Using Answer Set Programming in the Development of Verified Software

Florian Schanda[1] and Martin Brain[2]

1 Altran Praxis Limited
20 Manvers Street, Bath, BA1 1PX, UK
florian.schanda@altran-praxis.com

2 Department of Computer Science*
University of Oxford, Oxford, OX1 3QD, UK
martin.brain@cs.ox.ac.uk

Abstract

Software forms a key component of many modern safety and security critical systems. One approach to achieving the required levels of assurance is to prove that the software is free from bugs and meets its specification. If a proof cannot be constructed it is important to identify the root cause as it may be a flaw in the specification or a bug. Novice users often find this process frustrating and discouraging, and it can be time-consuming for experienced users. The paper describes a commercial application based on Answer Set Programming called Riposte. It generates simple counter-examples for false and unprovable verification conditions (VCs). These help users to understand why problematic VC are false and makes the development of verified software easier and faster.

1998 ACM Subject Classification F.3.1 Specifying and Verifying and Reasoning about Programs

Keywords and phrases Answer Set Programming, verification, SPARK, Ada, contract based verification, safety critical

Digital Object Identifier 10.4230/LIPIcs.ICLP.2012.72

1 Introduction

A critical system is one whose failure would cause serious injury, one or more fatalities, major environmental damage or significant damage to other assets. Software is a component of many critical systems and is playing an ever increasing role in their monitoring and control. For example in modern aircraft, both civil and military, there are complex flight control systems which must never 'go wrong'. Errors in algorithms may cause wrong behaviour; software crashes may result in catastrophic failures. Part of the argument for the safety of a system is verification – showing that the system meets its specification. For software the specification may include functional properties (things the system must do) and erroneous behaviour (things that the system must not do). Testing may be sufficient to show functional properties (i.e. the system can track flights) but is not able to guarantee the absence of errors – for example testing alone cannot show that a system will never crash. Critical systems require a higher level of assurance, formal verification systems, such as SPARK[1] can provide this kind of certainty.

* Work conducted while at University of Bath and Altran Praxis.
[1] The SPARK programming language is not sponsored by or affiliated with SPARC International Inc. and is not based on the SPARC™ architecture.

One of the major barriers to increasing the commercial adoption of formal verification is the perception that it is too expensive. Although formal development practices and verification have been shown to reduce total project costs [13, 21] as well as to increase levels of assurance, many companies focus only on the initial development time. "Programmers find verification hard, so it takes them longer and thus costs us more" is a common objection. This misses the wider context, but addressing this misapprehension is crucial to improving adoption. One route to doing so is to improve support tools for developing verified software. Given that developer time is at least 100 to 1000 times more expensive than CPU time, significant computation resources can be justified if they save developers' time.

The current proof tools for SPARK focus on the primary goal of quickly and easily discharging verification conditions (VCs). The proof of all verification conditions shows a number of properties about the system: for example that certain kinds of error cannot occur (for example buffer overruns), or that some security property holds (for example: only one door of an airlock must be open at any time).

There is only limited support for distinguishing between VCs that are unprovable due to incompleteness in the proof tools and those that are unprovable because they are false. When verifying finished and correct software, this is of little importance. However during development a significant minority of VCs may be unprovable. Distinguishing bugs (in specification or implementation) from areas of incompleteness is vital as the resolutions for each are very different and incorrectly classifying a verification failure can waste time and potentially introduce unsoundness (depending on the processes around the usage of the tool). Riposte is a tool based on Answer Set Programming (ASP) that supports developers in classifying and resolving verification failures by generating concrete counter-examples to false verification conditions. This paper:

- Overviews the architecture of Riposte and its usability features which are intended to produce more insightful counter-examples (Section 3).
- Describes the methodology used and experience gained in developing a commercial tool using ASP and the more unusual features of the problem encodings used by Riposte (Section 4).
- Gives statistics for the typical problem instances generated by Riposte and compares Riposte's performance to that of SMT solvers for analysing erroneous programs (Section 5).

2 SPARK

SPARK is a language and supporting toolset[2]; the primary design goal is the provision of an unambiguous language semantics and a sound verification system based on Hoare logic and theorem proving. The executable part of the language is a subset of Ada (83, 95 and 2005) and data flow and correctness contracts are given in Ada comments. Figure 2 shows a very simple SPARK program which will serve as an example throughout this paper, and Figure 1 illustrates the current architecture of the tool chain, with the phases of computation, flow of information and outputs. There are three key phases, referred to as examine, prove and summarise.

