manuscript No.
(will be inserted by the editor)

Proving SPARK Verification Conditions with SMT Solvers

Paul B. Jackson · **Grant Olney Passmore**

Received: date / Accepted: date

Abstract We have constructed a tool for using SMT (SAT Modulo Theories) solvers to discharge verification conditions (VCs) from programs written in the SPARK language. The tool has API interfaces for some solvers and can drive any solver supporting the SMT-LIB standard input language.

SPARK is a subset of Ada used primarily in high-integrity systems in the aerospace, defence, rail and security industries. Formal verification of SPARK programs is supported by tools produced by the UK company Altran Praxis.

We report in this paper on our experience in proving SPARK VCs using the popular SMT solvers CVC3, Yices, Z3 and Simplify, and compare these solvers with Praxis's automatic prover. We find that the SMT solvers can prove virtually all the VCs that are discharged by Praxis's prover, and sometimes more. Average run-times of the fastest SMT solvers are observed to be roughly $1-2\times$ that of the Praxis prover.

Significant work is sometimes needed in translating VCs into a form suitable for input to the SMT solvers. A major contribution of the paper is a detailed presentation of the translations we implement. This is expected to be of interest to other users of SMT solvers.

Keywords SMT solver · SAT modulo theories solver · Ada · SPARK · theory interpretation · data-type refinement

1 Introduction

1.1 Overview

Software is deployed in an ever increasing range of applications where its safety is paramount, in aerospace, rail and road transport, and medical equipment, for example. The UK com-

Part of this article appeared in preliminary form in AFM '07 [25]

Paul B. Jackson
School of Informatics, University of Edinburgh, UK
E-mail: Paul.Jackson@ed.ac.uk

Grant Olney Passmore
School of Informatics, University of Edinburgh, UK
E-mail: g.o.passmore@sms.ed.ac.uk

pany Altran Praxis provides verification tools that give mathematically-rigorous assurances of the correctness of SPARK-Ada programs. Examples of major projects that Praxis deploy SPARK and their tools on include an upgrade to the UK civilian air-traffic control system and monitoring software for jet engines. This paper reports on work to improve the capabilities of Praxis's verification tools. Specifically the paper is concerned with how SMT solvers could augment or replace Praxis's in-house automatic prover technology. Praxis's customers currently exert significant effort to work around the limits of this technology. Improvements in prover technology could broaden the range of projects on which use of the Praxis verification tools is cost effective and deepen the formal analysis the tools provide.

We find that significant formal engineering is required to make use of SMT solvers, and much of the paper is devoted to a careful exposition of what we have implemented. We expect this exposition to be of significant interest to others who are wanting to use SMT solvers for software or system verification.

1.2 Softare Verification using Verification Conditions

There are a variety of techniques currently used for formal verification of software. These include software model checking [26] and abstract interpretation [13]. Many involve attaching assertions to positions in the procedures and functions of programs. These assertions are predicates on the program state that are desired to be true whenever the flow of control passes them.

Praxis use a verification technique that involves generatiing and proving of predicate logic formulas called *verification conditions* (*VCs* for short). For each assertion, one can analyse the surrounding program structure and generate a set of VCs that, if proven, guarantee that the assertion will always be satisfied when reached. Usually VCs for an assertion are generated under the assumption that immediately prior assertions on the control flow path were satisfied. While VCs use mathematical analogs of program data types such as arrays, records and enumerated types, they are otherwise free of program syntax. A consequence is that provers for VCs need no knowledge of the semantics of the programming language beyond these mathematical data types. All relevant semantic information on how programming language statements execute is captured in the VC generation process.

1.3 SMT Solvers

SMT (SAT Modulo Theories) solvers combine recent advances in techniques for solving propositional satisfiability (SAT) problems [42] with the ability to handle first-order theories using approaches derived from Nelson and Oppen's work on cooperating decision procedures [35]. The core solvers work on quantifier free problems, but many also can instantiate quantifiers using heuristics developed for the non-SAT-based prover Simplify [14]. Common theories that SMT solvers handle include linear arithmetic over the integers and rationals, equality, uninterpreted functions, and datatypes such as arrays, bitvectors and records. Such theories are common in VCs, so SMT solvers are well suited to automatically proving them. An SMT solver proves a VC by checking that a conjunction of the VC's hypotheses and the negation of the VC's conclusion is unsatisfiable.

The experiments we report on here use three popular SMT solvers: CVC3 [1], Yices [16] and Z3 [33]. All these solvers featured in recent annual SMT-COMP competitions comparing SMT solvers[1] in categories which included handling quantifier instantiation. We also include Simplify in our evaluation because it is highly regarded and, despite its age (the latest public release was in 2002), it is still competitive with current SMT solvers. Simplify was used in the popular ESC/Java VC-based software verification tool [20] and continues to be the default prover for use with the successor tool ESC/Java2 [2]. And we include Praxis's automatic prover, which is the usual tool that SPARK users employ to discharge verification conditions.

One advantage that SMT solvers have over Praxis's prover is their ability to produce counterexample witnesses to VCs that are not valid. These counterexamples can be of great help to SPARK program developers and verifiers: they can point out scenarios highlighting program bugs, or indicate what extra assertions such as loop invariants need to be provided. They also can reduce wasted time spent in attempting to interactively prove false VCs.

1.4 Targetting the SPARK language

Tackling SPARK programs rather than say Java or C programs is appealing for a couple of reasons. Firstly, there is a community of industrial SPARK users who have a need for strong assurances of program correctness and who are already writing formal specifications and using formal analysis tools. This community is a receptive audience for our work and we have already received strong encouragement from Praxis. Secondly, SPARK is semantically relatively simple and well defined. This eases the challenges of achieving higher levels of VC proof automation.

1.5 Contributions of Paper

This paper makes two contributions:

1. It gives a detailed presentation of the process of translating VCs into forms suitable for passing to the SMT solvers. While some of the translation steps by themselves are well known and straightforward, several, especially those relating to translating finite types, are less so. We see value in presenting the details of them, explaining options and subtleties, and how the steps interact. This presentation could act as a guide to others needing to construct similar translations for SMT solvers.
2. It investigates how current SMT solvers perform on industrially relevant examples.

1.6 Wider Context of Reported Work

The longer-term goals of the work reported here are to improve the level of automation of SPARK VC verification and to extend the range of properties that can be automatically verified.

Often there is a requirement that all VCs associated with a program are checked by some means. Typically 95–98% of VCs are proved automatically by Praxis's prover. A large project might have 10^5 VCs, so the remaining several thousand VCs must be justified by other means. Alternative approaches for checking these VCs include checking them by hand and using an interactive theorem prover provided as part of the Praxis toolset. Interactive

[1] `http://www.smtcomp.org/`

proofs are usually brittle, they often fail when VCs change slightly because of changes to code or to annotations. Another approach that has been found more robust is to add axioms that provide hints to the automatic prover for completing VC proofs. Obviously, care is needed to avoid inadvertently introducing inconsistencies. All these approaches are highly skilled and very time consuming. Increasing the level of automation reduces the cost of complete VC checking, and makes complete checking affordable by a wider range of SPARK users.

These concerns over the cost of handling non-automatically-proven VCs impact the range of program properties that SPARK users try to check. If users try to check richer properties, the number of non-automatically-proved VCs increases and so does verification cost. Most SPARK users settle for verifying little more than the absence of run-time exceptions caused by arithmetic overflow, divide by zero, or array bounds violations.

Cost concerns also place constraints on SPARK programming style. SPARK users learn programming idioms that lead to the generation of VCs that are more likely to be proved by Praxis's automatic prover.

1.7 Organisation of Paper

Section 2 compares our VC translation approach to that of other popular VC-based program verification systems. Section 3 gives more background on SPARK. Section 4 gives an overview of our VC translation tool. The translation is presented in detail in Sections 5 to 13. Readers interested in the experiments may choose to skip these sections. Case study programs are summarised in Section 14, and Sections 15, 16 and 17 present our experiments on the VCs from these programs. Current and future work is covered in Section 18, and conclusions are in Section 19.

2 Related Work

We discuss here several related strands of research. In Sections 2.1 and 2.2 we consider verification-condition-based program verification, both for imperative languages in general and for Ada in particular. Then we look more broadly at research that has dealt with similar translations. SMT solvers support a variety of input languages and some of these languages have many of the features found in the FDL VC language our translation takes as a starting point. We discuss these input languages in Section 2.3. This survey of SMT solver input languages also serves to motivate the translation efforts we have gone to. Also there are similarities between many interactive theorem prover languages and FDL, and there has been strong interest in developing interfaces between interactive theorem provers and SMT solvers. We survey some work in this area in Section 2.4.

Our translations are theory interpretations of mathematical logic. In Section 2.5 we explore this formal basis for our translations and also briefly discuss the closely-related topic of theory interpretations in algebraic specification.

2.1 VC-based program verification

Systems for verifying programs by proving VCs have been around since the 1960s. King's PhD thesis [27] is the first description of such a system. Notable systems since include the

Stanford Pascal Verifier [30] Gypsy [21] and ESC/Java [20]. Popular contemporary systems include Why verification platform [19], the Spec# static program verifier [7], and ESC/Java2 [2].

ESC/Java2 generates VCs for Java programs. The standard VC language is that of the Simplify prover, though experimental translations into the SMT-LIB format (see Section 2.3) and into the input language of the PVS theorem prover[2] are also available. While PVS has a rich type system, the PVS translation translates to an embedding of the Simplify language, and so makes relatively little use of these types.

Spec# is targetted at the C# language. Originally it generated VCs in the Simplify language. Currently it proves VCs using the Z3 prover, though it is not known whether it continues to use the Simplify language as the interface language.

The Why tool provides a VC generator for the Why intermediate-level programming language (Why PL) and can translate these VCs into the input languages of both SMT solvers and interactive theorem provers [19]. The associated Krakatoa tool translates annotated Java into Why PL, and Caduceus and its successor Frama-C translate annotated C into Why PL. The VC language is a simply-typed polymorphic language without sub-types.

Both ESC/Java2 and Spec# also translate into a simple intermediate-level abstract progamming language before generating VCs. In the case of Spec#, there is an alternate front end for C and an alternate VC generator that outputs in the input syntax of the Isabelle/HOL interactive theorem prover[3].

In all the above cases, extensive axiomatisations of the source language data types and memory models has been carried out by the time VCs are generated. In the case of the Simplify language, the only interpreted type left is the integers; in the case of Why, there is also a Boolean interpreted type, for example. Why has a feature for allowing additional types to be interpreted. As far as we understand, this feature is used mainly when translating for VCs in interactive theorem prover languages. Nearly all this axiomatisation appears to happen at stages before the intermediate-level programming language representations are generated.

In contrast, with the VCs generated for SPARK programs, mathematical analogs of most of the SPARK level data-types survive in the VCs. That this is possible is in part due to the simplicity of the SPARK data types, memory model and mode of passing data between procedures: with SPARK there are no reference types or pointer types, there is no dynamically allocated memory, and all data appears to be passed by value on procedure calls and returns. This richer VC language then gives us more work when translating to a relatively simple language like SMT-LIB, where the only interpreted type we might make use of is the integer type.

There are some similarities between our translation steps and those employed in ESC/Java, ESC/Java2, Spec# and the Why front-ends before intermediate language generation. For example, our step for abstracting term-level Boolean operations (see Section 10) are derived from those in ESC/Java2. There are also significant differences. For example, our understanding is that the translations in these other systems are more monolithic than ours: they are not broken down into a series of distinct steps. And we have not seen parts of the translations in these other systems having a direct analog to our data refinement step (see Section 9). In these other systems, any data refinement is directly built into the introduced axioms.

A common observation in descriptions of these axiomatisations is the need to carefully phrase the axiomatisations and to provide hints on when and how to instantiate quantifiers

[2] `http://pvs.csl.sri.com`

[3] `http://www.cl.cam.ac.uk/research/hvg/Isabelle/`

involved in the axiomatisations. This attention can much improve the performance of the quantifier instantiation heuristics built-in to SMT solvers which otherwise can be very poor. In the work described here, our experience so far has been that our axiomatisations are handled relatively well by SMT solvers However, we are aware that most of the VC examples we have tried do not thoroughly exercise our axiomatisations, so further experimentation is necessary.

2.2 Verification of Ada programs

The Ada language was originally designed for use in mission-critical real-time and embedded systems. Users of the language have a natural interest in the safety and correctness of their programs and have supported the development of formal verification systems targeted at subsets of Ada such as SPARK.

Earlier examples of systems include Penelope [23] from Odyssey Research Associates and SDVS (State Delta Verification System) [32] from Aerospace Corporation. Both made use of an automatic prover from Aerospace Corporporation that was similar to that used in the Stanford Pascal Verifier. This prover used the Nelson Oppen technique [35] for combining provers for such theories as bit-strings, arrays, uninterpreted functions and linear integer arithmetic.

The Compliance Tool [37] takes as input SPARK programs and specifications written in the Z specification language. It generates VCs in Z which are then discharged either interactively or automatically using the ProofPower theorem prover [4]. The Compliance Tool is used in conjunction with the ClawZ system [3] for generating Z specifications of Simulink models of avionics systems. The Compliance Tool enables checking that SPARK code correctly implements the Simulink models.

The Hi-Lite[4] project currently underway is modifying the GNU GNAT compiler for Ada so it can handle SPARK annotations and generate intermediate-level code in the Why program verification language (see Section 2.1).

See Section 3 for a description of the formal verification capabilities of the SPARK toolset from Praxis.

2.3 SMT solver front-end translations

Both Yices and CVC3 have rich native input languages, with many of the features found in FDL. These SMT solvers both support (linear) arithmetic over the integers and reals, arrays, records and subtypes. Minor differences are that CVC3 makes a strict distinction between formulas and Boolean-valued terms and that neither support the ordered enumeration types found in FDL. The details of how these systems handle types such as records, arrays and subtypes are not well documented in published documents. In both cases there appears to be some translation away of subtypes similar to that which we consider in this paper. We expect that both systems avoid introducing the non-trivial equivalence relations on types we need to consider in some circumstances, as they have more control over the types that are directly supported by their core reasoning engines. For example, both support Boolean-valued terms, while some of the translations we need to consider have to translate to languages without Boolean-valued terms. We have observed experimentally that CVC3 does not handle array

[4] `http://www.open-do.org/projects/hi-lite/`

extensionality at all, as we do, though Yices does. At the time of writing, we had asked the Yices developers about how they handle array extensionality, but have not heard back from them. There are several published papers on how to reason about subsets of the quantified theory of arrays (see [11], for example) and we conjecture both systems implement special purpose translations for arrays, more sophisticated than what we consider here.

The Z3 prover native input language is simpler than that of Yices or CVC3 in that it does not support sub-types, but does support arrays and records.

The SMT-LIB initiative[5] has been promoting a common input language and standard background theories for SMT solvers since 2003. This is to facilitate research and development in SMT techniques and support an annual competition SMT-COMP between SMT solvers. However, the standard background theories supported by SMT-LIB are considerably simpler than the range of types found in FDL. Our understanding is that the SMT-LIB architects chose to keep things simple in order to minimise the extra effort required of potential SMT-COMP participants to support SMT-LIB.

