Automation for Exception Freedom Proofs

Bill J. Ellis and Andrew Ireland
School of Mathematical & Computer Sciences
Heriot-Watt University
Edinburgh, Scotland, UK
bill@macs.hw.ac.uk a.ireland@hw.ac.uk

Abstract

Run-time errors are typically seen as unacceptable within safety and security critical software. The SPARK approach to the development of high integrity software addresses the problem of run-time errors through the use of formal verification. Proofs are constructed to show that each run-time check will never raise an error, thus proving freedom from run-time exceptions. Here we build upon the success of the SPARK approach by increasing the level of automation that can be achieved in proving freedom from exceptions. Our approach is based upon proof planning and a form of abstract interpretation.

1 Introduction

Run-time errors may result in the catastrophic failure of high integrity software. This includes safety critical applications, *e.g.* an integer overflow run-time error led to the loss of Ariane 5 [14], and security critical applications, *e.g.* *"buffer overflows have been the most common form of security vulnerability in the last ten years"* [12]. The ability to statically prove that software applications are free from run-time errors has obvious social and economic benefits.

The SPARK [5] programming language has been designed for the development of high integrity software [22]. Here we present an automatic technique that supports the process of formally proving that software written in SPARK is free from run-time errors (exception freedom). SPARK is defined as a subset of Ada and can therefore be compiled using any Ada compiler. SPARK excludes all of the exceptions that can be raised within Ada except for index, range, division and overflow checks. The SPARK toolset supports the analysis of SPARK code. This includes static flow analysis and partial correctness proofs. In terms of guarding against run-time errors, the key tools and their functions are: EXAMINER - generates *run-time check* (RTC) verification conditions (VCs) based on the Ada run-time checks, consequently proving the RTC VCs guarantees exception freedom ; SIMPLIFIER - automatically discharges relatively simple VCs ; SPADE PROOF CHECKER - an interactive theorem prover.

There exists industrial strength evidence [4] showing that the SPARK toolset can automatically prove around 90% of RTC VCs. While this performance is impressive, the remaining 10% typically account for many hundreds of RTC VCs, each requiring a user guided proof. We target this 10% using a hybrid approach, integrating theorem proving and program analysis.

2 Run-time exceptions in SPARK

```
subtype R_Type is Integer;
subtype I_Type is Integer range 0..9;
type D_Type is array (I_Type) of Integer;
...
R:= 0;
for I in I_Type loop
   --# assert true;
   if D(I)>=0 and D(I)<=100 then
      R:=R+D(I);
   end if;
end loop;
```

Note that `--# assert` *introduces a loop invariant. The* EXAMINER *adds invariants where none are present and strengthens all invariants to include limited type information.*

Figure 1. A filter program written in SPARK

To illustrate the problems associated with proving RTC VCs consider the SPARK code given in figure 1. In particular, consider the assignment statement within the `then`-branch, *i.e.* `R:=R+D(I)`, whose corresponding RTC VC is given in figure 2. There are two aspects to proving that this assignment cannot raise an exception. Firstly, we must show that the value of `I` can never exceed the range of array `D`, *i.e.* `C1` and `C2`. Secondly, we must show that the value of the expression `R+D(I)` lies within the legal bounds of `R`, *i.e.* `C3` and `C4`. While proving `C1` and `C2` is trivial (match with `H2` and `H3` respectively), `C3` and `C4` are unprovable. This problem arises as there is not sufficient proof context.

```
procedure_filter_6.
H1:     for_all (i___1: integer, ((i___1 >= i_type__first) and (
           i___1 <= i_type__last)) -> ((element(d, [i___1]) >=
           integer__first) and (element(d, [i___1]) <= integer__last))) .
H2:    loop__1__i >= i_type__first .
H3:    loop__1__i <= i_type__last .
H4:    element(d, [loop__1__i]) >= 0 .
H5:    element(d, [loop__1__i]) <= 100 .
        ->
C1:    loop__1__i >= i_type__first .
C2:    loop__1__i <= i_type__last .
C3:    r + element(d, [loop__1__i]) >= integer__first .
C4:    r + element(d, [loop__1__i]) <= integer__last .
```

The EXAMINER *generates eight VCs for the code given in figure 1. Three of these correspond to exception freedom, while the rest correspond to any properties asserted by the programmer, in this case the invariant. The RTC VC above corresponds to proving that the assignment within the* `then` *branch can never raise an exception. Note that* `element(A,[I])` *denotes accessing array* `A` *at index* `I`.

