

Auditing User-Provided Axioms in Software Verification Conditions

Paul Jackson[1], Florian Schanda[2], and Angela Wallenburg[2]

[1] School of Informatics, University of Edinburgh, UK
pbj@inf.ed.ac.uk
[2] Altran, Bath, UK
{Florian.Schanda,Angela.Wallenburg}@altran.com

Abstract. A common approach to formally checking assertions inserted into a program is to first generate verification conditions, logical sentences that, if then proven, ensure the assertions are correct. Sometimes users provide axioms that get incorporated into verification conditions. Such axioms can capture aspects of the program's specification or can be hints to help automatic provers. There is always the danger of mistakes in these axioms. In the worst case these mistakes introduce inconsistencies and verification conditions become erroneously provable.
We discuss here our use of an SMT solver to investigate the quality of user-provided axioms, to check for inconsistencies in axioms and to verify expected relationships between axioms, for example.

1 Introduction

1.1 Use of Verification Conditions

One common approach to the formal verification of software involves the generation of verification conditions (VCs). VCs are typed first-order sentences that, if proven, assure the correctness of assertions that annotate programs. Often assertions check for the absence of exceptions, that arithmetic operations do not overflow or array indices are not out of bounds, for example. Sometimes assertions capture information from program specifications about intended behaviour. Typically one tries to discharge VCs using automatic theorem provers. SMT solvers such as Z3 [16], CVC4 [2] and Alt-Ergo [1] are popular choices. More rarely, particularly when the assertions are more complex, interactive semi-automated theorem provers such as PVS, Coq or Isabelle [17] are employed. Current examples of VC-based software verification tools and frameworks include Why3 [9], Boogie [6], Perfect Developer [13] and the SPARK tool-set [5]. These support verification of programs in (subsets of) such languages as C, C#, Java and Ada.

1.2 The Need for Axioms

It is often necessary for users of VC-based tools to write axioms which are added as assumptions to VCs. We describe below two classes of axioms: specification axioms and prover-hint axioms.

Specification axioms provide essential specification-related information. For example, such axioms can capture information about the environment a program is to operate in, information that might be difficult to capture in the preconditions of individual functions and procedures. Such axioms can also describe properties of constants, functions and relations that are introduced to help with program specification. For example, if verifying a sorting program, relations are needed to describe how the output is sorted and how the output is a permutation of the input.

Prover-hint axioms address incompletenesses in automatic provers, their failure to prove VCs that are logically valid. VCs frequently include quantified assumptions and can involve non-linear integer or real arithmetic. In general such VCs are intractable or undecidable. While automatic provers are continually improving, it is usually unrealistic to expect them to prove all VCs that might be generated for a program.

Sometimes VCs not proved automatically are verified by human inspection. Unfortunately this can be an exceptionally tedious and error prone process. An alternative to human inspection is the use of an interactive theorem prover such as PVS, Coq or Isabelle. Such systems rarely have soundness issues. However they have steep learning curves and place huge demands on the patience and mathematical sophistication of users.

Another approach is to use a combination of automatic proving, human inspection and axioms. The idea is to figure out why an automatic prover is failing to prove some given VC, and then to add one or more axioms that summarise logically-valid facts that the prover is missing and that are sufficient to enable the prover to complete the VC proof. Often these missing facts concern just a small part of the reasoning needed to justify the VC, so it is significantly simpler to manually check their correctness than to manually check the whole VC.

Sometimes, the automatic prover can be used to check the missing facts, even though the prover is not able to derive them or their equivalent when attempting the proof of the whole VC. Other times, it may be practical to use an interactive prover to check just these facts added as hints to the automatic prover.

In practice, this process of analysing failed automatic proofs is useful not only for identifying logically-valid prover-hint axioms, but also for discovering overlooked specification axioms.

Sometimes, because of the general nature of an axiom or because VCs are often similar, an axiom can turn out to be useful for multiple VCs.

One benefit of the user axiom approach to addressing VCs that fail to be automatically proved is that it potentially reduces the maintenance work needed when programs or program annotations change. If such VCs are manually checked, it is likely that they will need rechecking on every program or assertion change. If such VCs are discharged using an interactive theorem prover, the proofs might break and need fixing. However, commonly prover-hint axiom states a general truth that is independent of the particular context of the VCs it is needed in, so it remains valid as the program and program annotations evolve.