The SMT-LIB language distinguishes between formulas and terms. As the FDL language starting point of our translation does not, this is a distinction we need to introduce.

Background theories and restrictions on syntax (e.g. requiring that all arithmetic is linear or that there are no quantifiers) are grouped together into *sub-logics*. Developers of support for SMT-LIB choose to support certain of the sub-logics defined by SMT-LIB and a category of SMT-COMP is established for each of the sub-logics.

The sub-logics appropriate as a target from FDL include quantifiers, the theories of integer and real arithmetic, uninterpreted functions, and limited support for arrays. They do not include support for sub-types, record types or enumeration types. See Section 4.3 for further discussion of these sub-logics.

While it would be simpler for us to just support the native input languages of solvers such as Yices and CVC3, we have been keen to enable experimentation with as wide a range of solvers as possible, so we have gone to the extra effort of providing translations from FDL into appropriate SMT-LIB sub-logics. This has also enabled us to contribute VCs from the SPARK programs we examine to the SMT-LIB benchmarks collection. This collection is a valuable resource for all SMT solver developers and is used as a source of problems for SMT-COMP.

The Simplify input language just includes the type of integers. Because of the historical importance of Simplify and its continued competitive performance, we support a translation to its input language.

2.4 Interfaces between interactive theorem provers and SMT solvers

Developers and users of interactive theorem provers widely recognise the utility of the proof automation provided by SMT solvers.

The PVS interactive theorem prover links to the Yices solver, making use of Yices's native input language. Both Yices and PVS are developed within the same team at SRI, and, not suprisingly, the match between the languages is very good.

The 2011 release of the Isabelle/HOL prover[6] has interfaces to CVC3, Yices and Z3, and, independently, an interface *ismt* to Yices has been constructed [18]. HOL-Light has an interface [31] to CVC-LITE, a predecessor of CVC3, and HOL4 has an interface [41] to Z3.

[5] `http://www.smtlib.org/`

[6] `http://isabelle.in.tum.de/`

The HOL languages typically include recursive data-types, records, polymorphism, higher-order functions, and atomic types of reals, integers and Booleans. They do not support sub-typing directly: when sub-typing is needed, it is usually encoded into the term language in a similar way to that we describe in this paper. Different translations support all these types to varying degrees. Sometimes the translations are sound, but incomplete – axioms fully characterising some of the types and their associated operators are missing. The interfaces are both to native input languages and to SMT-LIB. In general, the translations to the native languages are more complete, as the work involved in creating the translation is less.

A common concern is handling polymorphic types: the translations typically handle these by introducing a distinct set of terms and axioms for each monomorphic instance of a polymorphic type. Our translation does something similar when handling FDL's array types.

A large concern of several of these interface projects is the trustworthiness of the SMT solver [31, 9]. Interactive theorem provers are typically engineered so that the correctness of all proofs relies on a small relatively-simple kernel of code. In contrast, SMT solvers have relatively-large code bases and employ highly-complex combinations of algorithms. These projects circumvent concerns about the correctness of SMT solvers by having the solvers output proofs that can be checked within the theorem prover or by some small independent proof checker tool.

Further examples of interfaces are the interface [12] between the Coq theorem prover and the Alt-Ergo SMT solver and the link [22] between Intel's Forte theorem prover and CVC-LITE.

A frustration in trying to analyse much of this work is the lack of proper documentation of what has been implemented.

2.5 Formal background for translations

Each of the translation steps we consider is formally described in mathematical logic as a *theory interpretation*. A sketch of the notion of a theory interpretation, appropriate for our purposes, is as follows. A *theory* consists of

- a signature which declares one or more type symbols, and uses these types in the specification of argument and value types as appropriate for some set of constant, function and relation symbols.
- a set of first-order-logic sentences over this signature,
- a subset of the set of all structures that model the sentences.

We allow the set of structures to be a subset of the set of all models of the theory sentences in order to permit some components of the signature to have fixed denotations and others to have their denotations unconstrained [7].

A theory interpretation is a map from some source theory to some target theory, where, in general, each element of the source signature is mapped to some type, term or formula built over the target signature. This mapping then induces a mapping that takes each sentence contructible over the source signature to some sentence over the target signature. An interpretation places some requirements on the relationship between the validity of sentences in the source theory and the validity of the mappings of these sentences in the target theory.

[7] Elsewhere in this paper, following common practice, we say that components with fixed denotations are *interpreted* and components with unconstrained denotations are *uninterpreted*. We avoid doing so in this section to avoid confusion with the primary subject of theory interpretations.

We will say more about this shortly. Usually interpretations describe how to map structures for the target signature to structures for the source signature.

The precise definition of a theory interpretation varies. A reasonable definition for our discussion here is that given by Hodges [24]. This treatment assumes a single type in both the source and destination signatures, though it does allow the image of the source type under the interpretation map to be a cartesian product of the target type, in general. It allows for the target theory having a predicate on the image of the source type that characterises which elements in this image are valid. It also allows for equality in the source theory to map to an equivalence relation in the target theory. Both these features arise in our treatment of type refinement in Section 9.

As a simple example, consider the interpretation of the theory of the rationals in the theory of the integers. The type of the rationals **Q** is mapped to the type of pairs of integers $\mathbf{Z} \times \mathbf{Z}$. The predicate restricts 2nd elements of these pairs to be non-zero, and equality is mapped to the equivalence relation on pairs

$$\langle a,b \rangle \equiv \langle c,d \rangle \quad \doteq \quad ad = bc \qquad .$$

Hodges discusses *admissibility conditions*—axioms introduced in the target theory to ensure equivalence relations are respected by functions and relations—which directly correspond to axioms we introduce in type refinement. Hodges names the map on sentences the *reduction* map and the map on structures the *co-ordinate* map. Hodges defines two properties concerning how an interpretation affects validity:

– *Left Totality:* The co-ordinate map maps every structure of the target theory to a structure of the source theory. This implies that if a sentence is valid in the source theory (true in all the structures associated with the source theory), its translation by the reduction map is valid in the target theory.
– *Right Totality:* For each source structure S there is a target structure that is mapped by the coordinate map to a structure isomorphic to S. This implies that if a translated sentence is valid in the target theory, the untranslated sentence is valid in the source theory.

Sometimes left-totality is built-in to the notion of a theory interpretation [17]. We do not do this, as we want to allow interpretations to weaker theories. For example, an interpreted function in the source theory might become uninterpreted in the target theory. We do always require our translations to be right-total in order for them to be sound: if an SMT solver establishes the validity of a translated VC, we want to know that the original VC is also valid.

The algebraic specification community has long formulated notions of theory interpretations for many-sorted theories (theories with many type constants), often for the purpose of modelling data-type refinement. For example, Blaine and Goldberg [8], following Turski and Maibaum [40] and drawing on the more abstract presentation of Sannella and Tarlecki [39], define theory interpretations that introduce quotient operations and *relativisation* predicates for restricting the target domain, much as we do. A primary interest is that facts that are true about a theory are preserved by interpretation maps, so interpretations in the algebraic specification literature are required to be left-total. While the algebraic specification literature considers some examples of data refinement, for example the refinement of finite sets by lists without duplicates, we have not been able to find presentations of the specific translations we consider here.

3 The SPARK Language and Toolset

The SPARK [5] subset of Ada was first defined in 1988 by Carré and Jennings at Southampton University and is currently supported by Praxis. The Ada subset was chosen to simplify verification: it excludes features such as dynamic heap-based data-structures that are hard to reason about automatically. SPARK adds to Ada a language of program annotations. These allow programmers to express assertions and attach them to flow-of-control points in programs. The program annotations take the form of Ada comments, so SPARK programs are compilable by standard Ada compilers.

SPARK inherits from Ada several less-common language features that build useful specification information into programs. This information then does not have to be explicitly included in program annotations. One can specify types that are subranges of integer, floating-point and enumeration types. For example, one can write:

```
subtype Index is Integer range 1 .. 10;
```

One can also define *modular* types which have values $0 \ldots n-1$ where n is some power of 2, and require all arithmetic on these values to be mod n. Modular types not only affect how Ada compilers treat arithmetic operations on those types, but also constrain integer values that can be injected into the types.

As with Ada, functions and procedures in SPARK are grouped into *packages*. A package can also contain other packages, so in general one has a hierarchy of packages. Packages always have two distinct parts, a *specification* and a *body* or implementation. Collectively, packages, functions and procedures are referred to as *program units*. Figure 1 shows a package definition containing a single procedure that does integer division by repeated subtraction [8]. The text `package P` introduces the specification of a package named `P`, and the text `package body P` introduces the definition of the body of package `P`. Lines starting with `--#` are SPARK annotations. Ada defines all text on a line after a `--` token as a comment, so these annotations are ignored by Ada compilers. The specification includes annotations for the precondition and postcondition of the `Divide` procedure. Preconditions and postconditions define assertions that are expected true at the start and end respectively of procedures and functions. The body also includes an assertion annotation that defines a loop invariant, a property true each time the start of the loop is reached.

The `derives` annotation concerns how output arguments are dependent on input arguments. Praxis's SPARK toolset checks `derives` annotations using an information flow analysis rather than generating and proving VCs.

The Examiner tool from Praxis generates VCs from SPARK programs. It is often very tedious for programmers to specify assertions using annotations, so, for common cases, the Examiner can add assertions automatically. For example, it can add type-safety side conditions for each expression and statement that check for the absence of run-time errors such as array index out of bounds, arithmetic overflow, violation of subtype constraints and division by zero.

The Examiner reads in files for the annotated source code of a program and writes the VCs for each program unit into 3 files:

- A *declarations* file declaring functions and constants and defining array, record and enumeration types,
- a *rule* file assigning values to constants and defining properties of data-types. For example, some properties axiomatically characterise functions mapping between enumeration types and sub-ranges of the integers.

[8] This example is drawn from the SPARK book [5]

```
package P is
   procedure Divide(M, N : in Integer;
                    Q, R : out Integer);
   --# derives Q, R from M,N;
   --# pre (M >= 0) and (N > 0);
   --# post (M = Q * N + R) and (R < N) and (R >= 0);
end P;

package body P is
   procedure Divide(M, N : in Integer;
                    Q, R : out Integer)
   is
   begin
      Q := 0;
      R := M;
      loop
         --# assert (M = Q * N + R) and (R >= 0);
         exit when R < N;
         Q := Q + 1;
         R := R - N;
      end loop;
   end Divide;
end P;
```

Fig. 1 A SPARK program for integer division

– a *verification condition goal* file containing a list of verification goals. A goal consists of a list of hypotheses and one or more conclusions. Conclusions are implicitly conjuncted rather than disjuncted as in some sequent calculi [28].

The language used in these files is known as FDL.

Figure 2 shows one of the 7 VC goals that the Examiner generates for the procedure shown in Figure 1. As the comment at the start of the goal indicates, this VC is for an execution path that starts and ends at the loop invariant assertion

```
assert (M = Q * N + R) and (R >= 0);
```

at the start of the main program loop. In other words, it is concerned with preservation of this loop invariant. Each label with prefix `H` introduces a hypothesis of the goal and each label with prefix `C` introduces a conclusion. As remarked above, the conclusions are implicitly conjoined, so each conclusion must be proved in order to prove the whole goal. Hypotheses `H1` and `H2` can be seen to come directly from the assertion of the loop invariant at the path start. Conclusions `C1` and `C2` are the weakest precondition [15] of the code in the loop body and the loop invariant assertion. The other hypotheses and conclusions are mostly concerned with machine bounds on the values of `Integer`-typed variables.

An excerpt of the accompanying declarations file is shown in Figure 3. Here, declarations are given of the constants and variables referred to in the goal. Semantically, constants and (free) variables in a goal are treated the same: both are implicitly universally quantified over. The difference is that FDL variables refer to values of program variables, whereas FDL constants have the same value in all program states.

An excerpt of the accompanying rule file is shown in Figure 4. Here we have definitions of the values of the constants in the goal: the `may_be_replaced_by` relation is logically the same as equality.

The VCs considered in our experiments often involve more first-order logic structure and a richer range of datatypes. An example VC is shown in abbreviated form in Figure 5.

```
For path(s) from assertion of line 17 to assertion of line 17:

procedure_divide_4.
H1:    m = q * n + r .
H2:    r >= 0 .
H3:    m >= integer__first .
H4:    m <= integer__last .
H5:    n >= integer__first .
H6:    n <= integer__last .
H7:    m >= 0 .
H8:    n > 0 .
H9:    r >= integer__first .
H10:   r <= integer__last .
H11:   not (r < n) .
H12:   q >= integer__first .
H13:   q <= integer__last .
H14:   q + 1 >= integer__first .
H15:   q + 1 <= integer__last .
H16:   r >= integer__first .
H17:   r <= integer__last .
H18:   r - n >= integer__first .
H19:   r - n <= integer__last .
        ->
C1:    m = (q + 1) * n + (r - n) .
C2:    r - n >= 0 .
C3:    m >= integer__first .
C4:    m <= integer__last .
C5:    n >= integer__first .
C6:    n <= integer__last .
C7:    m >= 0 .
C8:    n > 0 .
```

Fig. 2 Example VC goal from the integer division program in Figure 1

```
  const integer__size : integer = pending;
  const integer__last : integer = pending;
  const integer__first : integer = pending;
  var m : integer;
  var n : integer;
  var q : integer;
  var r : integer;
```

Fig. 3 Example declarations for the integer division program in Figure 1

```
divide_rules(4): integer__first may_be_replaced_by -2147483648.
divide_rules(5): integer__last may_be_replaced_by 2147483647.
divide_rules(6): integer__base__first may_be_replaced_by -2147483648.
divide_rules(7): integer__base__last may_be_replaced_by 2147483647.
```

Fig. 4 Example rules for the integer division program in Figure 1

This includes instances of operators on records (the field selectors `fld_msg_count` and `fld_initial`) and arrays (the 1 dimensional array element select function `element(_, [_])`), arithmetic operators and relations, and quantifiers (`for_all`).

The Simplifier tool from Praxis can automatically prove many verification goals. It is called the *Simplifier* because it returns simplified goals in cases when it cannot fully prove the goals generated by the Examiner. Users can then resort to an interactive proof tool to

```
...
H3:    subaddress_idx <= lru_subaddress_index__last .
...
H6:    for_all(i___2: word_index,
           ((i___2 >= word_index__first) and (
             i___2 <= word_index__last))
            -> (...))
...
H11:   fld_msg_count(element(bc_to_rt, [dest])) >=
           lru_subaddress_index__first .
...
H29:   fld_initial(element(bc_to_rt, [dest])) <=
           lru_start_index__last .
        ->
C1:    fld_initial(element(bc_to_rt, [dest])) + (
           subaddress_idx - 1) >= valid_msg_index__first .
C2:    fld_initial(element(bc_to_rt, [dest])) + (
           subaddress_idx - 1) <= valid_msg_index__last .
C3:    subaddress_idx - 1 >= all_msg_index__base__first .
C4:    subaddress_idx - 1 <= all_msg_index__base__last .
```

Fig. 5 An example VC involving an explicit quantifier and several datatypes

try to prove these remaining simplified goals. In practice, this proof tool requires rather specialised skills and is used much less frequently than the Simplifier.

