An Integration of Program Analysis and Automated Theorem Proving

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. Finding tractable methods for program reasoning remains a major research challenge. Here we address this challenge using an integrated approach to tackle a niche program reasoning application. The application is proving exception freedom, *i.e.* proving that a program is free from run-time exceptions. Exception freedom proofs are a significant task in the development of high integrity software, such as safety and security critical applications. The SPARK approach for the development of high integrity software provides a significant degree of automation in proving exception freedom. However, when the automation fails, user interaction is required. We build upon the SPARK approach to increase the amount of automation available. Our approach involves the integration of two static analysis techniques. We extend the proof planning paradigm with program analysis.

1 Introduction

Program reasoning has been an active area of research since the early days of computer science, as demonstrated by a program proof by Alan Turing [36]. However, as highlighted in [27] the search for "tractable methods" has remained a key research challenge. Here we address this challenge by considering the integration of two distinct static analysis techniques. The first is *proof planning* [4], a theorem proving technique developed by the automated deduction community. The second is *program analysis*, a general technique for automatically discovering interesting properties from a program's source code.

For our program reasoning we have focused on the SPARK programming language [1]. SPARK is designed for the development of high integrity software, as seen in safety and security critical applications. Our primary interest is in the development of automatic methods for proving exception freedom in SPARK programs, *i.e.* proving that a program is free from run-time exceptions. Such program reasoning represents an important task in the development of high integrity software. For instance, the loss of Ariane 5 was a result of an integer overflow run-time error [15], while buffer overflows are the most common form of security vulnerability [12]. The SPARK toolset supports proof of exception freedom using formal verification. This reduces the task of guaranteeing exception freedom to proving a number of theorems called *verification conditions* (VCs). Industrial strength evidence [9] shows that the SPARK toolset can typically prove around 90% of such VCs automactically. Our work targets the remaining 10%. These typically account for hundreds of VCs, each requiring user interaction to complete the proof.

Background material on SPARK and the nature of the verification problem being addressed is presented in §2. In §3 we compare proof using the SPARK toolset to proof following our approach. The details of our approach are presented in §4, §5, §6 and §7. In §8 related work is discussed while in §9 progress and future work are outlined. Our conclusions are presented in §10.

2 Background to the Problem

2.1 The SPARK Approach

The SPARK programming language is defined as a subset of Ada [26]. SPARK excludes many Ada constructs, such as pointers, dynamic memory allocation and recursion to make static analysis of SPARK feasible. SPARK includes an annotation language that supports flow analysis and formal proof. In the case of formal proof the annotations capture the program specification, asserting properties that must be true at particular program points. The annotations are supplied within regular Ada comments, allowing a SPARK compliant program to be compiled using any Ada compiler.

Compliance to the SPARK language is enforced by a static analyser called the EXAMINER. In addition, the EXAMINER performs data flow and information flow analysis [3]. The EXAMINER supports formal verification by building directly upon the Floyd/Hoare style of reasoning. VCs can be generated for proofs of both partial correctness and exception freedom. Two additional tools called the SPADE SIMPLIFIER and SPADE PROOF CHECKER are used to prove these VCs. The SIMPLIFIER is a special purpose theorem prover designed to automatically discharge relatively simple VCs while the PROOF CHECKER is an interactive proof development environment.

2.2 SPARK Exception Freedom

By its definition, SPARK eliminates many of the run-time exceptions that can be raised within Ada. However, index, range, division and overflow checks can still raise exceptions in SPARK code. The EXAMINER generates *run-time check* (RTC) VCs to statically guard against such exceptions. The RTC VCs are equivalent to the Ada run-time checks, consequently proving every RTC VC guarantees exception freedom. To generate VCs every loop must be annotated with an invariant. To support proof of exception freedom for sparsely annotated SPARK code, the EXAMINER automatically inserts invariants as will be described in §5.2.

To illustrate the problems associated with proving RTC VCs consider the SPARK code given in Figure 1. Note that this is used as a running example

```
package FilterPackage is
subtype AR_T is Integer
        range 0..9;
type A_T is array (AR_T)
     of Integer;
procedure Filter(A: in A_T;
                 R: out Integer);
--#derives R from A;
end FilterPackage;
```

```
package body FilterPackage is
procedure Filter(A: in A_T;
                 R: out Integer)
is
begin
  R:=0;
  for I in AR_T loop
    if A(I)>=0 and A(I)<=100 then
      R:=R+A(I);
    end if;
  end loop;
end Filter;
end FilterPackage;
```

Note that while `--#` represents an Ada comment it also denotes a SPARK annotation. Here the annotation is describing the flow information that `R` is derived from array `A` in subprogram `Filter`. This specification is checked automatically by the EXAMINER during its information flow analysis.

Fig. 1. Filter and sum values in an array

throughout the paper. Consider the assignment statement in the `then`-branch, *i.e.* `R:=R+A(I)`, whose corresponding RTC VC is given in Figure 2. There are two aspects to proving that this assignment can not raise an exception. Firstly, we must show that the value of `I` can never exceed the range of array `A`, *i.e.* `C1` and `C2`. Secondly, we must show that the value of the expression `R+A(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 insufficient proof context. Note that the RTC VCs involve proving that variables lie inside legal bounds. This is the case for all RTC VCs, allowing us to target our proof techniques and program analysis accordingly.