The 'front end' of the SPARK toolset is the Examiner. It checks the program for compliance with the SPARK subset, performs data flow analysis and generates VCs for each path between cut points (subprogram start and end, loops and assertions). VCs check contracts specified

[2] Available under the GPL from `http://libre.adacore.com/`

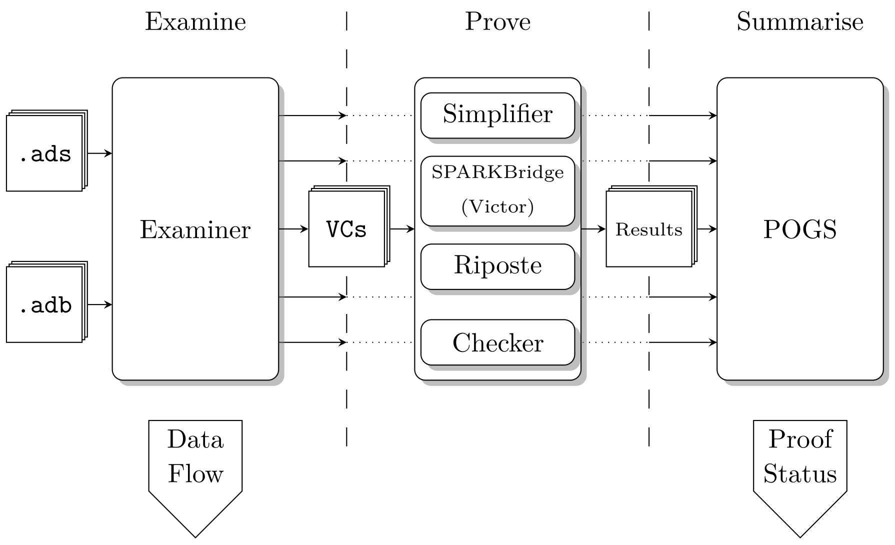

Figure 1 The SPARK tool chain.

by the user and freedom from run time exceptions such as integer overflow, array bounds checks and division by zero. Figure 3 shows a VC generated from the previous example program. VCs are expressed in functional description language (FDL), a simple intermediate language, and a variety of proof tools are available to discharge them. These include the Simplifier, a rewrite based automatic theorem prover; Victor, an SMT translator and prover driver [22] and the Checker, an interactive theorem prover. An Isabelle plug-in to read SPARK VCs [3] is also available. Finally the POGS tool is used to summarise the state of the proof of the entire system, giving statistics such as how many VCs there are in the system and how many of them are discharged.

SPARK is a mature system with the first version released in March 1987, and the SPARK tools have been used on a variety of industrial projects including applications such as flight control, rail signalling, and high-grade cryptographic systems.

SPARK places particular emphasis on modularity; this means it is common to verify software as it is being written, well before it is completed. Thus subprograms first analysed by the Examiner often contain errors and give undischarged VCs. Proof tools in earlier versions of SPARK did not distinguish between those VCs that are undischarged due to incompleteness and those undischarged because they are false. The resolution for these two kinds of failure are very different and misclassification can waste time and potentially introduce unsoundness. So there is a need for a counter-example generation tool to support users in locating the causes of verification failures.

3 Riposte architecture

Riposte consists of a front-end (implementing the parsing of FDL, interval reasoning, simple rewrite and the user interface) and a back-end (that is used to perform the actual search for counter-examples). The front-end generates an *AnsProlog* program for each conclusion analysed. The back-end is a further set of rules included by each program which encode the semantics of FDL. This program is then passed to an answer-set solver and any model returned by the answer-set solver represents a counter-example, which is then interpreted by

```
type Unsigned_Byte is new Integer range 0 .. 255;

function Add_UB (A, B: Unsigned_Byte)
                return Unsigned_Byte
--# return Value => (Value > A);
is
begin
   return A + B;
end Add_UB;
```

Figure 2 Example SPARK subprogram with several bugs. The line starting with `-#` is a SPARK contract specifying a postcondition for the function.

```
function_add_ub_2.
H1:     true .
H2:     a >= unsigned_byte__first .
H3:     a <= unsigned_byte__last .
H4:     b >= unsigned_byte__first .
H5:     b <= unsigned_byte__last .
H6:     a + b >= unsigned_byte__base__first .
H7:     a + b <= unsigned_byte__base__last .
         ->
C1:     a + b > a .
C2:     a + b >= unsigned_byte__first .
C3:     a + b <= unsigned_byte__last .
```

Figure 3 An interesting VC for the code from Figure 2. H2 - H5 are the hypotheses that give the bounds for a and b. H6 and H7 state that $a+b$ will not overflow the base type for Unsigned_Byte, in our case this is a 32-bit signed integer. C1 is the proof obligation required to show the postcondition of the function (as specified by the user); C2 and C3 are required to show absence of run-time errors as the addition may overflow the range allowed for Unsigned_Byte (this proof obligation is automatically generated by the Examiner).

the front-end and expressed in a user-friendly way. Figure 4 shows the overall architecture of Riposte, where `gringo` is the grounder and `clasp` the answer-set solver of the Potassco [18] tool-chain.

Riposte is designed to be *sound* but *not complete*, thus an absence of a model guarantees that a given conclusion is necessarily true. However, there may be spurious counter-examples generated by Riposte (in particular in the presence of complex quantified expressions). To mitigate this Riposte also attempts to check each counter-example to determine if it is a valid counter-example.

The back-end of Riposte contains approximately 4,700 lines of *AnsProlog*describing 1,000 rules; the front-end is around 12,000 lines of Python. To our knowledge it is one of the largest commercial deployment of an answer set program to date.