The Simplifier has been in development since at least far back as 1997 and drew on earlier code for an interactive proof checker. Praxis continues to improve it. It employs a number of heuristics involving applying predicate logic rules, rewriting, forward and backward chaining, and applying special purpose arithmetic rules. However it does not incorporate decision procedures for linear arithmetic or propositional reasoning, for example.

As of 2009, Praxis's SPARK toolkit is freely available under a GNU Public Licence [9]. This release includes both source code and user-level documentation for the Examiner and the Simplifier.

4 Architecture of the VC Translator and Prover Driver

4.1 Overview

Our VCT (VC Translator) tool reads in the VC file triples output by the Praxis VC generator tool, suitably translates the VCs for a selected prover, at present one of CVC3, Yices, Z3 or Simplify, and runs the prover on each VC goal. Fig. 6 provides an overview of the architecture. The tool is divided into three parts:

1. A *preprocessor* which parses the VC files and puts VCs into a standard internal form, resolving various features particular to the FDL language.
2. A *translator* which performs a variety of optional translation steps on the VCs in order to prepare them for the different provers.
3. A *driver* which translates to the concrete syntax or syntax tree data structures required by the provers, orchestrates invocations of the provers, and logs results.

[9] `http://libre.adacore.com/libre/`

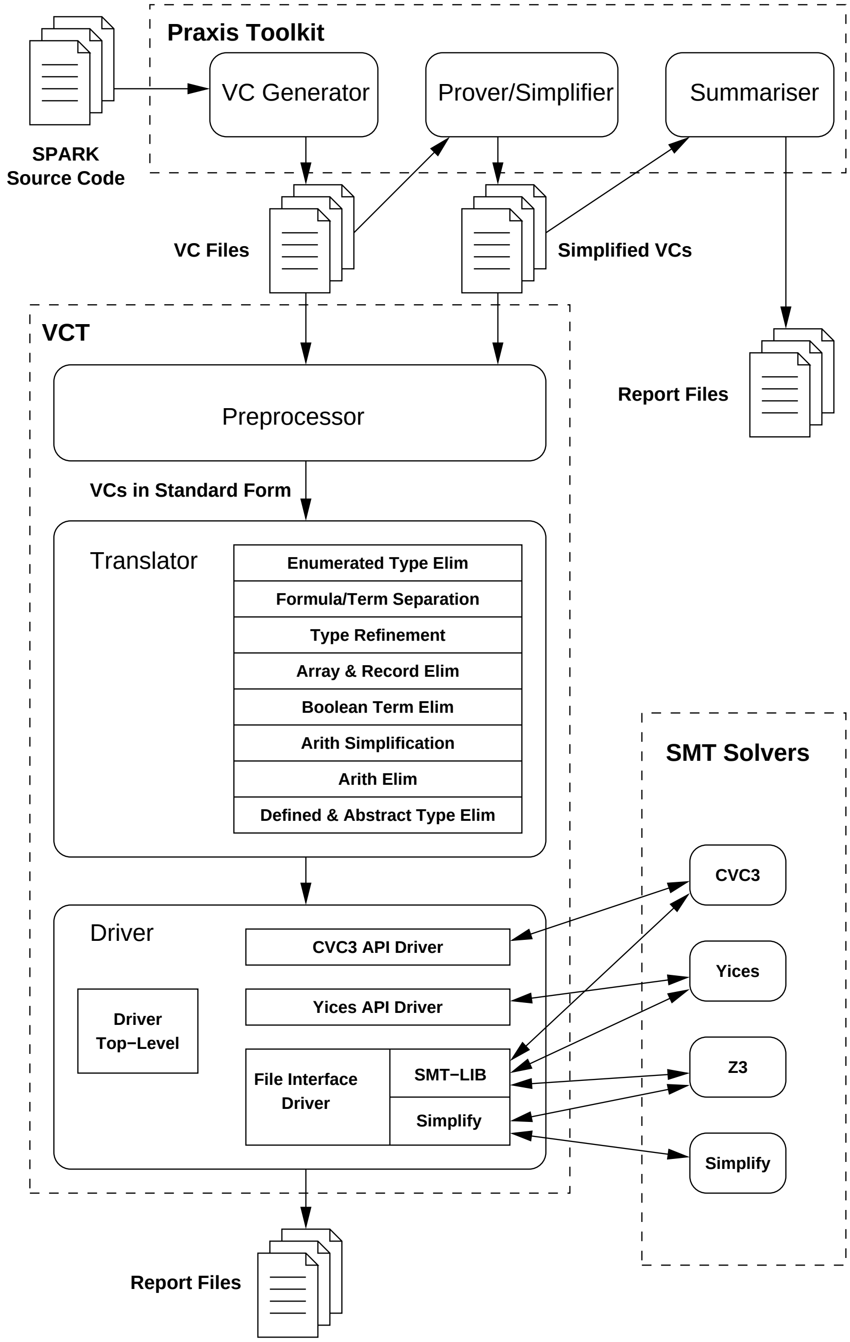

Fig. 6 Architecture of our VC translator and Prover Driver

These parts are described in more detail in the following subsections. We consider the preprocessor first, and then the driver before the translator, as the driver description motivates the discussion of the translation. A final subsection describes how VCs are represented in the translator.

Currently our tool consists of around 20,000 lines of C++ code, including comments and blank lines.

4.2 Preprocessor

Operations carried out by the preprocessor include:

- *Eliminating special rule syntax*: FDL rules give hints as to how they could be used. For example, an equality:

  ```
  integer__first = -2147483648
  ```

  that defines a value for a constant is expressed as:

  ```
  integer__first may_be_replaced_by -2147483648
  ```

  This special syntax was eliminated, as none of the provers we considered had any way of handling it.
- *Typing free variables in rules, closing rules*: FDL rules have untyped free variables, implicitly universally quantified. The preprocessor infers types for these variables from their contexts and adds explicit quantifiers to close the rules.
- *Adding missing declarations of constants*: FDL has some built-in assumptions about the definition of constants, for the lowest and highest values in integer and real subrange types, for example.
- *Reordering type declarations*: Most solver input languages require types to be declared before use, but such an ordering is not required in FDL.
- *Resolving polymorphism and overloading*: For instance, FDL uses the same symbols for the order relations and successor functions on integer and enumerated types, for arithmetic on the integers and reals, and for operations on arrays with different element and index types. After resolution, each function and relation has a definite concrete type.

4.3 Driver

There are various alternatives for interfacing with SMT solvers. We have experimented with several of these, partly out of necessity, partly to understand their pros and cons. The alternatives we have explored so far are as follows:

- SMT-LIB *file-level interface*
  The SMT-LIB standard input language for SMT solvers was introduced in Section 2.3. We translate into the SMT-LIB sub-logics:
  - AUFLIA: Closed linear formulas over the theory of integer arrays with free sort, function and predicate symbols,
  - AUFNIRA: Closed formulas with free function and predicate symbols over a theory of arrays with integer indices and real elements.

  In each case we just use the support for integer arithmetic: with the AUFLIA sub-logic the support is for linear integer arithmetic, with the AUFNIRA sub-logic the support is for possibly non-linear integer arithmetic. We do not currently make use of the support for arrays. This support is rather limited: our current translation requires any support for arrays to be over a range of index and element types. Extra translation work would be needed to make do with just the available index and element types. While we have a need for a theory of reals, the AUFNIRA sub-logic suprisingly provides no proper support for mixed integer real arithmetic: for example it is missing a function injecting the integers into the reals.

Currently, CVC3 and Z3 support AUFNIRA, and CVC3, Yices and Z3 and support AUFLIA.

Our SMT-LIB file-level interface writes SMT-LIB format files in either AUFLIA or AUFNIRA, runs an appropriate prover in a sub-process, and reads back the results.

The SMT-LIB standard treats formulas and terms as distinct syntactic categories, following the tradition in virtually all text-book presentations of first-order logic. This is in contrast to the case with the FDL language where formulas are just terms of Boolean type.

– *Simplify file-level interface*

  This interface uses the language of the Simplify prover. This language is essentially single-sorted. All functions and relations have integer argument sorts, and all functions have integer result sorts. Formulas are in a distinct syntactic category from terms. The Simplify language is accepted by Simplify itself and by Z3.

  Our Simplify interface shares much of its code with the SMT-LIB file-level interface.

– CVC*3 API interface*

  CVC3 supports a rich native input language. We translate FDL arrays, records, integers and reals directly to the corresponding types in this input language. We use CVC3's integer subrange type to realise translations for enumerated types.

  CVC3 requires a strict distinction between formulas and terms of Boolean type. Boolean terms are translated to the 1-bit bit-vector type.

  The interface uses functions in CVC3's C++ API to build the term and formula expressions .

– *Yices API interface*

  Yices's native input language is similar to CVC3's. The main difference is that Yices's language does not distinguish between formulas and terms of Boolean type.

We define a single API that is shared by all of the above interfaces. This API includes functions for initialising solvers, asserting formulas, calling solvers, and checking results. Our top-level driver module works above this API, sequencing the API function calls and performing other tasks such as collecting timing information and writing report files. The top-level module writes both to a log file and a comma-separated-value file where it records summaries of each solver invocation. This allows easy comparison between results from runs with different options and solvers.

4.4 Translator

Each translation step performed by the translator operates on sets of VCs in a standard internal form. The information held in this internal form is described in Section 4.5. An overview of the main translation steps is as follows. For each step we give references to later parts of this paper that cover the step in more detail.

– *Enumerated Type Elimination*

  This involves replacing uses of enumerated types with integer subrange types, and providing alternate definitions for functions and relations associated with enumerated types.

We use this step with all our driver options. It is for sure needed when translating to SMT-LIB or Simplify format, as neither supports enumerated types. Yices and CVC3 do support enumerated types via their APIs, but these types do not come with an order defined on them, and do not define successor and predecessor functions, as needed by FDL. We could introduce an order relation and the successor and predecessor functions axiomatically, but currently do not do so.
See Section 5 for details.

– *Formula/Term Separation*
When we need distinct syntactic categories of formulas and terms, we establish both term-level and formula-level versions of the propositional logic operators. As needed, we suitably resolve every occurrence of a function involving Boolean arguments or return value to be either at the formula or the term level. This sometimes results in a Boolean-valued term where a formula is expected, or vice versa, and, as necessary, we add in special operators that convert between Boolean terms and formulas. See Section 8 for details.
– *Type Refinement*
Type refinement carries out refinement translations that have the flavour of data-type refinements considered in the program refinement literature. When a type is refined, it is considered as a subtype of some new base type, and allowances are made for equality on the unrefined type possibly not corresponding to equality on the base type. Special treatment is given to arrays and records to allow arrays and records over base types to be used to model arrays and records over the original unrefined types.
The primary use of type refinement is to eliminate finite types such as integer subrange types and the Boolean type. These types are not supported by the SMT-LIB and Simplify input file formats.
See Section 9 for details.
– *Array and Record Elimination*
We can eliminate redundant array and record operators and can axiomatically characterise array and record types. An example of a redundant operator is a record constructor. This is redundant if a default record constant and record field update operators are available. The axiomatic characterisations are useful when the targetted solver or solver format does not provide explicit support for arrays and records. For example, we use axiomatic characterisations when translating for the SMT-LIB and Simplify formats.
See Section 6 for details on array elimination, and Section 7 for details on record elimination.
– *Boolean Term Elimination*
Term-level Boolean operations can be made uninterpreted and axioms can be introduced that express that the operations have the same behaviour as their formula-level counterparts. Also the Boolean type itself along with the true and false Boolean constants can be made uninterpreted. These steps are required by the SMT-LIB and Simplify formats.
See Section 10 for details.
– *Arithmetic Simplification*
We simplify arithmetic expressions that are semantically linear into expressions that are obviously syntactically linear. This improves what we can prove with Yices which rejects non-linear arithmetic expressions, and improves the quality of the VCs we can generate in linear SMT-LIB formats.
See Section 11 for details.
– *Arithmetic Elimination*
Options are provided for making uninterpreted various arithmetic operators that some

provers cannot handle, integer division and modulus, for example. In some cases, we add axioms that partially or fully characterise the behaviour of the operators. See Section 12 for details.

– *Defined Type and Abstract Type Elimination*
  To cope with the Simplify prover we need to eliminate all uninterpreted types and defined types. See Section 13 for details.

In Table 1 we summarise the steps that are used to at least some extent by each of the drivers.

Translation step	Yices API	CVC3 API	SMTLIB	Simplify
Enumerated Type Elimination	•	•	•	•
Formula/Term Separation		•	•	•
Type Refinement			•	•
Array & Record Elimination			•	•
Boolean Term Elimination			•	•
Arith Simplification	•	•	•	•
Arith Elimination	•	•	•	•
Defined & Abstract Type Elimination				•

Table 1 Translation steps used by the different prover drivers

The usual order of applying the steps is as they are listed above.

There are some dependencies between steps, so not all orderings are sensible. For example, Type Refinement has some special treatment for term-level Booleans, so it must come after they are introduced by Formula/Term Separation. Boolean Term Elimination is designed to come after Type Refinement.

Some ordering alternatives yield different translations. For example, the Array and Record Elimination is shown after Type Refinement, but it also could be positioned before, in which case the axioms introduced would be different at the end of the translation. See Section 17.2 for a discussion of some preliminary results on the effect of ordering on prover run-times. In other cases, the ordering is unimportant. For example, the arithmetic steps could be carried out at any stage with no change to the final result.

4.5 Standard Internal Representation for VCs

Each step of translation works on a *Verification Condition Unit* or *VC Unit*, for short. A VC Unit gathers all the VCs associated with a SPARK program unit (usually a procedure or a function) into a standard internal data-structure. The implementations of the translation steps then share a common set of utility functions for operating on this data-structure. The information in a VC unit is derived by the Preprocessor code from one of the 3-file sets output by the Praxis's VC generator as described in Section 3. In addition, each VC Unit extends this information about a particular set of VCs with information about the theory these VCs are over. This is helpful in tracking how the translation steps change the background theory of the VCs and in checking that translations have been correctly chosen and sequenced. Our notion of a VC Unit is a concrete realisation of the abstract notion of theory introduced in Section 2.5.

The elements making up a VC Unit include:

- *Identification of logic variant used*
  The variants are
  - *Strict First Order Logic* (*Strict FOL*) where formulas are a distinct syntactic category from terms.
  - *Quasi First Order Logic* (*Quasi FOL*) where formulas are terms of Boolean type.