1.3 Problems with using Axioms

A major issue with adding axioms is the possibility of introducing an inconsistency. Sometimes software verification engineers will not be experienced in the use of formal logic and in how to phrase axioms appropriately. Also, needed axioms often have much detail that is tedious and awkward to work through, and mistakes are consequently easy.

Introducing an inconsistency into a VC is obviously a bad thing: it makes the VC trivially true. If the automatic prover detects the inconsistency, the VC and corresponding program assertion will be claimed true by the prover, irrespective of whether they are actually true or not.

Diagnosing why a VC is not proven and crafting appropriate axioms takes time. In our experience, while this process is sometimes fast, taking just a fraction of an hour, other times it can be 1–2 hours or sometimes even days. On a large verification project, there could be several 100s of unproven VCs and perhaps 100s of axioms needed. In such cases, the extra time and cost can be significant, and support for reducing this time could be very useful.

As remarked above, sometimes user-provided axioms remain valid as programs and annotations change, but other times they need reworking and rereviewing. For example, axioms might refer to specification constants or functions whose definitions change. So user-provided axioms can still present a significant maintenance burden to any SPARK development project.

1.4 Formal Support for Auditing Axioms

The core issue we explore in this paper is the use of an automatic theorem prover to investigate the quality of axiom sets. Most critically we are interested in identifying inconsistencies, but also we investigate relationships between axioms. If these relationships are not as expected, then there might be mistakes in the phrasing of axioms. For example, we might discover that some axiom is derivable from two others, when we previously thought it was independent of those axioms. This study of relationships could help axiom writers find errors in their axioms more quickly. It could both boost confidence in the correctness of axiom sets and reduce their development time. Additional aspects we discuss include the determination of minimal sets of axioms needed for proving VCs and the resolution of ambiguities in axiom definitions.

Our specific context for the work reported is the tool-set from Altran UK (formerly Praxis) and AdaCore for the formal verification of programs in the SPARK [3] subset of Ada [5]. This tool-set allows the user to associate a set of axioms with each section of code being verified. These axioms are then added as extra assumptions to each VC associated with the relevant section.

We have extended this tool-set with features for investigating the quality of user-provided axioms. We describe in this paper our experiences of using these

[3] The SPARK Programming Language is not sponsored by or affiliated with SPARC International Inc and is not based on the SPARC® architecture.

features to audit user-provided axioms employed in several case-study SPARK programs.

2 Related Work

Systems for verifying programs by proving VCs have been around since the 1960s. For example, King's PhD thesis [15] is the first description of such a system. In these early systems the theorem provers were poor at discharging VCs. Boyer and Moore, in the preface to their 1979 book on their NQTHM inductive theorem prover [10], take aim the dangers in the practice of assuming as axioms any simplified VCs that could not be automatically proved. They remark that often such axioms are seen as obvious facts when they are not obvious to many, and sometimes are even false.

Aspects of the approaches we investigate here have been implemented in both the Boogie-based VCC tool for verifying concurrent C [12] and the Why3 software verification framework [8, 9]. VCC has an option for enabling the generation of *smoke test* VCs. These VCs are phrased as the unreachability of control points in code. However, if proven for obviously reachable control points, they indicate inconsistencies in specifications and axioms. VCC's documentation[4] remarks that these tests are useful, especially early in the verification/development cycle. Why3 supports a *bisect* feature for finding minimal subsets of declarations that can prove a given VC. *Declarations* in Why3 not only include declarations of constants, functions and relations, but also include user-asserted axioms. The Why3 documentation does not comment on the provision of the feature. However, its very existence suggests that someone thought it might be useful.

Beckert, Bormer and Klebanov [7] review the uses of in-program annotations in VC-based program verification systems. Some of these annotations have a function similar to the user axioms we consider. They make a distinction between *requirement annotations* that capture specification information and *auxiliary annotations* that are needed to guide proofs. They further sub-divide the auxiliary annotations into those needed only for efficiency reasons, e.g. hints for quantifier instantiation and intermediate lemmas, and those that are logically essential, e.g. loop invariants. The prover-hint axioms we identify serve the same purpose as their first class of auxiliary annotations.

