An Integrated Approach to High Integrity Software Verification

Andrew Ireland[1], Bill J. Ellis[1], Andrew Cook[1],
Roderick Chapman[2], Janet Barnes[2]

[1]School of Mathematical and Computer Sciences
Heriot-Watt University
Riccarton, Edinburgh, Scotland, EH14 4AS
a.ireland@hw.ac.uk

[2]Praxis High Integrity Systems Ltd,
Bath, England, BA1 1PX

Abstract. Using automated reasoning techniques, we tackle the niche activity of proving that a program is free from run-time exceptions. Such a property is particularly valuable in high integrity software, *e.g.* safety or security critical applications. The context for our work is the SPARK Approach for the development of high integrity software. The SPARK Approach provides a significant degree of automation in proving exception freedom. However, where this automation fails, the programmer is burdened with the task of interactively constructing a proof and possibly also having to supply auxiliary program annotations. We minimise this burden by increasing the automation, via an integration of proof planning and a program analysis oracle. We advocate a "co-operative" integration, where proof-failure analysis directly constrains the search for auxiliary program annotations. The approach has been successfully tested on industrial data.

1 Introduction

There is renewed interest in the formal verification of computer software. Various tools are emerging that use verification techniques to automatically reveal useful properties about software [1, 4, 25]. Such advances are supported through two key factors. Firstly, there is a shift away from full functional verification toward property based verification. By accepting more conservative verification the automation task becomes more tractable. Secondly, there exists a wealth of diverse automated reasoning tools. By exploiting and integrating these existing tools, a significant degree of automation can be realised. Thus, viable verification systems can be produced by matching the right kind of property verification with the right kind of automated tool support.

Here we follow this trend, applying automated reasoning techniques to the SPARK Approach [2], as developed by Praxis High Integrity Systems Ltd (henceforth Praxis). The SPARK Approach is designed for the development of high integrity software, as seen in safety and security critical applications. The SPARK

Approach advocates "correctness by construction", where the focus is on bug prevention rather than bug detection. SPARK has been applied successfully across a wide range of applications including railway signalling, smartcard security and avionics systems such as the Lockheed C130J and Eurofighter projects. The approach has been recognised by the US National Cyber Security Partnership as one of only three software development processes that can deliver sufficient assurance for security critical systems [44].

The formal verification capabilities of the SPARK Approach are most commonly used for *exception freedom proofs*, *i.e.* proving that a system is free from run-time exceptions. Such program reasoning represents an important task in the development of high integrity software. For instance, Ariane 5 was lost due to an integer overflow at run-time [22], and buffer overflows are the most common form of security vulnerability [18]. Industrial strength evidence [15] shows that the SPARK toolset can typically automate around 90% of the verification task for proving exception freedom. The remaining 10% must be manually discharged by the programmer. This task will involve interactively constructing proofs inside the SPARK proof tools and manually discovering any necessary program properties. For large systems, discharging the remaining 10% can present a significant challenge.

Our primary interest is in addressing this verification challenge by increasing the level of automation. Central to our approach is an integration of automated reasoning and program analysis. In particular, we use *proof planning* [8] in order to control the search for proofs and a program analysis oracle in order to generate auxiliary program properties. The novelty of our approach lies in the nature of our integration. We have developed a "co-operative" style of integration, where partial success during proof planning constrains subsequent program analysis.

Background material is presented in §2. In §3 we compare proof inside the SPARK Approach with our approach. The details of our approach are presented in §4, §5, §6, §7, and §8. Implementation details and our results are presented in §9 and §10 respectively. Related work is discussed in §11. The feasibility of transferring our approach into an industrial tool is explored in §12, while in §13 limitations of our approach and future work are outlined. Our conclusions are presented in §14.

2 Background

2.1 The SPARK Approach

At the heart of the SPARK Approach is the SPARK programming language. The SPARK programming language is defined as a subset of Ada [36]. To make static analysis feasible SPARK excludes many Ada constructs, such as pointers, dynamic memory allocation and recursion. SPARK includes an annotation language that supports flow analysis and formal verification. The annotations are supplied within regular Ada comments, allowing a SPARK compliant program to be compiled using any Ada compiler. An example SPARK subprogram called Filter is shown in Figure 1.

```
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
      --# assert true;
      if A(I)>=0 and A(I)<=100 then
        R:=R+A(I);
      end if;
    end loop;
  end Filter;
end FilterPackage;
```

Note that SPARK annotations are inserted inside Ada comments via the special prefix `--#`. The annotation `--# derives R from A;` conveys that the value of `R` is derived from array `A`. The Examiner checks this specification automatically via information flow analysis. The annotation `--# assert true;` represents an invariant. Explicit invariants are not mandatory as the Examiner will automatically insert default invariants in their absence. However, to facilitate understanding, we use trivially true invariants to highlight the location of the default invariant within the loop.

Fig. 1. SPARK code: Filter subprogram

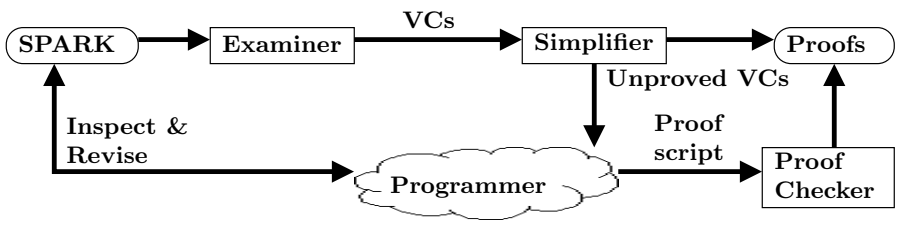

Fig. 2. The SPARK Approach

The SPARK Approach is supported through a collection of interacting tools, as shown in Figure 2. The Examiner performs the analysis of SPARK code. It ensures that the submitted code conforms to the SPARK language. Further, it conducts data flow and information flow analysis [5]. The Examiner also supports formal verification by building directly upon the Floyd/Hoare [26, 28] style of reasoning. Annotations may be inserted to supply a functional specification, in the form of preconditions and postconditions. The Examiner implicitly inserts an invariant at each loop, conveying limited type information. These default invariants may be strengthened by providing explicit invariant annotations. The Examiner includes a *verification condition generator* (VCG), reducing the task of verifying that a program meets its specification to proving a number of conjectures, called *verification conditions* (VCs). The Examiner can generate VCs stating both partial correctness and exception freedom.

Two additional tools support the proof of VCs. Firstly, there is the SPADE Simplifier (henceforth Simplifier), a special purpose theorem prover that automatically simplifies or discharges VCs. Secondly, the SPADE Proof Checker (henceforth Proof Checker) provides an interactive proof development environment. Where the Simplifier fails to prove a VC, the user must intervene. For each VC the Simplifier fails to discharge the user may attempt to:

- **Perform Proof** - Interactively prove the VC using the Proof Checker.
- **Strengthen Specification** - Strengthen the program specification, thereby enriching the properties in the VC, so that its proof can be found.
- **Identify Inconsistency** - Show that there is an inconsistency between the program and the specification by identifying a counter-example to the VC.

Note that the soundness of the SPARK Approach depends entirely on the soundness of the the SPARK toolset, *i.e.* the Examiner, the Simplifier and the Proof Checker.

Unsurprisingly, the VCs not discharged by the Simplifier tend to be the more difficult proof problems. Further, a typical high integrity system will generate thousands of VCs. Despite the success of the Simplifier, typically hundreds of proof failures need to be addressed per application. Additionally, interactive proofs will be tuned to a particular VC and hence a particular version of the system. As the system is changed these interactive proofs may break and require

refinement. Taken together, these factors present a significant bottle-neck to the practical completion of exception freedom proofs.

2.2 Proving Exception Freedom

By definition, SPARK eliminates many of the run-time exceptions that can be raised within Ada. However, index, range, overflow and division checks can still raise exceptions in SPARK code. The Examiner generates *exception freedom* VCs (EFVCs) that faithfully reflect the behaviour of these run-time checks. The index check ensures that an array access occurs within the bounds of the array. The range and overflow checks ensure that variables remain within their declared bounds. Finally, the division check prevents a division by zero, essentially restricting the denominator to bounds that exclude zero.

To illustrate the task of proving exception freedom we return to the Filter subprogram shown in Figure 1. Consider the assignment statement in the `then`-branch, *i.e.* `R:=R+A(I)`, whose corresponding EFVC is given in Figure 3. This particular statement generates two run-time checks within SPARK. Firstly, there is an index check to ensure that the value of `I` does not exceed the range of array `A`. This corresponds to proving conclusions `C1` and `C2`. Secondly, there is an overflow check to ensure that the expression `R+A(I)` assigned to `R` is within the type of `R`, *i.e.* `Integer`. This corresponds to proving conclusions `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 the default invariant is not sufficiently strong.

2.3 Proof Planning

Central to our work is an automated reasoning paradigm called *proof planning* [8]. Proof planning automates the search for proofs through the use of high-level proof outlines, known as *proof plans*. A proof plan is defined by a set of *methods*. Each method expresses preconditions for the applicability of a generic proof *tactic*. A method represents a partial specification of a generic tactic. For a given conjecture, method preconditions are used to control the selection and instantiation of generic tactics during proof planning. Once generated, an instantiated tactic can then be used to control proof construction within an appropriate tactic based proof checker.