3 Comparing Proof in SPARK with Our Approach

3.1 Proof via the SPARK Toolset

Completing a program proof in the SPARK toolset typically requires several steps of user interaction. The general proof process undertaken is summarised below.

1. **Incomplete proof:** For each VC yet to be proved the user must determine the reason for the failure, implement a suitable patch, and then repeat this proof process. Three reasons for failure are considered.
   (a) **Insufficient proof context:** The VC is unprovable as the proof context is not sufficiently strong. The user must introduce the required proof context by strengthening the program specification.

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

The EXAMINER generates eight VCs for the running example of Figure 1. Three of these are RTC VCs, while the rest are proving properties asserted by the EXAMINER. The RTC VC above corresponds to proving that the assignment in the `then`-branch can never raise an exception. Here `H1`, `H2` and `H3` are a result of the invariant automatically inserted by the EXAMINER. Note that $element(a, [i])$ denotes accessing array a at index i.

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

   (b) **Discovered a bug:** The VC can be proved to be false. This indicates the presence of a bug in the source code or specification. The VC will typically give a strong clue as to the nature of the bug. The user must modify the code or specification to eliminate the bug.
   (c) **Beyond the simplifier:** The VC is provable however its proof is beyond the scope of the SIMPLIFIER. The user must prove the VC via an interactive session with the PROOF CHECKER.
2. **Complete proof:** Every VC is discharged by the SIMPLIFIER and any user guided proofs created in the PROOF CHECKER.

This process is rarely intellectually demanding. However, typically many hundreds of proof failures need to be patched per application. Further, all interactive proofs will be tuned to a particular version of a program. As the program is changed these proofs may break and require refinement. Thus this task presents a significant bottle-neck to the practical completion of exception freedom proofs.

3.2 Proof via Our Approach

Our approach reduces the amount of user interaction required to complete a proof. We extend the existing SPARK toolset with a new tool called NUSPADE[1]. NUSPADE is a proof planner that also incorporates program analysis. By using NUSPADE aspects of the proof process outlined above can be automated.

1. **Incomplete proof:** Each VC yet to be proved is automatically tackled by NUSPADE. If NUSPADE successfully finds a proof plan then this is exported as

[1] The name NUSPADE emphasises that we are building upon SPADE.

a customised tactic for execution inside the PROOF CHECKER. If NUSPADE fails to find a proof plan three situations are possible.

(a) **Insufficient proof context:** If NUSPADE is able to identify missing proof context then it can exploit the services of a program analysis oracle to enhance the program specification accordingly.

(b) **Discovered a bug:** If NUSPADE reduces a VC to false then it must indicate a bug. Although not considered further in this paper, there is scope for configuring NUSPADE to actively detect common programing errors. This will involve targeting the forms of VCs that these errors tend to produce with suitable disproving methods.

(c) **Require user interaction:** If the above cases do not apply, NUSPADE is unable to progress. The user must pursue the interactive proof process outlined above in §3.1.

2. **Complete proof:** Every VC is discharged by the SIMPLIFIER, any tactics created by NUSPADE and any user guided proofs created in the PROOF CHECKER.

4 Proof Planning

Proof planning is an artificial intelligence technique for guiding tactic based theorem provers. It has been extensively investigated within the context of proof by mathematical induction [6]. A proof plan represents the pattern associated with a family of proofs and is used to guide the search for the proof of a given conjecture within the family. A successful search instantiates the proof plan for the given conjecture. From the instantiated proof plan a tactic can be mechanically extracted and automatically checked using an appropriate theorem prover. Adopting this approach passes the burden of soundness to the theorem prover. Free from the constraints of demonstrating soundness, greater flexibility is possible when planning a proof.

A proof plan corresponds to a set of *methods*. Each method expresses preconditions for the applicability of a particular tactic. The methods are typically less expensive to execute and more constrained than their corresponding tactics. Another significant component of proof planning is the *proof critics* mechanism [19, 21]. Proof critics are associated with the partial success of proof methods and provide a mechanism for patching failed proofs.

4.1 Exception Freedom Methods

Our exception freedom proof plan contains four methods as outlined below. Note that these appear in the order used within the proof planner, *i.e.* the simpler more immediate methods are tried first. Further, note that the details of these methods will be described in more detail in §7.2.

1. elementary: Applicable to goals that are automatically discharged by the PROOF CHECKER, modulo some minor simplifications.

Preconditions for transitivity method:

1. There exists a goal of the form: $E\ Rel\ C$.
2. For all variables V_i that occur within E there exists a hypothesis of the form: $V_i\ Rel\ E_i$.

Preconditions for transitivity critic:

1. Precondition 1 of the transitivity method holds, *i.e.* there exists a goal of the form: $E\ Rel\ C$.
2. Precondition 2 of the transitivity method fails, *i.e.* there exists a variable V_i that occurs within E such that there does not exist a hypothesis of the form: $V_i\ Rel\ E_i$.

Note that E and C range over expressions and constants respectively, while *Rel* denotes a transitive relation.

Fig. 3. Preconditions for the transitivity method and critic

2. fertilise: Applicable where part of a goal matches a hypothesis, producing a simplified goal.
3. decomposition: Applicable to a transitive relation within a goal, decomposing its term structure.
4. transitivity: Applicable to a goal involving a transitive relation, introducing a transitive step into the proof.