Ahn and Denney [3] use both an SMT solver and random testing to analyse axioms in a program verification system for verifying aerospace flight code. The approach tries to find false instances of axioms of form $\forall \boldsymbol{x}.\ A \Rightarrow B$. While A typically involves theories within the scope of an SMT solver (in their case Yices [14]), B often uses richer theories beyond the solver's capabilities, so they cannot use the SMT solver to check satisfiability of $\neg(A \Rightarrow B)$. They find that they can compute the truth value of ground instances of the axioms, so they exploit the Haskell-based QuickCheck library [11] to try a random search for falsifying instances. However, they observe this does not work well when random

[4] http://vcc.codeplex.com

instances of A are rarely true. In this case they first use the SMT solver to find satisfying instances of A and then use a random search of instantiations of the remaining variables in $\boldsymbol{x}$ to try to find an instantiation that makes B false and hence $A \Rightarrow B$ false.

A QuickCheck-based approach is also used in the Isabelle interactive theorem prover [17] for attempting to show conjectures false before users spend effort trying to prove the conjectures.

We have not yet ourselves investigated a testing approach for axiom auditing, but are considering it in future work.

Consistency checking is of importance in many other formal approaches to software and systems verification. For example, in model checking, it can be worth checking that at least some run satisfies the model and any environment assumptions, before going on to check temporal logic properties of runs: if there are no such runs, there is a problem with the combination of the model and the assumed environment.

3 Framework for Investigations

3.1 The SPARK language

SPARK is an Ada subset extended with an annotation language for expressing program specification information. It is designed for use in high-integrity applications. The Ada subset is tailored to make verification more straightforward. For example, it does not allow recursion, nor does it support heap-based data-structures. Several of the constraints of the SPARK language also correspond to common language constraints employed for embedded software in critical systems. For example, the no recursion and no heap restrictions ensure that a program's memory requirements are statically known.

SPARK has been used for high-integrity applications in sectors including aerospace, railways, automotive and nuclear. For example, it is currently being used in the development of the iFACTS support system for UK air traffic control, over 200K SLOC.

SPARK functions and procedures are collectively referred to as *subprograms*. Subprograms are grouped together into *packages*, and together subprograms and packages are examples of *program units*.

3.2 The SPARK formal verification tool-set

Virtually all developments in SPARK make use a formal verification tool-set developed by Altran UK. Traditionally the most-used tools are the *Examiner* which generates VCs from SPARK programs and the *Simplifier* automatic prover for discharging VCs. These VCs typically include *system axioms* which give definitions to standard constants and functions introduced by the VC generation process.

In addition, the user can provide *user axioms* which typically contain prover hints and information related to the specification of the particular program being

verified. The SPARK language has been recently extended to allow axiomatic information to be included in the bodies of program definitions and as annotations for specification functions declared in the SPARK program files. This improves the visibility of the axioms and helps software developers keep them synchronised with program changes. However, all user axioms considered in this paper were supplied in separate user-axiom files associated with each program unit.

3.3 The Victor VC translator and prover driver

Recently the SPARK tool-set incorporated a program *Victor*[5] developed by the first author which enables SMT solvers to be used as automatic provers for discharging VCs. Victor translates VCs in the FDL language output by the Examiner into API calls or standard languages accepted by SMT solvers. It also manages running the solvers and collecting results. Victor is most commonly used with the solvers Z3, CVC4 and Alt-Ergo. Of these, Z3 gives the best performance, and we report here only on experiments with Z3. Generally SMT solvers perform significantly better than the Simplifier. However Victor has only become a fully supported component of the tool-set in the past year, and many ongoing industrial users of the tool-set are still using the Simplifier as the primary tool for VC discharge.

Victor's translation typically involves introducing axiomatisations for various types such as the array, record and ordered enumeration types of the FDL language that do not have standard correspondences in the SMT solver input languages we use (SMTLIB 1.2 and 2.0). In this paper, we consider the axioms introduced by Victor as additional system axioms.

The prover driver component of Victor has been adapted to enable the exploration of the axiom auditing features introduced in Section 4 of this paper.

4 Axiom Analysis

4.1 Exploring Properties of Axioms

We describe here how we approach examining the consistency and inter-dependency of user-provided axioms.

A VC sentence generated by the SPARK tool-set has the general structure

$$S \wedge U \wedge H \Rightarrow C$$

where S, U, H and C are each an implicitly-conjoined set of closed formulas expressed in a typed first-order logic. More specifically:

- S is a set of system axioms,
- U is a set of user axioms,

[5] http://code.google.com/p/vct/

– H is a set of hypotheses,
– C is a set of conclusions.