Within proof planning, methods are complemented by proof *critics* [29]. Critics are associated with the partial success of proof methods and support the automatic analysis and patching of failed proof attempts. Applications of proof-failure analysis and proof patching include conjecture generalization and lemma discovery [30, 31], loop invariant discovery [34, 49], and refining faulty conjectures [43]. A key feature of most critics is the use of meta-variables in delaying choice during proof search, known as *middle-out reasoning* [10]. Middle-out reasoning is not restricted to proof patching, for instance is has been used in guiding proof search within the context of program synthesis [35, 40].

```
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 .
H6: r >= integer__first .
H7: r <= integer__last .
     ->
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 Filter subprogram in Figure 1, three of which are EFVCs. The EFVC above corresponds to proving that the assignment `R:=R+A(I)` can never raise an exception. Note that `element(a, [loop__1__i])` denotes accessing array `a` at index `loop__1__i`. Note also that the EFVC is presented in the format generated by the Examiner. The VC contains four implicitly conjoined conclusions, *i.e.* `C1` through to `C4`. We consider each conclusion individually, thus this VC corresponds to four distinct goals, with each goal sharing the same hypotheses. In this EFVC, `H1`, `H2` and `H3` are a consequence of the default invariant automatically inserted by the Examiner.

Fig. 3. An exception freedom verification condition (EFVC)

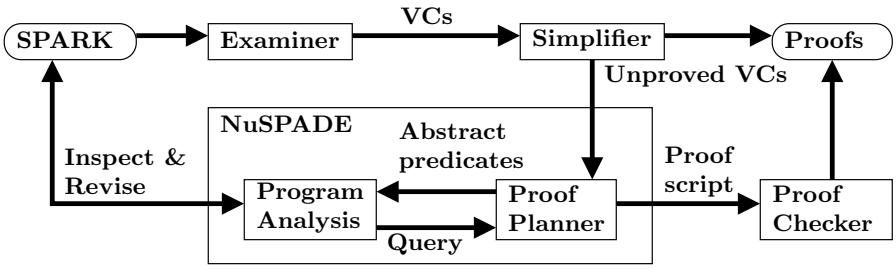

Fig. 4. NuSPADE and the SPARK Approach

2.4 Program Analysis

Program analysis involves automatically generating program properties via code level analysis. The field covers a diverse range of techniques, *e.g.* data flow analysis, information flow analysis, constraint based analysis and abstract interpretation [45]. For our application we are focusing on proving exception freedom within SPARK. This task reduces to proving that variables lie within legal bounds. In general, the SPARK type system reveals strong constraints on variables, supporting exception freedom proofs. However, more sophisticated constraints are often required when variables are modified within a loop. Consequently, we are primarily interested in program analysis techniques that automatically discover loop invariants.

The Runcheck verifier [27] was probably the first system to tackle exception freedom verification. The system included program analysis, building on recurrence relations, to automatically discover loop invariants. The technique was first introduced by Elspas *et al* [21] as the "difference equations method". A similar approach was adopted by Katz and Manna [38]. As noted by Cousot [17], the use of recurrence relations in this manner fits within the general methodology of abstract interpretation. The limitation of using recurrence relations as a basis for generating loop invariants are well known [14]. However, in special purpose applications, such as proving exception freedom, the technique has proved very useful in practice.

3 NuSPADE and the SPARK Approach

NuSPADE[1] aims to increase automation within the SPARK Approach. It fits directly inside the SPARK Approach, automating the tasks currently undertaken by the programmer, as illustrated in Figure 4.

The two key components in NuSPADE are a proof planner and a program analysis oracle. The proof planner provides overall control, exploiting the services

[1] The name 'NuSPADE' emphasises that we are extending the capabilities of the SPADE proof tools.

of the program analysis oracle where necessary. Each VC not proved by the Simplifier is sent to the proof planner. Where proof planning successfully produces a tactic, a proof script is automatically generated from the tactic and used to construct a proof within the Proof Checker. If proof planning fails, proof-failure analysis may identify missing proof context. The form of this proof context is described via *abstract predicates*, *i.e.* simple patterns that describe the structure of desired program properties. These abstract predicates are provided to the program analysis oracle. The program analyser examines the code corresponding to the targeted VC, searching for properties which match the abstract predicates. Where successful, the discovered properties are used to revise the program specification. The overall process is iterative, *i.e.* once a specification is revised the process of VC generation and proof planning is repeated. The expectation is that on each iteration progress will be made towards completing the verification. If NuSPADE fails, the programmer will still need to intervene.

As NuSPADE aims to emulate the activity of the programmer, its behaviour can be summarised in this context. For each VC that the Simplifier fails to discharge NuSPADE will attempt to:

- **Perform Proof** - Prove the VC using proof planning. Where a proof plan is found it is translated into a proof script and checked within the Proof Checker.
- **Strengthen Specification** - Where a proof planning attempt fails, proof-failure analysis combined with program analysis is used to strengthen a program specification.
- **Identify Inconsistency** - Proof planning searches to identify the VC as being trivially false.

Note that every NuSPADE action occurs inside the context of the SPARK toolset. Thus, where employing NuSPADE, the soundness of the SPARK Approach remains entirely dependent on the soundness of the SPARK toolset.

Below we explain in detail the relationship between proof planning, proof-failure analysis and program analysis. Our techniques are illustrated using the Filter subprogram shown in Figure 1.

4 Proof Planning

Here we describe two proof plans used to control proof search within NuSPADE. The first deals with exception freedom VCs while the second deals with the VCs associated with loop invariants.

4.1 Exception Freedom Proof Plan

Proving exception freedom typically involves reasoning about inequalities. Our exception freedom proof plan defines a strategy for decomposing inequality conclusions so that the available hypotheses can be applied. The decomposition of

inequalities requires the discovery of an intermediate bound. Our proof plan exploits middle-out reasoning to find a suitable intermediate bound. The methods that define the exception freedom proof plan are described below, in the order in which they are used within proof planning.

Elementary Method The elementary method is applicable to trivial goals that will be automatically discharged by the Proof Checker. The method closes the current goal.

Simplify Method The simplify method is applicable to goals whose complexity can be reduced through an available substitution law. For example, the simplify method may replace constants with explicit values. In particular, simplify aims to transform goals so that the elementary method becomes applicable.

Fertilise Method Any occurrence of a hypothesis within the conclusion may be replaced with *true*. The fertilise method (preconditions given in Figure 5) seeks to perform this simplification by finding such a match. To extend applicability the matching process may involve elementary forward chaining and hypothesis instantiation.

Preconditions for fertilise method:

1. There exists a hypothesis H that matches a subterm of conclusion C, modulo elementary forward chaining and hypothesis instantiation.

Fig. 5. Preconditions for the fertilise method

Transitivity Method The transitivity method (preconditions given in Figure 6) begins a sequence of reasoning aimed at discharging conclusions that specify bounds on an expression. Key to the success of this reasoning is having explicit bounds on all variables in the expression. Consider, for example, C4 in Figure 3

$$r + element(a, [i]) \leq integer\_last \ . \tag{1}$$

An application of the transitivity method to (1) gives

$$(r + element(a, [i]) \leq X_1) \wedge (X_1 \leq integer\_last) \ . \tag{2}$$

Note that the introduction of meta-variable X_1 prepares the way for the decomposition of $r + element(a, [i])$, *i.e.* an application of the decomposition method.

Preconditions for transitivity **method:**

1. There exists a conclusion of the form E *Rel* C.
2. For all variables V_i that occur within E there exists hypotheses of the form V_i *Rel* E_i and E_i *Rel* V_i.

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

Fig. 6. Preconditions for the transitivity method

Decomposition Method The decomposition method (preconditions given in Figure 7) is applicable to a conclusion subterm that involves a transitive relation. The aim of the decomposition method is to reduce this transitive relation into a number of simpler relations. For example, the left conjunct of (2) can be decomposed, giving

$$(r \leq X_2) \wedge (element(a, [i]) \leq X_3) \wedge (X_2 + X_3 \leq integer\_last) \ .$$

The decompositions considered by the decomposition method are supported through suitable substitution laws, *i.e.* equivalence or implication. Note that a proof plan may require multiple applications of the decomposition method.

Preconditions for decomposition **method:**

1. There exists a conclusion of the form E_1 *Rel* E_2.
2. There exists a substitution law for *Rel* justifying the decomposition of the conclusion.

Note that E_1 and E_2 denote expressions, while *Rel* denotes a transitive relation.

Fig. 7. Preconditions for the decomposition method

4.2 Loop Invariant and Inductive Proof Plans

Our loop invariant proof plan contains three methods in addition to those described above. These focus on verifying loop invariants and any auxiliary subgoals that arise.

Rippling Methods The rewriting strategy called *rippling* was originally developed to automate proof by mathematical induction [11, 12]. However, rippling has been shown to be applicable to a wider class of problems. In particular, rippling can be applied to the verification of loop invariants, as initially proposed

in [33, 34, 49]. Rippling works by identifying and reducing syntactic differences between formulae. We exploit the rippling strategy in our loop invariant proof plan. Below we provide a short description of rippling. For a full account see [3, 9, 11].

We implement rippling via an annotate method and a wave method. The annotate method automatically introduces meta-level annotations into a conclusion, identifying the syntactic differences between the conclusion and a given hypothesis. For example, given a hypothesis $f(i)$ and a conclusion $f(i+1)$, the annotate method will annotate the conclusion as

$$f(\boxed{i+1}^{\uparrow}) \,. \tag{3}$$

The annotated portion of the term, represented by shading, is know as the *wave-front*. This denotes the syntactic mismatch between the conclusion and the hypothesis. The arrow is used to indicate the direction in which the wave-front is moving, *i.e.* either outward or inward. Directed wave-fronts are used to guarantee termination of the method.

Wave-fronts are manipulated via annotated rewrite rules called *wave-rules*. Wave-rules are annotated in the same manner as the conclusion. For example, the rewrite rule[2]

$$f(X+1) \Rightarrow f(X) \wedge g(X)$$

may be annotated as

$$f(\boxed{X+1}^{\uparrow}) \Rightarrow \boxed{f(X) \wedge g(X)}^{\uparrow} \,. \tag{4}$$

Wave-rules target syntactic differences by only manipulating annotated terms in the conclusion. The wave method controls the application of wave-rules. For example, applying wave-rule (4) to (3) gives

$$\boxed{f(i) \wedge g(i)}^{\uparrow} \,. \tag{5}$$

In general, rippling will involve an arbitrary number of wave-rule applications. Eventually, the unannotated part of the conclusion will match the given hypothesis. For example, (5) now matches with the given hypothesis. At this stage, our fertilise method applies, leaving the simplified conclusion $g(i)$.

Induction Method Like rippling, the induction method is reused from previous work on proof by mathematical induction [11, 12]. Although rare within the application domain, the need for inductive proof arises where an additional lemma is required in order to complete a proof, *i.e.* situations where none of the SPADE axioms or rules are applicable.

[2] We use $\Rightarrow$ to denote rewrite rules and $\rightarrow$ to denote logical implication.

Generalise Method The generalise method is strongly linked with the induction method, since a generalisation step may be required in order to obtain a stronger induction hypothesis. Our generalise method uses the relatively simple heuristic of replacing common subterms by a universally quantified variable, as found in Nqthm [7].

5 Proof-Failure Analysis

Within NuSPADE, we have extended the role of proof planning critics. We use critics to provide an interface between our proof planner and our program analysis oracle. This interface enables critics to request additional program properties. Below we outline the four critics that were developed to support the automation of exception freedom proof.

5.1 Elementary Critic

The elementary critic (preconditions given in Figure 8) identifies unprovable goals by discovering counter-examples, *i.e.* values for variables that satisfy the hypotheses but not the conclusion. Patching the proof requires imposing tighter constraints on at least one of these variables. Constraint solving is used to find counter-examples. To illustrate, consider again `C4` in Figure 3

$$r + element(a, [i]) \leq integer\_last\ . \tag{6}$$

The elementary method fails to prove conclusion (6), leading to an invocation of the elementary critic. Note that *integer_last* is a constant, whose value is set by the programmer depending on the behaviour of their target compiler. Here we assume that *integer_first* and *integer_last* have the values -32768 and 32767 respectively. By inspecting the hypotheses associated with the goal (see Figure 3) we know that

$$element(a, [i]) \geq 0 \wedge element(a, [i]) \leq 100$$

and that

$$r \geq integer\_first \wedge r \leq integer\_last\ .$$

It then follows that $r + element(a, [i])$ can raise an overflow exception if r is in the range $integer\_last - 99 \ldots integer\_last$, *i.e.* $32668 \ldots 32767$. This reasoning is achieved automatically by the constraint solver. The counter-example identifies that r or $element(a, [i])$ must be constrained to complete a proof. In general, array constraints are unusual and, where they do exist, are difficult to find. Thus, the elementary critic guides the program analysis toward finding constraints on r by generating abstract predicates of the form

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

Preconditions for elementary **critic:**

- All preconditions for the elementary method fail.
- There exists a top-level conclusion of the form E *Rel* C.

Patch: Search for a counter-example to show that the given hypotheses are insufficient to prove exception freedom. If a counter-example is identified then abstract predicates are used to request tighter bounds from the program analysis oracle.

Note that E ranges over expressions, while C denotes a constant. *Rel* denotes a transitive relation.

Fig. 8. Preconditions for the elementary critic

While our constraint solving system works well in practise, it fails in the presence of large integers, *i.e.* numbers that lie beyond $-(2^{25}) \ldots 2^{25} - 1$. In these cases the elementary critic is not applied. Instead, the proof search progresses as normal, decomposing the goal into simpler subgoals. This can admit the application of the elementary critic or allow for false to be trivially derived.

5.2 Transitivity Critic

The transitivity critic (preconditions given in Figure 9) identifies missing hypotheses. The goal is patched by requesting that the program analysis oracle introduces the missing hypotheses.

Preconditions for transitivity **critic:**

- Precondition 1 of the transitivity method holds, *i.e.*
  . There exists a conclusion of the form E *Rel* C.
- Precondition 2 of the transitivity method fails, *i.e.*
  . There exists at least one variable V_i that occurs within E such that there does not exist a hypothesis within the proof context of the form V_i *Rel* E_i or E_i *Rel* V_i.

Patch: Generate abstract predicates which specify the missing hypotheses and send them to the program analysis oracle.

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

Fig. 9. Preconditions for the transitivity critic

5.3 Fertilise Critic

The fertilise critic (preconditions given in Figure 10) extends the applicability of the fertilise method by recognising a near match and transforming the goal accordingly. In particular, we focus on strengthening an inequality hypothesis to match a strict inequality conclusion. The fertilise critic has two alternative patches. The first works strictly at the proof planning level, modifying the goal to allow the fertilise method to succeed. This is possible where a property exists allowing the nearly matching hypothesis to be be strengthened and achieve an exact match with the conclusion. The second patch involves the program analysis oracle searching for the required property.

Preconditions for fertilise critic:

- All preconditions for the fertilise method fail, *i.e.*
  - There does not exist a hypothesis H that matches a subterm of conclusion C.
- There exists a hypothesis H of the form $A \leq B$ or $A \geq B$ and the conclusion C is strictly stronger, taking the form $A < B$ or $A > B$ respectively.

Patch: Conditional on a hypothesis H' of the form $\neg(A = B)$:

- Where H' exists: Combine H' with H to infer a strict inequality, supporting a match with the conclusion.
- Where H' does not exist: Generate a predicate to introduce a property of the form H'.

Fig. 10. Preconditions for the fertilise critic

5.4 Decomposition Critic

The decomposition critic (preconditions given in Figure 11) identifies a missing substitution law. Where this occurs, the user is informed of the problem, and asked to supply additional properties.

6 Program Analysis

6.1 Program Analysis Oracle

Our program analysis offers no soundness guarantees. To emphasise this point, we refer to our system as a *program analysis oracle*. The soundness of our overall approach is guaranteed, *i.e.* it relies upon the soundness of VC generator and Proof Checker. The program analysis oracle has strong similarities with abstract interpretation. The source code is translated into a flowgraph. Each variable at

Preconditions for decomposition **critic:**

- Precondition 1 of the decomposition method holds, *i.e.*
  . There exists a conclusion of the form E_1 *Rel* E_2.
- Precondition 2 of the decomposition method fails, *i.e.*
  . There exists no substitution law for *Rel* justifying the decomposition of the conclusion.

Patch: Report the need for additional properties.

Note that E_1 and E_2 denote expressions, while *Rel* denotes a transitive relation.

Fig. 11. Preconditions for the decomposition critic

each program point is associated with an abstract state. This state aims to describe the possible values that its corresponding variable may take. Operations on the abstract states are defined, allowing the flowgraph to be symbolically executed. Once the symbolic execution is complete, the final values of the abstract states define constraints on the program variables. We represent our program analysis heuristics as *program analysis methods*. Each method describes an abstract state and corresponding operations on this abstract state. Most methods build upon the results of earlier methods. Thus, methods are executed in order of complexity, gradually discovering increasingly rich abstract states. Once all of the methods have executed the strongest abstract states satisfying the abstract predicates are exported as the final result.

Our adoption of a program analysis oracle brings several key benefits. Primarily, we avoid having to formalise and prove the soundness of our program analysis heuristics. This enables the rapid development and reconfiguration of sophisticated heuristics. Further, external tools may be exploited during program analysis without having to justify their correctness. In particular, given our domain of exception freedom, we require equational reasoning services. This is achieved by exporting conjectures to our proof planner. The resulting plans are not explicitly checked, as our oracle does not require such guarantees.

6.2 Method: Type

Type information directly reveals the legal bounds of every variable. Such bounds are valuable in exception freedom proofs. This is particularly the case for high integrity software, which tends to adopt tightly constrained types wherever possible. A variable is only within its type once it has been assigned to. Thus this method involves retrieving the type of variables and identifying the code points where variables must have been assigned to.

For example, consider the PolishFlag subprogram in Figure 12. The variables `I` and `J` are declared to be of type `ARPO_T` and variable `T` is declared to be of type `AC_T`. While `I` and `J` are always assigned values prior to reaching the invariant

the same is not true of T. Thus, at the invariant, the abstract states will only reveal the type of I and J. This information can be expressed via the following candidate invariant.

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