3.1 Rewrite

After parsing, Riposte performs a number of rewrites and simplifications. These include putting the hypotheses and conclusions in prenex normal form, Skolemisation to remove existential quantifiers and elimination of redundant quantifiers. Normalising expressions makes the subsequent processing simpler and can make the problems easier.

3.2 Interval reasoning

The integers in FDL are mathematical and thus ‘infinite precision’. However in the VCs generated from SPARK programs every program variable is bounded. These bounds are

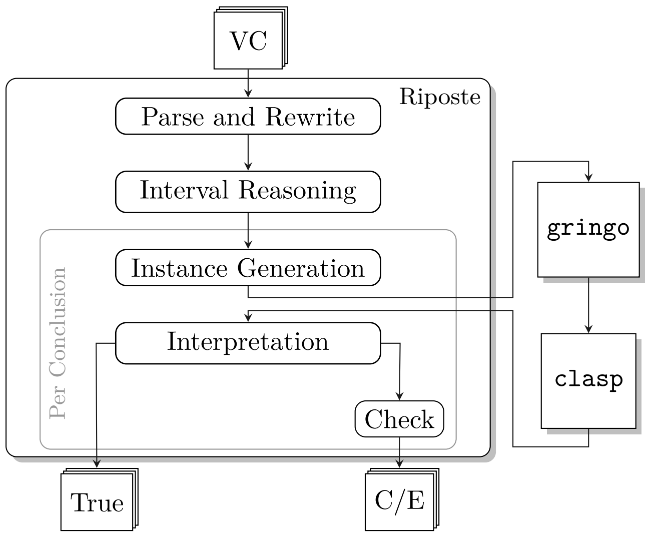

Figure 4 Architecture of Riposte.

found and interval reasoning techniques similar to bounds propagation algorithms used in CSP (Constraint Satisfaction Problem) solvers [16] are used to soundly refine the bounds. Sometimes this reasoning is sufficient to show a conclusion must always be true, in which case it can be discharged.

For example in Figure 3, Riposte can determine that the range of both a and b is $[0; 255]$, and the range for $a + b$ is $[0; 510]$. Riposte can now immediately rewrite conclusion 2 to 'true' as unsigned_byte___first is 0.

3.3 Program generation

Once bounds are established for all program variables and the formulae simplified, Riposte handles each conclusion individually. Although this requires more calls to the solver, the program variables and hypotheses used in each search can be reduced to only those that are necessary for a given conclusion. This not only makes the search faster, it significantly simplifies the counter-examples generated, as assignments for irrelevant program variables are not generated.

We now present a few interesting lines from the program generated for conclusion 3. Note that more of the encoding is described in Section 4, but most names should be fairly obvious.

A few important background literals required by the rest of the encoding are defined, the only one relevant for our example is wordLength; from interval arithmetic we know that the largest values (the base types for a and b) can fit into a signed 32-bit integer in our example.

```
%%% Background
wordLength(32).
literalIntegerLow(0).
literalIntegerHigh(1).
optimisationLength(8).
```

For each program variable present in the VC, Riposte generates an 'input variable', this defines the search space for the program.

```
variable(a,bitInteger,input).
variable(b,bitInteger,input).
```

Each VC may also make use of some numeric constants (0 and 255 in our example); the bit-patterns for those are also defined in the program, a snippet for the first 3 bits of constant 255 is shown below:

```
variable(bi_const_255,bitInteger,constant).
bitValue(bi_const_255,0,1).
bitValue(bi_const_255,1,1).
bitValue(bi_const_255,2,1).
bitValue(bi_const_255,3,1).
```

Riposte then encodes each hypothesis and the currently analysed conclusion. The encoding of a hypotheses (H2) is as follows:

```
%%% H2: a >= 0
variable(bi_leq_s(bi_const_0,a),boolean,expression).
computation(bi_leq_s(bi_const_0,a),
            bi_leq_s,
            bi_const_0,
            a).

hypothesis(bi_leq_s(bi_const_0,a)).
```

The `variable` literal declares the Boolean expression `bi_leq_s(bi_const_0, a)` with the `computation` literal generates the rules that compute its value and the `hypothesis` literal denotes that it is a hypothesis and thus must be true in all models.

Finally, the encoding of the conclusion analysed, C3, is shown below. Note the naming of the variables for expressions, such as `bi_plus_s(a, b)`; this avoids generating two calculations for the same expression twice and it also allows easy identification of what a variable represents from the name only.

```
%%% C3: a + b <= 255
variable(bi_plus_s(a,b),bitInteger,expression).
computation(bi_plus_s(a,b),
            bi_plus_s,
            a,
            b).

variable(bi_leq_s(bi_plus_s(a,b),bi_const_255),boolean,expression).
computation(bi_leq_s(bi_plus_s(a,b),bi_const_255),
            bi_leq_s,
            bi_plus_s(a,b),
            bi_const_255).

conclusion(bi_leq_s(bi_plus_s(a,b),bi_const_255)).
```