A VC is considered to be valid when it is satisfied by all possible interpretations for uninterpreted functions, constants and types, and by standard interpretations for interpreted functions, constants and types. A standard example of interpreted constants, functions, and types is natural number literals, integer arithmetic operations and the integer type.

The SPARK tool-set groups VCs according to the program unit they are associated with. Across a set of VCs for given program unit, the Hs and Cs vary, but the Ss and Us are the same.

To explore issues of consistency and inter-relatedness between user-provided axioms, Victor generates special kinds of goals, and attempts proof of each using a designated automatic theorem prover. The kinds of goals are shown in Table 1. In the table, we assume that the set of user axioms for some given program unit

Kind	Goal shape	Description
S-incon	$S \Rightarrow \bot$	Are system axioms inconsistent?
U-incon	$S \wedge U \Rightarrow \bot$	Are user axioms inconsistent?
u-incon	$S \wedge u_i \Rightarrow \bot$	Is user axiom u_i inconsistent?
u-taut	$S \Rightarrow u_i$	Is user axiom u_i always true?
u-deriv	$S \wedge (U \setminus \{u_i\}) \Rightarrow u_i$	Does user axiom u_i follow from other user axioms?

Table 1. Kinds of Axiom Audit Goals

is $U = \{u_1, \ldots, u_n\}$, so, for that program unit, n goals of each of kinds u-incon, u-taut and u-deriv are generated, but only 1 goal of each of kinds S-incon and U-incon is generated. The symbol $\bot$ is for falsity. Note that no use is made of the hypotheses H and conclusions C of each original VC.

Validity of all goals is considered with the system axioms S assumed, i.e. validity is considered in the combination of theories described both by these axioms and built-in to the prover used. Examples of built-in theories are theories of integer and real arithmetic.

The S-incon goals are baseline checks. We expect inconsistencies in system-provided axioms very rarely and the main focus of our work has been to examine the user axioms. In other VC-based verification environments, the system axioms can be much richer than the system axioms we encounter, and in such cases it would definitely make sense to also audit the system axioms more thoroughly.

The U-incon goals directly look for some inconsistency involving one or more goals, whereas u-incon goals consider the consistency of each axiom on its own. The u-taut goals check whether axioms are tautologies. We hope that most axioms added as prover hints are tautologies. The u-deriv goals look at relationships between axioms, whether each axiom depends on the others. Generally u-deriv goals are only relevant if the corresponding u-incon and u-taut goals are unproven and there are no inconsistencies in the user-provided axioms.

It is useful to know not just whether an auditing goal is provable but also, if it is provable, which formulas are needed for the proof. Automatic provers can provide this information in various ways. Some, like the Simplifier prover provided with the SPARK tool-set, output a trace of their deductions, and this trace includes information on the formulas used. Several SMT solvers have facilities for outputing proof certificates. These certificates are independently checkable and provide evidence for the correctness of their deductions. The formulas used can be read off these deductions.

Another approach is to make use of a facility some SMT solvers have for generating *unsat cores*. When an SMT solver is used as a prover, the negation of the sentence to be proved is passed to the solver. This negation usually has the form of a conjunction. For example, for a VC sentence of the form shown above, it has form

$$S \wedge U \wedge H \wedge \neg C \qquad .$$

Here, as before, each set of formulas is implicitly conjoined. If the solver finds this negated sentence to be unsatisfiable, i.e. it deduces there are no models of the negated sentence, then the sentence itself is true in all models, i.e. it is valid. An unsat core is a (usually) minimal subset of the conjuncts of the conjunction that itself is unsatisfiable. From the point of view of provability of the original sentence, this subset is those formulas whose consideration is sufficient to show the sentence's validity. In our work we have experimented with generating and examining these unsat cores for provable auditing goals.

Interpretation of the results of these tests has to bear in mind the usual incompleteness of the used prover. Proved goals indicate validity of the goals, but if a goal is unproven, the goal might or might not be valid - we don't know. Failure for a goal to be proven is just a suggestion that it might not be valid.

4.2 Finding Minimal Sets of Axioms

It is useful to be able to identify minimal sets of user axioms needed for the proof of individual VCs or collections of VCs. Issues with axioms can be diagnosed if these sets are not as expected. And, once minimal sets are identified, unrequired axioms can be deleted in order to reduce the future axiom maintenance burden. Also, the fewer axioms there are, the more likely it is that axioms will be well written, compact and general, and the less likely it is that anything will need changing.