Figure 2. A run-time check verification condition (RTC VC)

Currently, the SPARK approach to overcoming such failures requires user interaction, *i.e.*

1. user identifies the need for additional hypotheses relating to the bounds on variables.
2. user strengthens the default loop invariant to introduce the required hypotheses.
3. user constructs proofs for the loop invariant VCs and revised RTC VCs via the PROOF CHECKER[1].

With a suitable invariant in place, the associated proofs are relatively uncomplicated and exhibit common patterns of reasoning. However, as an application typically requires many hundreds of proof failures to be patched this task presents a significant bottle-neck in practice.

3 Overview of our approach

Our approach combines the *proof planning* [6] theorem proving technique with program analysis based upon a form of *abstract interpretation* [11]. In particular, our approach replaces the user interaction outlined above, *i.e.*

1. proof planning identifies the need for additional hypotheses relating to the bounds on variables.
2. program analysis strengthens the default loop invariant to introduce the required hypotheses.
3. proof planning constructs proof tactics for the loop invariant VC and revised RTC VCs suitable for execution within the PROOF CHECKER.

Below we outline the proof planning and program analysis aspects of our approach. Due to limited space, we give prominence to the program analysis.

[1]The SIMPLIFIER is typically unable to discharge loop invariant VCs and RTC VCs whose proof relies on user supplied lemmas.

3.1 Proof planning

Proof planning is an Artificial Intelligence technique for guiding tactic based theorem provers. A *proof plan* is defined in terms of a set of *methods* that express preconditions for the applicability of a set of general purpose tactics. Each method can be viewed as a partial specification of a general purpose tactic. Given a particular proof obligation, methods are used by a planning system to automatically construct a customised tactic.

Our exception freedom proof plan contains five methods[2]. The failure of these methods is used to identify the need for additional hypotheses. This role is achieved through proof *critics* [18, 19]. Proof critics support the automatic analysis and patching of failed proof attempts. Here we use proof critics to identify a *schematic hypothesis* that is used to constrain our program analysis. Proof critics are typically linked to methods. The triggering of a proof critic is associated with a particular pattern of partial success of a method. For our example (see figure 2), the critics suggest a schematic hypothesis of the form:

$$r \geq \mathcal{A} \ \wedge \ r \leq \mathcal{B}$$

This schema suggests that additional information on the bounds of r needs to be introduced through the discovery of a stronger loop invariant.

3.2 Program analysis

Abstract interpretation provides a theoretical framework for the development of program analysis techniques. It operates by approximating the semantics of a programming

[2]The methods exploit the common patterns seen in discharging RTC VCs, focusing on the manipulation of inequalities, including the *isolate* method originally seen in [9].

language, replacing concrete values with more general *abstract values*. Abstract interpretation techniques are guaranteed to produce correct results. However, these guarantees are often achieved through approximations.

The emphasis on correctness tends to restrict the complexity of the heuristics used in abstract interpretation. However, the discovery of interesting program properties requires many and varied heuristics, as seen in [21, 17]. Our solution is to follow the proof planning paradigm. Program analysis is treated as an oracle, providing suggestions that the planner attempts to proof plan. By removing the burden of correctness, program analysis can flexibly operate using various heuristics. The soundness of the entire approach is ensured through the execution of the tactics generated by the proof planner.

3.2.1 Recurrence relation heuristic

The use of *recurrence relations* plays a key role in our program analysis heuristics. Significant tool support exists to automatically solve certain classes of recurrence relations, *e.g.* MATLAB [1]. We have focused on PURRS [3] for our current work. However, as our technique requires the services of a generic recurrence relation solver, we are not tied to PURRS.

The transformations applied to program variables within a loop can be expressed as a series of recurrence relations. That is, the value of each variable on the n^{th} iteration is expressed in terms of variables on previous iterations, usually the $(n-1)^{th}$ iteration. Solving these recurrence relations produces an invariant equating the value of a variable to an expression involving n. For example, in figure 1 `I` is implicitly initialised to 0 and the assignment statement `I:=I+1` is implicitly seen after each iteration of the loop. This can be expressed as the recurrence relation $i_{(0)} = 0, i_{(n)} = i_{(n-1)} + 1$, which can be solved as $i_{(n)} = i_{(0)} + n$ and reduced to $i_{(n)} = n$.