3.4 Interpretation

After the program has been generated and passed to `gringo` and `clasp`, a model may be returned. Each model contains a valuation for each input variable (`bitValue` for each bit of a bitIntegers, `boolenValue` for booleans, etc.).

```
*** Found a counter-example to function_add_ub_2, conclusion C3:
    (For path(s) from start to finish:)
H2: a >= 0
H3: a <= unsigned_byte__last
H4: b >= 0
H5: b <= unsigned_byte__last
  ->
C3: a + b <= unsigned_byte__last

This conclusion is false if:
   a = unsigned_byte__last
   b = 1
```

A number of basic, but effective, usability features have been implemented in order to assist the user with understanding the counter-example. In the output reproduced above for conclusion 3 of our example, large numbers are translated back to the original constants or an easier expression[3]; in this case unsigned_byte___last is really 255, but it is shown using the original name used in the VC. Furthermore, in order to reduce visual clutter only the hypotheses which are relevant to our conclusion are printed.

Riposte also checks that the counter-example given does actually make all hypotheses true and the conclusion false. This currently functions as an integrity check but will be used to refine the modelling if spurious counter-examples are generated.

Caching of previous results using `Memcached` is also performed to allow incremental and distributed computation [8].

4 Methodology and Modelling

The experience of developing a commercial scale application using ASP has yielded some insights into the development process and some useful encoding techniques.

4.1 Methodology

Riposte was developed using the methodology described in [6]. The map from informal concepts (such as "the B'th bit of variable N has value V") to literals was the first thing developed. Using this a number of simple programs were encoded manually and an interpretation script was developed. This allowed models to be understood in terms of the informal concepts rather than as a set of literals, and was invaluable in locating faults. The search space (each possible assignment to the variables in the FDL) was modelled and manually checked. Then the program was developed incrementally, one instruction at a time, with the behaviour of each section checked before proceeding. This greatly simplified debugging as the faults were almost always in the most recently added rules and their effects, in terms of the concepts they were supposed to represent, were easily visible. Three additional techniques were used to locate and prevent faults: random testing of individual instructions, system level regression testing and test driven development, and explicit modelling of assumptions about the model.

To test the individual instructions, a simple application was written that picked input values (covering all of the combinations of common 'edge' and extreme values and some random values), emulated the instruction and then produced a program that checked that the *AnsProlog* model gave the correct result. This proved useful while modelling the instruction as it allowed the partially completed model to be checked. In at least one case a discrepancy between the declarative *AnsProlog* and the procedural emulation in the test system was found to be a fault in the emulation!

At a higher level, system level test cases (VCs with annotations of which conclusions were supposed to have counter examples) were extensively used. Often suites of tests for a feature were written before they were implemented; in a fashion similar to test driven development. Once features were implemented, these test suites were used as whole system regression tests. This approach proved very effective and when the system was used on commercial code bases, very few faults were found.

The third technique for fault minimisation is more specific to *AnsProlog*. When developing a model there are normally a number of undocumented assumptions about the programs

[3] For example instead of printing 4503599627370495, Riposte will print $2 * *52 - 1$, which we contend is much more helpful.

and encodings. For example that each FDL program variable modelled is given only one type or that any bit is either 1 or 0. In the case of the program, these are normally regarded to be obvious from the informal meaning of the predicates and it is left to the programmer to generate valid programs. Implicit properties of the encoding can be given as auxiliary constraints if that helps inference. In Riposte assumptions about programs and the encoding are explicitly stated using rules that derive a "model error" literal. For example, separate literals are used to state when a bit is 1 or when it is 0 and the following rules are used to express the link between them.

```
%% Each bit of bitIntegers must be 1 (x)or 0
modelError(bit_is_both_1_and_0(N,B)) :-
   bitValue(N,B,1), bitValue(N,B,0), variable(N,bitInteger,R).
modelError(bit_is_neither_1_nor_0(N,B)) :-
   not bitValue(N,B,1), not bitValue(N,B,0), variable(N,bitInteger,R), bit(B).
```

There are two uses for these rules. During development it is possible to search for answer sets with model errors. This gives meaningful explanations of which implicit properties of the model have been broken, rather than yielding models. When Riposte is run in production mode, a constraint is added stating there are no modelling errors. Thus all of the rules describing modelling errors effectively become constraints, allowing equivalence preprocessing [19] to collapse the separate literals to one. This is an evolution of the techniques for error diagnosis used in *Anton* [5].

4.2 Encoding

A number of encoding techniques were developed to improve the performance and capacity of Riposte.

Variables are a central part of the model used in Riposte. They model FDL variables, constants and the values of expressions. For example, if the expression $a+b>0$ appears in a hypothesis or conclusion, there will be five variables modelled; two FDL, or input integers, a and b, one integer constant, 0, and two expression variables, an integer for $a+b$ and a Boolean for $a+b>0$. Choice rules are used so that FDL variables are assigned non-deterministic values. Constants are assigned direct values and the values of expression variables are given by the rules modelling their instruction. One key innovation was to name expression variables by the expression they compute. For example the variable corresponding to the expression $a+b$ would be named `bi_plus_s(a,b)`. This meant that all of the hypotheses and conclusions that referred to $a+b$ would automatically use the same variable and thus the same literals. Not only did this reduce the size of the programs generated, it also helped eliminate symmetries introduced by having multiple variables record the value of the same expression, and thus improved propagation.