The minimal set for a VC can be smaller than the set that might be identified from a proof certificate or unsat core. The reason has to do with the fact that a user axiom is sometimes added to make proofs easier, but the VC is still valid without the axiom. In these cases a prover might be able to prove the VC without the axiom, but, if the axiom is present, it might use it because it provides a short-cut.

In our experiments, we worked with SPARK case studies where the user axioms had originally been added when the Simplifier SPARK tool-set prover

had been used. However, we checked the auditing goals using stronger SMT-solver-based provers, principally Z3. This made it all the more likely that we would encounter user-provided axioms that were unnecessary.

As each user axiom file is associated with a whole program unit, not an individual VC, each user axiom potentially could be used in the proof of multiple VCs. In order to identify unused axioms, we identify user axioms that do not feature in the minimal sets for any of the VCs for a program unit.

We adopt a simple approach to computing a minimal set of user axioms needed for proving a given VC. First we establish whether the original VC of form

$$S \wedge U \wedge H \Rightarrow C$$

is provable. If it is, we then in turn try removing each user axiom from U and see whether the resulting goal is provable: when it is, we leave the axiom out, when it is not, we add the axiom back in.

In general of course there might be multiple minimal sets, and the minimal set we find might not be a set of smallest size. However, we decided to start with just one minimal set to explore its potential usefulness.

4.3 Resolving Ambiguities in Axioms

A particular issue we ran into with user-provided axioms was that they were not always unambiguous. Typically user-provided axioms are of form

$$\forall x_1 : T_1, \ldots, x_n : T_n.\ P \quad ,$$

where $T_1, \ldots, T_n$ are the types of variables $x_1, \ldots, x_n$. However the concrete syntax for axioms permits these outermost universal typed quantifiers to be implicit, and the common practice to date by user-rule writers has been almost always to leave these quantifiers implicit. In some scenarios this is not a problem: one can always deduce the types of the quantified variables from the immediate context of their occurrences. However, with the concrete syntax for VCs, many operators are overloaded. For example, the same successor function is used for integer, real and enumeration types. With this overloading, variable types sometimes cannot be deduced from their immediate context. And further, sometimes the overloading cannot be resolved until types of free variables have been deduced.

We implemented an algorithm that combines operator overload resolution and variable type inference. This tries to make as much progress on both fronts, and warns the user when it does not complete. The expectation is then that the user goes in and adds sufficient explicit quantifiers to the user axioms that both overload resolution and type inference can complete.

All the experiments we report on here were run on axiom sets where we first had gone through the process of making sure all the axioms were unambiguous.

Resolving this ambiguity carefully is critical for soundness reasons. It is easy to construct examples where there are multiple ways of resolving operator overloading and variable typing, and some ways yield a sound axiom, other ways an unsound axiom.

The general lesson here for designers of languages for expressing logical formulas is that they should be very wary of adopting convenient notational ambiguities.

5 Experimental Results

5.1 General Setup

The evaluation we present here is based on our examination of recent SPARK program developments that we have access to and that make use of user axioms. We select two developments that provide interesting illustrations of the potential of rule auditing. For each development, we first resolved variable type and operator overloading ambiguities in the axioms as described in Section 4.3, before running the two kinds of analysis described in Sections 4.1 and 4.2.

We have found the Z3 SMT solver currently the best to use for proving VCs. It is generally faster than the competition and can almost always prove more VCs than the others. We ran Z3 both its default mode for checking satisfiability and in an unsat core generation mode. We used a development version of release 4.3.2 from March 2013 that included patches to fix a bug we encountered in the unsat core functionality of the 4.3.1 release

5.2 Case study 1: Tokeneer ID Station

Overview The Tokeneer ID Station (TIS) [4] was a research project commissioned by the US NSA (National Security Agency) to develop part of an existing secure system in accordance with a high-integrity development process advocated by Altran UK. One phase of the project involved developing and verifying SPARK code. This comprised about 10k lines of declarations and executable code, and 2k lines of SPARK proof annotations. All materials from this project are now publically available.