Often there is not sufficient information to solve a variable's recurrence relation. In such cases an approximation is made, generalising the search of finding an equation for the the n^{th} iteration to finding an interval on the n^{th} iteration. The end points of these intervals are described using what we call *extreme recurrence relations*, giving the extreme upper or lower limits of a variable on the n^{th} iteration. In figure 1, `R` is initialised to 0 and the assignment statement `R:=R+D(I)` is seen within the true branch of the conditional test `D(I)>=0 and D(I)<=100`. The recurrence relation for the false path is $r_{(0)} = 0, r_{(n)} = r_{(n-1)}$. Note that r remains unchanged so can be solved as $r_{(n)} = 0$. However, the recurrence relation for the true path[3], $r_{(0)} = 0, r_{(n)} = r_{(n-1)} + ele(d, i)$, cannot be solved. The problem is that $ele(d, i)$ is not expressed in terms of n. To eliminate this problem term it is replaced by its extreme bounds, giving two extreme recurrence relations bounding the unsorted interval $sort([r_{(n)} = r_{(n-1)} + 0, r_{(n)} = r_{(n-1)} + 100])$. These can be solved and subsequently sorted giving the interval $[0, n * 100]$.

3.2.2 Extracting loop invariants

Once the abstract interpreter has terminated, a candidate invariant can be extracted from the solved recurrence relations. The form of this invariant is guided by the schematic hypothesis discovered earlier. The solved recurrence relations can not be added directly as they may refer to the loop iteration variable n, which does not exist within the code. It is necessary to replace n with an expression in terms of the known program variables. An expression for n is found by isolating n in an equation relating a program variable to an expression including n. For example, the abstract value for the invariant of figure 1 includes $i_{(n)} = n$ and $r_{(n)} = [0, n * 100]$. The schematic hypothesis identifies that the lower and upper bounds on r should be introduced. The upper bound of r is in terms of n, however, exploiting $i_{(n)} = n$, n can be replaced with i giving the invariant:

```
--# assert R>=0 and R<=(I*100);
```

3.3 Planning the revised VCs

Adding the candidate invariant produces revised RTC VCs with additional hypotheses. Further, the invariant VCs have additional conclusions to prove the strengthened invariant. Although these VCs are provable, they are not discharged by the SIMPLIFIER. With the added hypotheses our exception freedom proof plan is now able to prove the RTC VCs. The loop invariant VCs are tackled using the *ripple* proof method [7]. Space precludes further discussion, however, the applicability of the ripple method to the verification of loop invariants has been previously investigated [20].

4 Implementation and results

Our exception freedom proof plan has been implemented within the CLAM proof planner [8]. The implementation of the tactics is ongoing. A prototype of the program analyser has been developed within Prolog. The proof planner and program analyser will eventually be integrated within a new system called NUSPADE. Evaluation of our approach is in the early stages, but our initial results are encouraging. We are about to embark on a more detailed testing and evaluation phase using industrial strength examples.

5 Related work

Probably the first system to prove exception freedom is the RUNCHECK verifier [17]. RUNCHECK operated on Pascal programs, employing a number of heuristics to discover

[3]*ele* is an abbreviation of the array access function *element*.

invariants and tackling RTC VC proofs with an external theorem prover. One of its heuristics involved the calculation of recurrence relations as *change vectors*, ignoring program context and collecting transformations made to variables. These change vectors were subsequently solved using a few rewrite rules that targeted common patterns. Our approach has a tighter integration between theorem proving and program analysis. In addition, our program analyser solves recurrence relations using a powerful recurrence relation solver tool. Further, our program analysis exploits program context and approximates to intervals where equality solutions can not be found.

The use of recurrence relations in generating loop-invariants was first reported by Elspas *et al* [13] and was also used by Katz and Manna [21]. Although the limits of recurrence relations as a basis for generating loop-invariants are well known [10], they have proved to be very useful for our niche application.

Recently there has been renewed interest in approaches that employ theorem proving to strengthen program development. The focus tends to be on finding errors rather than proving correctness. For example, ESC/JAVA [16] is an extended static checker for Java. Like SPARK, ESC/JAVA requires program annotations. HOUDINI [15] is able to automatically generate many of the annotations required by ESC/JAVA using predicate abstraction. There also exist systems based purely on abstract interpretation that require no program annotations. Most noteworthy are MERLE [23] and POLYSPACE [2]. Although these recent systems do not target proof, there results may still be valuable for our approach. In particular, we intend to conduct an investigation of POLYSPACE in the near future.