```
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+1;
    loop
      --# assert true;
      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. 12. SPARK code: PolishFlag subprogram

6.3 Method: For Loop Range

Every SPARK for loop counter variable has a declared type. Further, this type may be constrained by imposing an additional range restriction. Similar to type

information, this range directly reveals the bounds of the variable and is valuable in exception freedom proofs.

For example, consider the Search subprogram in Figure 13. The loop counter variable `I` is declared to be of type `AR_T` and is constrained to be inside the range `L` to `U`. This inspires abstract states which can be expressed with the following candidate invariant.

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

```
package SearchPackage 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 Search(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 SearchPackage;
```

```
package body SearchPackage  is
  procedure Search(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
      --# assert true;
      if A(I)=F then
        R:=I;
        exit;
      end if;
    end loop;
  end Search;
end SearchPackage;
```

Fig. 13. SPARK code: Search subprogram

6.4 Method: Non-looping Code

Non-looping code is more susceptible to static analysis than looping code. Without loops, the number of paths through the code can be statically determined,

allowing each path to be considered individually. Note that subprograms containing loops will still contain sections of non-looping code.

At the start of a SPARK subprogram an arbitrary variable `x` will either have been assigned some unknown initial value ($x^{\sim}$) or will be undefined (*undef*). There then follows non-looping code, where variables are modified through assignments until a loop or the end of the subprogram is reached. Each assignment to `x` will take the general form

$$\mathtt{x} := F(\mathtt{v}_1, \mathtt{v}_2, \ldots \mathtt{v}_n) \ ,$$

where $F(\mathtt{v}_1, \mathtt{v}_2, \ldots \mathtt{v}_n)$ represents arbitrary function calls on program variables $\mathtt{v}_1, \mathtt{v}_2, \ldots \mathtt{v}_n$. Every program variable `v`, at each program point p, has an abstract state $\mathtt{v}^p$. To minimise complexity the abstract states are restricted to a particular class of formula. The formula only contains single value variables, constants, and regular arithmetic and relational functions. In general, following an assignment to `x`, its abstract state will take the following form

$$\mathtt{x}^p = F(\mathtt{v}_1^p, \mathtt{v}_2^p, \ldots \mathtt{v}_n^p) \ .$$

This general mechanism supports the propagation of abstract states through non-looping code. However, to remain within our restricted representation for abstract states, it is often necessary to apply approximations. As only single value variables are represented directly, array elements are approximated to the extreme bounds of their type. Similarly, as only standard functions are considered, any arbitrary function call is approximated to the extreme bounds of its return type. Despite these approximations, relatively complex abstract states can still emerge. However, by exploiting contextual information and employing lightweight equality reasoning, effective simplifications are often possible.