In the formal verification of the absence of runtime errors, 7001 verification conditions were generated and 107 user axioms were written. 40 of these user axioms were hints to the Simplifier prover of the SPARK tool-set. All these prover-hint axioms had outermost universal quantifiers surrounding quantifier-free formulas involving propositional variables, equalities, function symbols and integer arithmetic operators and constants. Some of the arithmetic involved integer division operators and non-linear multiplication. The truth of the axioms did not rely on the interpretation of any of the function symbols. It was clear from their form that all these prover-hint axioms were expected to be tautologies. The other 67 axioms concerned properties of SPARK subprograms and of specification functions and relations.

Inconsistency Checking Checks of u-incon goals with Z3 directly identified two inconsistent prover-hint user axioms. One of the inconsistent axioms, in the form in which it was written, is:

```
 B1 and Op = Op_1 -> B2
 may_be_deduced_from
[ St = St_1 or (St = St_2 or St = St_3),
  St_1 <> St_2,
  St_1 <> St_3,
  St_2 <> St_3,
  St = St_1 or St = St_2 -> B1 and (B3 and Op = Op_2),
  Op_1 <> Op_2,
  St = S_3 -> not B1 ].
```

In this SPARK syntax for axioms, `->` is implication, `may_be_deduced_from` is reverse implication, `<>` is disequality, formulas within the `[]` list are implicitly conjoined, and all the variables are implicitly universally quantified. The reason this axiom is inconsistent is that it contains a typo: in the last line the `S_3` should be `St_3`. As written, the types of the equalities and inequalities are ambiguous. Before we carried out the rule audit, we added extra type constraints to this axiom to ensure the ambiguities were resolved.

This inconsistency was not detected with the VC of kind U-incon where all 6 user axioms for the relevant program unit were included as hypotheses. We often observe that Z3's performance with problems involving quantified hypotheses is sensitive to the number and ordering of these hypotheses.

This inconsistency was only detected when we ran Z3 in its unsat-core generation mode. When we ran it in its normal mode this inconsistency was missed. When producing unsat cores, Z3 is inhibited from making certain preprocessing simplifications to input problems. In our case, the change happened to make a difference to the way Z3 explored the problem search space and helped it find the inconsistency.

The other inconsistency picked up by Z3 concerned an axiom expressing a property of the integer division operator. Again, in SPARK axiom syntax, we have:

```
X - (Y - 1) * 100 <= 200 -> Y + 1 = (X - 1) div 100 + 1
                  may_be_deduced_from
                [ 100 < X - (Y - 1) * 100,
                  goal(checktype(X, integer)),
                  goal(checktype(Y, integer)) ] .
```

Here, we see how one expresses constraints on the type of variables `X` and `Y`. Without the constraint on `Y`, it is unclear if `Y` is integer or real. To see the falsity, consider when `X = 0`, `Y = -1`, and note that integer division in the FDL language is real division rounded towards zero.

With the VCs of kind u-taut, Z3 demonstrated that 37 of the 40 prover-hint axioms were indeed tautologies. Two of the 3 remaining are those shown above. The other remaining was a false statement of non-linear arithmetic:

```
(B - 1) * X + A < Z may_be_deduced_from
[ A < X or B < Y,
  A <= X,
```

```
B <= Y,
X >= 0,
X * Y = Z ] .
```

To see the falsity of this, consider the axiom when `X = Z = A = B = 0`, `Y = 1`. While Z3 could not prove the u-incon check for this axiom, it could prove a variation of this check where all the system axioms were deleted. It is easy to see that these axioms were a significant distraction: there were over 300 system axioms, including over 100 with universal quantifiers.

These latter two axioms shown above are incorrect over-generalisations of two VC subgoals the Simplifier prover could not prove. For example, the axiom concerning division should have had a pre-condition that `X >= 1`.

The VCs the above axioms were intended as hints for are all valid. Indeed, Z3 is able to prove them all (without these inconsistent axioms) in 10s of milliseconds.

Axiom inter-relationships The u-deriv checks along with unsat cores revealed a number of relationships between the specification axioms. For example, among one set of axioms $A_1, \ldots A_9$ for a package program unit, we had

$$\begin{array}{l} A_2 \Rightarrow A_1, \\ A_1 \wedge A_7 \Rightarrow A_2, \\ A_4 \Rightarrow A_3, \\ A_3 \wedge A_8 \Rightarrow A_4, \\ A_6 \Rightarrow A_5, \\ A_5 \wedge A_9 \Rightarrow A_6, \\ A_2 \Rightarrow A_7, \\ A_4 \Rightarrow A_8, \\ A_6 \Rightarrow A_9. \end{array}$$

Here the relationships are in the context of the system axioms. These relationships can be more succinctly expressed as:

$$\begin{array}{l} A_2 \Leftrightarrow A_1 \wedge A_7, \\ A_4 \Leftrightarrow A_3 \wedge A_8, \\ A_6 \Leftrightarrow A_5 \wedge A_9. \end{array}$$

While one might think that axioms A_2, A_4 and A_6 would be sufficient hints, it appears that the Simplifier prover needed more of the axioms. For example, its proofs made use of both A_2 and A_1, though A_7 was unused.