  For simplicity, we present the rest of the VC Unit elements for the Strict FOL variant. The changes for Quasi FOL are straightforward. For example, relation declarations are not distinct from function declarations—they are just declarations of functions of Boolean result type.
- *Type-constant declarations and definitions*
  This introduces the set of type constant names that can be used in type expressions. It includes constants for both interpreted types such as the reals and integers, and uninterpreted types. We write $C : \mathsf{Type}$ to declare that C is a type-constant, and $C : \mathsf{Type} = T$ to define type constant C as a definition for type expression T.
  For convenience, we assume that there are sufficient type definitions that every type in a formula and every argument type to a type constructor on the right-hand side of a type definition can be a declared or defined type. We do not allow type constructors at such positions. A similar condition is enforced in the SPARK subset of Ada and the FDL VC language.
- *Type constructors*
  Each *type constructor* constructs a new type from 0 or more existing types and possibly other information. Examples include: enumerated types, array types, record types and integer subrange types.
  Taken together, the type-constants and the type constructors generate the language of types.
  All the type constructors we consider have intended interpretations, usually parameterised by the interpretations of their components.
- *Term signature*
  This declares constants and functions. We write $c : T$ to declare that constant c has type T and $f : (S_1, \ldots, S_n)T$ to declare that function f has argument types $S_1, \ldots, S_n$ and result type T. We keep track of whether each has some intended interpretation, and, if so, what that interpretation is.
  We assume that there is no overloading or polymorphism: every constant or function has a unique type. To enforce this condition, we create monomorphic instances of naturally polymorphic operators in FDL, such as the functions for updating and accessing array elements. We structure the constant and function identifiers such that polymorphic base names are easily extractable. This is needed when handing off VC goals to SMT solvers that expect some polymorphic operators.
  The term signature along with typed variables generates the language of terms.
  We optionally allow into the language of terms *if-then-else* operators of form $\mathsf{ITE}_T(\phi, a, b)$, where ϕ is a formula and a and b are terms of type T. $\mathsf{ITE}_T(\phi, a, b)$ is equal to a when ϕ is true, and b when ϕ is false.
- *Relation signature*
  The relation signature declares atomic relations. We write $R : (S_1, \ldots, S_n)$ to declare that relation r has argument types $S_1, \ldots, S_n$. As with the term signature, we track any intended interpretations and assume all relations are monomorphic. In particular, we create a monomorphic instance of equality $=_T$ for each type T we need to express equality at.

The relation signature, together with usual propositional logic operators ($\wedge, \vee, \neg, \Leftrightarrow, \Rightarrow$) and typed existential and universal quantifiers ($\forall x : T.\ \phi$ and $\exists x : T.\ \phi$), generate the languagee of formulas.

– *Intended interpretations*
We keep track of whether each declared type constant, term constant, function and relation has an intended interpretation or is uninterpreted. If an entity is interpreted, then we also keep track of the nature of that interpretation. Usually the interpretations are the expected ones: the type Int is interpreted as the integers. Occasionally, the interpretations are not the ones immediately suggested by the entity names. For example, we sometimes interpret the type Bool as the integers when the prover language being translated to does not have any type containing just two-elements. This is the case with the input language of the Simplify prover and the SMT-LIB sub-logics we use.
– *Rules*
Rules are formulas. Commonly they introduce equality-based definitions of term constants and function constants, and, more generally, provide axiomatic characterisations of types and associated terms. It is expected that rules are always satisfiable.
– *Goals*
A goal is composed of a list of hypothesis formulas and a list of conclusion formulas. The logical sense of a goal is that the conjunction of the hypotheses implies the conjunction of the conclusions. A goal is considered valid if it true in all interpretations satisfying the rules and giving interpreted types, constants, functions and relations their intended interpretations.

The next sections of this paper give details on how each of the translation steps introduced in Section transforms a VC Unit.

5 Enumerated Type Elimination

5.1 Enumerated Types in FDL

A named enumerated type E containing constants $k_0, \ldots, k_{n-1}$ is introduced with the type definition

$$E : \mathsf{Type} \;=\; \{k_0, \ldots, k_{n-1}\} \quad .$$

Associated with E are operators

$$\begin{aligned} \mathsf{pos}_E &: (E)\mathsf{Int} \\ \mathsf{val}_E &: (\mathsf{Int})E \\ \mathsf{succ}_E &: (E)E \\ \mathsf{pred}_E &: (E)E \end{aligned}$$

and relations

$$\begin{aligned} \leq_E &: (E,E)\mathsf{Bool} \\ <_E &: (E,E)\mathsf{Bool} \quad . \end{aligned}$$

We usually write the relations using infix notation.

These functions and relations are not primitive in FDL: instead they are uninterpreted and are characterised by axioms. The pos_E and val_E functions define an isomorphism between the type S and the integer subrange $\{0, \ldots, n-1\}$ such that $\mathsf{pos}_E(k_i) = i$. The succ_E and pred_E functions are successor and predecessor functions. For example, $\mathsf{succ}_E(k_i) = k_{i+1}$ when $i < n-1$. The axioms leave $\mathsf{succ}_E(k_{n-1})$ and $\mathsf{pred}_E(k_0)$ unconstrained.

5.2 Elimination by Translation to Integer Subranges

We change the type definition to

$$E : \mathsf{Type} \;=\; \{0 \mathinner{..} n-1\} \quad ,$$

so the typename E is just a name for the integer subrange type $\{0 \mathinner{..} n-1\}$. We declare the enumerated type constants as uninterpreted constants, and add axioms

$$\begin{aligned} k_0 &= 0 \\ &\vdots \\ k_{n-1} &= n-1 \quad . \end{aligned}$$

We remove all original axioms characterising the enumerated type operators and relations, replacing them with the axioms

$$\begin{aligned} &\forall x : E.\, \mathsf{val}_E(x) =_{\mathsf{Int}} x \\ &\forall x : E.\, \mathsf{pos}_E(x) =_{\mathsf{Int}} x \\ &\forall x : E.\, x < n-1 \Rightarrow \mathsf{succ}_E(x) =_{\mathsf{Int}} x+1 \\ &\forall x : E.\, 0 < x \Rightarrow \mathsf{pred}_E(x) =_{\mathsf{Int}} x-1 \quad . \end{aligned}$$

We replace all occurrences of the $\leq_E$ and $<_E$ relations in rules and goals with the integer relations $\leq$ and $<$.

When we use integer subrange types, it is not the case that argument types of functions and relations always match expected types exactly. In general type checking which such subrange types can involve arbitary non-linear arithmetic reasoning. In practice so far we have found we can type check VC Units using just syntactic checks. Typechecking currently just uses the conventional integer typing for $+$ and $-$, the knowledge that E is a subtype of Int, and the typing $\{k \mathinner{..} k\}$ for integer literal k.

6 Array Elimination

6.1 Arrays in FDL

The SPARK FDL language has primitive n dimensional arrays. A type definition of form

$$A : \mathsf{Type} \;=\; \mathsf{Array}(S_1, \ldots, S_n, T)$$

introduces an n dimensional array named A with S_i the ith index type and T is the type of elements. The index types are usually integers, integer subranges or enumeration types. The element type can be any type.

For simplicity, we consider here the 1 dimensional case

$$A : \mathsf{Type} \;=\; \mathsf{Array}(S, T) \quad .$$

The generalisation to n dimensional arrays is straightforward.

Associated with the array type A are

– *Array constructors* of form

$$\mathsf{mk\_array}_A(t_0,\ [s_1] := t_1, \ldots,\ [s_k] := t_k) \quad \text{for } k \geq 0$$

or of form

$$\mathsf{mk\_array}_A([s_1] := t_1, \ldots,\ [s_k] := t_k) \quad \text{for } k > 0 \quad .$$

These constructors make an array with t_i at index s_i. With the first form a default value t_0 is provided. With the second, the assumption is that the values at all indices are explicitly set. Here we use extra syntactic sugar to improve readability. Without this sugar the function names would need further decoration so the constructors of different arities have different names. FDL also allows for assigning a value to a range of indices. The latest versions of our tool provides support for this, but we do not describe our support in this paper.
– A *select* function $\mathsf{select}_A(a,s)$ for selecting the element of array a at index s. The *select* function is sometimes known as an *array read* function.
– An *update* function $\mathsf{update}_A(a,s,t)$ for updating the element of array a at index s to new value t. The *update* function is sometimes known as an *array write* function.

6.2 Eliminating array constructors

We introduce a constant and operator

$$\begin{aligned} \mathsf{default}_A &: A \\ \mathsf{const}_A &: (T)A \end{aligned}$$

with a characterising axiom

$$\forall t : T.\ \forall s : S.\ \mathsf{select}_A(\mathsf{const}_A(t), s) \ =_T \ t \quad .$$

The constructor $\mathsf{mk\_array}_A(t_0, [s_1] := t_1, \ldots, [s_k] := t_k)$ is replaced by the term a_k, recursively defined by

$$\begin{aligned} a_0 &= \mathsf{const}_A(t_0) \\ a_i &= \mathsf{update}_A(a_{i-1}, s_i, t_i) \qquad \text{for } 0 < i \leq k \quad . \end{aligned}$$

The constructor $\mathsf{mk\_array}_A([s_1] := t_1, \ldots, [s_k] := t_k)$ is replaced by the term a_k, recursively defined by

$$\begin{aligned} a_0 &= \mathsf{default}_A \\ a_i &= \mathsf{update}_A(a_{i-1}, s_i, t_i) \qquad \text{for } 0 < i \leq k \quad . \end{aligned}$$

6.3 Eliminating interpreted arrays

We eliminate the need to have a standard interpretation for array type A and functions select_A and update_A by introducing suitable axioms. Assume we have the $\mathsf{default}_A$ and const_A constant and function introduced above in Section 6.2. The axioms are

$$\forall a : A.\ \forall s : S.\ \forall t : T.\ \mathsf{select}_A(\mathsf{update}_A(a,s,t), s) =_T t$$

$$\forall a : A.\ \forall s, s' : S.\ \forall t : T.\ s \neq_S s' \ \Rightarrow \ \mathsf{select}_A(\mathsf{update}_A(a,s,t), s') =_T \mathsf{select}_A(a, s')$$

$$\forall a, a' : A.\ (\forall s : S.\ \mathsf{select}_A(a,s) =_T \mathsf{select}_A(a',s)) \quad \Rightarrow \quad a =_A a'$$

The first two of the axioms are often called *read-write* axioms. The first axiom describes how, if we write value t at index s in array a and then read from the same index, we get back value t. The second describes how, if we read at index s' after writing to a distinct index s, we get the same result as if we had performed the read before the write. The third is a statement of *array extensionality*: it states that two arrays should be considered equal when they contain the same elements. The extensionality axiom could also be stated with $\Leftrightarrow$ rather than $\Rightarrow$. We choose the form with $\Rightarrow$, as the axiom for $\Leftarrow$ is just a trivial statement that select_A respects equality in its first argument. All provers have built-in knowledge of this. With these axioms, we drop the type definition $A : \mathsf{Type} \;=\; \mathsf{Array}(S,T)$, but retain a type declaration for A, so A is now an uninterpreted type.

Some provers are not able to use extensionality axioms exactly as stated here, because they cannot use the formula $a = a'$ as a pattern to match against in order to derive instantiations. To this end, we provide the option of replacing each equality at an array type in rules and goals with a new relation eq_A with trivial defining axiom

$$\forall a, a' : A.\; \mathsf{eq}_A(a, a') \quad \Leftrightarrow \quad a =_A a' \quad .$$

These axioms only characterise the array type up to isomorphism if the index type S is finite. If S is infinite, one model involves A denoting the subset of functions of type $S \rightarrow T$ with all but finite number of values the same: the array operators only allow us to explicitly construct such functions. Another model, non-isomorphic to this one, uses all functions of type $S \rightarrow T$.

While arrays with integer rather than finite range indices are common at various stages of translation, arrays always start off as having finite index types in SPARK programs. We expect any VCs involving cardinalities of array types to have their truth values maintained by our translation steps, without us adding extra axioms that ensure that abstract types for arrays always have the expected cardinalities.

7 Record Elimination

7.1 Records in FDL

A type definition

$$R : \mathsf{Type} \;=\; \mathsf{Record}(f_1 : T_1, \ldots, f_n : T_n)$$

introduces a record type named R with fields $f_1, \ldots, f_n$ of types $T_1, \ldots, T_n$ respectively.

For simplicity, we consider here a record with two fields:

$$R : \mathsf{Type} \;=\; \mathsf{Record}(\mathsf{fst} : S,\; \mathsf{snd} : T) \qquad .$$

Associated with the record type R are

- a *record constructor* of form

$$\mathsf{mk\_record}_R(\mathsf{fst} := s,\; \mathsf{snd} := t) \quad .$$

As a prefix operator, we can write this as $\mathsf{mk\_record}_R(s,t)$ and declare it with

$$\mathsf{mk\_record}_R \;:\; (S,T)R \qquad ,$$

though here we will continue using the more verbose syntax.

– record *field select* operators

$$\mathsf{select\_fst} : (R)S$$
$$\mathsf{select\_snd} : (R)T \quad .$$

– record *field update* operators

$$\mathsf{default}_R : R$$
$$\mathsf{update\_fst} : (R,S)R$$
$$\mathsf{update\_snd} : (R,T)R \quad .$$

For example, $\mathsf{update\_fst}(r,s)$ updates the fst field of record r with value s.

The generalisation to the case of a record with more fields is straightforward. In the general case, for a record with n fields, we have a constructor that takes n arguments, n field select operators (one for each field), still a single default constant, and n field update operators (again, one for each field).

7.2 Eliminating record constructors

We replace the record constructor $\mathsf{mk\_record}_R(\mathsf{fst} := s,\ \mathsf{snd} := t)$ with

$$\mathsf{update\_snd}_R(\mathsf{update\_fst}_R(\mathsf{default}_R, s), t) \quad ,$$

where $\mathsf{default}_R$ is a new uninterpreted constant of type R.

7.3 Eliminating record updates

We can choose to keep record constructors and have the update operations be derived. We have the identities

$$\mathsf{update\_fst}_R(r,s) = \mathsf{mk\_record}_R(\mathsf{fst} := s, \mathsf{snd} := \mathsf{select\_snd}_R(r))$$
$$\mathsf{update\_snd}_R(r,t) = \mathsf{mk\_record}_R(\mathsf{fst} := \mathsf{select\_fst}_R(r), \mathsf{snd} := t,)$$

There is the choice of either applying these identities to eliminate all occurrences of the update operators, or making the update operators uninterpreted and adding the identities as axioms. If we eliminate update operators of an n-field record, we get a factor of n increase in size of each update expression, and the sub-expression r needs replicating $n-2$ times. If records have high numbers of fields, updates are nested, and there is no structure sharing in expressions, this replication could result in a huge increase in expression size. For this reason we currently introduce the identities as quantified axioms.

7.4 Eliminating interpreted records

We eliminate the need to have a standard interpretation for record type R and associated operators by introducing suitable axioms. We implement two approaches, depending on whether constructors or updates are first eliminated.

If constructors have been eliminated, we use axioms

$$\forall r : R.\ \forall s : S.\ \mathsf{select\_fst}_R(\mathsf{update\_fst}_R(r,s)) =_S s$$
$$\forall r : R.\ \forall s : S.\ \mathsf{select\_snd}_R(\mathsf{update\_fst}_R(r,s)) =_T \mathsf{select\_snd}_R(r)$$
$$\forall r : R.\ \forall t : T.\ \mathsf{select\_fst}_R(\mathsf{update\_snd}_R(r,t)) =_S \mathsf{select\_fst}_R(r)$$
$$\forall r : R.\ \forall t : T.\ \mathsf{select\_snd}_R(\mathsf{update\_snd}_R(r,t)) =_T t$$

These axioms are somewhat similar to the array read-write axioms discussed in Section 6.3. The 1st and 4th axioms here state that if we access a record field that just has been updated, we get the updated value. The 2nd and 3rd axioms state that if we access some field of a record distinct from a field that just has been updated, we get back the same result as if we had accessed the same field before the update. For a record type with n fields, we need n^2 such axioms, one for each choice of field being updated and of field being accessed.