4.2 Exception Freedom Critics

In our exception freedom proof plan a proof critic is associated with the transitivity method. The transitivity critic detects insufficient proof context. It describes the missing proof context using *hypotheses schemata*. The preconditions for the transitivity method and critic are presented in Figure 3. Note that the preconditions of the transitivity critic are expressed in terms of the partial success of the preconditions of the transitivity method.

4.3 Failed Proof Plan

To illustrate the behaviour of our exception freedom proof plan we return to our running example of Figure 1 and its corresponding RTC VC for the `then`-branch shown in Figure 2. We focus on the goal of proving `C4`,

$$r + element(a, i) \leq integer\_last\ ,$$

noting that the proof context includes the hypothesis `H5`

$$element(a, i) \leq 100\ .$$

All methods completely fail except the transitivity method which is partially successful. In the following A, B, X and Y range over expressions. The goal satisfies the first precondition of the transitivity method as there exists a goal of the form $E\ Rel\ C$. However, the second precondition fails, as variable r exists within $r + element(a, i)$ yet there is no hypothesis matching $r \leq B$. Note that $element(a, i)$ does not cause the second precondition to fail as there does exist a hypothesis of the form $element(a, i) \leq Y$. The proof plan for the lower bound of $r + element(a, i)$ similarly fails as there does not exist a hypothesis that matches $r \geq A$ and there does exist a hypothesis that matches $element(a, i) \geq X$. Each of these failure patterns trigger the transitivity critic, suggesting the need for additional hypotheses corresponding to the schema

$$(r \geq A) \wedge (r \leq B)\ .$$

This schema suggests that additional information on the bounds of r needs to be introduced through the discovery of a stronger loop invariant. Below in §5 we describe how this discovery is automated through program analysis.

5 Program Analysis

Program analysis involves automatically calculating interesting properties about source code. Different program analysis techniques have been presented, including flow analysis [3], performance analysis [14] and discovering constraints on variables [11].

Although VCs are generated by combining source code with its specification they typically reveal only a subset of this information. Thus, it is reasonable to return to the source code and its corresponding specification. For example, in [28] invariant discovery is tackled through *top-down* and *bottom-up* approaches, exploiting the specification and source code respectively. Top-down approaches are more applicable in the presence of a strong specification. As exception freedom proofs are typically performed on minimally annotated code the top-down approach is less effective. However, we believe that top-down approaches have a significant role to play in assisting partial correctness proofs [23]. Bottom-up approaches are more applicable where low level implementation detail is desired. This is especially suitable for exception freedom proofs, which involve reasoning about the low level details of an algorithm. Thus we focus on extracting properties from the source code using program analysis.

5.1 Program Analysis Oracle

Program analysis for program verification typically involves the use of heuristic based techniques, as seen in [28, 18]. These techniques can be quite unstructured, with different techniques interacting in various ways and often targeting a particular area of a program. In particular, these techniques often produce candidate properties that require nontrivial reasoning in order to prove their correctness. Thus it is not practical to capture such imprecise techniques in a formal manner. Our strategy is to view program analysis as an oracle. The system produces candidate properties for use during proof planning. The soundness of the entire approach is ensured by the execution of the tactics generated by the proof planner.

We capture distinct program analysis heuristics as *program analysis methods*. Our program analysis begins by first translating the input source code into a flowchart. The program analysis methods are then called in series to annotate the flowchart with *abstract values*, *i.e.* approximate descriptions of program variables. Each method employs a suitable representation to describe these abstract values. Once all of the methods have completed, a collection of *program properties* are extracted from the annotated flowchart. These properties may be accessed during proof planning to assist in the verification effort.

5.2 Program Analysis Performed by the Examiner

As mentioned in §2.2, the EXAMINER automatically inserts invariants to enable proof of exception freedom in minimally annotated SPARK. In addition to this the EXAMINER also inserts preconditions to enrich the program specification. This behaviour captures the spirit of our program analysis, *i.e.* exploiting information in the source code to automatically discover useful properties.

The EXAMINER adds a precondition that every imported subprogram parameter is within its type. Further, the EXAMINER adds a default invariant of true for each loop. This is strengthened by asserting that `for` loop iterators are within their type. Further, any precondition is copied into the invariant adjusting all variables to refer to their initial, rather than current, value.

5.3 Program Analysis Methods

Based on industrial strength examples and focusing on exception freedom proofs a small collection of program analysis methods have been established. These are presented in the following sections. For brevity the examples presented focus on regular program variables. However, they can be naturally extended to deal with arrays and records, the two main SPARK structures.

