Using SMT solvers to verify high-integrity programs*

Paul B. Jackson
School of Informatics
University of Edinburgh
King's Bldgs
Edinburgh, EH9 3JZ
United Kingdom
Paul.Jackson@ed.ac.uk

Bill J. Ellis
School of Mathematical and Computer Sciences
Heriot-Watt University
Riccarton
Edinburgh, EH14 4AS
United Kingdom
bill@macs.hw.ac.uk

Kathleen Sharp
IBM United Kingdom Ltd
Hursley House, Hursley Park
Winchester, SO21 2JN
United Kingdom
Kathleen.sharp@uk.ibm.com

ABSTRACT

In this paper we report on our experiments in using the currently popular SMT (SAT Modulo Theories) solvers Yices [10] and CVC3 [1] and the Simplify theorem prover [9] to discharge verification conditions (VCs) from programs written in the SPARK language [5]. 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 Praxis High Integrity Systems. These tools include a VC generator and an automatic prover for VCs.

We find that Praxis's prover can prove more VCs than Yices, CVC3 or Simplify because it can handle some relatively simple non-linear problems, though, by adding some axioms about division and modulo operators to Yices, CVC3 and Simplify, we can narrow the gap. One advantage of Yices, CVC3 and Simplify is their ability to produce counter-example witnesses to VCs that are not valid.

This work is the first step in a project to increase the fraction of VCs from current SPARK programs that can be proved automatically and to broaden the range of properties that can be automatically checked. For example, we are interested in improving support for non-linear arithmetic and automatic loop invariant generation.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*assertion checkers, formal methods*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*mechanical verification, assertions, invariants, pre- and post-conditions*

General Terms

Experimentation, Performance, Verification

*This work was funded in part by UK EPSRC Grant GR/S01771/01.

AFM '07, November 6, Atlanta, GA, USA.

Keywords

SMT solver, SAT modulo theories solver, Ada, SPARK

1. INTRODUCTION

SMT (SAT Modulo Theories) solvers combine recent advances in techniques for solving propositional satisfiability (SAT) problems [33] with the ability to handle first-order theories using approaches derived from Nelson and Oppen's work on cooperating decision procedures [24]. The core solvers work on quantifier free problems, but many also can instantiate quantifiers using heuristics developed for the non-SAT-based prover Simplify [9]. 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 and so SMT solvers are well suited to automatically proving VCs.

SMT solvers are currently used to prove Java VCs in the ESC/Java2 [2] tool and C# VCs in the Spec# static program verifier [6]. The Simplify prover was used to prove VCs in the ESC/Modula-3 and original ESC/Java tools.

We evaluate here the current releases of two popular SMT solvers: CVC3 [1] and Yices [10]. Both these systems featured in the 2006 and 2007 SMT-COMP competitions comparing SMT solvers[1] in the category which included handling quantifier instantiation. We also evaluate Simplify because it is highly regarded and, despite its age (the latest public release is over 5 years old), it is still competitive with current SMT solvers.

One advantage that SMT solvers and Simplify have over Praxis's Simplifier is their ability to produce counter-example witnesses to VCs that are not valid. These counter-examples 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.

The work reported here is the first step in a project to improve the level of automation of SPARK VC verification and extend the range of properties that can be automatically verified. Typical properties that SPARK users currently verify are those which check for the absence of run-time exceptions caused by arithmetic overflow, divide by zero, or array bounds violations. Particular new kinds of properties we are interested in include those which involve non-linear

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

arithmetic and which involve significant quantifier reasoning. A key obstacle to improving automation is the need to provide suitable loop invariants. We are therefore interested in also exploring appropriate techniques for automatic invariant generation.

Tackling SPARK programs rather than say Java or C programs is appealing for a couple of reasons. Firstly, there is a community of 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.

Notable earlier work on improving the verification of SPARK programs is by Ellis and Ireland [18]. They used a proof-planning and recursion analysis approach to analyse failed proofs of VCs involving loops to identify how to strengthen loop invariants.

Some success has been had at NASA in using first-order automatic theorem provers for discharging VCs [8]. A major problem with such an approach is the poor support for arithmetic in such provers. This work succeeds by axiomatically characterising a fragment of linear arithmetic that is just sufficient for the kinds of VCs encountered.