Redundant Axioms The redundant axiom analysis identified 50 redundant axioms, including the 40 prover-hint axioms. The u-deriv checks indicated that 7 of the remaining 10 specification axioms were subsumed by other specification axioms.

5.3 Case study 2: Mixed floating-point and integer arithmetic

Overview This case study considered SPARK code of industrial origin that comprised a collection of around 30 functions and procedures for carrying out computations in a mix of floating point and integer arithmetic. Assertions and user axioms were added by the company developing the SPARK code as part of the company's evaluation of Altran's SPARK tool-set. Of particular interest to us was a collection of 25 or so specification axioms that concerned conversions from floating point numbers to integers.

In the SPARK tool-set, computations with floating-point numbers are reasoned about approximately by using the mathematical reals in VCs.

Inconsistency Checking The floating-point to integer conversions were characterised by floor and ceiling functions which rounded towards $-\infty$ or $+\infty$ respectively. For example, the ceiling axioms had form

$$c0 : \forall x : R.\ x \leq k - 1 \Rightarrow \text{ceil}(x) \leq x + 1$$
$$c1 : \forall x : R.\ x \leq k - 1 \Rightarrow \text{ceil}(x) \leq k$$
$$c2 : \forall x : R.\ x \leq k - 1 \Rightarrow x \leq \text{ceil}(x)$$
$$c3 : \forall x : R.\ x \leq k - 1 \Rightarrow -k \leq \text{ceil}(x)$$

Here k was an integer constant for the largest representable floating point number.

The U-incon check identified that axioms $c0$ and $c3$ were mutually contradictory: consider instantiating both with $-k - 2$. $c0$ then says that $\text{ceil}(-k - 2) \leq -k - 1$, but $c3$ says that $\text{ceil}(-k - 2) \geq -k$. However Z3 missed a very similar U-incon check for inconsistency between axioms for a floor function. Interestingly, a u-deriv check succeeded involving the 2 mutually-inconsistent floor axioms as hypotheses and an axiom involving a round to nearest function as conclusion, though the unsat core showed the conclusion playing no formal part in the truth of the check. Nevertheless, we suspect that the conclusion provided a hint to Z3 as to how to instantiate the hypotheses to obtain the contradiction.

Inter-relationship Checking This identified for example that $c0 \Rightarrow c1$, i.e. that $c1$ is redundant if $c0$ is retained in sorting out the inconsistency.

6 Discussion

There are different scenarios in which one could envisage undertaking axiom auditing. Firstly, after a SPARK development has reached some milestone, as part of a review process. Here the identification of inconsistencies would be of definite interest, as would unused axioms. However, it's not clear how much use inter-relationship information would be. Secondly, during a SPARK development, as user axioms are being written. Here it seems that the the axiom creator could

perhaps benefit more from the feedback provided by the inter-relationship analysis, as the feedback would help confirm and correct expectations about the axioms.

There are questions about how representative the examples are of issues from the previous section. The examples partly relied on the fact that the user axioms had originally been developed with provers with significantly weaker arithmetic capabilities than Z3. Still, non-linear arithmetic reasoning is a real challenge area for SMT solvers, and one could expect users in the near future will still need to provide some help with more complex non-linear VCs.

As noted, Z3 sometimes struggles and has unpredictable behaviour with handling the quantifiers in axioms. This is not just an issue with Z3, all SMT solvers have similar issues.

7 Conclusions and Future Work

We have discussed here several approaches we have investigated for checking the quality of user-provided axioms employed in the formal verification of SPARK Ada programs. The main approaches are to check for inconsistencies, tautologies and dependencies in axiom sets, and to derive minimal axioms sets. We also needed to resolve ambiguities in the axioms that were a consequence of the particular language they were expressed in.

We have shown the value of these approaches on two case study SPARK programs, most significantly finding inconsistent axioms in both cases.