For example, consider the Clip subprogram shown in Figure 14. There are three paths through this code. At initialisation $\mathtt{v}^0 = v^{\sim}$, as `v` is an import (`in`) variable, and $\mathtt{r}^0 = \mathit{undef}$ as `r` is an export (`out`) variable. The first path involves entering the outermost `then` branch yielding the property $\mathtt{v} < \mathtt{i\_t'first}$ and an assignment giving $\mathtt{r}^1 = \mathtt{i\_t'first}$. The second path involves failing the outermost `then` branch and entering the innermost `then` branch. This yields the properties $\neg(\mathtt{v} < \mathtt{i\_t'first})$ and $\mathtt{v} > \mathtt{i\_t'last}$ and an assignment giving $\mathtt{r}^2 = \mathtt{i\_t'last}$. The final path enters the outermost `else` branch. This yields the properties $\neg(\mathtt{v} < \mathtt{i\_t'first})$ and $\neg(\mathtt{v} > \mathtt{i\_t'last})$ and an assignment giving $\mathtt{r}^3 = \mathtt{v}^{\sim}$. As the value $\mathtt{v}^{\sim}$ is unknown it offers a weak constraint. However, by exploiting the type information gathered by earlier methods, it can be found that $\mathtt{r}^3 \geq \mathtt{i\_t'first} \wedge \mathtt{r}^3 \leq \mathtt{i\_t'last}$. At this stage the three separate paths merge to give

$$\begin{aligned} \mathtt{r}^4 &= \mathtt{i\_t'first} \quad \vee \\ \mathtt{r}^4 &= \mathtt{i\_t'last} \quad \vee \\ (\mathtt{r}^4 &\geq \mathtt{i\_t'first} \wedge \mathtt{r}^4 \leq \mathtt{i\_t'last}) \end{aligned}$$

which can be simplified as

$$\mathtt{r}^4 \geq \mathtt{i\_t'first} \wedge \mathtt{r}^4 \leq \mathtt{i\_t'last} \tag{7}$$

and expressed as the following candidate assertion.

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

```
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. 14. SPARK code: Clip subprogram

6.5 Method: Looping Code

Looping code introduces significant complexities over non-looping code. The discovered abstract states must be invariant to accommodate every potential loop iteration. As noted in §2.4, invariant properties can be discovered by solving recurrence relations. We build upon this observation, exploiting the services of a powerful recurrence relation solver. We decompose our looping code analysis into four sub-methods. The first two sub-methods discover generic recurrence relations to describe the available loops. The third sub-method instantiates these generic recurrence relations based on the behaviour of the surrounding code. The fourth sub-method structures the discovered recurrence relations for inclusion as invariants.

Sub-Method: Generate And Solve Recurrence Relations This method identifies and solves recurrence relations. The innermost loop without solved recurrence relations is tackled next. To represent a general iteration n every variable's abstract state $\mathtt{v}^p$ is treated as $\mathtt{v}^p_{(n)}$. To begin, every variable is initialised to its value on the previous iteration as $\mathtt{v}^p_{(n)} = \mathtt{v}^p_{(n-1)}$. With this initialisation our non-looping method for one iteration of the loop generates abstract states that can be extracted as recurrence relations.

For example, return to the Filter example in Figure 1. Variables `i` and `r` are initialised to their values on the previous iteration. Variable `i` is implicitly assigned once each iteration via `I:=I+1`, generating a final abstract state that can be expressed as the recurrence relation $\mathtt{i}_{(n)} = \mathtt{i}_{(n-1)} + 1$. This can then be solved as $\mathtt{i}_{(n)} = \mathtt{i}_{(0)} + n$, *i.e.* the value of `i` on iteration n is equal to the initial value of `i` ($\mathtt{i}_{(0)}$) plus the current iteration number (n).

There are two separate paths to consider for variable `r`. The first path involves not entering the `if` statement. Consequently, `r` is unchanged and the final abstract state for `r` is its initial value. This gives the recurrence relation $\mathtt{r}_{(n)} = \mathtt{r}_{(n-1)}$, which is solved as $\mathtt{r}_{(n)} = \mathtt{r}_{(0)}$. The second path involves entering the `if` statement. The condition `A(I)>=0 and A(I)<=100` reveals the property $\mathit{element}(\mathtt{a}, \mathtt{i}) \geq 0 \wedge \mathit{element}(\mathtt{a}, \mathtt{i}) \leq 100$, and the assignment statement `R:=R+A(I)` leads to the abstract state $\mathtt{r}^1_{(n)} = \mathtt{r}_{(n-1)} + \mathit{element}(\mathtt{a}, \mathtt{i})$. As noted in §6.4, our analysis approximates in the presence of array elements. In this case, the context information leads to the following two extreme cases

$$\left[\mathtt{r}^1_{(n)} = \mathtt{r}_{(n-1)} + 0, \mathtt{r}^1_{(n)} = \mathtt{r}_{(n-1)} + 100\right] .$$

Note that $[c_1, c_2, \ldots, c_n]$ defines a range of values through a collection of extreme cases. Once the details of each case are known the collection may be ordered and simplified into regular inequality bounds. Here the abstract state is extracted as two extreme recurrence relations and solved as

$$\left[\mathtt{r}_{(n)} = \mathtt{r}_{(0)}, \mathtt{r}_{(n)} = \mathtt{r}_{(0)} + 100 * n\right] .$$

Note that the extreme recurrence relations subsume the case where not entering the `if` statement.

Sub-Method: Generalise Loop The generation and solving of recurrence relations is applicable to a single loop. To support the analysis of loops that contain loops, the modifications made to a variable within a loop is abstracted to a single assignment. This abstracted assignment conceals the nested loop, allowing the outer loop to be analysed.

Unsurprisingly, the quality of the recurrence relations found on the outer loop depend strongly on the quality of the abstracted assignment. In general, this technique is most effective where the nested loop performs simple iteration. For example, this situation might occur where using two nested `for` loops to iterate over a two dimensional array.

Sub-Method: Instantiate and Simplify All of the solved recurrence relations above are general, referencing undefined initial values of the form $\mathtt{v}_{(0)}$. Once the outermost loops have been solved the actual initial values may be inserted. This process works in reverse to the discovery of recurrence relations, instantiating from the outermost loop toward the inner loop. These instantiations introduce specific values, often supporting additional simplifications.

For example, return to the Filter example in Figure 1. At entry to the loop the abstract state found for both `i` and `r` will be 0. Thus, both $\mathtt{i}_{(0)}$ and $\mathtt{r}_{(0)}$ may be replaced with 0 giving the simplified recurrence relations

$$\begin{gathered} i_{(n)} = n \\ \left[\mathtt{r}_{(n)} = 0, \mathtt{r}_{(n)} = 100 * n\right] \ . \end{gathered}$$

This may be simplified further by translating the collection into inequalities

$$\begin{gathered} \mathtt{i}_{(n)} = n \\ \mathtt{r}_{(n)} \geq 0 \wedge \mathtt{r}_{(n)} \leq n * 100 \ . \end{gathered}$$

Sub-Method: Restructure for Invariant Based on the abstract predicates the abstract states are extracted as candidate invariants. In most cases the abstract states can be directly expressed as candidate invariants. However, for solved recurrence relations, it is necessary to eliminate references to the artificial loop iteration variable n. Such an elimination can be achieved by deriving an expression for n in terms of the actual program variables. In practise, this can often be achieved by transforming a recurrence relation associated with a loop counter variable.

For example, return to the Filter example in Figure 1. At the invariant it is known that $\mathtt{i}_{(n)} = n$, thus all occurrences of n can be replaced with the variable i. Exploiting this transformation the abstract value for variable `r` may now be expressed as the following candidate invariant.

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

6.6 Method: For Loop Entry

The end-point of a for-loop range may be an expression containing program variables. The Ada semantics specify that the end-point is evaluated only when the loop is entered. However, any variables referenced in the end-point may be modified within the loop. Therefore the evaluation of an end-point expression at loop entry may differ from its evaluation on subsequent loop iterations. To encode these semantics the Examiner clones any variables in an end-point expression as entry variables. These variables may be referenced in annotations, with a variable `X` having a corresponding entry variable `X%`.

Where a variable `v` remains unchanged within a loop it will generate a recurrence relation of the form $\mathtt{v}_{(n)} = \mathtt{v}_{(n-1)}$. If this variable is also an entry variable

it is valid, and useful, to equate the entry variable with its regular variable within the loop.

For example, consider the Search subprogram shown in Figure 13. This includes a for-loop that is constrained to be within a range. The end-point of this range is the variable `U`. The abstract state generated for `U` will reveal that the variable remains unchanged within the loop. This property can be expressed with the following candidate invariant.

```
--#assert U=U%;
```

7 Filter Subprogram Revisited

To describe the overall behaviour of our program analysis we return to the Filter subprogram in Figure 1. In first analysing this example the Examiner generates VCs, not all of which are proved by the Simplifier. These remaining VCs trigger the first iteration of NuSPADE. While our proof planning on the remaining VCs fails, the elementary critic (see §5.1) generates an abstract predicate of the form

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

Our program analysis oracle is invoked, with the looping method (see §6.5) satisfying the abstract predicate above with the following candidate invariant.

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

The addition of this invariant revises the subprogram specification, requiring the Examiner to regenerate the VCs. Two of the resulting VCs are not proved by the Simplifier, triggering a second iteration of NuSPADE. The proof planning on the remaining VCs is successful, and completes the exception freedom proofs. The details of the remaining VCs and the associated proof planning is described below.

7.1 Exception Freedom Proofs

Here we focus on the proof of the remaining EFVC, shown in Figure 15. We focus on proving the conclusion

$$r + element(a, [i]) \leq integer\_last \ . \tag{8}$$

Note that the proof context now includes the hypothesis

$$r \leq i * 100 \tag{9}$$

as well as the hypothesis

$$element(a, [i]) \leq 100 \ . \tag{10}$$

```
H1: r >= 0 .
H2: r <= loop__1__i * 100 .
H3: 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))) .
H4: loop__1__i >= ar_t__first .
H5: loop__1__i <= ar_t__last .
H6: element(a, [loop__1__i]) >= 0 .
H7: element(a, [loop__1__i]) <= 100 .
H8: r >= integer__first .
H9: r <= integer__last .
     ->
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 .
```

Fig. 15. Revised exception freedom verification condition (EFVC)

The proof planning of conclusion (8) begins with an application of the transitivity method, giving rise to a conjunction of the form

$$(r + \mathit{element}(a, [i]) \leq X_1) \wedge (X_1 \leq \mathit{integer\_last}) \; . \tag{11}$$

Recall that X_1 denotes a meta-variable. The decomposition method uses substitution laws to decompose the inequalities. Here the decomposition method applies the following substitution law

$$(W \leq Y) \wedge (X \leq Z) \rightarrow (W + X) \leq (Y + Z) \; . \tag{12}$$

Given that we are performing a backward style of proof, the application of (12) to the left-hand conjunct of (11) gives rise to

$$(r \leq X_2) \wedge (\mathit{element}(a, [i]) \leq X_3) \wedge (X_2 + X_3 \leq \mathit{integer\_last}) \; . \tag{13}$$

Note that as a side-effect of the decomposition step, X_1 is instantiated to be $X_2 + X_3$. The fertilise method is now applicable. Exploiting hypotheses (9) and (10), fertilisation simplifies (13), instantiating X_2 to be $i * 100$ and X_3 to be 100 in the process. This leaves a proof residue of the form

$$(i * 100) + 100 \leq \mathit{integer\_last} \; . \tag{14}$$

Assuming that *integer_last* is the constant 32767, the remaining goal (14), is trivial and is discharged by the simplify and elementary methods.