Section 2 gives more background on SPARK. Section 3 describes the current implementation of our interface to Yices and CVC3. Case study programs are summarised in Section 4 and Sections 5 and 6 present the results of experiments on the VCs from these programs. Future work, both near term and longer term is discussed in Section 7 and conclusions are in Section 8.

2. THE SPARK LANGUAGE AND TOOLKIT

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 in a language of program annotations for specifying intended properties such as pre and post conditions and assertions. These annotations take the form of Ada comments so SPARK programs are compilable by standard Ada compilers.

A feature of SPARK inherited from Ada particularly relevant for verification purposes is the ability to specify subtypes, types which are subranges of integer, floating-point and enumeration types. For example, one can write:

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

Such types allow programmers to easily include in their programs extra specification constraints without having to supply explicit annotations. It is then possible, either dynamically or statically, to check that these constraints are satisfied.

The Examiner tool from Praxis generates verification conditions from programs. Annotations are often very tedious for programmers to write, so the Examiner can generate automatically some kinds of annotations. For example, it can generate annotations for checking the absence of run-time errors such as array index out of bounds, arithmetic overflow, violation of subtype constraints and division by zero.

As with Ada, a SPARK program is divided into program units, each usually corresponding to a single function or procedure. The Examiner reads in files for the annotated source code of a set of related units and writes the VCs for each unit into 3 files:

- A *declarations* file declaring functions and constants and defining array, record and enumeration types,
- a *rules* file assigning values to constants and defining properties of datatypes,
- 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.

The language used in these files is known as FDL.

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 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. To avoid the need to use the tool or to manually review verification goals, users are often inclined to limit the kinds of program properties they try to verify to ones that can be verified by the Simplifier. They also learn programming idioms that lead to automatically provable goals.

3. SMT SOLVER INTERFACE

Our interface program reads in the VC file triples output by the Examiner tool and generates a series of *goal slices*. A goal slice is a single conclusion packaged with its associated hypotheses, rules and declarations. After suitable processing, each goal slice is passed in turn to a selected prover, at present one of Yices, CVC3 or Simplify. The processing includes:

- Handling enumerated types.

  The Examiner generates rules that define each enumerated type as isomorphic to a subrange of the integers. Explicit functions defining the isomorphism are declared and the rules for example relate order relations and successor functions on the enumeration types to the corresponding relations and functions on the integers.

  We experiment with 3 options:

  1. Mapping each FDL enumerated type to the enumerated type of the SMT solver. We keep the rules since neither solver publically supports ordering of the elements of enumerated types.
  2. Mapping each FDL enumerated type to an abstract type in the SMT solver, so the solver has to rely fully on the supplied enumeration type rules to reason about the enumeration constants.
  3. Defining each enumeration type as an integer subrange, defining each type element as equal to an appropriate integer, and eliminating all rules.

The last option generally gives the best performance, but the first two are better from a counter-example point of view: counter-examples involving the enumeration types are more readable since they use the enumeration constants rather than the corresponding integers.

- Resolving overloaded functions and relations.

  For example, FDL uses the same symbols for the order relations and successor functions on all types. Resolution involves inferring types of sub-expressions and type checking the FDL.

- Inferring types of free variables in rules and adding quantifiers to close the rules.

  FDL is rather lax in declaring types of these variables and so types must be inferred from context.

- Resolving the distinction between booleans as individuals and booleans as propositions. The FDL language uses the same type for both, as does Yices. However, CVC3 and Simplify take a stricter first-order point of view and require the distinction.

- 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 and we needed to make these explicit.

- Reordering type declarations so types are defined before they are used. Such an ordering is not required in FDL, but is needed by Yices and CVC3.

We carry out all the above processing in a prover independent setting as much as possible in order to keep the driver code for individual provers compact and ease the development of further drivers.

The match between the FDL language of the SPARK VCs and the input languages of both Yices and CVC3 is good. The FDL language includes quantified first-order formulae and datatypes including the booleans, integers, reals, enumerations, records and arrays, all of which are supported by both solvers.

At present we are interfacing to Yices and CVC3 using their C and C++ APIs respectively. An alternative is to write goal slices to files in the specific input languages of the respective solvers and call the solvers in sub-processes. We take this latter approach with Simplify since it does not have a readily-available C or C++ API.