5.4 Method: Type

SPARK adopts the strong Ada type system, imposing some additional constraints to ease static analysis. As type information directly reveals a variables legal bounds it is especially valuable in exception freedom proofs. For example, consider the source code in Figure 4. The variables `I` and `J` are declared to be of type `ARPO_T`. Thus the method will find abstract values for `I` and `J` which may be expressed using the following candidate invariant. Note that SPARK code assertions, including invariants, are annotated as `--#assert`.

```
--#assert (I>=ARPO_T'First and I<=ARPO_T'Last) and
--#       (J>=ARPO_T'First and J<=ARPO_T'Last);
```

This invariant is required to prove exception freedom. Note that it is impossible to prove that a variable is within its type until it has been assigned a value, ruling out the candidate invariant property that `T` is inside its type `AC_T`.

```
package PolishFlagPackage is
subtype AR_T is Integer
        range 1..4;
type AC_T is (Red, White);
type A_T is array (AR_T)
     of AC_T;
procedure PolishFlag(A: in out A_T);
--# derives A from A;
end PolishFlagPackage;
```

```
package body PolishFlagPackage is
procedure PolishFlag(A: in out A_T)
is
  subtype ARPO_T is Integer
          range A'First..A'Last+1;
  I,J: ARPO_T;
  T: AC_T;
begin
  I:=ARPO_T'First; J:=ARPO_T'Last;
  loop
    exit when I=J;
    if A(I)=Red then
      I:=I+1;
    else
      J:=J-1; T:=A(I);
      A(I):=A(J); A(J):=T;
    end if;
  end loop;
end PolishFlag;
end PolishFlagPackage;
```

Fig. 4. Sort two value array

5.5 Method: For Loop Range

Each SPARK `for` loop iterator must have a declared type. This type may be constrained by imposing an additional range restriction. For example, consider the source code in Figure 5. The loop iterator `I` is declared to be of type `AR_T` and is constrained to be inside a range from `L` to `U`. This inspires abstract values which may be expressed as the following candidate invariant.

```
--#assert I>=L and I<=U;
```

Note that the property that loop iterators are within their type, as asserted by the EXAMINER, is usually sufficient for exception freedom proofs. However, the more constrained property found here would likely assist a partial correctness proof.

5.6 Method: Non-looping Code

At the start of a SPARK subprogram an arbitrary variable `X` will either have its initial value ($x^{\sim}$) or be undefined ($undef$). Following each assignment to `X` its value will change accordingly. Essentially it is straight forward to propagate the value of variables through non-looping code.

For example, consider the source code shown in Figure 6. At the start of subprogram `Clip` $v = v^{\sim}$ and $r = undef$. Entering the `then` branch of the outermost `if` statement yields $r = I\_T'First$. Entering the `else` branch enters the innermost `if` statement. The `then` branch yields $r = I\_T'Last$ while the `else` branch yields $r = v^{\sim}$. As either branch of the innermost `if` statement may be

```
package FindPackage is
  subtype AR_T is Integer
          range 1..10;
  subtype ARMO_T is Integer
          range 0..10;
  type AC_T is
       range -1000..1000;
  type A_T is array (AR_T)
       of AC_T;
  procedure Find(A: in A_T;
                 L,U: in AR_T;
                 F: in AC_T;
                 R: out ARMO_T);
  --#derives R from A,L,U,F;
end FindPackage;
```

```
package body FindPackage is
procedure Find(A: in A_T;
               L,U: in AR_T;
               F: in AC_T;
               R: out ARMO_T)
is
begin
  R:=0;
  for I in AR_T range L..U loop
    if A(I)=F then
      R:=I;
      exit;
    end if;
  end loop;
end Find;
end FindPackage;
```

Fig. 5. Find first index in array, between bounds, containing target

taken a disjunction is required giving $(r = I\_T'Last) \vee (r = v^{\sim})$. This is repeated for the outermost `if` statement giving $(r = I\_T'First) \vee ((r = I\_T'Last) \vee (r = v^{\sim}))$. These abstract values may be expressed through the following candidate assertion. Note that as `V` is an import variable of mode `in` it can not be changed and thus implicitly refers to its initial value.

```
--#assert (R=I_T'First) or ((R=I_T'Last) or (R=V));
```

However, where variables are assigned expressions involving variables, as in `R:=V`, it is often the case that conditional information can be exploited to constrain the abstract values. For example, consider the disjunct $r = v^{\sim}$ generated above. All that will be known about v is that it is inside its type. As v is declared as an integer this provides a weak constraint on the value of r. Where `R:=V` is encountered it is known that $\neg(v < I\_T'First) \wedge \neg(v > I\_T'Last)$. Using inequality reasoning this can be simplified to $(v \geq I\_T'First) \wedge (v \leq I\_T'Last)$. Replacing $v^{\sim}$ with this gives a more constrained abstract value following the outermost `if` $r = I\_T'First \vee (r = I\_T'Last \vee ((r \geq I\_T'First) \wedge (r \leq I\_T'Last)))$, which can be simplified to $(r \geq I\_T'First) \wedge (r \leq I\_T'Last)$. This abstract value would be expressed using the following candidate assertion.

```
--#assert (R>=I_T'First and R<=I_T'Last);
```

Consistently performing such reasoning for the general case would become difficult. However, reasonable progress can be made by targeting variables occurring in assigned expressions and employing lightweight inequality reasoning.

5.7 Method: Looping Code

Looping code presents problems over non-looping code as the abstract values found for variables within the loop should be general enough to describe every

```
package ClipPackage
is
subtype I_T is Integer
        range 1..4;
procedure Clip(V: in Integer;
               R: out I_T);
--# derives R from V;
end ClipPackage;
```

```
package body ClipPackage is
procedure Clip(V: in Integer;
               R: out I_T)
is
begin
  if V<I_T'First then
    R:=I_T'First;
  else
    if V>I_T'Last then
      R:=I_T'Last;
    else
      R:=V;
    end if;
  end if;
end Clip;
end ClipPackage;
```

Fig. 6. Clip from integer to more constrained type

iteration. We use *recurrence relations* to describe the value of a variable on an arbitrary iteration. Powerful tools exist to automatically solve certain classes of recurrence relations, *e.g.* MATLAB [31]. Although we have focused on PURRS [33] as we only require a generic recurrence relation solver we are not tied to PURRS.