```
H1:  r >= 0 .
H2:  r <= loop__1__i * 100 .
H3:  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))) .
H4:  loop__1__i >= ar_t__first .
H5:  loop__1__i <= ar_t__last .
H6:  element(a, [loop__1__i]) >= 0 .
H7:  element(a, [loop__1__i]) <= 100 .
H8:  r >= integer__first .
H9:  r <= integer__last .
H10: r + element(a, [loop__1__i]) >= integer__first .
H11: r + element(a, [loop__1__i]) <= integer__last .
H12: loop__1__i >= ar_t__first .
H13: loop__1__i <= ar_t__last .
H14: not (loop__1__i = ar_t__last) .
      ->
C1:  r + element(a, [loop__1__i]) >= 0 .
C2:  r + element(a, [loop__1__i]) <= (loop__1__i + 1) * 100 .
C3:  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))) .
C4:  loop__1__i + 1 >= ar_t__first .
```

Fig. 16. Revised loop invariant verification condition

7.2 Loop Invariant Proofs

Here we focus on the proof of the remaining loop invariant VC, shown in Figure 16. Given the hypotheses

$$r \leq i * 100 \tag{15}$$
$$element(a, [i]) \leq 100 \tag{16}$$

we focus on proving the conclusion

$$r + element(a, [i]) \leq (i + 1) * 100 \; . \tag{17}$$

The annotate method identifies the difference between (17) and hypothesis (15), giving

$$\boxed{r} + element(a, [i])^{\uparrow} \leq (\boxed{i} + 1)^{\uparrow} * 100 \; . \tag{18}$$

The wave-rules and rewrite rule required for the proof are as follows

$$(\boxed{X} + Y)^{\uparrow} * Z \Rightarrow (\boxed{X * Z}) + (Y * Z)^{\uparrow} \tag{19}$$
$$(\boxed{W} + X)^{\uparrow} \leq (\boxed{Y} + Z)^{\uparrow} \Rightarrow \boxed{W \leq Y} \wedge X \leq Z^{\uparrow} \tag{20}$$
$$1 * X \Rightarrow X \; . \tag{21}$$

The wave method applies (19) to the right-hand side of (18) to give

$$\boxed{r} + element(a, [i])^{\uparrow} \leq (\boxed{i * 100}) + (1 * 100)^{\uparrow} \; . \tag{22}$$

Using wave-rule (20), the wave method further ripples (22) to give

$$(\boxed{r \leq i * 100}) \wedge (element(a, [i]) \leq 1 * 100)^{\uparrow} \; . \tag{23}$$

Rippling is complete and the fertilise method applies, matching with hypothesis (15), leaving a proof residue of the form

$$element(a, [i]) \leq 1 * 100 \; . \tag{24}$$

The simplify method, using (21), reduces (24) to give

$$element(a, [i]) \leq 100 \; . \tag{25}$$

Matching against hypothesis (16) the fertilise method reduces (25) to *true*, and the elementary method completes the proof.

8 Proof Checking in the SPARK Approach

Proof planning decouples the processes of proof search and proof checking. As mentioned in §2.3, a successful proof planning attempt will generate a tactic. A tactic is a program which controls the the application of low-level inference rules. Tactics are able to perform calculations and make dynamic decisions. For example, a tactic might search through the available hypotheses to retrieve a desired formula.

Unfortunately, the Proof Checker is not tactic based. As there is no support for dynamic calculations, each theorem proving step is achieved via an explicit proof command. Consequently, in NuSPADE, the tactics generated must be translated into a prescriptive sequence of proof commands, that we call a *proof script*. Note that the Proof Checker is designed for interactive use, actively seeking to assist the user by automating small proof steps. While valuable to the user, these unplanned proof steps introduce significant complexities in generating prescriptive proof scripts. To overcome this problem we introduced a small collection of new commands to the Proof Checker. These commands simply bypass the automatic support, making the Proof Checker more controllable.

To illustrate, the tactic generated by NuSPADE for conclusion `C4` in Figure 15 is shown in Figure 17 and the tactic for conclusion `C2` in Figure 16 is shown in Figure 18. Taking a closer look, consider the fourth line of the tactic in Figure 18

```
wave(inequals(80),[],imp)
```

This corresponds to a single application of the `wave` method. In particular, it describes an application of rewrite rule

$$(W + X) \leq (Y + Z) \Rightarrow W \leq Y \wedge X \leq Z \ .$$

The proof script segment generated for this single rewrite step is shown in Figure 19. The verbosity is the result of two factors. Firstly, the lack of dynamic calculation naturally leads to a more detailed proof. Secondly, to gain control, we tended to use smaller grain proof commands.

9 Implementation

The two core components in NuSPADE are a proof planner and a program analysis oracle. The proof planner is implemented in SICStus Prolog. We built upon the Clam proof planner [13], in particular the critics enabled version [29, 30]. We modified Clam to make it aware of SPARK VCs and support our methods and critics. During our proof planning we exploit the services of a constraint solver. We simply used the constraint logic programming (CLP) capability provided with SICStus Prolog.

The program analysis oracle requires the translation from SPARK to a flowchart. Praxis provided the SPARK grammar and a tokeniser that was extracted from the Examiner. Exploiting these components the Stratego [50] system was used to

```
plan(vc6_4,c4,
simplify(filter_rules(3),[2],equ) then
  simplify(filter_rules(4),[2],equ) then
    simplify(filter_rules(7),[2],equ) then
      simplify(filter_rules(8),[2],equ) then
        trans(transitivity(1),loop__1__i*100+100) then
          decomp(inequals(80),[1]) then
            fertilize(h2) then
              fertilize(h12) then
                simplify(logical_and(5),[1],equ) then
                  simplify(logical_and(2),[],equ) then
                    simplify(filter_rules(4),[2]) then
                      elementary)
```

Fig. 17. Exception freedom verification condition (EFVC) tactic

```
plan(vc3_2,c2,
annotate(c2,h2) then
  wave(distribute(1),[2],equ) then
    wave(inequals(80),[],imp) then
      fertilize(h2) then
        simplify(logical_and(2),[],equ) then
          simplify(filter_rules(7),[2],equ) then
            simplify(filter_rules(8),[2],equ) then
              simplify(arith(2),[2],equ) then
                elementary)
```

Fig. 18. Loop invariant verification condition tactic