6 Conclusion

Building upon the SPARK approach, we have developed a hybrid technique for increasing the automation of exception freedom proofs. Program analysis is used to generate missing proof assertions. Proof planning provides the trigger for this analysis and proof automation for the associated proof obligations. Encouraged by our initial results, we believe that this approach will deliver significant improvements in the automation of exception freedom proofs within industrial strength applications.

Acknowledgements: In particular we would like to thank Peter Amey and Rod Chapman for their support in our research. Thanks also go to Alan Bundy, Jonathan Hammond, Ian O'Neill, Phil Thornley, Benjamin Gorry, Tommy Ingulfsen, Julian Richardson and Maria McCann for their feedback and encouragement. The research reported in this paper is supported by EPSRC grant GR/R24081 and is a collaboration with Praxis Critical Systems Ltd.

References

[1] Matlab. http://www.mathworks.com/.

[2] Polyspace-technologies. http://www.polyspace.com/.

[3] Purrs: The parma university's recurrence relation solver. http://www.cs.unipr.it/purrs/.

[4] P. Amey and R. Chapman. Industrial strength exception freedom. In *Proceedings of ACM SigAda*. 2002.

[5] J. Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison-Wesley, 2003.

[6] A. Bundy. The use of explicit plans to guide inductive proofs. In *Proceedings of CADE-9*. Springer-Verlag, 1988.

[7] A. Bundy, A. Stevens, F. van Harmelen, A. Ireland, and A. Smaill. Rippling: A heuristic for guiding inductive proofs. *Artificial Intelligence*, 62, 1993.

[8] A. Bundy, F. van Harmelen, C. Horn, and A. Smaill. The Oyster-Clam system. In *CADE-10*. Springer-Verlag, 1990.

[9] A. Bundy and B. Welham. Using meta-level inference for selective application of multiple rewrite rules in algebraic manipulation. *Artificial Intelligence*, 16(2), 1981.

[10] M. Caplain. Finding invariant assertions for proving programs. In *Proceedings of the International Conference on Reliable Software*, Los Angeles, CA, 1975.

[11] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL-4*. ACM Press, 1977.

[12] C. Cowan, P. Wagle, C. Pu, S. Beattie, and J. Walpole. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In *DARPA Information Survivability Conference and Expo (DISCEX)*. 2000.

[13] D. Elspas, M.W. Green, K.N. Levitt, and R.J. Waldinger. Research in interactive program-proving techniques. In *SRI, Menlo Park, CA*. 1972.

[14] ESA. Ariane 5 - flight 501 failure. Board of inquiry report, European Space Agency, 1996.

[15] C. Flanagan, K. Rustan, and M. Leino. Houdini, an annotation assistant for ESC/Java. In *Proceedings of FME 2001*. LNCS 2021, Springer-Verlag, 2001.

[16] C. Flanagan, K. Rustan, M. Leino, M. Lillibridge, G. Nelson, J. Saxe, and R. Stata. Extended static checking for Java. In *Proceedings of PLDI*, 2002.

[17] S.M. German. Automating proof of the absence of common runtime errors. In *Proceedings of POPL-5*. 1978.

[18] A. Ireland. The Use of Planning Critics in Mechanizing Inductive Proofs. In *Proceedings of LPAR'92*. LNAI 624, Springer-Verlag, 1992.

[19] A. Ireland and A. Bundy. Productive use of failure in inductive proof. *Journal of Automated Reasoning*, 16(1–2), 1996.

[20] A. Ireland and J. Stark. Proof planning for strategy development. *Annals of Mathematics and Artificial Intelligence*, 29(1-4), 2001.

[21] S.M. Katz and Z. Manna. Logical analysis of programs. *Communications of the ACM*, 19(4), 1976.

[22] S. King, J. Hammond, R. Chapman, and A. Pryor. Is proof more cost effective than testing? *IEEE Trans. on SE*, 26(8), 2000.

[23] Liz Whiting and Mike Hill. Safety analysis of hawk in flight monitor. In *Workshop on Program Analysis For Software Tools and Engineering*, 1999.