The transformations applied to a program variable in a loop are expressed as a recurrence relation, *i.e.* the value of a 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 on the n^{th} iteration to an expression involving n. To extract usable properties from the solved recurrence relations it is necessary to eliminate this n.

Solving a variable's recurrence relation may require solutions to other recurrence relations. For this reason the method is separated into sub-methods with the more immediate sub-methods being applied first. Note that the sub-methods are shown below in the order in which they are applied.

Sub-Method: Unchanged: This targets variables that are unchanged inside a loop. Any import variables of mode `in` must remain unchanged throughout a subprogram. These are identified by examining the subprogram's parameter list. Other variables must change inside the subprogram but may remain unchanged inside a loop. These are identified by finding no assignments to the variable inside the loop.

For example, consider the source code shown in Figure 5. By examining the subprogram parameter list it is found that `A`, `L`, `U` and `F` are import variables of mode `in`. Recurrence relations are calculated for the remaining variable `R`. The initial value of r is 0 and no assignments are made to r inside the loop (the only assignment to r takes place on the loop exit). Thus the recurrence relation found for r is $r_{(0)} = 0, r_{(n)} = r_{(n-1)}$ which is solved as $r_{(n)} = 0$. These abstract values may be expressed as the following candidate invariant.

```
    --#assert A=A and L=L and U=U and F=F and R=0;
```

Note that the only descriptive property is `R=0`. However, by successfully solving all of the variables the loop analysis of this subprogram can now terminate.

Sub-Method: Constant Change: It is common to modify a variable by a constant value in each iteration of a loop. These are identified by finding that every assignment to a variable occurs outside conditional statements and the assigned expressions only involve this variable and constant values.

For example, in the running example of 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 is expressed as the recurrence relation $i_{(0)} = 0, i_{(n)} = i_{(n-1)} + 1$, which is solved as $i_{(n)} = i_{(0)} + n$ and reduced to $i_{(n)} = n$. As this abstract value contains n it can not yet be presented as a candidate invariant.

Sub-Method: Variable Change: A variable may be modified by a variable amount in each iteration of a loop. This can occur in several cases including assigning to a variable inside a conditional statement and assigning a variable an expression which takes different values from an array. In such cases there is not sufficient information to describe the exact value of a variable on the n^{th} iteration. Thus an approximation is made, generalising the search to finding the bounds of all possible values on the n^{th} iteration. We model the extreme end points of these bounds using what we call *extreme recurrence relations.*

For example, in the running example of Figure 1, `R` is initialised to 0 and the assignment statement `R:=R+A(I)` is seen within the `then` branch of the `if` statement which is conditional on `A(I)>=0 and A(I)<=100`. The recurrence relation for not entering the `if` statement is $r_{(0)} = 0, r_{(n)} = r_{(n-1)}$, which is solved as $r_{(n)} = 0$. However, the recurrence relation for entering the `if` statement, $r_{(0)} = 0, r_{(n)} = r_{(n-1)} + \mathit{element}(a, i)$, cannot be solved. The problem is that $\mathit{element}(a, i)$ represents a variable change. This problem term is eliminated by generalising to its extreme bounds. Exploiting context information reveals these bounds to be between $r_{(n)} = r_{(n-1)} + 0$ and $r_{(n)} = r_{(n-1)} + 100$. Each of these can be solved and expressed as a range giving the abstract value $(r_{(n)} \geq 0) \wedge (r_{(n)} \leq (n * 100))$. Once again, as this abstract value contains n it can not yet be presented as a candidate invariant.

Sub-Method: Counter Variables: During the execution of a loop the value of variables may change. Those variables found to monotonically increase or decrease by one are classified as *counter variables.* Counter variables are very common and often key to understanding an algorithm, motivating their special classification.

Counter variables can be identified by exploiting the abstract values found by the constant change and variable change sub-methods. For example, in the description of the constant change sub-method it was shown how the abstract value $i_{(n)} = n$ would be found for variable `I` in the running example of Figure 1.

Although the presence of n prevents this from being presented as a candidate invariant, it is straight forward to determine that `I` is an increasing counter variable initialised at zero (as n can be thought of as an increasing counter variable initialised at zero). There would be little benefit in expressing this property as a program assertion. However, this information can be collected as program properties and be exploited during proof planning. For example, in [23] the counter variable classification is instrumental in progressing an otherwise failed program proof.

Sub-Method: Extracting Properties: Following the loop analysis it is necessary to post-process the solved recurrence relations into new abstract values that eliminate all references to n. This is achieved by replacing n with an expression in terms of the known program variables.

For example, in the running example of Figure 1 the initial abstract values are $i_{(n)} = n$ and $(r_{(n)} \geq 0) \wedge (r_{(n)} \leq (n*100))$. The upper bound of r is expressed in terms of n, however, exploiting $i_{(n)} = n$, n is replaced by i giving the new abstract value $(r_{(n)} \geq 0) \wedge (r_{(n)} \leq (i_{(n)} * 100))$ which may be expressed as the following candidate invariant.