```
...
dosubgoalproperty r<=loop__1__i*100 and
element(a,[loop__1__i])<=1*100.
...
done c#1.
alldone.
dosubgoalproperty
r+element(a,[loop__1__i])<=loop__1__i*100+1*100.
   dosubgoalproperty r<=loop__1__i*100 and
                     element(a,[loop__1__i])<=1*100->
                     r+element(a,[loop__1__i])<=
                     loop__1__i*100+1*100.
      prove c#1 by implication.
      dosimplifyhypsconj.
      infer r+element(a,[loop__1__i])<=loop__1__i*100+1*100
                                        using inequals(80).
      done c#1.
      alldone.
   done c#1.
   alldone.
forwardchain h#21.
done c#1.
alldone.
...
```

Fig. 19. Proof script extract

translate a core subset of SPARK into our flowchart representation. The program analyser itself is implemented in SICStus Prolog. During execution the program analysis exploits results from the PURRS [47] recurrence relation solver. Further, the program analysis relies on our proof planner, and its constraint services, to perform various equational reasoning tasks.

The Proof Checker is implemented in Poplog Prolog. While not essential, developing our systems in Prolog eased the task of integrating NuSPADE within the SPARK Approach. Some changes were made to the Proof Checker to support the execution of automatically generated proof scripts. These changes involved very little new code, simply introducing more constrained versions of the existing proof commands.

10 Results

The analysis of NuSPADE drew upon two sources of data. Firstly, Praxis provided access to two safety critical industrial applications written in SPARK. One of the industrial applications was the Ship Helicopter Operating Limits Information System (SHOLIS) [39]. SHOLIS was the first system developed to meet the UK Ministry of Defence Interim Defence Standards 00-55 [42] and 00-56 [41]. The systems contains roughly 15,000 lines of executable code, leading to roughly 7000 VCs. Further details of the industrial applications are confidential. Our second set of examples are non-industrial and were drawn from text books.

The Simplifier is very effective at proving EFVCs, typically proving around 90% automatically [15]. Our techniques focus on the VCs the Simplifier fails to prove. Code containing loops tends to present the more difficult automation problems, thus we concentrated our efforts in this area. While industrial strength critical software systems are engineered to minimise the number and complexity of loops, we found that 80% of the loops that we did encounter were provable using our techniques. That is, our program analysis, guided by proof-failure analysis, automatically generated auxiliary program annotations that enabled subsequent proof planning and proof checking attempts to succeed. Two key reasons were identified for the 20% of loops that our techniques failed to prove. Firstly, in some situations a stronger precondition to the enclosing subprogram was required in order to complete a proof. Secondly, our program analysis is sometimes too coarse grained, *e.g.* insufficient discrimination between conditional branches. These limitations represent opportunities for future work.

Providing additional properties that correspond to program variable type information increases the Simplifier's success rate by around 2%. During the development of NuSPADE, Praxis extended the behaviour of the Examiner to automatically present type information for all variables that appear on the right hand side of an assignment. In isolation this technique is unsound as variables may not have been initialised. However, the Examiner's data flow analysis checks that every variable is assigned a value before its use, eliminating the potential error. The advantage of our technique is conducting explicit proofs rather than relying on the correctness of the Examiner's data flow analysis.

The actions taken by NuSPADE in tackling a subprogram depends on the nature of the subprogram, and capabilities of the Simplifier. Consequently, individual subprograms can exhibit quite different patterns of behaviour. However, in considering a collection of examples some general patterns begin to emerge. Our results on 21 loop based examples are presented in Figure 20. A more detailed analysis of these results is presented in the appendix.

Number of examples	**Goals at iteration**					
	1		**2**		**3**	
	U	**P**	**U**	**P**	**U**	**P**
6	12	0	-	-	-	-
14	37	1	14	43	-	-
1	8	0	4	6	0	6

In the table above we separate our 21 examples depending on the number of NuSPADE iterations involved. On each iteration we list the number of goals remaining after an invocation of the Simplifier. We partition these goals into **U**, those that are unprovable, and **P** those that are provable.

Fig. 20. Results summary table

The insertion of an invariant leads to an additional VC to prove that the invariant is correct. While the Simplifier is very effective on general exception freedom verification tasks, it is less capable in reasoning about loops. These factors explain why the number of remaining goals tends to increase after the first iteration of NuSPADE.

Following the final iteration of NuSPADE some goals remain unproved. Three goals required the insertion of preconditions, which is not considered by NuSPADE. Due to resource constraints, the Examiner does not generate exhaustive rules for large constant structures. The resulting missing information leads to eight unproved goals. In such situations NuSPADE directs the user to provide the missing rules. Once the rules are in place NuSPADE successfully plans the remaining goals.

Note that one example required three iterations of NuSPADE to complete. This unusual situation exists as two distinct kinds of properties were required. The first iteration of NuSPADE leads to one property being inserted. With this property in place the second iteration of NuSPADE progresses further, leading to the second property being inserted. Finally, once both properties are in place, the remaining VCs are proved by NuSPADE.

More generally, NuSPADE provided evidence as to the effectiveness of proof planning. Existing proof methods were incorporated into NuSPADE with relative ease. This highlights how proof planning facilitates the reuse of proof strategies.

In our particular domain of automated program reasoning we wanted to introduce a program analysis component. Proof planning naturally supported this integration via proof-failure analysis and its critics mechanism. This highlights the flexibility of proof planning, and its ability to be specialised for particular application domains.

11 Related Work

The Runcheck system [27] aimed to prove exception freedom for Pascal programs. Runcheck employs various heuristics to discover invariants and tackles proof with an external theorem prover. One of its heuristics involves the calculation of recurrence relations as *change vectors*, ignoring program context and collecting transformations made to variables. These change vectors are subsequently solved using a few rewrite rules that target 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.

Recently there has been renewed interest in approaches that employ theorem proving to support program development. The focus tends to be on finding errors rather than proving correctness. For example, ESC/Java [25] is an extended static checker for Java. Like SPARK, ESC/Java requires program annotations. Houdini [24] is able to automatically generate many of the annotations required by ESC/Java using predicate abstraction.

Our approach has similarities to Caveat [4], a static analysis tool designed for safety critical software written in a subset of ANSI C. It is developed by the French nuclear agency (CEA) and is used by Airbus in the formal verification of avionics software. Caveat relies upon the programmer to provide loop invariants. In the longer term, however, it is planned to integrate Caveat with an abstract interpretation tool in order to automatically discover invariants [4].

An integration of abstract interpretation and program proof has been explored in [48] for a simple imperative programming language. The approach uses abstract interpretation to generate program annotations. An algorithm is then used to generate Hoare style proofs for the annotated programs. The work currently focuses on verifying properties of integer ranges for program variables. Our approach differs in that we use proof-failure analysis to focus our program analysis efforts.

There exist program analysis systems that are formulated inside the abstract interpretation framework [16]. These systems tend to pinpoint the location of errors rather than prove correctness. The most noteworthy systems are Merle [51] and Polyspace [46]. 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. Our program analysis exploits the strong SPARK type model and program annotations to gain effective constraints on variables where analysing individual subprograms. Consequently, our analysis is

fairly cheap to perform and is applicable early in program development. Further, we replace the formalisation aspects of the abstract interpretation framework with explicit proofs. This simultaneously frees the development of our program analysis and offers strong correctness guarantees.

An alternative approach to developing high integrity software involves the generation of annotations during the construction of the program. This works well for niche application areas, as exemplified by the AutoBayes program synthesis system [6].

Finally, it should be noted that the work presented here represents a continuation of the work in [19, 20]. In particular, we significantly extended both the proof methods and critics. Further, we broadened the application of our program analysis to deal with nested loops. In addition, we extended our empirical testing of NuSPADE to include industrial test data.

12 SPADEase: Towards Technology Transfer

Following NuSPADE, the SPADEase project involved a six month industrial secondment. The SPADEase project aimed to facilitate the transfer of the ideas embodied in NuSPADE to an industrial environment. In practise, we planned to stimulate this knowledge transfer by developing an industrial minded version of NuSPADE, called SPADEase.

NuSPADE was developed as a research system, focusing on the hard automation problems. Consequently, NuSPADE lacks the integrity and accessibility expected of industrial tools. Thus, the primary goal of SPADEase was to re-factor NuSPADE as a practical system for industrial use. Given the short project time, emphasis was placed on consolidating the proof planning aspect of NuSPADE.

Like the NuSPADE proof planner, SPADEase is implemented in SICStus Prolog. This allowed SPADEase to reuse various predicates from the NuSPADE system, for example to support the rippling heuristic. To improve control and traceability, a new core planning engine was created for SPADEase.

As SPADEase was fundamentally a knowledge transfer project it did not extend the core functionality of NuSPADE. Nevertheless, SPADEase represents a significant advance over NuSPADE in terms of ease of use and deployment. SPADEase seamlessly integrates within the SPARK Approach, appearing to the external user as an enhanced version of the SPADE Simplifier. While SPADEase lacks a program analysis component its modular design readily supports the integration of this facility in the future.

13 Limitations and Future Work

In §5.1 we observed that the constraint solving system employed in NuSPADE fails on large numbers. Unfortunately, such large numbers can occur in EFVCs. In such cases the elementary critic is deactivated, allowing the proof plan to continue and other methods detect unprovability. Ideally, the constraint solver

should be sufficiently powerful to tackle the problems that occur in practise. This may be achieved by exploiting a more powerful constraint solver.

The transitivity method is most effective where considering linear expressions. Moreover, our translation from solved recurrence relations to inequality bounds does not support non-linear expressions. It would be possible to enrich the behaviour of our techniques to deal more effectively with non-linear expressions. However, as the vast majority of the programs we encountered led to liner expressions, we did not find a need for such enhancements in practice.

The decomposition method exploits substitution laws to decompose inequalities. If multiple substitution laws are applicable the decomposition method represents a choice point in the search for a proof. Ideally, each of these choice points should be explored before reporting on the success or otherwise of the proof plan. However, our elementary critic is not aware of alternative choice points and will detect and report the first false goal it finds. In principle, this weakness could mean not exploring proof paths that lead to proof. However, this potential problem has not arisen in practise, as the decomposition method is relatively constrained, leading to a sparse number of choice points. Nevertheless, this weakness may be addressed by introducing more global analysis into our critics.

In §10 we note that the introduction of a precondition can be a key step in completing a proof. Consequently, we are interested in investigating the automatic generation of preconditions. This process would involve discovering preconditions based on the form of the code. If the code contains errors the generated precondition could be flawed. Thus, any generated precondition would need to be manually reviewed to maintain the integrity of the system. Nevertheless, we feel that valuable automation may be achieved in the area of precondition discovery.

A possibility which we have not considered is exploiting the services of decision procedures. The use of decision procedures within proof planning is an active area of research [37]. Further, there exists powerful off-the-shelf decision procedures, *e.g.* the Integrated Canonizer and Solver (ICS) [23]. It is likely that decision procedures would bring valuable reasoning to NuSPADE.

Our focus here has been on proving exception freedom within the SPARK Approach. However, our approach can be naturally extended to tackle other program verification tasks. In particular, we are interested in applying our approach to automate partial correctness proofs. While this task represents a significant verification challenge our initial results [34, 32] suggest that some valuable progress can be made in this area.

Aspects of our program analysis reflect the behaviour of existing abstract interpretation systems, such as Merle and Polyspace. While these systems are not designed for formal verification their results could assist in the discovery of a formal proof. Consequently, we are interested in investigating the practicalities of integrating these tools into our program analysis oracle.

Finally, we view SPADEase as a first step toward technology transfer. We are actively looking to enhance our tool support further, with the intention of employing our techniques during the development of a *live* software project.

14 Conclusion

NuSPADE increases the level of automation when proving exception freedom in the SPARK Approach. NuSPADE has been successfully evaluated on industrial data, producing encouraging results. Based upon this work we developed SPADEase, providing an initial step toward technology transfer. Our approach tackles the verification task on two fronts by automating both proof search and specification strengthening. We build upon the proof planning framework. In particular, we use proof-failure analysis to guide the search for program properties. Our approach highlights the leverage that can be gained by a "co-operative" integration of distinct static analysis techniques.

Acknowledgements

In particular we would like to thank Peter Amey for his 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 a collaboration with Praxis High Integrity Systems Ltd. The NuSPADE project was supported by EPSRC grant GR/R24081 and the SPADEase project was supported by EPSRC Collaborative Training Account GR/T11289.

References

1. T. Ball and S.K. Rajamani. The SLAM project: Debugging system software via static analysis. In *Conference Record of POPL'02: The 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–3, Portland, Oregon, 2002.