Simplify's input language is rather different from FDL. All individual expressions in Simplify are untyped except for those which are arguments to arithmetic relations or are the arguments or results of arithmetic operations - these expressions are of integer type. We handle both arrays and records using Simplify's built-in axiomatic theory of maps. For example, one axiom is stated as:

```
(FORALL (m i x)
  (PATS (select (store m i x) i))
  (EQ (select (store m i x) i) x)
)
```

Here the `PATS` expression is a hint used to suggest to Simplify a sub-expression to use as a trigger pattern when searching for matches that could provide instantiations. Handling enumeration types is straightforward: the FDL generated by Praxis's Examiner provides enough inequality predicates bounding enumeration type values to allow the interpretation of enumeration types themselves as integers. A problem with Simplify is that all arithmetic is fixed precision. We follow the approach taken when Simplify is used with ESC/Java where all constants with magnitude greater than some threshold are represented symbolically and axioms are asserted concerning how these constants are ordered [21].

To aid in analysis of results, we provide various options for writing information to a log file, as well as writing comma-separated-value run summaries. These allow easy comparison between results from runs with different options and solvers.

4. CASE STUDY SPARK PROGRAMS

For our experiments we work with two 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[2] under the GNU General Public Licence.

Some brief statistics on each of these examples and the corresponding verification conditions are given in Table 1. The

	Autopilot	Simulator
Lines of code	1075	19259
No. units	17	330
No. annotations	17	37
No. VC goals	133	1799
No. VC goal slices	576	6595

Table 1: Statistics on Case Studies

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 is the number of SPARK precondition and assertion annotations in all the Ada specification and body files. No postconditions were specified in either example.

In both cases the VCs are primarily from exception freedom checks, e.g. checking that arithmetic values and array indices are always appropriately bounded.

The VCs from both examples involve enumerated types, linear and non-linear integer arithmetic, integer division and uninterpreted functions. In addition, the Simulator example includes VCs with records, arrays and the modulo operator.

5. EXPERIMENTAL RESULTS

Results are presented for the tools

- Yices 1.0.9,

- CVC3 development version 20071001,

- Simplify 1.5.4,

[2] `http://www.suslik.org/Simulator/index.html`

	Yices		CVC3		Simplify		Simplifier	
proven	510	88.5%	518	89.9%	516	89.6%	572	99.3%
unproven	66	11.5%	58	10.1%	60	10.4%	4	0.7%
timeout	0	0 %	0	0 %	0	0 %	0	0 %
error	0	0 %	0	0 %	0	0 %	0	0 %
total	576		576		576		576	

Table 2: Coverage of Autopilot goal slices

	Yices		CVC3		Simplify		Simplifier	
proven	6004	91.0%	6074	92.1%	5940	90.1%	6410	97.2%
unproven	591	9.0%	337	5.1%	646	9.8%	185	2.8%
timeout	0	0 %	184	2.8%	0	0 %	0	0 %
error	0	0 %	0	0 %	9	0.1%	0	0 %
total	6595		6595		6595		6595	

Table 3: Coverage of Simulator goal slices

- Simplifier 2.32, part of the 7.5 release of Praxis's tools.

We needed a development version of CVC3 as the latest available release (1.2.1) had problems with a significant fraction of our VCs.

Unless otherwise stated, the experiments were run on a 3GHz Pentium 4D CPU with 1GB of physical memory running Linux Fedora Core 6. We used the translation option to eliminate the enumeration type occurrences, as this yields the best performance. With Simplify, the threshold for making constants symbolic was set at 100,000.

The coverage obtained with each tool is summarised in Tables 2 and 3. The tables show the results of running Yices, CVC3, Simplify and Praxis's Simplifier on each goal slice from the VCs in each of the case studies. The *proven* counts are for when the prover returned claiming that a goal is true. The *unproven* counts are for when the prover returned without having proven the goal. The *timeout* counts are for when the prover reached a time or resource limit. The only limit reached in the tests was a resource limit for CVC3: CVC3 usually reached the set limit of 100,000 in 8-10 sec. This limit was only specified for the CVC3 runs on the Simulator VCs. The *error* counts are for when the prover had a segmentation fault, encountered an assertion failure or diverged, never reaching a resource limit after a few minutes.