If we choose to treat the record constructor as primitive and update operators as derived, an alternative axiom set is

$$\forall s : S.\ \forall t : T.\ \mathsf{select\_fst}_R(\mathsf{mk\_record}_R(\mathsf{fst} := s, \mathsf{snd} := t)) =_S s$$
$$\forall s : S.\ \forall t : T.\ \mathsf{select\_snd}_R(\mathsf{mk\_record}_R(\mathsf{fst} := s, \mathsf{snd} := t)) =_T t$$

For a record type with n fields, we need n such axioms, one for each choice of selected field. While this approach yields fewer axioms than when constructors have been eliminated, it is not clear which approach might give best prover performance.

There are two ways of axiomatising record extensionality. The first

$$\forall r : R.\ \forall r' : R.\ \mathsf{select\_fst}_R(r) =_S \mathsf{select\_fst}_R(r') \wedge \mathsf{select\_snd}_R(r) =_T \mathsf{select\_snd}_R(r') \Rightarrow r =_R r'$$

only makes use of the select operators. It states that two records should be considered equal when their fields are equal. The second way

$$\forall r : R.\ \mathsf{mk\_record}_R(\mathsf{fst} := \mathsf{select\_fst}_R(r),\ \mathsf{snd} := \mathsf{select\_snd}_R(r)) =_R r$$

relies on constructors not being eliminated. The two ways are easily shown as equivalent. For example, the second can be derived from the first by specialising r in the first to

$$\mathsf{mk\_record}_R(\mathsf{fst} := \mathsf{select\_fst}_R(r'),\ \mathsf{snd} := \mathsf{select\_snd}_R(r'))$$

and simplifying using the select-constructor axioms given above. We implement both approaches. As with arrays, we have the option of introducing a defined relation for equality at record types in order to make the first style of extensionality axiom easier to instantiate. We suspect the that most provers can make little use of the second axiom, unless they resort to instantiating universally quantified hypotheses with any terms of the correct type, which can be very costly.

8 Separation of Formulas and Terms

The FDL language does not make the traditional first-order-logic distinction between formulas and terms: formulas in FDL are terms of Boolean type. While some provers do not make this distinction, some do, and so we implement a translation step that starts with a VC unit where no distinction is made, and introduce the distinction.

The translation is in two phases:

1. Resolve each occurrence of a logical connective, quantifier, Boolean-valued function, or Boolean constant to either a formula or a term level version.
2. Add appropriate operators to convert between terms with Boolean type and formulas in order to ensure well-formedness—that we do not have a term where a formula is expected, or vice versa.

The scope for what resolutions are available depends on the conversion operators used. We define an operator b2p from the Boolean type Bool to propositions (formulas) as

$$\mathsf{b2p}(x) \quad \doteq \quad x =_{\mathsf{Bool}} \mathsf{true_b}$$

and an operator p2b the other way as

$$\mathsf{p2b}(p) \quad \doteq \quad \mathsf{ITE}_{\mathsf{Bool}}(p, \mathsf{true_b}, \mathsf{false_b})$$

Here, $\mathsf{ITE}_T(p,x,y)$ (ITE standing for *if-then-else*) is equal to the term x of type T when the formula p is true and to the term y of type T when the formula p is false, and $\mathsf{true_b}$ and $\mathsf{false_b}$ are the Bool-typed constants for truth and falsity. Some provers and prover formats support an ITE construct, others do not. Even if it is not supported, it can be eliminated using, for example, the identity

$$\phi[\mathsf{ITE}_T(p,e_1,e_2)] \quad \Leftrightarrow \quad (p \wedge \phi[e_1]) \vee (\neg p \wedge \phi[e_2])$$

where $\phi[\cdot]$ is an atomic formula with a sub-term '$\cdot$'. However, this identity must be used with care, as in general it can result in exponential growth in formula size.

We describe below how we carry out the resolutions, both in the case that a p2b operator is available, and in the case it is not.

8.1 Resolution into formulas and terms

Our implementation by default adopts two basic heuristics:

1. Use formula versions when possible, arguing that this ought to enable provers to run more efficiently as they have special built-in support for formula-level reasoning.
2. Avoid if possible introducing two versions, because this complicates and slows provers reasoning.

In what follows, let us refer to rules, goal hypotheses and goal conclusions collectively as *clauses*.

The resolution procedure examines in turn every subterm of every clause of a VC Unit in order to identify occurrences of terms that need resolving. This examination is completed before the resolutions are actually carried out.

The resolution distinguishes whether a subterm is in a *formula context* or a *term context*. A subterm is in a formula context if all the operators above it—up to the root of the clause—are just formula constructors (propositional logic connectives and predicate logic quantifiers). Otherwise it is in a term context.

Resolution of each kind of operator is as follows by default:

– **logical connectives** ($\wedge$, $\vee$, $\neg$, $\Leftrightarrow$, $\Rightarrow$) and **logical constants** (true, false): If the connective or constant is in a term context and p2b is not available, use a term version. Otherwise use a formula version. We use b suffixes to distinguish term versions of these connectives and constants from the formula versions. For example, we write $\wedge_b$ for the term-level version of $\wedge$.

– **quantifiers** (∀, ∃): No provers requiring term/formula separation support term-level quantifiers, so we always use formula versions.
– Bool**-valued functions** and Bool**-valued uninterpreted constants**: If there is at least one occurrence of the function or constant in a term context, use a term level function or constant for all occurrences. Otherwise use a relation or propositional variable for all occurrences.
  One exception is with relations for which provers have built-in support: equality and order relations on integers. In this case, a term version is used only when essential, that is, when the occurrence is in a term context and p2b is not available. This strategy, in general, results in VC Units that contain instances of both term-level and formula-level versions of each of these relations. When we get both versions of a relation, we add an axiom asserting that they are equivalent.
  Another exception is with array and record select operators, when the array happens to have a Bool element type or the record field select function is for a Bool-valued field. In this case, we always use a term-level function to ensure treatment of array and record operator typing is always uniform.

8.2 Insertion of operators converting between formulas and terms

We insert a b2p operator whenever a Bool-typed term is at a position where a formula is expected, and we insert a p2b operator whenever a formula is at a position where a Bool-typed term is expected. This ensures that each of our VC unit clauses is a well-formed strict first-order-logic formula.

8.3 Options

It is not clear if the resolution heuristics described above should alway be applied, and we have options to enable other heuristics, such as always prefer term-level versions, or always prefer formula level versions, whenever possible.

We also implement an option to initially convert equalities over terms of type Bool into if-and-only-if formulas. This is in line with the heuristic to maximise the amount of structure resolved to the formula level.

9 Finite Type Elimination by Type Refinement

We consider here a translation for eliminating finite types, for example, for replacing the Boolean type Bool and an integer subrange type $\{0\,..\,9\}$ with the integer type Int, and a type

$$\mathsf{Array}(\{0\,..\,9\}, \mathsf{Record}(\mathsf{fst} : \{0\,..\,9\}, \mathsf{snd} : \mathsf{Bool}))$$

with the type

$$\mathsf{Array}(\mathsf{Int}, \mathsf{Record}(\mathsf{fst} : \mathsf{Int}, \mathsf{snd} : \mathsf{Int})) \quad .$$

These type changes are accompanied by changes to formulas and the addition of axioms, in order to ensure the validity of each goal in a VC unit is unchanged. We call this translation a *type refinement* translation, as the translations of each type are similar to data-type refinements. See the end of Section 2.5 for further information and references. We first give a simplified account of the translation, and later, in Section 9.6, discuss a few details of how the translation is actually implemented.

The translation works simultaneously on all types of a VC unit. For each named type T, we introduce

- a type T^+, the *base type* for T
- a unary relation $\in_T$ on T^+, the *membership predicate* for T,
- a binary relation $\equiv_T$ on T^+, the *equivalence relation* for T.

Usually applications of $\equiv_T$ are infix, so we write $x \equiv_T y$ rather than $\equiv_T (x,y)$. The intent is that $\equiv_T$ is an equivalence relation when restricted to $\{x : T^+ | \in_T (x)\}$, and there is a 1-to-1 correspondence between the equivalence classes of $\equiv_T$ restricted to $\{x : T^+ | \in_T (x)\}$ and the elements of T. We place no requirements on $\equiv_T$ when either argument does not satisfy $\in_T$. We say a membership predicate $\in_T$ is *trivial* if $\in_T (x)$ is true for all x. We say an equivalence relation $\equiv_T$ is *trivial* if $\equiv_T (x,y)$ is the same as $x =_T y$ for all x,y.

Sometimes we have intended interpretations for T^+, $\in_T$ and $\equiv_T$. Other times T^+ might be a defined type, and we introduce axioms characterising $\in_T$ and $\equiv_T$.

9.1 Translation of theory elements

- **Type constant declaration** $C : \mathsf{Type}$.
  Replace by type constant declaration $C^+ : \mathsf{Type}$.
  If C is uninterpreted, we declare that $\equiv_C$ is trivial, and allow the option of declaring that $\in_C$ is trivial. See Section 13 for discussion of when this option is useful.
  If C has an intended interpretation, there might be type-specific modifications to the declaration or the interpretation. Currently, there are optional modifications for the Bool type constant. See Section 10 for details. For the other interpreted type constants (Int, Real), there are no changes.
- **Type constant definition** $C : \mathsf{Type} = T$.
  The expected cases for T are
  - Array type
  - Record type
  - Integer subrange type
  - Type constant

  Enumerated types are not expected. For the first 3 cases, see the appropriate section below for changes to the definition and other theory elements. For T a type constant, replace the definition with type constant definition

  $$C^+ = T^+$$

  and add axioms

  $$\forall x : T^+.\ \in_C (x) \quad \Leftrightarrow \ \in_T (x)$$
  $$\forall x,y : T^+.\ x \equiv_C y \ \Leftrightarrow \ x \equiv_T y \quad .$$

  Refinement of array and record types is not strictly necessary for the SMT-LIB and Simplify translation targets: these types can be eliminated before type refinement. We consider their refinement, as the SMT provers might be more efficient with elimination of these types after refinement. We are also looking forward to translating for Z3's native language and the Higher-Order-Logic languages of popular interactive theorem provers. All these languages have support for arrays and records, but not sub-types.

- **Constant declaration** $c : T$
  Replace by constant declaration $c : T^+$. If c is uninterpreted, add a subtyping axiom $\in_T (c)$. If c is interpreted before refinement, a new interpretation needs to be specified.
- **Function declaration** $f : (S)T$.
  Replace by function declaration $f : (S^+)T^+$.
  If the function is uninterpreted, add a subtyping axiom and a functionality axiom

  $$\begin{array}{l} \forall x : S^+.\ \in_S (x) \Rightarrow \in_T (f(x)) \\ \forall x,y : S^+.\ \in_S (x) \wedge \in_S (y) \wedge x \equiv_S y \Rightarrow f(x) \equiv_T f(y) \quad . \end{array}$$

  These axioms ensure that each model of the function after translation can be translated into a model of the function before translation. As remarked in Section 2.5, part of the mathematics of theory interpretations involves constructing maps of structures of the target theory into structures of the source theory.
  An alternative to the above subtyping axiom is the stronger axiom

  $$\forall x : S^+.\ \in_T (f(x))$$

  where the $\in_S$ precondition is omitted. A model will still exist, providing we are careful in ensuring that all axioms constraining f are translated properly so they provide no constraints on values of f on arguments not satisfying $\in_S$. Using stronger axioms of this kind should result in better prover performance, since less work is required in producing useful instantiations of them.
  It is generally not consistent to omit the $\in_S$ preconditions in the functionality axiom.
  If the function f is interpreted before refinement, a new interpretation needs to be specified.
  The generalisation for *n*-ary functions is straightforward.
- **Relation declaration** $r : (T)$.
  Replace by relation declaration $r : (T^+)$. If the relation r is uninterpreted, add a functionality axiom

  $$\forall x,y : S^+.\ \in_S (x) \wedge \in_S (y) \wedge x \equiv_S y \Rightarrow r(x) \Leftrightarrow r(y) \quad .$$

  This axiom ensures that each model of the relation after translation can be translated into a model of the relation before translation. If r is interpreted before refinement, a new interpretation afterwards is needed.
  The generalisation for *n*-ary relations is straightforward.
- **Formulas**.
  Formula $\forall x : T.\ P(x)$ becomes $\forall x : T^+.\ \in_T (x) \Rightarrow P'(x)$, where $P'(x)$ is the translation of $P(x)$.
  Formula $\exists x : T.\ P(x)$ becomes $\exists x : T^+.\ \in_T (x) \wedge P'(x)$.
  Formula $s =_T t$ becomes $s \equiv_T t$.
  All other formulas are unchanged.
  This translation of quantifiers is commonly referred to as *relativisation*. As a simple example, consider a theory interpretation from the naturals to the integers: the formula $\forall x : \natural.\ P(x)$ translates to $\forall x : \mathbf{Z}.\ x \geq 0 \Rightarrow P'(x)$.
  If we are in strict first-order logic, we introduce both term-level and formula-level versions of $s \equiv_T t$, corresponding to the term and formula level versions of $s =_T t$, and we add an axiom stating how they correspond.
- **Intended interpretations**
  The changes required are described in Sections 9.2–9.5.

In many cases, when $\in_T (x)$ is always true or when $x \equiv_T y$ is simply $x =_{T^+} y$, the added axioms simplify, sometimes to the extent that they become tautologies and are unnecessary.

9.2 Translation of Array Types

We consider here translating a one dimensional array with type definition

$$A : \mathsf{Type} = \mathsf{Array}(S,T) \qquad .$$

The generalisation to multi-dimensional arrays is straightforward.

The translation of the index type S and element type T induces a translation of the array type A. We consider that refinement of the element type T may introduce a non-trivial base type T^+, a non-trivial membership predicate $\in_T$ and a non-trivial equivalence relation $\equiv_T$, and refinement of the index type S may introduce a non-trivial base type S^+ and a non-trivial membership predicate $\in_S$. However, we assume that $\equiv_S$ is trivial. We need to do this to keep *update* operators straightforwardly defined in possible later translation stages that introduce axiomatic characterisations of these operators. This is a reasonable assumption as each S_i is normally the integers, some subrange of the integers, or an enumerated type. If ever there were some reason for wanting to relax this assumption, it would not be difficult to do so.