2. J. Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison-Wesley, 2003.
3. D. Basin and T. Walsh. A calculus for and termination of rippling. *Journal of Automated Reasoning*, 16(1–2):147–180, 1996.
4. P. Baudin, A. Pacalet, J. Raguideau, D. Schoen, and N. Williams. Caveat: A tool for software validation. In *International Conference on Dependable Systems and Networks (DSN02)*. IEEE Computer Society, 2002.
5. Jean-Francois Bergeretti and Bernard A. Carré. Information-flow and data-flow analysis of while-programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(1), 1985.
6. B.Fischer and J.Schumann. Autobayes: A system for generating data analysis programs from statistical models. *Journal of Functional Programming*, 13(3):483–508, 2003.
7. R. S. Boyer and J S. Moore. *A Computational Logic Handbook*. Academic Press, 1988. Perspectives in Computing, Vol 23.
8. A. Bundy. The use of explicit plans to guide inductive proofs. In R. Lusk and R. Overbeek, editors, *9th International Conference on Automated Deduction*, pages 111–120. Springer-Verlag, 1988. Longer version available from Edinburgh as DAI Research Paper No. 349.

9. A. Bundy, D. Basin, D. Hutter, and A. Ireland. *Rippling: Meta-level Guidance for Mathematical Reasoning*. Cambridge University Press, 2005.
10. A. Bundy, A. Smaill, and J. Hesketh. Turning eureka steps into calculations in automatic program synthesis. In S. L.H. Clarke, editor, *Proceedings of UK IT 90*, pages 221–6. IEE, 1990. Also available from Edinburgh as DAI Research Paper 448.
11. A. Bundy, A. Stevens, F. van Harmelen, A. Ireland, and A. Smaill. Rippling: A heuristic for guiding inductive proofs. *Artificial Intelligence*, 62:185–253, 1993. Also available from Edinburgh as DAI Research Paper No. 567.
12. A. Bundy, F. van Harmelen, J. Hesketh, and A. Smaill. Experiments with proof plans for induction. *Journal of Automated Reasoning*, 7:303–324, 1991. Earlier version available from Edinburgh as DAI Research Paper No 413.
13. A. Bundy, F. van Harmelen, C. Horn, and A. Smaill. The Oyster-Clam system. In M. E. Stickel, editor, *10th International Conference on Automated Deduction*, pages 647–648. Springer-Verlag, 1990. Lecture Notes in Artificial Intelligence No. 449. Also available from Edinburgh as DAI Research Paper 507.
14. M. Caplain. Finding invariant assertions for proving programs. In *Proceedings of the International Conference on Reliable Software*, Los Angeles, CA, 1975.
15. R. Chapman and P. Amey. Industrial strength exception freedom. In *Proceedings of ACM SigAda*. 2002.
16. 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.
17. P. Cousot and R. Cousot. Automatic synthesis of optimal invariant assertions: mathematical foundations. In *ACM Symposium on Artificial Intelligence & Programming Languages*, volume 12, pages 1–12. ACM SIGPLAN Notices, 1977.
18. 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.
19. B.J. Ellis and A. Ireland. Automation for exception freedom proofs. In *Proceedings of the* 18^{th} *IEEE International Conference on Automated Software Engineering*, pages 343–346. IEEE Computer Society, 2003. Also available from the School of Mathematical and Computer Sciences, Heriot-Watt University, as Technical Report HW-MACS-TR-0010.
20. B.J. Ellis and A. Ireland. An integration of program analysis and automated theorem proving. In E.A. Boiten, J. Derrick, and G. Smith, editors, *Proceedings of 4th International Conference on Integrated Formal Methods (IFM-04)*, volume 2999 of *Lecture Notes in Computer Science*, pages 67–86. Springer Verlag, 2004. Also available from the School of Mathematical and Computer Sciences, Heriot-Watt University, as Technical Report HW-MACS-TR-0014.
21. D. Elspas, M.W. Green, K.N. Levitt, and R.J. Waldinger. Research in interactive program-proving techniques. In *SRI, Menlo Park, CA*. 1972.
22. ESA. Ariane 5 - flight 501 failure. Board of inquiry report, European Space Agency, 1996.
23. Jean-Christophe Filliâtre, Sam Owre, Harald Rueß, and N. Shankar. ICS: integrated canonizer and solver. In *Proceedings of CAV 2001*, 2001.
24. C. Flanagan and K. Rustan M. Leino. Houdini, an annotation assistant for ESC/Java. In *Proceedings of FME 2001*. LNCS 2021, Springer-Verlag, 2001.
25. 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.

26. R. W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science, Proceedings of Symposia in Applied Mathematics 19*, pages 19–32. American Mathematical Society, 1967.
27. S.M. German. Automating proof of the absence of common runtime errors. In *Proceedings of 5th ACM Conference on Principles of Programming Languages*. 1978.
28. C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–583, 1969.
29. A. Ireland. The Use of Planning Critics in Mechanizing Inductive Proofs. In A. Voronkov, editor, *International Conference on Logic Programming and Automated Reasoning (LPAR'92), St. Petersburg*, Lecture Notes in Artificial Intelligence No. 624, pages 178–189. Springer-Verlag, 1992. Also available from Edinburgh as DAI Research Paper 592.
30. A. Ireland and A. Bundy. Productive use of failure in inductive proof. *Journal of Automated Reasoning*, 16(1–2):79–111, 1996. Also available as DAI Research Paper No 716, Dept. of Artificial Intelligence, Edinburgh.
31. 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):225–245, March 1999. A longer version is available from Dept. of Computing and Electrical Engineering, Heriot-Watt University, Research Memo RM/97/11.
32. A. Ireland, B.J. Ellis, and T. Ingulfsen. Invariant patterns for program reasoning. In R. Monroy, G. Arroyo-Figueroa, L.E. Sucar, and H. Sossa, editors, *Proceedings of 3rd Mexican International Conference on Artificial Intelligence (MICAI-04)*, volume 2972 of *Lecture Notes in Artificial Intelligence*, pages 190–201. Springer Verlag, 2004. Also available from the School of Mathematical and Computer Sciences, Heriot-Watt University, as Technical Report HW-MACS-TR-0011.
33. A. Ireland and J. Stark. On the Automatic Discovery of Loop Invariants. In *Proceedings of the Fourth NASA Langley Formal Methods Workshop – NASA Conference Publication 3356*, 1997. Also available from Dept. of Computing and Electrical Engineering, Heriot-Watt University, Research Memo RM/97/1.
34. A. Ireland and J. Stark. Proof planning for strategy development. *Annals of Mathematics and Artificial Intelligence*, 29(1-4):65–97, February 2001. An earlier version is available as Research Memo RM/00/3, Dept. of Computing and Electrical Engineering, Heriot-Watt University.
35. A. Ireland and J. Stark. Combining proof plans with partial order planning for imperative program synthesis. *Journal of Automated Software Engineering*, 13(1):65–105, 2005. An earlier version is available from the School of Mathematical and Computer Sciences, Heriot-Watt University, Technical Report HW-MACS-TR-0026.
36. ISO. Reference manual for the ada programming language. ISO/IEC 8652, International Standards Organization, 1995.
37. P. Janičić and A. Bundy. A general setting for flexibly combining and augmenting decision procedures. *Journal of Automated Reasoning*, 28(3):257–305, April 2002.
38. S.M. Katz and Z. Manna. Logical analysis of programs. *Communications of the ACM*, 19(4):188–206, 1976.
39. S. King, J. Hammond, R. Chapman, and A. Pryor. Is proof more cost effective than testing? *IEEE Trans. on SE*, 26(8), 2000.
40. I. Kraan, D. Basin, and A. Bundy. Middle-out reasoning for logic program synthesis. In D. S. Warren, editor, *Proceedings of the Tenth International Conference on Logic Programming*. MIT Press, 1993. Also available as Max-Planck-Institut für Informatik Report MPI-I-93-214 and Edinburgh DAI Research Report 638.

41. MoD. Hazard analysis and safety classification of the computer and programmable electronic system elements of defence equipment. Interim Defence Standard 00-56, Issue 1, Ministry of Defence, Directorate of Standardization, Kentigern House, 65 Brown Street, Glasgow G2 8EX, UK, April 1991.
42. MoD. The procurement of safety critical software in defence equipment (part 1: Requirements, part 2: Guidance). Interim Defence Standard 00-55, Issue 1, Ministry of Defence, Directorate of Standardization, Kentigern House, 65 Brown Street, Glasgow G2 8EX, UK, April 1991.
43. R. Monroy, A. Bundy, and A. Ireland. Proof Plans for the Correction of False Conjectures. In F. Pfenning, editor, *5th International Conference on Logic Programming and Automated Reasoning, LPAR'94*, Lecture Notes in Artificial Intelligence, v. 822, pages 54–68, Kiev, Ukraine, 1994. Springer-Verlag. Also available from Edinburgh as DAI Research Paper 681.
44. National Cyber Security Partnership (NCSP). Improving security across the software development lifecycle. 2004. http://www.cyberpartnership.org.
45. F. Nielson, H.R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 1999.
46. PolySpace-Technologies. http://www.polyspace.com/.
47. PURRS: The parma university's recurrence relation solver. http://www.cs.unipr.it/purrs/.
48. Sunae Seo, Hongseok Yang, and Kwangkeun Yi. Automatic construction of hoare proofs from abstract interpretation results. In Atsushi Ohori, editor, *Programming Languages and Systems, First Asian Symposium, APLAS 2003, Beijing, China, November 27-29, 2003, Proceedings*, volume 2895 of *Lecture Notes in Computer Science*, pages 230–245. Springer, 2003.
49. J. Stark and A. Ireland. Invariant discovery via failed proof attempts. In P. Flener, editor, *Logic-Based Program Synthesis and Transformation*, LNCS 1559, pages 271–288. Springer-Verlag, 1998. An earlier version is available from the Dept. of Computing and Electrical Engineering, Heriot-Watt University, Research Memo RM/98/2.
50. 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.
51. L. Whiting and M. Hill. Safety analysis of hawk in flight monitor. In *Workshop on Program Analysis For Software Tools and Engineering*, 1999.

A Introduction to NuSPADE Results

Formal proof using NuSPADE typically involves a number of iterations. On each iteration an application of the Examiner and Simplifier proceeds an application of NuSPADE. That is, NuSPADE is only applied to the VCs that are not proved by the Simplifier. Note that where NuSPADE modifies a program specification, *e.g.* loop invariant strengthening, another iteration of the analysis and proof process is required.

For each example below we describe the behaviour of NuSPADE on successive iterations. For each goal considered by NuSPADE we record which: i) proof plan applied, ii) critics applied, and iii) program analysis methods applied. For space reasons, the following key is used to describe the table headings:

Plan	**a1**	**exception freedom**	(see §4.1)
	a2	**loop invariant**	(see §4.2)
Critic	**b1**	**elementary**	(see §5.1)
	b2	**transitivity**	(see §5.2)
	b3	**fertilise**	(see §5.3)
PropGen	**c1**	**for-loop range**	(see §6.3)
	c2	**looping code**	(see §6.5)
	c3	**for-loop entry**	(see §6.6)

Note that in these tables the symbol • denotes success while ◦ denotes partial success, for example when a plan fails but a critic successfully fires.

B Industrial Examples

B.1 Example 1

Iteration	**Goal**	**Plan**		**Critic**			**PropGen**		
		a1	**a2**	**b1**	**b2**	**b3**	**c1**	**c2**	**c3**
I1	I1_1	◦			•			•	
	I1_2	◦			•			•	
	I1_3	◦			•			•	
	I1_4	◦			•			•	
I2	I2_1		•						
	I2_2		•						
	I2_3	•							
	I2_4	•							
	I2_5	•							
	I3_6	•							

Due to resource constraints the Examiner does not generate exhaustive rules for large constant data structures. This can leave information gaps in the VCs, preventing proof. In I1_1, I1_2, I1_3 and I1_4, the critic requests additional information that corresponds to such a data structure. It would be unsound for our

tools to add this information directly, however it would be possible for our tools to communicate with the Examiner to request a targeted subset of the constant information. Currently this information must be introduced by the user creating explicit rules, ensuring that these rules faithfully reflect the code. Note that another example exhibited exactly the same pattern as this example.

B.2 Example 2

Iteration	Goal	Plan		Critic			PropGen		
		a1	a2	b1	b2	b3	c1	c2	c3
I1	I1_1		○	●			●		
	I1_2		○	●			●		
	I1_3	○		●				●	
	I1_4	○		●				●	
	I1_5	○		●			●		
	I1_6	○		●			●		
	I1_7	○		●				●	
	I1_8	○		●				●	
I2	I2_1		○		●		●		
	I2_2		●						
	I2_3		○			●			●
	I2_4		○		●		●		
	I2_5		●						
	I2_6		●						
	I2_7		○			●			●
	I2_8	●							
	I2_9	●							
	I2_10	●							
I3	I3_1		●						
	I3_2		●						
	I3_3		●						
	I3_4	●							
	I3_5	●							
	I3_6	●							

B.3 Example 3

Iteration	Goal	Plan		Critic			PropGen		
		a1	a2	b1	b2	b3	c1	c2	c3
I1	I1_1		○			●			●
	I1_2		○			●			●
	I1_3	○		●			●		
	I1_4	○		●				●	
	I1_5	○		●				●	
I2	I2_1		●						
	I2_2		●						
	I2_3	●							
	I2_4	●							

B.4 Example 4

Iteration	Goal	Plan		Critic			PropGen		
		a1	a2	b1	b2	b3	c1	c2	c3
I1	I1_1		○			●			●
	I1_2		○	●			●		
I2	I2_1		○		●			●	

B.5 Example 5

Iteration	Goal	Plan		Critic			PropGen		
		a1	a2	b1	b2	b3	c1	c2	c3
I1	I1_1		○			●			●
	I1_2		○	●			●		

B.6 Example 6

Iteration	Goal	Plan		Critic			PropGen		
		a1	a2	b1	b2	b3	c1	c2	c3
I1	I1_1		○	●				●	

B.7 Example 7

Iteration	Goal	Plan		Critic			PropGen		
		a1	a2	b1	b2	b3	c1	c2	c3
I1	I1_1	○		●				●	
	I1_2	○		●				●	

B.8 Example 8

Iteration	Goal	Plan		Critic			PropGen		
		a1	a2	b1	b2	b3	c1	c2	c3
I1	I1_1		○	●			●		
	I1_2		○	●			●		
	I1_3		○	●			●		
	I1_4		○	●			●		
	I1_5		○	●			●		

B.9 Example 9

Iteration	Goal	Plan		Critic			PropGen		
		a1	a2	b1	b2	b3	c1	c2	c3
I1	I1_1		○	●			●		

B.10 Example 10

Iteration	Goal	Plan		Critic			PropGen		
		a1	a2	b1	b2	b3	c1	c2	c3
I1	I1_1		○			●			●
I2	I2_1		○	●			●		

Note that this subprogram calls a function with preconditions, resulting in VCs to prove that the preconditions hold. As NuSPADE focuses on exception freedom it is unable to make progress on these partial correctness problems.

Note that three other examples exhibited exactly the same pattern as this example.

B.11 Example 11

Iteration	Goal	Plan		Critic			PropGen		
		a1	a2	b1	b2	b3	c1	c2	c3
I1	I1_1		○			●			●
	I1_2		○			●			●
I2	I2_1		○	●			●		
	I2_2		○	●			●		

B.12 Example 12

Iteration	Goal	Plan		Critic			PropGen		
		a1	a2	b1	b2	b3	c1	c2	c3
I1	I1_1	○		•				•	
I2	I2_1		•						
	I2_2		•						
	I2_3		•						
	I2_4		•						
	I2_5		•						
	I2_6		•						
	I2_7		•						
	I2_8		•						
	I2_9	•							

By default the SPARK tools come complete with a large collection of rules, suitable to complete the proof of most exception freedom VCs. However, the proof of this example required some very specific rewrite steps, not available in SPARK. To overcome this problem the proof plan exploits both generalisation and induction methods. Although these methods are infrequently required, they extend the reach of the proof planner beyond the default rules allowing for the successful proof of more complicated examples.

C Non-industrial Examples

C.1 Example Filter

Iteration	Goal	Plan		Critic			PropGen		
		a1	a2	b1	b2	b3	c1	c2	c3
I1	I1_1	○		•				•	
I2	I2_1		•						
	I2_2	•							

C.2 Example: Filter2

Iteration	Goal	Plan		Critic			PropGen		
		a1	a2	b1	b2	b3	c1	c2	c3
I1	I1_1	○		•				•	
	I1_2	○		•				•	
	I1_3	•							
I2	I2_1		•						
	I2_2		•						
	I2_3	•							
	I2_4	•							
	I2_5	•							

C.3 Example: Average

Iteration	Goal	Plan		Critic			PropGen		
		a1	a2	b1	b2	b3	c1	c2	c3
I1	I1_1		○	●				●	
	I1_2		○	●				●	
	I1_3	○		●				●	
	I1_4	○		●				●	
I2	I2_1		●						
	I2_2		●						
	I2_3	●							
	I2_4	●							
	I2_5	●							
	I2_6	●							

C.4 Example: Power

Iteration	Goal	Plan		Critic			PropGen		
		a1	a2	b1	b2	b3	c1	c2	c3
I1	I1_1	○		●				●	
	I1_2	○		●				●	
	I1_3	○		●				●	
	I1_4	○		●				●	
I2	I2_1		●						
	I2_2	●							

C.5 Example: Find

Iteration	Goal	Plan		Critic			PropGen		
		a1	a2	b1	b2	b3	c1	c2	c3
I1	I1_1	○		●				●	

C.6 Example: Matrix Multiplication

Iteration	Goal	Plan		Critic			PropGen		
		a1	a2	b1	b2	b3	c1	c2	c3
I1	I1_1	○		●				●	
	I1_2	○		●				●	
	I1_3	○		●				●	
	I1_4	○		●				●	
I2	I2_1		●						
	I2_2		●						
	I2_3	●							
	I2_4	●							
	I2_5	●							
	I2_6	●							

C.7 Example: Matrix Filter

Iteration	Goal	Plan		Critic			Method		
		a1	a2	b1	b2	b3	c1	c2	c3
I1	I1_1	○		●				●	
I2	I2_1		●						
	I2_2		●						
	I2_3		●						
	I2_4		●						
	I2_5	●							

D Non-industrial Examples Code

D.1 Example Code: Filter

The code for this example has already been presented in Figure 1.

D.2 Example Code: Filter2