To indicate the performance of each prover, we have gathered and sorted run times of each prover on each goal slice. Table 4 shows the run times at a few percentiles. 'TO' indicates that the timeout resource limit was reached. With the CVC3 Simulator runs, this was reached at the 97% level.

	Autopilot			Simulator		
	Yices	CVC3	Smpfy	Yices	CVC3	Smpfy
50%	.01	.02	.01	.01	.05	.04
90%	.02	.04	.02	.03	.11	.07
99%	.02	4.75	.03	.04	TO	.08
99.9%				.16		.08
max.	.03	17.20	.03	.56	>10	.09

Table 4: Run time distributions (times in sec.)

Numbers are not given for the Simplifier in this table as it does not provide a breakdown of its run time on individual goal slices.

Table 5 provides a comparison of the total run times of each prover on all goal slices. These times also include the overhead time for reading in the various VC files and suitably translating them. For Yices and CVC3, this overhead is at most a few percent.

	Autopilot	Simulator
Yices	5.63	109.6
CVC3	83.11	2928.0
Simplify	8.09	293.1
Simplifier	6.51	226.9

Table 5: Total run times (sec)

The number for Praxis's Simplifier running on the Simulator goal slices is an estimate based on running a Solaris version of the Simplifier on a slower SPARC machine and obtaining a scaling factor by running both Solaris and Linux versions on the Autopilot goal slices. Praxis only release a Linux version of the Simplifier for demonstration purposes, and this version cannot handle the size of the Simulator example.

Both the total and individual goal slice times for Simplify appear to be significant over-estimates of the time spent executing Simplify's code. For example, a preliminary investigation shows that for only 15-30% of the total run times for the Simplify tests is the CPU in user mode executing the child processes running Simplify. Much of the rest of the time seems to be spent in file input/output (communication with Simplify is via temporary files) and sub-process creation and clean-up.

6. DISCUSSION OF RESULTS

Coverage.

All the goal slices unproven by Yices, CVC3 or Simplify, but certified true by Praxis's Simplifier, involve non-linear arithmetic with some combination of non-linear multiplication, integer division and modulo operators. These slices nearly all involve checking properties to do with bounds on the values of expressions. It is fairly simple to see these properties are true from considering elementary bounding properties of arithmetic functions, for example, that the product of two non-negative values is non-negative. See the section below on *incompleteness* for a discussion of an experiment

involving adding axioms concerning bounding properties.

The 4 Autopilot goal slices unproven by Simplifier are all true. They all have similar form: for example, one goal slice in essence in FDL syntax is:

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

The 2.8% Simulator goal slices unproven by Simplifier appear to be nearly all false. They are derived from 47 of the 330 Simplifier sub-programs. The author of the Simulator case study code had neither the time nor the need to ensure that all goal slices for all sub-programs were true.

The slightly better coverage obtained with CVC3 over Yices on the Autopilot example is partly because Yices prunes any hypothesis or conclusion with a non-linear expression, whereas CVC3 accepts non-linear multiplication and knows some properties of it. For example, it proved the goal slices:

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

Run times.

As can be seen from the distributions, Yices, CVC3 and Simplify all have run times within a factor of 5 of each other on many problems. CVC3's total run times at 15-20× those of Yices seems to be due to it trying for longer on problems where it returns *unproven* or *timeout* results: all but 16 of the problems it proves are proven in under 0.11s.

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 package such as *gmp* which is used by Yices and CVC3. Also, the goal slices typically have a simple propositional structure and it seems that relatively few goals involve instantiating quantifiers which brings in more propositional structure.

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 happily prove:

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

As mentioned earlier, an attempt to soften the impact of this soundness when Simplify was used with ESC/Java involved replacing all integer constants with magnitude above a threshold by symbolic constants. When we tried this approach with a threshold of 100,000, several examples of false goal slices were certified 'true' by Simplify. 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 an assertion failure. 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 like in the above example. 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 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.

Timeouts.

To enable working through large sets of problems in reasonable times, it is very useful to be able to interrupt runs after a controllable interval. We found the resource limit of CVC3 allowed fairly reliable stopping of code.