The refinement introduces a new array type definition

$$A^+ : \mathsf{Type} = \mathsf{Array}(S^+,T^+) \qquad .$$

The functions and constants associated with array A acquire new type declarations, as described above in Section 9.1.

$$\begin{array}{l} \mathsf{default}_A : A^+ \\ \mathsf{const}_A : (T^+)A^+ \\ \mathsf{select}_A : (A^+,S^+)T^+ \\ \mathsf{update}_A : (A^+,S^+,T^+)A^+ \end{array}$$

After the translation, $\mathsf{default}_A$ and const_A remain uninterpreted, and select_A and update_A now have interpretations as the select and update operators for the type $\mathsf{Array}(S^+,T^+)$. Also as described above in Section 9.1, the axiom for const_A introduced in Section 6.2 is suitably relativised, and new functionality and subtyping axioms are introduced for $\mathsf{default}_A$ and const_A.

Now let us consider how to suitably define $\in_A$ and $\equiv_A$, and, if needed, add axioms, so that the use of the refined array type is essentially isomorphic to the orginal type. We ensure that new arrays store elements satisfying $\in_T$ at indices satisfying $\in_S$. We consider two options for what happens at indices not satisfying $\in_S$: either require that some default element of $\in_T$ always be stored, or place no constraints. How the translations are tailored for each of these cases is as follows.

– **Out-of-bounds elements constrained**

We use the definitions

$$\begin{array}{lll} \in_A (a) & \doteq \forall s : S^+. & (\in_S (s) \Rightarrow \in_T (\mathsf{select}_A(a,s))) \\ & & \wedge\, (\neg \in_S (s) \Rightarrow \mathsf{select}_A(a,s) =_T \mathsf{any\_element}_A) \\ \equiv_A (a,a') & \doteq \forall s : S^+. & \mathsf{select}_A(a,s) \equiv_T \mathsf{select}_A(a',s) \end{array}$$

where $\mathsf{any\_element}_A$ has declaration

$$\mathsf{any\_element}_A : T^+$$

and no constraining axioms. In the event that $\equiv_T$ is trivial, the definition of $\equiv_A (a, a')$ amounts to extensional array equality and so we can use instead

$$\equiv_A (a, a') \doteq a =_{A^+} a' \quad .$$

- **Out-of-bounds elements unconstrained**
We use the definitions

$$\begin{array}{ll} \in_A (a) & \doteq \forall s : S^+ .\ \in_S (s) \Rightarrow \in_T (\mathsf{select}_A(a,s)) \\ \equiv_A (a,a') & \doteq \forall s : S^+ .\ \in_S (s) \Rightarrow \mathsf{select}_A(a,s) \equiv_T \mathsf{select}_A(a',s) \end{array}$$

9.3 Translation of Record Types

For simplicity we consider refining only two field records.

$$R : \mathsf{Type} = \mathsf{Record}(\mathsf{fst} : S,\ \mathsf{snd} : T) \qquad .$$

The generalisation to records with other numbers of fields is straightforward.

We have that

$$\begin{array}{ll} R^+ & \doteq \mathsf{Record}(\mathsf{fst} : S^+,\ \mathsf{snd} : T^+) \\ \in_R (r) & \doteq \in_S (\mathsf{select\_fst}(r)) \wedge \in_T (\mathsf{select\_snd}(r)) \\ \equiv_R (r,r') & \doteq \mathsf{select\_fst}(r) \equiv_S \mathsf{select\_fst}(r') \wedge \mathsf{select\_snd}(r) \equiv_T \mathsf{select\_snd}(r') \qquad . \end{array}$$

9.4 Relaxing integer subrange types to the Int type

We refine an integer subrange constant definition

$$S : \mathsf{Type} = \{j, \ldots, k\} \quad ,$$

where $j \leq k$, using the definitions

$$\begin{array}{ll} S^+ & \doteq \mathsf{Int} \\ \in_S (x) & \doteq j \leq x \wedge x \leq k \\ \equiv_S (x,y) & \doteq x =_{\mathsf{Int}} y \quad . \end{array}$$

9.5 Relaxing the Boolean type to the integer type

We implement two alternative translations that use Int as a base type:

$$\mathsf{Bool}^+ \doteq \mathsf{Int} \quad .$$

The translations apply if initially the Bool type has an interpretation as some two element type containing distinct interpretations of the constants true_b and false_b and the logical operators all have their usual interpretations on this type.

With both alternatives, we interpret $\mathsf{true_b}$ as 1 and $\mathsf{false_b}$ as 0, and require new interpretations for the Boolean logical operators and Boolean-valued relations that treat 1 as true and all other integers as false, and that only have values 0 or 1.

9.5.1 *Booleans as subtype of integers*

We consider the Bool type as a 2 element subset of the integer type. We use the definitions

$$\in_{\mathsf{Bool}}(x) \doteq x =_{\mathsf{Int}} 0 \lor x =_{\mathsf{Int}} 1 \qquad (\text{or } 0 \leq x \leq 1)$$
$$x \equiv_{\mathsf{Bool}} y \doteq x =_{\mathsf{Int}} y$$

where x, y are of type Int.

9.5.2 *Booleans as quotient of integers*

We consider the Bool type as being derived from two equivalence classes of integers. Introduce

$$\in_{\mathsf{Bool}}(x) \doteq \mathsf{True}$$
$$x \equiv_{\mathsf{Bool}} y \doteq \mathsf{b2p}(x) \Leftrightarrow \mathsf{b2p}(y) \quad .$$

9.6 Implementation details

- We do not invent new type names for the base types T^+. Instead we just reuse the name T.
- We track the trivialness of the membership predicate $\in_T$ and equivalence relation $\equiv_T$ for each type T, and use this information to simplify and sometimes eliminate the new axioms introduced by the translation. For example, functionality axioms for functions are unneeded when the equivalence relations for all the argument types are trivial. This requires that the translation works on types in the order they are defined, and works through the function, constant and relation declarations after the types have been considered.

10 Boolean Type Elimination

We consider here eliminating the Boolean type and associated interpreted constants, functions and relations. We allow for the interpretation of the Boolean type Bool initially being the integers as well as some two element domain.

10.1 Eliminating Boolean-valued functions and relations

We introduce the axioms

$$\forall p : \mathsf{Bool}.\ \mathsf{b2p}(\neg_b p) \Leftrightarrow \neg\mathsf{b2p}(p)$$
$$\forall p, q : \mathsf{Bool}.\ \mathsf{b2p}(p \land_b q) \Leftrightarrow \mathsf{b2p}(p) \land \mathsf{b2p}(q)$$
$$\forall p, q : \mathsf{Bool}.\ \mathsf{b2p}(p \lor_b q) \Leftrightarrow \mathsf{b2p}(p) \lor \mathsf{b2p}(q)$$
$$\forall p, q : \mathsf{Bool}.\ \mathsf{b2p}(p \Leftrightarrow_b q) \Leftrightarrow \mathsf{b2p}(p) \Leftrightarrow \mathsf{b2p}(q)$$
$$\forall x, y : T.\ \mathsf{b2p}(\mathsf{term\_eq}_T(x, y)) \Leftrightarrow x =_T y$$
$$\forall x, y : T^+.\ \mathsf{b2p}(\mathsf{term\_equiv}_T(x, y)) \Leftrightarrow x \equiv_T y$$
$$\forall i, j : \mathsf{Int}.\ \mathsf{b2p}(\mathsf{term\_le}_{\mathsf{Int}}(i, j)) \Leftrightarrow i \leq j$$
$$\forall x : T.\ \mathsf{b2p}(\mathsf{term\_}r(x)) \Leftrightarrow r(x)$$

and remove the requirements that the functions and relations have intended interpretations. Here $\mathsf{term\_eq}_T$ is the term-level version of formula-level equality $=_T$, $\mathsf{term\_equiv}_T$ is the term-level version of the equivalence relation $\equiv_T$ introduced by type refinement, $\mathsf{term\_le}_{\mathsf{Int}}$ is the term-level version of $\leq$ over the integers, and $\mathsf{term\_}r$ is the term-level version of uninterpreted relation r. These axioms are consistent with the initial explicit interpretations of the functions, whether Bool is interpreted as the integers or some two element domain.

We introduce these axioms after type refinement rather than before, as this avoids the introduction of relativisation preconditions that might slow provers. For example, if we were to introduce the axiom for $\wedge_b$ before type refinement and we requested refinement to refine the type Bool to be a subtype of the integers, the axiom after refinement would be

$$\forall p,q : \mathsf{Bool}.\ \in_{\mathsf{Bool}}(p) \wedge \in_{\mathsf{Bool}}(q) \Rightarrow \mathsf{b2p}(p \wedge_b q) \Leftrightarrow \mathsf{b2p}(p) \wedge \mathsf{b2p}(q) \quad .$$

Also, if we eliminated the Boolean propositional logic operators before refinement, we would also get refinement adding extra unnecessary subtyping axioms such as

$$\forall p,q : \mathsf{Bool}.\ \in_{\mathsf{Bool}}(p \wedge_b q)$$

or

$$\forall p,q : \mathsf{Bool}.\ \in_{\mathsf{Bool}}(p) \wedge \in_{\mathsf{Bool}}(q) \Rightarrow \in_{\mathsf{Bool}}(p \wedge_b q)$$

depending on whether generation of strong subtyping axioms was chosen or not.

10.2 Eliminating coercions between formulas and terms

We substitute out occurrences of the b2p coercion from term-level Booleans to formulas and the p2b coercion from formulas to term-level Booleans using the identities mentioned earlier in Section 8:

$$\begin{aligned} \mathsf{b2p}(x) &= x =_{\mathsf{Bool}} \mathsf{true_b} \\ \mathsf{p2b}(p) &= \mathsf{ITE}_{\mathsf{Bool}}(p, \mathsf{true_b}, \mathsf{false_b}) \quad . \end{aligned}$$

10.3 Eliminating the Boolean type and constants

We implement two alternatives for when we remove intended interpretations of the Boolean type Bool and the logical constants $\mathsf{true_b}$ and $\mathsf{false_b}$.

If the Boolean type Bool has interpretation as the integers, we change the type declaration of Bool to a type definition

$$\mathsf{Bool} : \mathsf{Type} = \mathsf{Int}$$

and add axioms

$$\begin{aligned} \mathsf{false_b} &=_{\mathsf{Int}} 0 \\ \mathsf{true_b} &=_{\mathsf{Int}} 1 \quad . \end{aligned}$$

If Bool is interpreted as some abstract two element type, we keep its type declaration

$$\mathsf{Bool} : \mathsf{Type}$$

and add axioms

$$\begin{aligned} &\forall p : \mathsf{Bool}.\ p =_{\mathsf{Bool}} \mathsf{true_b} \vee_b p =_{\mathsf{Bool}} \mathsf{false_b} \\ &\mathsf{true_b} \neq \mathsf{false_b} \end{aligned}$$

The first axiom could be hard for automatic provers to use efficiently, so this may not be a desirable option.

11 Arithmetic Simplification

We use various simplifications to turn arithmetic expressions that are semantically linear into expressions that are obviously syntactically linear. For example, we

- substitute out constants c if there is some hypothesis that $c = k$ where k is an integer literal. Such hypotheses are very common in the VCs generated by Praxis's SPARK VC generator tool.
- normalise arithmetic expressions involving multiplication and integer division by constants.
- evaluate ground arithmetic expressions involving multiplication, exponentiation by non-negative integers, integer division and the modulus function.

Examples of the normalisation are replacing $(k \times e) \times (k' \times e')$ with $(k \times k') \times (e \times e')$ and replacing $(k \times e)\ \mathsf{div}\ k'$ with $(k\ \mathsf{div}\ k') \times e$ when k' divides k. Here k, k' are integer constants and e, e' are arbitrary integer-valued terms.

We also allow exponentiation by non-negative integers to be expanded away, for when solvers can handle non-linear arithmetic, but not exponentiation.

12 Elimination of Arithmetic Types and Operators

Options we support include

- Replace natural number literals above some threshold t with a new uninterpreted constants $n_1 \ldots n_k$ and add axioms $t < n_1 < n_2 \ldots < n_k$ asserting how these constants are ordered.
  This is an attempt to avoid arithmetic overflow in provers such as Simplify that use fixed precision rather than bignum arithmetic. This approach is used with ESC/Java when it uses the Simplify solver [29].
- Replace all integer and real multiplications that are not obviously syntactically linear by new uninterpreted functions. This forces non-linear arithmetic expressions to look linear, as required by several solvers.
- Make exponentiation of integer and real expressions by non-negative integers uninterpreted.
- Make integer division and the modulus function uninterpreted. Add characterising axioms such as:

$$\forall x, y : \mathsf{Int}.\ 0 < y \ \Rightarrow\ 0 \leq x\ \mathsf{mod}\ y$$
$$\forall x, y : \mathsf{Int}.\ 0 < y \ \Rightarrow\ x\ \mathsf{mod}\ y < y$$
$$\forall x, y : \mathsf{Int}.\ 0 \leq x \ \wedge\ 0 < y \ \Rightarrow\ y \times (x\ \mathsf{div}\ y) \ \leq\ x$$
$$\forall x, y : \mathsf{Int}.\ 0 \leq x \ \wedge\ 0 < y \ \Rightarrow\ x - y \ <\ y \times (x\ \mathsf{div}\ y)$$
$$\forall x, y : \mathsf{Int}.\ x \leq 0 \ \wedge\ 0 < y \ \Rightarrow\ x \ \leq\ y \times (x\ \mathsf{div}\ y)$$
$$\forall x, y : \mathsf{Int}.\ x \leq 0 \ \wedge\ 0 < y \ \Rightarrow\ y \times (x\ \mathsf{div}\ y) \ <\ x + y$$

- Make real division uninterpreted.
- Make the real type and all functions involving reals uninterpreted.

- Make uninterpreted functions over integers expressing effect of bit-wise operations. Add characterising axioms for these such as:

$$0 \leq x \wedge 0 \leq y \Rightarrow 0 \leq \mathsf{bit\_or}(x,y)$$
$$\forall x,y : \mathsf{Int}.\ 0 \leq x \wedge 0 \leq y \Rightarrow x \leq \mathsf{bit\_or}(x,y)$$
$$\forall x,y : \mathsf{Int}.\ 0 \leq x \wedge 0 \leq y \Rightarrow y \leq \mathsf{bit\_or}(x,y)$$
$$\forall x,y : \mathsf{Int}.\ 0 \leq x \wedge 0 \leq y \Rightarrow \mathsf{bit\_or}(x,y) \leq x+y \quad .$$

13 Uninterpreted Type and Defined Type Elimination

The prover Simplify does not support uninterpreted types and type definitions. Essentially it assumes that all functions and relations are on the single sort of integers.

As observed by Bouillaguet et al. [10], if it is consistent for all uninterpreted types to have interpretations with the same cardinality, then it is not necessary to use a many-to-single sort relativisation translation where a predicate is defined carving out each of the many sorts from a single sorted universe. Instead, it is consistent to drop these predicates and give all uninterpreted types the same interpretation.

We have not established that every uninterpreted type in SPARK VC units is free from any axiomatic constraints that rule out the integers as a possible model. There might be constraints that only allow finite models of some uninterpreted type. Types with natural models with larger cardinality than the integers (e.g. the real type) are not an issue, as the Downward Löwenheim-Skolem theorem guarantees in these cases that a countable model also exists. We therefore refine every uninterpreted type using an uninterpreted membership predicate function (see Section 9.1) in order to ensure every uninterpreted type can be modelled by the integers.