One of the key challenges in modelling was how to deal with quantified expressions. As soon as the target application contains arrays, quantified hypotheses are unavoidable as even the simplest statement of type safety about arrays requires quantifiers. To illustrate Riposte's handling of quantifiers, consider the following (contrived) example:

```
function Contrived (A : Unsigned_Byte)
                  return Boolean
--# pre for all I in Unsigned_Byte range 50 .. 100 => (A /= I);
--# return A > 60 -> A > 150;
is
begin
   return True;
end Contrived;
```

The hypotheses which represents the precondition (effectively $a \notin [50,100]$) is expressed in FDL translated as follows (note that the identifier I has been renamed by Riposte).

```
%%% H1: for_all(riposte____qid_1: unsigned_byte,
%%%              riposte____qid_1 >= 50 and riposte____qid_1 <= 100 ->
%%%                 not a = riposte____qid_1)
```

Riposte handles quantifiers using the sound but not complete technique of instantiation. Every quantified hypothesis is replaced by a number of copies representing a subset of the possible bindings for the quantifier. Omitting particular bindings can fail to remove models (giving incompleteness) but cannot add models to a problem with no models (thus giving soundness). Due to space constraints we show this only for part of the statement, *not* $a = riposte\_\_\_\_qid\_1$. Note that RIPOSTE____QID_1 is variable whose instantiation is determined by the literal `hypothesisInstantiation`. `qi_h1` is an arbitrary constant identifying the relevant expression.

```
variable(bi_equal_l(a,RIPOSTE____QID_1),boolean,expression)
   :- hypothesisInstantiation(qi_h1,RIPOSTE____QID_1).
computation(bi_equal_l(a,RIPOSTE____QID_1),
            bi_equal_l,
            a,
            RIPOSTE____QID_1)
   :- hypothesisInstantiation(qi_h1,RIPOSTE____QID_1).

variable(b_not_l(bi_equal_l(a,RIPOSTE____QID_1)),boolean,expression)
   :- hypothesisInstantiation(qi_h1,RIPOSTE____QID_1).
computation(b_not_l(bi_equal_l(a,RIPOSTE____QID_1)),
            b_not_l,
            bi_equal_l(a,RIPOSTE____QID_1))
   :- hypothesisInstantiation(qi_h1,RIPOSTE____QID_1).
```

And finally the rule which encodes our *simple* but *surprisingly effective* instantiation heuristic. We essentially instantiate the quantified expression for all variables which are not expressions (i.e. for constants and input variables only).

```
hypothesisInstantiation(qi_h1,RIPOSTE____QID_1) :-
   variable(RIPOSTE____QID_1, bitInteger,R1), R1 != expression.
```

For our example this means that the quantified hypothesis is instantiated for $\{a, 0, 1, 50, 60, 100, 150, 255\}$ and Riposte gives $i = 101$ as a counter-example.

The last two encoding techniques improved the performance and completeness of Riposte, the next technique is focused on improving usability. Considering the program given in Figure 3, $a = 91$ and $b = 214$ is a counter example to conclusion 3. While this is an entirely correct counter-example it is perhaps not the most informative. To produce more helpful counter examples, Riposte makes use of the optimisation features of the answer set solver. This is used to produce counter examples in which the FDL or input variables are as close to zero as possible. By using an arbitrary order across the input variables and individual optimise statements for each variable, counter examples will often end up minimising some program variables and maximising other. For example in the case above, $a = 255$ and $b = 1$ is given. One of the advantages of using an answer set solver is being able to perform this optimisation.

5 Evaluation

This section gives two evaluations of Riposte. The first focuses on false VCs and compares Riposte with Victor and a variety of different SMT solvers. This evaluates Riposte in its intended usage scenario – finding counter examples to individual false VCs. The second experiment uses a large set of true VCs for a number of commercial applications and shows the distribution of problem size and run-time across real applications.

5.1 Comparison

As Riposte is a developer support tool, a key requirement is that it produces responses quickly and consistently across a range of real world programs. To test this a set of programs with undischarged VCs was created from publicly available SPARK applications: libsparkcrypto [26], Tokeneer [1] and SPARKSkein [9]. A number of subprograms whose proofs require non-formalised background information (for example, the number of certificates that can fit on the removable storage) were taken from Tokeneer. Subprograms taken from libsparkcrypto and SPARKSkein were modified to contain common bugs such as off by one errors, missing preconditions, indexing errors and insufficient loop invariants. The Examiner was used to generate VCs for these subprograms and the Simplifier used to remove simple true VCs, leaving a test set of 45 VCs. All experiments were run on an Intel i7 860 (2.8 GHz, 4 cores) desktop computer running Debian GNU/Linux, using a 20 minutes time limit per VC.