The Yices developers recommended an approach involving using timer interrupts and setting a certain variable in the interrupt handler. However, this feature depended on using an alternate API to the one we used, and this alternate API did not support the creation of quantified formulae. Fortunately, we have not yet needed a timeout feature with our runs of Yices.

We did implement a timer-driven interrupt feature for stopping subprocesses which might be useful for stopping Simplify at some in the future.

Robustness.

When working with an earlier version of CVC3, we had significant problems with it generating segmentation faults and diverging. Because of our interface to CVC3 through its API, every fault would bring down our iteration through the goal slices of one of the examples. We resorted to a tedious process of recording goal slices to be excluded from runs in a special file, with a new entry manually added to this file after each observed crash or divergence. Fortunately the CVC3 developers are responsive to bug reports.

We have found Yices pleasingly stable: to date we have observed only one case in which it has generated a segmentation fault. (This occurred in an experiment not reported in the figures here.)

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

Incompleteness.

As a first step towards improving the coverage possible with Yices, CVC3 and Simplify, we added axiomatic rules describing the bounding properties of the integer division

	Yices		CVC3		Simplify	
proven	554	96.2%	554	96.2%	560	97.2%
unproven	22	3.8%	0	0 %	16	2.8%
timeout	0	0 %	12	2.1%	0	0 %
error	0	0 %	10	1.7%	0	0 %
total	576		576		576	

Table 6: Autopilot coverage with div & mod rules

	Yices		CVC3		Simplify	
proven	6216	94.3%	6256	94.9%	6045	91.7%
unproven	379	5.7%	108	1.6%	388	5.9%
timeout	0	0 %	231	3.5%	0	0 %
error	0	0 %	0	0 %	162	2.5%
total	6595		6595		6595	

Table 7: Simulator coverage with div & mod rules

and modulo operators:

$$\forall x, y : \mathbf{Z}.\ 0 < y \ \Rightarrow\ 0 \leq x \bmod y$$
$$\forall x, y : \mathbf{Z}.\ 0 < y \ \Rightarrow\ x \bmod y < y$$
$$\forall x, y : \mathbf{Z}.\ 0 \leq x \ \wedge\ 0 < y \ \Rightarrow\ y \times (x \div y) \ \leq\ x$$
$$\forall x, y : \mathbf{Z}.\ 0 \leq x \ \wedge\ 0 < y \ \Rightarrow\ x - y \ <\ y \times (x \div y)$$
$$\forall x, y : \mathbf{Z}.\ x \leq 0 \ \wedge\ 0 < y \ \Rightarrow\ x \ \leq\ y \times (x \div y)$$
$$\forall x, y : \mathbf{Z}.\ x \leq 0 \ \wedge\ 0 < y \ \Rightarrow\ y \times (x \div y) \ <\ x + y$$

Rules characterising $\div$ and mod when their second argument is negative can also be formulated, but these were not needed for our examples.

The coverage obtained with these additional rules is shown in Tables 6 and 7.

The observed total run times of the provers were 15-30% slower than without the additional rules.

The extra goal slices proven nearly all involve integer division with a constant divisor. Such instances of division yield instances of the axioms with linear multiplications which the provers can then work with. Most remaining unproven goal slices that were proved true by the Simplifier involved non-constant divisors or non-linear multiplications. We have experimented a little with adding in axioms involving inequalities and non-linear multiplication, but so far have not had much success. A problem is phrasing the axioms so that the instantiation heuristics, perhaps guided by explicit identification of sub-expressions to use for matching, can work productively.

7. FUTURE WORK

7.1 Near term work with SMT solver interface

One objective in the next month or two is to get the interface code into a state in which we can publically release it.

We expect that the main initial use will be in exploiting the counter-example capability to debug code and specifications. SPARK users would be reluctant to trust direct judgement provided by SMT solvers on goal slice truth. However Praxis's interactive prover has been through some certification process and a worthwhile sub-project would be to translate proof objects output by e.g. CVC3 into commands for the interactive prover. Ellis and Ireland in their work also generated proof scripts for the interactive prover from their proof plans that successfully proved VCs.

Another objective is to establish an automatic means for translating SPARK VCs into the SMT-LIB format. This ought to be straightforward given CVC3's capabilities for dumping problems in this format. This would provide a useful way of augmenting the SMT-LIB with the VCs from SPARK code examples such as the Simulator and Autopilot used in our evaluation.