We hope shortly to experiment with axiom auditing on much larger SPARK examples than those case studies we have considered to date. We are also looking for an opportunity to engage with SPARK verification engineers on a live project, to have them explore how axiom auditing might improve their confidence in the correctness of the axioms.

The results obtained with main approaches are dependent on the theorem proving power of the selected prover. We have had good results with the Z3 SMT solver, but also have seen its behaviour is not always consistent.

It is clear that large sets of system axioms can be a significant distraction to Z3, and we need to look at pruning these axiom sets to those of obvious relevance to the user axioms being investigated.

It would be interesting to investigate the use of other provers such as resolution-based automated theorem provers which have more mature technology for handling quantifiers, though their arithmetic support is not as strong as that of SMT solvers. We also will consider experimenting with a testing-based approach for attempting to falsify axioms, as described in Section 2. A relatively lightweight way of doing this would be to exploit a feature of Victor for generating VCs in the theory language of the Isabelle/HOL theorem prover [17], and using Isabelle's built-in QuickCheck procedures.

References

1. Alt-Ergo: an OCAML SMT solver for software verification, homepage at http://alt-ergo.lri.fr/
2. CVC3: an automatic theorem prover for Satisfiability Modulo Theories (SMT), homepage at http://cvc4.cs.nyu.edu
3. Ahn, K.Y., Denney, E.: A framework for testing first-order logic axioms in program verification. Software Quality Journal 21, 159–200 (2013)
4. Barnes, J., Chapman, R., Johnson, R., Widmaier, J., Cooper, D., Everett, B.: Engineering the Tokeneer enclave protection software. In: Secure Software Engineering, 1st International Symposium (ISSSE). IEEE (2006), http://www.adacore.com/sparkpro/tokeneer
5. Barnes, J., with Altran Praxis: SPARK: the proven approach to high Integrity Software. Altran Praxis (2012)
6. Barnett, M., Chang, B.Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented program. In: Formal Methods for Components and Objects: 4th International Symposium, FMCO 2005. Lecture Notes in Computer Science, vol. 4111, pp. 364–387. Springer (September 2006), visit http://research.microsoft.com/en-us/projects/boogie/ for current information on Boogie.
7. Beckert, B., Bormer, T., Klebanov, V.: On essential program annotations and completeness of verifying compilers. In: Filliâtre, J.C., Freitas, L. (eds.) Proceedings, Workshop on Verified Software: Theory, Tools, and Experiments (VSTTE) (2009)
8. Bobot, F., Filliâtre, J.C., Marché, C., , Melquiond, G., Paskevich, A.: The why3 platform, version 0.80. Tech. rep., University Paris-Sud, CNRS, Inria (October 2012), http://why3.lri.fr
9. Bobot, F., Filliâtre, J.C., Marché, C., Paskevich, A.: Why3: Shepherd your herd of provers. In: Leino, K.R.M., Moskal, M. (eds.) Boogie 2011: First International Workshop on Intermediate Verification Languages. pp. 53–64 (August 2011), http://proval.lri.fr/publications/boogie11final.pdf
10. Boyer, R.S., Moore, J.S.: A Computational Logic. Academic Press (1979), http://www.cs.utexas.edu/users/boyer/acl.pdf
11. 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. pp. 268–279 (2000)
12. Cohen, E., Dahlweid, M., Hillebrand, M.A., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S.: VCC: A practical system for verifying concurrent C. In: Theorem Proving in Higher Order Logics, 22nd International Conference. Lecture Notes in Computer Science, vol. 5674, pp. 23–42. Springer (2009)
13. Crocker, D., Carlton, J.: Verification of c programs using automated reasoning. In: Fifth IEEE International Conference on Software Engineering and Formal Methods (SEFM). pp. 7–14. IEEE Computer Society (2007)
14. Dutertre, B., de Moura, L.: The Yices SMT solver (August 2006), tool paper at http://yices.csl.sri.com/tool-paper.pdf
15. King, J.C.: A Program Verifier. Ph.D. thesis, Carnegie-Mellon University (1969)
16. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Tools and Algorithms for the Construction and Analysis of Systems, TACAS. LNCS, vol. 4963, pp. 337–340. Springer (2008)
17. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic, Lecture Notes in Computer Science, vol. 2283. Springer (2002), see http://www.cl.cam.ac.uk/research/hvg/Isabelle/ for current information