Figure 5 gives a graph of the cumulative time taken for Riposte to produce answers for the benchmark VCs. Results are also given for Victor, the SMT based prover for Spark, using a variety of back end SMT solvers: Alt-Ergo [12], CVC3 [14] and Z3 [15]. These are included to give an idea of what constituted reasonable amounts of time and completeness, rather than for direct comparison.

Although the SMT solvers outperform Riposte for the easy VCs, the more complex VCs containing bugs are resolved much more quickly by Riposte; resulting in the overall fastest time to process all VCs. Riposte is the only tool that renders a verdict on all benchmark VCs within the time limit. The division between grounder and solver causes slightly higher overheads for Riposte, giving the lower results on the left hand side of the graph. However the total time taken by Riposte on all resolved VCs is significantly lower (Riposte 1600s, CVC3 9000s, Alt-Ergo 11100s, Z3 20800s; to the nearest 100s) even though coverage is higher.

Riposte's performance on these benchmarks is fairly typical. During development it has been used on over 22,500 VCs (including four industrial applications, one unknown to the tools authors) resolving 95% or more. When counter examples are found, they are typically found rapidly and within the time developers are willing to wait.

5.2 Program statistics

We have also used Riposte to analyse the three code-bases mentioned above in their original form to generate some statistics on the size and run-times of programs generated. Figure 6 shows the distribution of sizes the ground programs in terms of atoms and rules. It can be seen that most of the programs are small (≤ 25000 atoms/rules), but a few are very large ($\geq 1,1$ million atoms and $\geq 1,2$ million rules).

Figure 7 plots the time taken to ground each program against the time taken to solve. It can be seen that most programs take longer to ground than to actually solve and even then the combined time is usually significantly less than around 10 seconds.

6 Related work

A key precedent for using ASP to reason about programs is the TOAST superoptimiser [7]. Its model of instructions was somewhat simpler as it was modelling hardware and thus only had the register 'type' to consider. In comparison, Riposte's models include a much richer type system (as it is modelling a typed language) and supports both quantifiers and axioms for reasoning about more complex types.

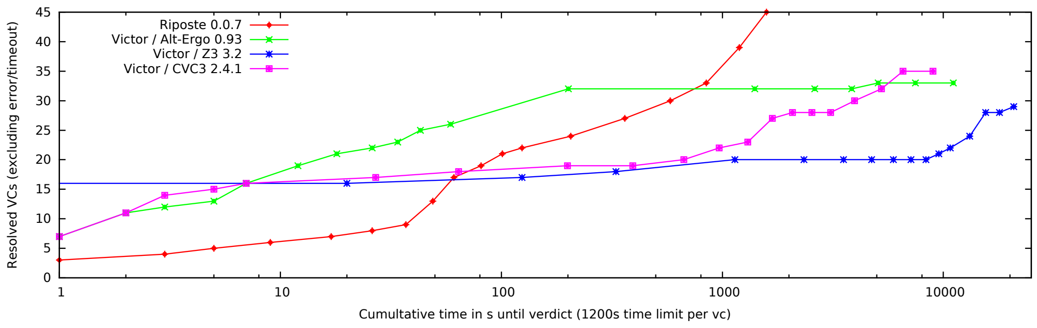

Figure 5 Cumulative time for returning a verdict on VCs for erroneous code.

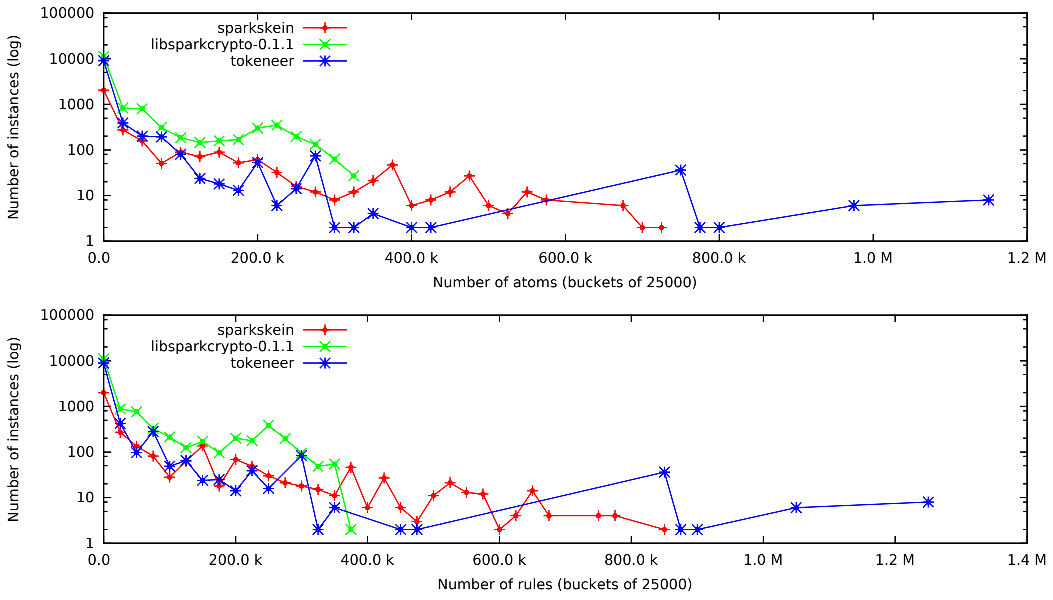

Figure 6 Program sizes in terms of atoms (above) and rules (below).