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

Note that it is difficult to eliminate n in $i_{(n)} = n$ as r is not described as an equality with n. This failure means that if an invariant property describing `I` is required then a suitable abstract value discovered from another method must be used instead. Further note that although the loop analysis does not suggest a candidate invariant property for `I` it does successfully classify it as a counter variable.

5.8 Method: Loop Guards

The loop analysis involves recurrence relations, expressing constraints on variables on the n^{th} iteration. The loop exit could be modeled by constraining the range of n and analysing the recurrence relations associated with variables. However, such an approach can be quite complicated and often unnecessary. Thus we instead consider the loop exit as a special case distinct from the recurrence relation analysis. We focus on finding properties that describe relationships between variables in the loop guard. In particular we check to see if an inequality relationship holds between these variables.

The loop guard is significant as its negation becomes available in the loops iteration VCs. This is the only property that constrains loop iterations and thus must be exploited to show that monotonically increasing (or decreasing) variables do not increase (or decrease) forever and exceed their legal bounds.

For example, in the running example of Figure 1 it must be proved that `I<=AR_T'Last` is invariant, *i.e.* that `I` does not exceed the upper bound of its type. This loop implicitly has a loop guard of the form `I=AR_T'Last`. Thus the induction hypothesis, $i \leq AR\_T'Last$, and the negation of the loop guard,

$\neg(i = AR\_T'Last)$, are hypotheses in the loop iteration VCs. Crucially, these can be combined to provide a single inequality constraint $i < AR\_T'Last$. As i is an increasing counter variable the induction conclusion will take the form $i+1 \leq AR\_T'Last$ which is trivially true given the inequality constraint hypothesis.

However, cases exist where the negation of the loop guard is not sufficiently strong to support such a proof. For example, consider the source code in Figure 4. Assume the following loop invariant, discovered in §5.4, has been added to prove that I and J do not exceed their type.

```
--#assert (I>=ARPO_T'First and I<=ARPO_T'Last) and
--#       (J>=ARPO_T'First and J<=ARPO_T'Last);
```

The loop guard is I=J, introducing the hypothesis $\neg(i = j)$ during loop iteration. Knowing that i and j have different values does not constrain the bounds of i and j. The counter variable sub-method reveals that I is an increasing counter variable, J is a decreasing counter variable and that I starts below J. As the loop exits at I=J, I can never exceed J, discovering the candidate invariant property $i \leq j$. Adding this invariant property introduces the new hypothesis $i \leq j$ into the loop iteration VCs. This can now be combined with the negation of the loop guard to introduce the inequality constraint hypothesis $i < j$. This is sufficiently strong to prove that both I and J remain within the bounds of their type.

6 Patching Proof Failure

We return to our running example of Figure 1 and proving the RTC VC for the then-branch as shown in Figure 2. Our initial proof plan in §4.3 failed with the transitivity critic requesting additional hypotheses corresponding to the schema

$$(r \geq A) \wedge (r \leq B)\ .$$

This failure activates program analysis of the relevant subprogram generating a collection of program properties. These properties are searched for a suitable candidate invariant constraint on r, guided by the schema above. Such an invariant was discovered in §5.7 and is repeated below.

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

Adding this invariant leads to revised RTC VCs, adding the following two hypotheses to the RTC VC shown in Figure 2.

```
H6:    r >= 0 .
H7:    r <= loop__1__i * 100 .
```

7 Planning the Revised VCs

7.1 Loop Invariant Methods

As illustrated above, it is often the case that an invariant must be strengthened before an exception freedom proof can be completed. The stronger invariant properties must be proved. The proof planner tackles loop invariant VCs using the ripple method [2, 6]. Although space precludes further discussion we note that proving loop invariants via the ripple method has been previously investigated and reported [24, 34, 25]. For example, in the running example of Figure 1 the strengthened invariant results in two loop invariant VCs, neither of which are automatically discharged by the SIMPLIFIER. However, by proof planning using the ripple method these proofs can be automated.

7.2 Revisiting the Exception Freedom Methods

We now consider the methods introduced in §4.1 in more detail. Recall that proving exception freedom involves showing that a variable does not violate its legal upper and lower bounds. Let the general value of a variable be denoted by the term $T(V_1, \ldots, V_n)$, where V_i, $1 \leq i \leq n$, denote variables. Further let L and U denote the lower and upper constants of a bound. Thus a variable's lower and upper bound checks give rise to goals of the form in (1) and (2) respectively.

$$T(V_1, \ldots, V_n) \geq L \tag{1}$$
$$T(V_1, \ldots, V_n) \leq U \tag{2}$$

Although we focus on the upper bound case (2), the same general pattern of proof is also applicable to the lower bound case (1). The proof context associated with (2) should contain hypotheses expressing the upper bounds of V_i

$$V_i \leq U_i \ . \tag{3}$$

Note that the absence of such hypotheses triggers the transitivity critic which aims to introduce the missing hypotheses by exploiting our program analysis.