We also eventually would like to try further examples. A problem is the dearth of medium or large SPARK examples in the public domain. Praxis have access in house to some suitable interesting examples, and we hope through collaboration and site visits to also experiment with these.

7.2 The larger picture

Three areas we are considering exploring are non-linear arithmetic, improved quantifier support and automatic invariant generation. Improvements in these areas would be of significant help in increasing the level of automation of VC proof, especially for VCs coming from typical SPARK applications.

Currently we have identified some major lines of relevant work in each area, but have not yet narrowed down on which approaches would be most productive to pursue.

Below we survey some of the literature we have come across on reasoning in these areas and speculate on architectures we might adopt for a full verification system.

7.2.1 Non-linear arithmetic

We are interested in being able to prove problems involving non-linear arithmetic over both the integers and reals. Arithmetic on the reals is of interest for several reasons. Real problems are easier to decide than integer problems and the kinds of integer arithmetic problems that frequently come up in VCs are often also true over reals. Often algorithms for solving integer problems extend algorithms for real problems. For example, this is the case with integer linear programming and mixed integer real non-linear programming. Real arithmetic is also of interest because it is often used to approximate floating-point arithmetic.

The theory of real closed fields (first order formulae over equalities and inequalities involving polynomials over reals) is decidable and decision procedures involving cylindrical algebraic decomposition are under active development [16]. These procedures have high time complexities and are usually only practical on small problems. As we have not yet identified programs yielding interesting VCs in this class, it is difficult to say whether such procedures could be useful to us.

There is much promising work on incomplete proof techniques for quantifier free problems involving polynomials over real numbers. These techniques are observed to be sufficient for problems that come up in practice that are significantly larger than can be handled by complete techniques. For example, Tiwari has investigated using Gröbner bases [30], Parrilo uses sum of squares decompositions and semi-definite programming (a non-linear extension of linear programming) [26] and the ACL2 theorem prover has extensions to support some non-linear resoning [17].

Akbarpour and Paulson are currently exploring heuristically reducing problems involving functions such as sine, ex-

ponentials and logarithms to real closed field problems [3].

We have not come across specific work addressing reasoning with integer division and modulo operators. Since these operators can be fully characterised in terms of integer addition and multiplication with a few first order axioms, we hope that it might be possible to make significant headway with some appropriate combination of first-order reasoning and reasoning about integer polynomial arithmetic. Any such techniques will almost certainly be heuristic in nature, since the problem of solving equations involving polynomials with integer coefficients (Diophantine equations) is undecidable.

7.2.2 Quantifiers

The support for first-order reasoning in the Simplify prover and SMT solvers such as Yices and CVC3 is certainly very useful, but it falls far short of what automated first-order provers are capable of.

Integrating refutation complete first-order provers with reasoning in specific theories such as integer linear arithmetic is known to be a very hard problem.

A promising new direction in first order theorem proving research is that of applying *instantiation-based methods* [13]. Here the aim is to produce refutation complete procedures for first order logic which work by running a SAT solver on successively larger ground instantiations of first-order problem. Given that SMT solvers also use SAT solver algorithms at heart, a natural question that several have asked is whether instance-based and SMT algorithms could be fruitfully combined. Such a combination may well be substantially incomplete, but could still be very useful in practice.

7.2.3 Invariant generation

The need and opportunities for automatic support in the inference of loop invariants have long been recognised [32]. In the last decade there has been a revival in interest in the problem [7, 31].

Flanagan and Leino [11] demonstrate a lightweight generate and test approach for the ESC/Java system: a large number of candidate annotations are generated, inspired by the program structure, and the VC prover then prunes these down.

The technique of *predicate abstraction* [14], a form of abstract interpretation, has been very useful in software model checkers such as Microsoft's SLAM [4] and Berkeley's BLAST [15]. Flanagan and Qadeer [12] explain how to use predicate abstraction to generate loop invariants. An interesting feature of their work is the ability to infer loop invariants with quantifiers, something often necessary when verifying programs involving arrays.