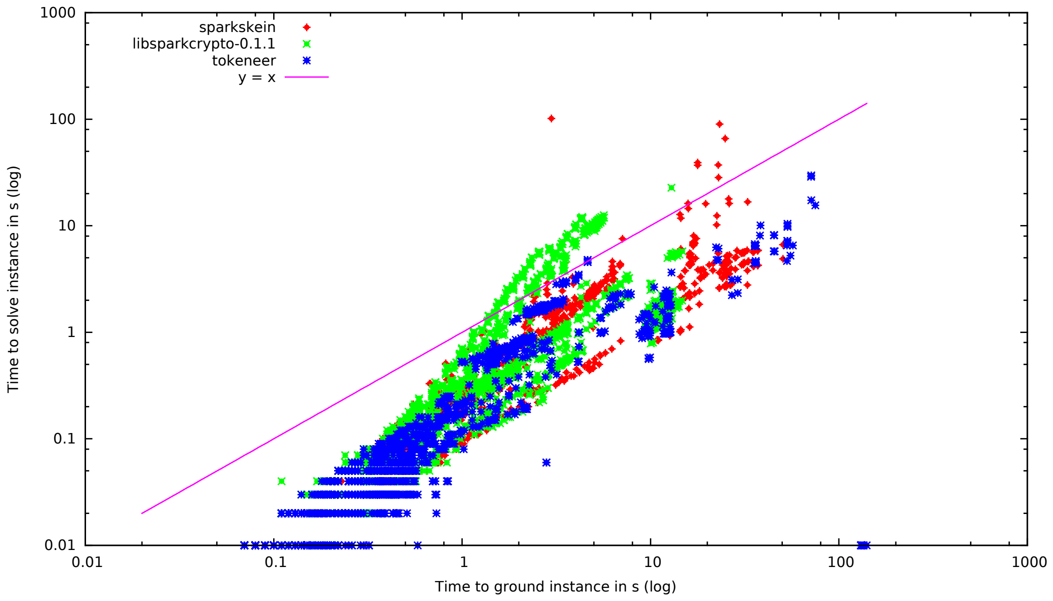

Figure 7 Time taken to ground each program v.s. time taken to solve.

The closest system to Riposte, both in terms of architecture and approach, is Nitpick [4], Isabelle's counter-example generator. It uses KodKod to generate counter-examples to HOL theorems. Although the theoretical foundations of answer set semantics and KodKod's FORL are very different, there are many parallels between the two systems, making KodKod effectively an equivalent approach. The one key difference is that Nitpick has to deal with infinite objects, making the encoding significantly more complex.

Systems that use SAT, SMT or other model generation solvers to discharge VCs (for example [11]) can potentially generate counter-examples directly from failed proof attempts. However there are a number of practical problems involving the size and complexity of the counter-examples generated [25].

A key problem is that compact VC generation algorithms [2] make it difficult to identify the root cause of a counter-example (as well as potentially significantly increasing the cost of verification [24]). One option is to 'tag' the VC with explanations. Tags can be additional propositions [23] or meta-information annotations [17]. As SPARK uses a more verbose but significantly simpler VC generation system (see Section 2), these are not needed in Riposte, since the failing condition (and why it is generated) is already available to the user.

Another area of research concerns the development of user interfaces to view and explore counter-examples once they have been generated. The VCC Model Viewer [11] and its successor, the Boogie Verification Debugger [20], show the power of integrating counter-example display into an IDE. An innovative approach to doing this is generating a program that triggers a bug corresponding to the counter-example [25] and then using a conventional debugger interface.

Finally, counter-examples play a key role in checking and refining abstraction in model checking systems, although this tends to be automatic (for example systems based on CEGAR [10]) rather than aimed at supporting end-users.

7 Conclusion and Future Work

This paper presents Riposte, a successful commercial application of answer set programming. Its performance is state of the art, as shown in Section 5. Furthermore it validates previous work on development methodologies [6] by showing it is possible to develop large application using them.

The next step for Riposte is integration into the next commercially supported release of the SPARK tools. This will definitely yield challenging examples generated from VCs for real world systems. It is hoped that these will be useful in improving the performance and capacity of answer set programming tools. One area of particular interest is improvement in the performance of grounders. As shown in Figure 7, grounding time is often the dominant factor in Riposte's performance. This is unusual as when the grounding is a bottleneck it is normally a space issue rather than run-time.

Another challenging area is moving counter examples beyond assignments of values to program variables. In some cases it is possible to produce expressions that describe a set of counter examples and are more informative than a single counter example. It may be possible to use the skeptical query mode of answer set solvers to find expressions that hold for every counter example. More generally, techniques for summarising the answer sets of a program would be of use.