We allow type definitions to be eliminated by expanding the definitions.

14 Case Study SPARK Programs

For our experiments we work with three readily available examples.

- **Autopilot**: the largest case study distributed with the SPARK book [5]. It is for an autopilot control system for controlling the altitude and heading of an aircraft.
- **Simulator**: a missile guidance system simulator written by Adrian Hilton as part of his PhD project. It is freely available on the web[10] under the GNU General Public Licence.
- **Tokeneer**: the Tokeneer ID Station is a biometric software system for managing access to a secure area [6]. This case study was commissioned by the US National Security Administration in order to evaluate Praxis's 'Correct by Construction' SPARK-based high-integrity software development methodology. All the materials from this case study were made publically available on the web late 2008 [11].

Some brief statistics on each of these examples and the corresponding verification conditions are given in Table 2.

The lines-of-code estimates are rather generous, being simply the sum of the number of lines in the Ada specification and body files for each example. The *annotations* count

[10] `http://www.suslik.org/Simulator/index.html`
[11] `http://www.adacore.com/tokeneer`

Table 2 Statistics on Case Studies

	Autopilot	Simulator	Tokeneer
Lines of code	1075	19259	30441
No. funcs & procs	17	330	286
No. annotations	17	37	194
No. VC goals	133	1806	1880

is the number of SPARK precondition, postcondition and assertion annotations in all the Ada specification and body files. In the Autopilot and Simulator examples, almost all the annotations were assertions. In the Tokeneer example, there were roughly equal number of the three kinds. The VC goal counts are for the goals output by the Examiner, excluding those goals the Examiner proves internally. The Examiner provides no information about these goals other than that it discharged them, so there is little point in us considering them.

In all cases, most of the VCs are from exception freedom checks inserted by the Examiner tool. The VCs from all examples involve enumerated types, linear and non-linear integer arithmetic, integer division and uninterpreted functions. In addition, the Simulator and Tokeneer examples includes VCs with records, arrays and the modulo operator.

15 Experimental Conditions

The provers tools we linked to our VCT tool were:

- CVC3 2.2,
- Yices 1.0.24,
- Z3 2.3.1,
- Simplify 1.5.4.

We compared our results against those obtained with the Praxis automatic prover/simplifier from the 8.1.1 GPL release of Praxis's SPARK toolkit. As explained in the Introduction, our interest is to do better than this prover, so it is important we compare against it. All experiments used a 2.67 GHz Intel Xeon X5550 4 core processor with 50 GB of physical memory and running Scientific Linux 5.

As distributed, all the Tokeneer VCs are described as true, though not all are necessarily directly machine provable. The distributed VC goals fall into 3 categories:

- (94.1%) those proved using Simplifier, Praxis's automatic prover,
- (2.3%) those proved using Checker, Praxis's interactive prover, and
- (3.6%) those deemed true by inspection.

The interactive proofs drew on auxiliary rule files that included definitions of specification functions used in the SPARK program annotations. Whenever some of the VCs of a program unit were proved using the Checker tool and the Checker made use of an auxiliary rule file, we also read in that rule file when attempting proof of VCs of that unit. For a fair comparison, we report in our results section below on the Praxis automatic prover's performance running with these auxiliary rule files. It seems the Tokeneer developers never tried this, perhaps because the earlier version of the automatic prover they used did not have this option.

We report here on experiments with 6 choices of SMT solver and interface mode.

- **CVC3/API**

– **Yices/API**. Here we let Yices reject individual hypotheses and conclusions that it deems non-linear. It does accept universally quantified hypotheses with non-linear multiplications, and does find useful linear instantiations of these hypotheses.
– **CVC3/SMT-LIB file interface**, using the AUFNIRA SMT-LIB sub-logic.
– **Yices/SMT-LIB file interface**, using the AUFLIA SMT-LIB sub-logic. Here we needed to abstract all non-linear multiplications, including those in quantified hypotheses, in order to conform to the AUFLIA requirements.
– **Z3/SMT-LIB file interface**, using the AUFNIRA SMT-LIB sub-logic.
– **Simplify/Simplify file interface**

Unless otherwise stated, all solvers were run with a 1 second timeout, except for Yices with the API interface, since the Yices API we use provides no functionality for setting timeouts. We refer to each of these setups of a prover with some interface mode as a *test configuration*. For convenience we also refer to running the Praxis prover as a test configuration.

16 Experimental Results

In this section we report our observations of the coverage obtained with each test configuration and of the distribution of prover run-times on the different problems. Our VCT tool can work through all the VCs for all the program units of a case study in a single run, and output a comma-separated-value record of data concerning each goal. This made it straightforward to produce the various statistics listed in this section.

In Section 17 we give an analysis of these observations, and show examples of VCs that illustrate differences between solvers. Section 17 also includes remarks on soundness and robustness issues encountered in the experiments.

Table 3 Coverage of VC goals (%)

Prover Interface	CVC3 API	Yices API	CVC3 SMT-LIB	Yices SMT-LIB	Z3 SMT-LIB	Simplify file	Praxis
Autopilot	96.2	95.5	96.2	91.7	98.5	96.2	97.0
Simulator	94.6	94.0	94.5	93.6	95.5	93.2	95.5
Tokeneer	96.6	97.0	95.3	95.7	97.0	86.4	95.0

The coverage obtained with each test configuration is summarised in Table 3. The table shows the percentage of VC goals from each case study that are claimed true with each configuration.

Some of the Simplify runs halted on Simplify failing an internal runtime assertion check. This happened on 2.3% of the Simulator goals, and 0.5% of the Tokeneer goals.

Table 4 Average run time per goal (msec)

Prover Interface	CVC3 API	Yices API	CVC3 SMT-LIB	Yices SMT-LIB	Z3 SMT-LIB	Simplify file	Praxis
Autopilot	111 (100)	18 (7)	91 (73)	32 (15)	42 (25)	34 (17)	16
Simulator	190 (173)	25 (8)	171 (146)	51 (26)	74 (50)	69 (44)	33
Tokeneer	358 (322)	53 (18)	251 (206)	85 (40)	83 (38)	415 (370)	50

Table 4 shows the total run time for each test configuration on each case study. The unparenthesised times are normalised by being divided by the number of goals in each case. The parenthesised numbers are normalised estimates of the time spent in the actual prover code rather than the VCT tool's code. In the case of Yices with the API interface, it is estimated that, if there had been support to enforce a 1 second timeout, the Tokeneer times would have been 7sec shorter and there would have been no change to the Autopilot and Simulator times.

Table 5 Run time distribution for Tokeneer case study goals (sec)

Prover Interface	Cvc3 API	Yices API	Cvc3 SMT-LIB	Yices SMT-LIB	Z3 SMT-LIB	Simplify file
30%	0.11	0.02	0.04	0.03	0.02	0.05
50%	0.25	0.03	0.06	0.03	0.03	0.28
70%	0.48	0.04	019	0.04	0.04	0.58
90%	0.66	0.05	0.71	0.05	0.06	1.01
95%	0.73	0.06	1.00	0.06	0.07	1.10
98%	0.81	0.07	>20.00	0.11	0.10	>20
99%	5.49	0.16	>20.00	4.05	>20.00	>20

Table 6 Run time distribution for Tokeneer case study goals (sec) (only proven goals)

Prover Interface	Cvc3 API	Yices API	Cvc3 SMT-LIB	Yices SMT-LIB	Z3 SMT-LIB	Simplify file
30%	0.11	0.02	0.04	0.02	0.02	0.04
50%	0.25	0.03	0.05	0.03	0.03	0.29
70%	0.47	0.04	0.15	0.04	0.04	0.56
90%	0.65	0.05	0.62	0.05	0.05	0.98
95%	0.70	0.05	0.79	0.05	0.07	1.04
98%	0.76	0.07	0.99	0.06	0.08	1.13
99%	0.78	0.08	1.13	0.07	0.09	1.42
100%	0.85	0.27	12.34	0.26	0.82	12.65

The average run times for the provers are often heavily skewed by long run times for relatively few of the goals, especially as it is common for provers to time out rather than terminate on goals they cannot prove. To give an indication of how run times on goals are distributed, we sorted the run times in each case, and show in Table 5 these goal run times at a few percentiles. For example, the 50% line in the table gives the median run times. We ran the tests for this data with a timeout of 20sec rather 1sec to improve the quality of the data on slower goals. It is also interesting to look at the distribution of run-times for just the goals that each prover is able to prove. This makes it easy to see how timeout thresholds affect the coverage. This data is shown in Table 6. The entry for some test configuration on the 50% row shows that 50% of the final coverage for a 20sec timeout with that configuration was obtained with run-times of the indicated value or less.

Numbers are not given for the Praxis's prover in these tables, as its log files do not provide a breakdown of its run time on individual goals.

17 Discussion of Results

17.1 Coverage

We discuss in this section the coverage results summarised in Table 3 in the previous section, considering each case study in turn.

Autopilot

The goals in this case study are all thought to be true, and, indeed, with a timeout of of 10 seconds rather than 1 second, Z3 reports them all to be true.

The goals that failed to be proved under one or more test configuration all involved bounding properties of arithmetic formulas that included integer division or the modulo operator. For example, the goal

```
H1:   j >= 0 .
H2:   j <= 100 .
H3:   k > 0 .
H4:   j <= k .
H5:   m <= 45 .
H6:   m > 0 .
      ->
C1:  (m * j) div k <= 45 .
```

was not provable in any of the test configurations, though a goal with the same hypotheses and the similar conclusion

```
C1:  (m * j) div k >= -45 .
```

was proved with the Praxis and Z3 configurations. These and other goals presented in this section are all abstracted and simplified to show the essential structure: common subexpressions are abstracted to variables, irrelevant hypotheses and conclusions are removed, and constants with literal values are often substituted out.

A slightly harder example of a bounds theorem that cannot be solved just by considering how the bounds on each argument to the division operator affect the bounds of its value is:

```
H1:    f > 0 .
H2:    f <= 100 .
H3:    v >= 0 .
H4:    v <= 100 .
       ->
C1:    (100 * f) div (f + v) <= 100 .
```

This was proved in the Z3 configuration and also in the CVC3-API configuration if we raised the timeout to 20sec.

The coverage with Yices/API was lower because Yices/API rejected most hypotheses and conclusions with non-linear multiplication, whereas non-linear multiplication was accepted in all other configurations except YicesSMT-LIB. Usefully, Yices via its API accepted non-linear multiplication within universally quantified hypotheses, and permitted linear instantiations of these hypotheses. For example, in proving

```
H1:    f >= -1000 .
H2:    f <= 1000 .
H3:    t >= -1000 .
H4:    t <= 1000 .
        ->
C1:    (t - f) div 12 >=  -180 .
```

for the case when $t - f$ is non negative, Yices can instantiate the hypothesis

$$\forall x, y : \mathsf{Int}.\ 0 \leq x \ \wedge\ 0 < y \ \Rightarrow\ x - y < y \times (x \ \mathsf{div}\ y)$$

to derive the new linear hypothesis that

$$t - f - 12 < 12 \times ((t - f)\mathsf{div}12)$$

from which the conclusion

$$(t - f)\mathsf{div}12 \geq -180$$

follows. Unfortunately, should Yices find a non-linear instantiation, it currently immediately terminates rather than ignoring the instantiation.

One reason for the lower coverage with Yices/SMT-LIB is that then, with linearity required everywhere, the non-linear multiplication in quantified hypothesis such as above is abstracted to an uninterpreted function. This makes such a hypothesis much less useful.

CVC3, Z3 and Simplify all accept non-linear multiplications everywhere in their input formulas.

Simulator

While the VC goals here were richer than with the Autopilot case study in that they also involved array and record expressions, the goals on which provers gave different results again all involved arithmetic beyond linear arithmetic. For example, Z3 and the Praxis prover both proved the goal

```
H1:    s >= 0 .
H2:    s <= 971 .
        ->
C1:    43 + s * (37 + s * (19 + s)) >= 0 .
C2:    43 + s * (37 + s * (19 + s)) <= 214783647 .
```

and the goal

```
H1:  m = 971 .
H2:  k0 = 0 .
H3:  k1 = 2^32 - 1 .
   ->
C1:   e1 mod m * (e2 mod m) mod m  >= k0 .
C2:   e1 mod m * (e2 mod m) mod m  <= k1 .
```

The rounding of the coverage figures for Z3 and the Praxis prover hides the fact that the Praxis prover discharages 1 more goal. This in essence is:

```
H1:    p >= 1 .
H2:    p <= 1000 .
H3:    d >= 0
H4:    d <= 92
H5:   r  >= 0 .
H6:   r  <= 100 .
        ->
C1:    (942 + d * (d * d) div 2000) * r div 100 * p div 2 >= -1000000 .
C2:    (942 + d * (d * d) div 2000) * r div 100 * p div 2 <= 1000000 .
```

To read the conclusions, note that `*` and integer division `div` have the same precedence and are left associative. The conclusions follow by interval arithmetic and bounding properties of `div`: one can compute that the left-hand-side expression in the conclusion is in the range $0 \ldots 665672$.

The remaining 3% of unproved goals are all false as far as we can tell. The author of the Simulator case study code had neither the time nor the need to ensure that all goals for all sub-programs were true.

Coverage is obviously sensitive to how timeout values are set: increase the timeout value and often coverage increases too. However, there usually is a timeout value beyond which no further coverage is obtained. For example, with Z3 there is no increase in coverage with a timeout of 20sec rather than 1sec, and both CVC3/API and CVC3/SMT-LIBconverge on proving the same 94.6% of goals at a 20sec timeout.

Tokeneer

The best coverage was obtained with the Yices/API and Z3 configurations. They succeeded in proving all 94.1% of goals originally proven by the Praxis prover, all 2.3% of goals that were originally proven by the interactive Checker tool, as well as 0.6% of the 3.6% proven by manual review. We have inspected the goals unproven by Yices/API and Z3, and in every case it seems there are missing hypotheses, making these goals as stated false. Many of the goals are missing hypotheses characterising specification functions.

Praxis's automatic prover was able to use the rules originally introduced for the interactive prover to increase its coverage by 0.9%. All these goals it newly proved were goals originally proved using the interactive prover.

The goals that Yices and Z3 prove and Praxis's automatic prover misses appear to mostly involve straightforward linear arithmetic and Boolean reasoning. The issue here is that Praxis's prover does not implement decision procedures for linear arithmetic and Boolean reasoning, rather it uses a set of finely-tuned heuristic procedures.
One slightly more interesting example of such a goal is

```
H2   p < (f - 1) div 100 + 1
H3   1 <= f
->
C1:   f - (p - 1) * 100 >= 101
```

The drop in Simplify's coverage compared to that of Z3 is due to a combination of a low timeout, Simplify halting on assertion failure, and the incompleteness introduced by making large constants symbolic. With a timeout of 20sec rather than 1sec, Simplify's coverage increased from 86.4% to 94%. See Section 17.3 for more discussion of the latter 2 issues.