Leino and Logozzo [20] use failure of the VC prover to guide the refinement of conjectured loop invariants just for those program executions associated with the failure. Whereas this work employs abstract interpretation, McMillan [22] achieves a somewhat similar end through the use of Craig interpolants.

Nearly all the above cited work focusses on invariants involving only linear arithmetic expressions. Recently Gröbner basis techniques have been used for handling polynomial arithmetic expressions over the reals [28, 19].

As mentioned in the introduction, Ellis and Ireland [18] have used proof-planning to identify how to strengthen loop invariants.

7.2.4 Verification system architectures

How can different approaches to proving VCs be successfully integrated? One approach is to use a theorem proving environment for programming strategies for refining proof goals and for interfacing to individual provers such as SMT provers, non-linear arithmetic provers and first-order provers. Theorem proving environments allow for rapid experimentation with proof strategies and already have many individual provers of interest either linked in or built in. This approach of using a programmable theorem proving environment as a hub prover was advocated in the PROSPER project [23] and is employed in the Forte system at Intel [29]. Theorem proving environments that look appealing for such a role include Isabelle [25], HOL, HOL Light and PVS.

A finer grain approach might be to investigate adding new procedures as extra theory solvers within an SMT solver. For example, maybe a non-linear arithmetic procedure could be integrated into an SMT solver.

Exploring techniques for invariant generation will require program analysis capabilities and collaboration between a variety of different reasoning tools. SRI have proposed an *evidential tool bus* as an architecture for linking together verification components [27]. They have expressed a specific interest in it being used for generating invariants and supporting software verification. It would be very interesting if we could make use of this and possibly assist in its development.

8. CONCLUSIONS

We have presented some preliminary encouraging results in using the state-of-the-art SMT solvers Yices and CVC3 and the Simplifier prover to discharge verification conditions arising from SPARK programs.

Around 90% of the problems we examined involved no non-linear arithmetic reasoning and were usually solved in under 0.1s by all tools. Another 3-7% were solvable when simple axioms were added concerning bounding properties of the modulo and integer division operators. Many of the remaining true problems are of a slightly more essential non-linear character and are beyond what is provable by the tools even with these axioms. However these problems are mostly still easy to see true, and Praxis's Simplifier prover appears able to handle them in most cases.

We expect shortly to publically release the code we have developed so SPARK users can experiment with it. Our code will also soon provide an easy way of producing SMT challenge problems in the standard SMT-LIB format from SPARK program VCs.

Longer term, we see this work as a first step in a research programme to improve the level of automation in the formal verification of programs written in SPARK and SPARK like subsets of other programming languages.

9. 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] B. Akbarpour and L. C. Paulson. Towards automatic proofs of inequalities involving elementary functions. In *PDPAR: Workshop on Pragmatics of Decision Procedures in Automated Reasoning*, 2006.

[4] T. Ball, B. Cook, V. Levin, and S. K. Rajamani. SLAM and Static Driver Verifier: technology transfer of formal methods inside Microsoft. Technical Report MSR-TR-2004-08, Microsoft Research, 2004. The SDV homepage is `http://www.microsoft.com/whdc/devtools/tools/sdv.mspx`.

[5] J. Barnes. *High Integrity Software: The SPARK approach to safety and security*. Addison Wesley, 2003.

[6] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *Post workshop proceedings of CASSIS: Construction and Analysis of Safe, Secure and Interoperable Smart devices*, volume 3362 of *Lecture Notes in Computer Science*. Springer, 2004.

[7] S. Bensalem and Y. Lakhnech. Automatic generation of invariants. *Formal Methods in System Design*, 15:75–92, 1999.

[8] E. Denney, B. Fischer, and J. Schumann. An empirical evaluation of automated theorem provers in software certification. *International Journal of AI tools*, 15(1):81–107, 2006.

[9] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: a theorem prover for proof checking. *Journal of the ACM*, 52(3):365–473, 2005.

[10] B. Dutertre and L. de Moura. The Yices SMT solver. Tool paper at `http://yices.csl.sri.com/tool-paper.pdf`, August 2006.

[11] C. Flanagan and K. R. M. Leino. Houdini, an annotation assistant for ESC/Java. In *FME: International Symposium of Formal Methods Europe*, volume 2021 of *Lecture Notes in Computer Science*, pages 500–517. Springer, 2001.