References

1 Janet Barnes, Roderick Chapman, Randy Johnson, James Widmaier, Bill Everett, and David Cooper. Engineering the Tokeneer enclave protection software. In *ISSSE '06*. IEEE, 2006.

2 Mike Barnett and K. Rustan M. Leino. Weakest-precondition of unstructured programs. In *Proceedings of the 6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 82–87. ACM, September 2005.

3 Stefan Berghofer. Verification of Dependable Software using SPARK and Isabelle. In Jörg Brauer, Marco Roveri, and Hendrik Tews, editors, *Proceedings of the 6th International Workshop on Systems Software Verification (SSV 2011)*, pages 48–65. TU Dresden, August 2011. Technical report TUD–FI11.

4 Jasmin Christian Blanchette and Tobias Nipkow. Nitpick: A counterexample generator for higher-order logic based on a relational model finder. In *Interactive Theorem Proving*, volume 6172, pages 131–146. Springer, July 2010.

5 Georg Boenn, Martin Brain, Marina De Vos, and John ffitch. Automatic music composition using answer set programming. *The Theory and Practise of Logic Programming*, 11(2-3):397–427, February 2011.

6 Martin Brain, Owen Cliffe, and Marina De Vos. A pragmatic programmer's guide to answer set programming. In Marina De Vos and Torsten Schaub, editors, *Proceedings of SEA09*, pages 49–63. Electronic proceedings at http://sea09.cs.bath.ac.uk, September 2009.

7 Martin Brain, Tom Crick, Marina De Vos, and John Fitch. Toast: Applying answer set programming to superoptimisation. In Sandro Etalle and Mirosław Truszczyński, editors, *Proceedings of ICLP06*, volume 4079, pages 270–284. Springer, 2006.

8 Martin Brain and Florian Schanda. A low cost technique for distributed and incremental verification. In *Verified Software: Theories, Tools and Experiments*, pages 114–129. Springer, 2012.

9 Roderick Chapman, Eric Botcazou, and Angela Wallenburg. SPARKSkein: A Formal and Fast Reference Implementation of Skein. In *SBMF 2011*, volume 7021 of *LNCS*, pages 16–27. Springer, 2011.

10 Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification*, volume 1855 of *LNCS*, pages 154–169, 2000.

11 Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, MichałMoskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: A practical system for verifying concurrent c. In *Theorem Proving in Higher Order Logics*, volume 5674 of *LNCS*, pages 23–42. Springer, August 2009.

12 Sylvain Conchon, Evelyne Contejean, and Johannes Kanig. Ergo: A theorem prover for polymorphic first-order logic modulo theories, 2006.

13 M. Croxford and J. Sutton. Breaking through the V and V Bottleneck. In *Ada in Europe 1995*, volume 1031 of *LNCS*. Springer, 1996.

14 CVC3: An automatic theorem prover for satisfiability modulo theories (SMT). `http://www.cs.nyu.edu/acsys/cvc3`, 2006.

15 L. de Moura and N. Bjørner. Z3: An efficient SMT solver. *Tools and Algorithms for the Construction and Analysis of Systems*, 4963:337–340, 2008.

16 Rina Dechter. *Constraint Processing*. Morgan Kaufmann Publishers, 2003.

17 Ewen Denney and Bernd Fischer. Explaining verification conditions. In *Algebraic Methodology and Software Technology*, volume 5140 of *LNCS*, pages 145–159. Springer, 2008.

18 M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, and M. Schneider. Potassco: The Potsdam answer set solving collection. *AI Communications*, 24(2):105–124, 2011.

19 M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. Advanced preprocessing for answer set solving. In M. Ghallab, C. Spyropoulos, N. Fakotakis, and N. Avouris, editors, *Proceedings of the Eighteenth European Conference on Artificial Intelligence (ECAI'08)*, pages 15–19. IOS Press, 2008.

20 Claire Le Goues, K. Rustan M. Leino, and MichałMoskal. The Boogie verification debugger. In *Software Engineering and Formal Methods*, volume 7041 of *LNCS*, pages 407–414. Springer, November 2011.

21 Anthony Hall and Roderick Chapman. Correctness By Construction: Developing a Commercial Secure System. *IEEE Software*, pages 18–25, Jan/Feb 2002.

22 Paul B. Jackson and Grant Olney Passmore. Proving SPARK Verification Conditions with SMT solvers. `http://homepages.inf.ed.ac.uk/pbj/papers/vct-dec09-draft.pdf`, December 2009.

23 K. Rustan M. Leino, Todd Millstein, and James B. Saxe. Generating error traces from verification-condition counterexamples. *Science of Computer Programming*, 55(1-3):209–226, March 2005.

24 K. Rustan M. Leino, MichałMoskal, and Wolfram Schulte. Verification condition splitting. Unpublished report., 2008.

25 Peter Müller and Joseph N. Ruskiewicz. Using debuggers to understand failed verification attempts. In *FM 2011: Formal Methods*, volume 6664 of *LNCS*, pages 73–87. Springer, 2011.

26 Alexander Senier. libsparkcrypto - a cryptographic library implemented in SPARK. `http://senier.net/libsparkcrypto`, 2010.