The first step involves the transitivity method, reducing (2) to give

$$(T(V_1, \ldots, V_n) \leq X_1) \wedge (X_1 \leq U) \ . \tag{4}$$

The introduction of the meta-variable X_1 prepares the way for the decomposition of $T(V_1, \ldots, V_n)$. The second step calls the decomposition method to decompose $T(V_1, \ldots, V_n)$. This draws upon a collection of substitution axioms for inequalities. The aim of this method is to express the left hand side conjunct of (4) as a conjunction of inequalities of the form

$$V_i \leq X_j \ . \tag{5}$$

Note that the complete decomposition of $T(V_1, \ldots, V_n)$ may require the application of multiple substitution axioms. The third step calls the fertilise method to match the decomposed inequalities against the inequality hypotheses. Matching (5) against (3) instantiates X_j to U_i. This has the effect of instantiating the right hand side conjunct of (4) to give

$$T(U_1, \ldots, U_n) \leq U \ . \tag{6}$$

The fourth and final step involves the elementary method, simplifying (6) such that it can be trivially discharged by the PROOF CHECKER.

The key to the proof plan is the transitivity method as described in Figure 3. Note that the transitivity method introduces a first-order meta-variable into the goal structure that is incrementally instantiated during subsequent proof planning steps. This use of meta-variables is known as *middle-out reasoning* [5] and has been used effectively in guiding proof search within the context of program synthesis [30, 35], proof patching [20–22] and loop invariant discovery [24, 34, 25].

7.3 Successful Proof Plan

We now return to proving the `then`-branch of the code in Figure 1 following the patch of an invariant. Once again, we focus on the goal of proving `C4`

$$(r + element(a,i)) \leq integer\_last \; , \tag{7}$$

noting that the proof context includes the two hypotheses `H7` and `H5`,

$$(r \leq i * 100) \wedge (element(a,i) \leq 100) \; . \tag{8}$$

The proof planning begins with an application of the transitivity method, rewriting (7) to a conjunction

$$((r + element(a,i)) \leq X_1) \wedge (X_1 \leq integer\_last) \; . \tag{9}$$

The decomposition method searches for a substitution axiom involving $\leq$, finding the rewrite rule

$$(W + X) \leq (Y + Z) \Rightarrow (W \leq Y) \wedge (X \leq Z) \tag{10}$$

which is applied to (9) giving

$$((r \leq X_2) \wedge (element(a,i) \leq X_3)) \wedge ((X_2 + X_3) \leq integer\_last) \; . \tag{11}$$

Note that as a side-effect of applying (10), X_1 has been instantiated to X_2+X_3 in (11). Given (8), the fertilise method applies to the conjuncts on the left hand side of (11) resulting in X_2 and X_3 being instantiated to $i * 100$ and 100 respectively. The remaining goal takes the form

$$((i * 100) + 100) \leq integer\_last \; .$$

Given that *integer_last* has a known concrete value this goal is trivial and can be discharged by the elementary method.

8 Related Work

Probably the first system to prove exception freedom was the RUNCHECK verifier [18]. RUNCHECK operated on Pascal programs, employing a number of heuristics to discover 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 ranges 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 [29]. Although the limits of recurrence relations as a basis for generating loop-invariants are well known [8], 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 [17] is an extended static checker for Java. Like SPARK, ESC/JAVA requires program annotations. HOUDINI [16] is able to automatically generate many of the annotations required by ESC/JAVA using predicate abstraction.

There exists systems that employ program analysis to pinpoint unfavourable behaviour. These systems are typically formulated inside the abstract interpretation framework [10]. By observing this framework the program analysis will ensure correctness by allowing for approximate results. The most noteworthy systems are MERLE [38] and POLYSPACE [32]. Although these systems do not target proof, their results might be used to assist a formal proof. Rather than use annotations these systems gain constraints on variables by analysing a program in its entirety. This process can be computationally expensive and requires a complete program for input. As our program analysis targets individual subprograms it is fairly cheap to perform and is applicable early in program development. Further, by avoiding the abstract interpretation framework we have the flexibility to implement heuristic based program analysis techniques. As we treat our program analysis as an oracle that guides search for a formal proof, we can adopt this approach without sacrificing correctness.

9 Progress & Future Work

Our NUSPADE tool has been prototyped as separate components. The proof plans presented here have been implemented within the CLAM proof planner [7]. Note that this prototype does not support the extraction of a customised tactic from discovered proof plans. The program analysis has been prototyped in a system with a limited SPARK parser. This is sufficient to explore the program analysis methods. Work is underway on completing the NUSPADE system. Currently we have developed a suitable proof planning infrastructure in Prolog and are building a stronger program analysis system, exploiting the STRATEGO [37] program transformation tool.

Using our prototype systems we have successfully demonstrated the applicability of our technique on a collection of isolated subprograms. These subprograms are representative of the kinds of subprograms that are seen within a high integrity software system. The next step is to tackle proof of exception freedom for an entire industrial strength high integrity software system. We also envisage a comparative study between our approach and non-theorem proving techniques, such as MERLE and POLYSPACE. It may be found that such systems can be packaged as additional program analysis oracles for use in our approach.

10 Conclusion

Building upon the SPARK toolset, we have developed an approach for increasing the automation of exception freedom proofs. Our approach is formulated within the proof planning framework. Under certain patterns of failure, critics are invoked which in turn appeal to a program analysis oracle. This oracle aims to discover program properties that patch the failed proof, allowing the proof planning to progress.

Our approach demonstrates that program verification can be tackled on more than one front. By integrating the distinct static analysis techniques of proof planning and program analysis a more capable automatic program verification system can be can be constructed.