17.2 Run times

Average run times are shown in Table 4 and the distribution of runtimes for the Tokeneer case study is shown in Tables 5 and 6. We make here some general remarks on these results.

It is important not to read too much into the numbers. SMT solvers have many options for selecting alternative heuristics, problem transformations and resource limits, all of which can significantly affect performance. The numbers here are for the default settings of the solvers, which in some cases (e.g. Z3) involve the solver automatically choosing some parameter settings based on the input problem. We have not attempted to tune option settings for the SPARK VCs. In very preliminary investigations, we have found it easy to get factor of two changes in run times. Also, we have made no attempt so far to optimise our tool to reduce the often significant contribution it makes to the overall run times.

Looking at the run-time distributions, CVC3/API is an order of magnitude slower than Yices/API, Yices/SMT-LIB or Z3at most percentiles.

The CVC3/SMT-LIB configuration is significantly faster than the CVC3/API configuration at lower percentiles, but slower at the highest. This is no doubt at least partly due to the different nature of the translations in the two cases. For example, with the API translation, CVC3 can bring to bear specialised handling for the different types in goals. With the SMT-LIB translation, there are many more quantified axioms introduced to characterise the different types, and CVC3 has to fall back on its default heuristics for instantiating these axioms. This might account for the better performance at high percentiles with the API translation.

Yices/API and Yices/SMT-LIB run time distributions are similar, except at the highest run times, maybe again because, with the API, each type can be given individualised treatment.

The performance of Simplify is impressive, especially given its age (the version used dates from 2002) and that it does not employ the core SAT algorithms used in the SMT solvers. Part of this performance edge must be due to the use of fixed-precision integer arithmetic rather than some multi-precision arithmetic package such as *gmp* which is used by Yices and CVC3. We are not sure of why there is a slip in the comparative speed of Simplify on the Tokeneer case study. Perhaps it is related to the higher number of explicit assertions in the Tokeneer code that then results in more complex VCs.

Also too, we observe that Praxis's prover has run times comparable to the best observed with any of the other configurations.

We have carried some preliminary experiments to see what effects the translation options have on SMT-LIB and Simplify run times. So far we see at best relatively small changes in the overall run times. For example, if we use the constructor-select rather than the update-select axiomatisation of records, Z3 runs about 10% faster, but there is little change Yices's run time.

17.3 Soundness

The use of fixed-precision 32-bit arithmetic by Simplify with little or no overflow checking is rather alarming from a soundness point of view. For example, Simplify will claim

```
(IMPLIES
    (EQ x 2000000000)
    (EQ (+ x x) (- 294967296)))
```

to be valid.

As mentioned earlier, when Simplify was used with ESC/Java, an attempt was made to soften the impact of this soundness problem by replacing all integer constants with magnitude above a threshold by symbolic constants. When we tried this approach with a threshold of 100,000, the value suggested in the ESC/Java paper [29], several examples of false goal slices from the Simulator example were asserted to be valid by Simplify. One such slice in essence was

```
H1:    lo >= 0 .
H2:    lo <= 65535 .
H3:    hi >= 0 .
H4:    hi <= 65535 .
H5:    100000 < k200000
       ->
C1:    lo + hi * 65536 <= k200000 .
```

where k200000 is the symbolic constant replacing the integer 200000. These particular goals became unproven with a slightly lower threshold of 50,000.

One indicator of when overflow is happening is when Simplify aborts because of the failure of a run-time assertion left enabled in its code. All the reported errors in the Simplify runs are due to failure of an assertion checking that an integer input to a function is positive. We guess this is due to silent arithmetic overflow. Of course, arithmetic overflow can easily result in a positive integer, so this check only catches some overflows.

We investigated how low a threshold was needed for eliminating the errors with the Simulator VCs and found all errors did not go away until we reduced the threshold to 500.

To get a handle on the impact of using a threshold on provability, we reran the Yices/API test on the Simulator example using various thresholds. With 100,000 the fraction of goals proven by Yices dropped to 90.8%, with 500 to 90.4% and with 20 to 89.6%. Since Yices rejects any additional hypotheses or conclusions which are made non-linear by the introduction of symbolic versions of integer constants, these results indicate that under 2% of the Simulator goal slices involve linear arithmetic problems with multiplication by constants greater than 20.

17.4 Robustness

Over the course of developing our prover interface tool, we have worked with several versions of different provers, and have found some versions prone to generating segmentation faults or running into failed assertions. This was particularly a problem when interfacing to the prover through its API, because every fault would bring down our iteration through the goals of a case study. We resorted to a tedious process of recording goals to be excluded from runs in a special file, with a new entry manually added to this file after each observed crash. Fortunately prover developers are generally responsive to bug reports.

One incentive for running provers in a subprocess is that the calling program is insulated from crashes of the subprocess.

18 Current and future work

One aim of this work is to get the SPARK user community engaged with the latest state-of-the-art provers for their VCs. To this end, we publically released our tool in 2010 under a GPL licence [12]. Also in 2010, Praxis integrated an experimental release of our tool into their SPARK toolset and have distributed it to all their customers. A GPL version of this toolset is now available[13].

Another aim is to provide VC challenge problems to the automated reasoning research community. We provided the Tokeneer VCs in the SMT-LIB format to the 2009 SMT competition, and hope that members of the SPARK user community will in future use our tool to generate further benchmarks.

Next steps in the development of our VCT tool include:

– Extending coverage of the FDL VC language, especially including support for the reals which are currently used for modeling floating-point numbers. Many SPARK users make much use of floating-point arithmetic.

[12] Visit http://homepages.inf.ed.ac.uk/pbj/spark/victor.html

[13] Visit https://libre.adacore.com/libre/tools/spark-gpl-edition/

- Adding support for the SMT-LIB 2.0 format introduced in 2010[14]. This promises to simplify providing support for the reals.
- Improving interfaces for interactive theorem provers. For example, we already have 2 versions of a preliminary interface to the Isabelle theorem prover [36].
- Exploring how to provide proof explanations that are comprehensible by software engineers and that could be used in proof review processes.
- Figuring out how best to present VC counterexamples to SPARK users.
- Adding an alternate front-end preprocessor for VC Units in a more vanilla standardised syntax, so the VCT tool could easily be used with VCs generated from other languages.

We are also working in several directions to improve automation options. These include building translations to the input languages of popular interactive theorem provers, and exploring integrating a variety of existing techniques for proving problems involving non-linear arithmetic [38]. Some of this work is in conjunction with the Z3 development team who have made significant improvements to Z3's non-linear capabilities [34].

19 Conclusions

We have demonstrated that state-of-the-art SMT solvers such as Yices, Z3 and CVC3 are well able to discharge verification conditions arising from SPARK programs. These solvers are able to prove nearly the same VCs as Praxis's prover. Out of the nearly 4000 VCs considered, we found 42 proved by solvers and not Praxis's prover: these highlighted incompletenesses in the heuristic proof strategy employed by Praxis's prover. Many involved simple linear arithmetic and propositional reasoning, We also found one VC discharged by Praxis's prover and not any SMT solver involving non-linear interval arithmetic calculations. We observed average run-times for the fastest of the solvers of roughly $1-2\times$ that of Praxis's prover.

In this article we have described the architecture of our VCT tool for translating VCs into input formats of SMT solvers and for driving those solvers. The translation involves a number of steps such as eliminating array and record types, undertaking data type refinements, and separating formulas and terms. There are a number of options, subtleties and interactions of these steps. We have given a detailed presentation of these steps as a guide to others who wish to implement similar translations, and to encourage discussion of improvements to such translations.

Acknowledgements: Thanks to Angela Wallenburg at Altran Praxis and the anonymous reviewers of this article for their helpful and constructive comments.

References

1. CVC3: an automatic theorem prover for Satisfiability Modulo Theories (SMT). Homepage at `http://www.cs.nyu.edu/acsys/cvc3/`
2. ESC/Java2: Extended Static Checker for Java version 2. Development coordinated by KindSoftware at University College Dublin. Homepage at `http://secure.ucd.ie/products/opensource/ESCJava2/`
3. Arthan, R., Caseley, P., OHalloran, C., Smith, A.: ClawZ: Control laws in Z. In: Third IEEE International Conference on Formal Engineering Methods (ICFEM'00), pp. 169–176 (2000)
4. Arthan, R., Jones, R.B.: Z in HOL in ProofPower. FACS FACTS: Newsletter of BCS Specialist Group for Formal Aspects of Computing Science (2005-1), 39–55 (2005)

[14] `http:www.smtlib.org`

5. Barnes, J.: High Integrity Software: The SPARK approach to safety and security. Addison Wesley (2003)
6. 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)
7. Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# programming system: An overview. In: Post workshop proceedings of CASSIS: Construction and Analysis of Safe, Secure and Interoperable Smart devices, *Lecture Notes in Computer Science*, vol. 3362. Springer (2004)
8. Blaine, L., Goldberg, A.: DTRE - a semi-automatic transformation system. In: Constructing Programs from Specifications, pp. 165–204. Elsevier (1991)
9. Böhme, S., Weber, T.: Fast LCF-style proof reconstruction for Z3. In: M. Kaufmann, L. Paulson (eds.) Interactive Theorem Proving, *Lecture Notes in Computer Science*, vol. 6172, pp. 179–194. Springer (2010)
10. Bouillaguet, C., Kuncak, V., Wies, T., Zee, K., Rinard, M.: Using first-order theorem provers in the Jahob data structure verification system. In: Verification, Model Checking, and Abstract Interpretation (VMCAI), *Lecture Notes in Computer Science*, vol. 4349, pp. 74–88. Springer (2007)
11. Bradley, A.R., Manna, Z.: The Calculus of Computation: Decision Procedures with Applications to Verification. Springer (2007)
12. Conchon, S., Contejean, E., Kanig, J., Lescuyer, S.: Lightweight integration of the ergo theorem prover inside a proof assistant. In: Proceedings of Workshop on Automated Formal Methods (AFM '07) (2007)
13. Cousot, P.: Abstract interpretation. ACM Computing Surveys **28**, 324–328 (1996)
14. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: a theorem prover for proof checking. Journal of the ACM **52**(3), 365–473 (2005)
15. Dijkstra, E.W.: Guarded command, non-determinacy and the formal derivation of programs. Communications of the ACM **18**(8), 453–457 (1975)
16. Dutertre, B., de Moura, L.: The Yices SMT solver (2006). Tool paper at `http://yices.csl.sri.com/tool-paper.pdf`
17. Enderton, H.B.: Introduction to Mathematical Logic. Academic Press (1972)
18. Erkök, L., Matthews, J.: Using Yices as an automated solver in Isabelle/HOL. In: Workshop on Automated Formal Methods (2008)
19. Filliâtre, J.C., Marché, C.: The Why/Krakatoa/Caduceus platform for deductive program verification. In: W. Damm, H. Hermanns (eds.) Computer Aided Verification, 19th International Conference, CAV 2007, *LNCS*, vol. 4590, pp. 173–177. Springer (2007)
20. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended Static Checking for Java. In: PLDI: Programming Language Design and Implementation, pp. 234–245. ACM (2002). For current work, see `http://secure.ucd.ie/products/opensource/ESCJava2/`
21. Good, D.I., Wichmann, B.A.: Mechanical proofs about computer programs. Philosophical Transactions of the Royal Society of London. Series A, Mathematical and Physical Sciences **312**(1522), 389–409 (1984)
22. Grundy, J., Melham, T., Krstić, S., McLaughlin, S.: Tool building requirements for an API to first-order solvers. Electronic Notes in Theoretical Computer Science **144**, 15–26 (2006). Proceedings of the Third Workshop on Pragmatics of Decision Procedures in Automated Reasoning (PDPAR 2005)
23. Guaspari, D.: Penelope, an Ada verification system. In: Tri-Ada '89: Ada technology in context: application, development, and deployment, pp. 216–224. ACM (1989)
24. Hodges, W.: Model Theory. Cambridge University Press (1993)
25. Jackson, P.B., Ellis, B.J., Sharp, K.: Using SMT solvers to verify high-integrity programs. In: J. Rushby, N. Shankar (eds.) Automated Formal Methods, 2nd Workshop, AFM 07, pp. 60–68. ACM (2007). Preprint available at `http://fm.csl.sri.com/AFM07/afm07-preprint.pdf`
26. Jhala, R., Majumdar, R.: Software model checking. ACM Computing Surveys **41**, 21:1–21:54 (2009)
27. King, J.C.: A program verifier. Ph.D. thesis, Carnegie-Mellon University (1969)
28. Kleene, S.C.: Introduction to Meta-Mathematics. North-Holland (1952)
29. Leino, K.R.M., Saxe, J., Flanagan, C., Kiniry, J., et al.: The logics and calculi of ESC/Java2, revision 2060. Tech. rep., University College Dublin (2008). Available from the documentation section of the ESC/Java2 web pages.
30. Luckham, D.C., German, S.M., von Henke, F.W., Karp, R.A., Milne, P.W., Oppen, D.C., Polak, W., Scherlis, W.L.: Stanford pascal verififier user manual. Tech. Rep. CS-TR-79-731, Computer Science Department, Stanford University (1979)
31. McLaughlin, S., Barrett, C., Ge, Y.: Cooperating theorem provers: A case study combining HOL-Light and CVC Lite. Electronic Notes in Theoretical Computer Science **144**(2), 43–51 (2006). Proceedings of the Third Workshop on Pragmatics of Decision Procedures in Automated Reasoning (PDPAR 2005)
32. Menas, T.K., Bouler, J.M., Doner, J.E., Filippenko, I.V., Levy, B.H.: Using SDVS to assess the correctness of Ada software used in the Midcourse Space Experiment. Tech. Rep. ATR-94(4778)-1, Aerospace Corporation (1994)

33. 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)
34. de Moura, L., Passmore, G.O.: Superfluous S-polynomials in strategy-independent Groebner bases. In: 11th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC 2009. IEEE Computer Society (2009)
35. Nelson, G., Oppen, D.C.: Simplification by cooperating decision procedures. ACM Trans. on programming Languages and Systems **1**(2), 245–257 (1979)
36. 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
37. O'Halloran, C., Arthan, R., King, D.: Using a formal specification contractually. Formal Aspects of Computing **9**(4), 349–358 (1997)
38. Passmore, G.O., Jackson, P.B.: Combined decision techniques for the existential theory of the reals. In: 16th Symposuim on the Integration of Symbolic COmputation Mechanised Reasoning (Calculemus 2009), *Lecture Notes in Computer Science*, vol. 5625. Springer (2009)
39. Sannella, D., Tarlecki, A.: Towards formal development of programs from algebraic specifications: implementations revisited. Acta Informatica **25**(3), 233–281 (1988)
40. Turski, W.M., Maibaum, T.E.: The Specification of Computer Programs. Addison-Wesley (1987)
41. Weber, T.: SMT solvers: New oracles for the HOL theorem prover. In: Workshop on Verified Software: Theory, Tools, and Experiments (VSTTE 2009) (2009)
42. Zhang, L., Malik, S.: The quest for efficient boolean satisfiability solvers. In: CAV: Computer Aided Verification, *Lecture Notes in Computer Science*, vol. 2404, pp. 17–36. Springer (2002)