[12] C. Flanagan and S. Qadeer. Predicate abstraction for software verification. In *POPL: Principles of Programming Languages*, pages 191–202. ACM, 2002.

[13] H. Ganzinger and K. Korovin. New directions in instantiation-based theorem proving. In *LICS: Logic in Computer Science*, pages 55–64. IEEE, 2003.

[14] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *CAV: Computer Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83. Springer, 1997.

[15] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with BLAST. In *SPIN: workshop on model checking software*, volume 2648 of *Lecture Notes in Computer Science*, pages 235–239. Springer, 2003.

[16] H. Hong and C. Brown. QEPCAD: Quantifier elimination by partial cylindrical algebraic decomposition, 2004. See `http://www.cs.usna.edu/~qepcad/B/QEPCAD.html` for current implementation.

[17] W. A. Hunt, Jr., R. B. Krug, and J. Moore. Linear and nonlinear arithmetic in ACL2. In *CHARME: Correct Hardware Design and Verification Methods*, volume 2860 of *Lecture Notes in Computer Science*, pages 319–333. Springer, 2003.

[18] A. Ireland, B. Ellis, A. Cook, R. Chapman, and J. Barnes. An integrated approach to high integrity software verification. *Journal of Automated Reasoning*, 36(4):379–410, 2006.

[19] L. I. Kovács and T. Jebelean. An algorithm for automated generation of invariants for loops with conditionals. In *SYNASC: Symbolic and Numeric Algorithms for Scientific Computing*. IEEE, 2005.

[20] K. R. M. Leino and F. Logozzo. Loop invariants on demand. In *APLAS: Programming Languages and Systems, Third Asian Symposium*, volume 3780 of *Lecture Notes in Computer Science*, pages 119–134. Springer, 2005.

[21] K. R. M. Leino, J. Saxe, C. Flanagan, J. Kiniry, et al. The logics and calculi of ESC/Java2, revision 1.10. Technical report, University College Dublin, November 2004. Available from the documentation section of the ESC/Java2 web pages.

[22] K. L. McMillan. Lazy abstraction with interpolants. In *CAV: Computer Aided Verification*, volume 4144 of *Lecture Notes in Computer Science*, pages 123–136. Springer, 2006.

[23] T. F. Melham. PROSPER: an investigation into software architecture for embedded proof engines. In *FRoCoS: Frontiers of Combining Systems*, volume 2309 of *Lecture Notes in Artificial Intelligence*, pages 193–206. Springer, 2002.

[24] G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. on programming Languages and Systems*, 1(2):245–257, 1979.

[25] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002. See `http://www.cl.cam.ac.uk/research/hvg/Isabelle/` for current information.

[26] P. A. Parrilo. Semidefinite programming relaxations for semialgebraic problems. *Mathematical Programming*, 96(2):293–320, 2003.

[27] J. Rushby. Harnessing disruptive innovation in formal verification. In *SEFM: Software Engineering and Formal Methods*. IEEE, 2006.

[28] S. Sankaranarayanan, H. B. Sipma, and Z. Manna. Non-linear loop invariant generation using Gröbner bases. In *POPL: Principles of Programming Languages*, pages 318–329. ACM, 2004.

[29] C.-J. Seger, R. B. Jones, J. W. O'Leary, T. Melham, M. D. Aagaard, C. Barrett, and D. Syme. An industrially effective environment for formal hardware verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(9):1381–1405, 2005.

[30] A. Tiwari. An algebraic approach for the unsatisfiability of nonlinear constraints. In *CSL: Computer Science Logic*, volume 3634 of *Lecture Notes in Computer Science*, pages 248–262. Springer, 2005.

[31] A. Tiwari, H. Rueß, H. Saïdi, and N. Shankar. A technique for invariant generation. In *TACAS: tools and algorithms for the construction and analysis of systems*, volume 2031 of *Lecture Notes in Computer Science*, pages 113–127. Springer, 2001.

[32] B. Wegbreit. The synthesis of loop predicates. *Communications of the ACM*, 17(2), 1974.

[33] L. Zhang and S. Malik. The quest for efficient boolean satisfiability solvers. In *CAV: Computer Aided Verification*, volume 2404 of *Lecture Notes in Computer Science*, pages 17–36. Springer, 2002.