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. J. Barnes. *High Integrity Software: The SPARK Approach to Safety and Security.* Addison-Wesley, 2003.
2. D. Basin and T. Walsh. A calculus for and termination of rippling. *Journal of Automated Reasoning*, 16(1–2), 1996.
3. J. Bergeretti and B.A. Carré. Information-flow and data-flow analysis of while-programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(1), 1985.
4. A. Bundy. The use of explicit plans to guide inductive proofs. In *CADE-9*. Springer-Verlag, 1988.
5. A. Bundy, A. Smaill, and J. Hesketh. Turning eureka steps into calculations in automatic program synthesis. In *Proceedings of UK IT*, 1990.
6. A. Bundy, A. Stevens, F. van Harmelen, A. Ireland, and A. Smaill. Rippling: A heuristic for guiding inductive proofs. *Artificial Intelligence*, 62, 1993.

7. A. Bundy, F. van Harmelen, C. Horn, and A. Smaill. The Oyster-Clam system. In 10th *International Conference on Automated Deduction*, 1990.
8. M. Caplain. Finding invariant assertions for proving programs. In *Proceedings of the International Conference on Reliable Software*, 1975.
9. R. Chapman and P. Amey. Industrial strength exception freedom. In *Proceedings of ACM SigAda*. Addison-Wesley, 2002.
10. 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, 1977.
11. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL-5*. ACM, 1978.
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)*. IEEE Computer Society Press, 2000.
13. D. Elspas, M.W. Green, K.N. Levitt, and R.J. Waldinger. Research in interactive program-proving techniques. In *SRI*. 1972.
14. A. Ermedahl and J. Gustafsson. Deriving annotations for tight calculation of execution time. In *European Conference on Parallel Processing*, 1997.
15. ESA. Ariane 5 - flight 501 failure. Board of inquiry report, European Space Agency, 1996.
16. C. Flanagan and K. Rustan M. Leino. Houdini, an annotation assistant for ESC/Java. In *Proceedings of FME*. Springer-Verlag, 2001.
17. 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.
18. S.M. German. Automating proof of the absence of common runtime errors. In *POPL-5*. ACM, 1978.
19. A. Ireland. The use of planning critics in mechanizing inductive proofs. In *International Conference on Logic Programming and Automated Reasoning (LPAR'92)*, LNAI No. 624. Springer-Verlag, 1992.
20. A. Ireland and A. Bundy. Extensions to a generalization critic for inductive proof. In 13th *Conference on Automated Deduction*, 1996.
21. A. Ireland and A. Bundy. Productive use of failure in inductive proof. *Journal of Automated Reasoning*, 16(1–2), 1996.
22. A. Ireland and A. Bundy. Automatic verification of functions with accumulating parameters. *Journal of Functional Programming: Special Issue on Theorem Proving & Functional Programming*, 9(2), 1999.
23. A. Ireland, B.J. Ellis, and T. Ingulfsen. Invariant patterns for program reasoning. Technical Report HW-MACS-TR-0011, School of Mathematical and Computer Sciences, Heriot-Watt University, 2004. Also to appear in the Proceedings of the Mexican International Conference on Artificial Intelligence 2004 (MICAI-04).
24. A. Ireland and J. Stark. On the automatic discovery of loop invariants. In *Proceedings of the 4th NASA Langley Formal Methods Workshop – NASA Conference Publication 3356*, 1997.
25. A. Ireland and J. Stark. Proof planning for strategy development. *Annals of Mathematics and Artificial Intelligence*, 29(1–4), 2001.
26. ISO. Reference manual for the Ada programming language. ISO/IEC 8652, International Standards Organization, 1995.
27. C.B. Jones. The early search for tractable ways of reasoning about programs. In *IEEE Annals of the History of Computing*. IEEE Computer Society, 2003.
28. S.M. Katz and Z. Manna. A heuristic approach to program verification. In *Proceedings of IJCAI-73*. IJCAI, 1973.

29. S.M. Katz and Z. Manna. Logical analysis of programs. *Communications of the ACM*, 19(4), 1976.
30. I. Kraan, D. Basin, and A. Bundy. Middle-out reasoning for logic program synthesis. In *Proceedings of the* 10th *International Conference on Logic Programming*, 1993.
31. MatLab. http://www.mathworks.com/.
32. PolySpace-Technologies. http://www.polyspace.com/.
33. PURRS: The parma university's recurrence relation solver. http://www.cs.unipr.it/purrs/.
34. J. Stark and A. Ireland. Invariant discovery via failed proof attempts. In *Logic-Based Program Synthesis and Transformation*, number 1559 in LNCS. Springer-Verlag, 1998.
35. J. Stark and A. Ireland. Towards automatic imperative program synthesis through proof planning. In 14th *IEEE International Conference on Automated Software Engineering*. IEEE Computer Society, 1999.
36. A.M. Turing. Checking a large routine. In *Report of a Conference on High Speed Automatic Calculating Machines*. University Mathematical Laboratory, Cambridge, UK, 1949.
37. E. Visser. Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5. In *Rewriting Techniques and Applications (RTA)*, LNCS, 2001.
38. L. Whiting and M. Hill. Safety analysis of hawk in flight monitor. In *Workshop on Program Analysis For Software Tools and Engineering*, 1999.