```
package Filter2Package is
  subtype P_Type is Integer;
  subtype I_Type is Integer range 0..9;
  subtype Dc_Type is Integer range -10000..10000;
  type D_Type is array (I_Type) of Dc_Type;
  procedure Filter2(D1, D2: in D_Type; P: out P_Type);
  --# derives P from D1, D2;
end Filter2Package;
```

```
package body Filter2Package is
  procedure Filter2(D1, D2: in D_Type; P: out P_Type)
  is
  begin
    P:= 0;
    for I in I_Type loop
      --# assert true;
      if D1(I) >= 0 and D1(I) <= 20 and
         D2(I) >= 0 and D2(I) <= 20 then
         P:= P + (D1(I) * D2(I));
      end if;
    end loop;
  end Filter2;
end Filter2Package;
```

D.3 Example Code: Average

```
package AveragePackage
is
  subtype S_Type is Integer;
  subtype I_Type is Integer range 1..10;
  subtype ADc_Type is Integer range -100..100;
  type D_Type is array(I_Type) of ADc_Type;
  procedure Average(D: in D_Type; S: out S_Type;
                    A: out ADc_Type);
  --# derives S, A from D;
end AveragePackage;
```

```
package body AveragePackage
is
  procedure Average(D: in D_Type; S: out S_Type;
                    A: out ADc_Type)
  is
  begin
    S:= 0;
    for I in I_Type loop
      --# assert true;
      S:= S + D(I);
    end loop;
    A:= S/((I_type'Last - I_Type'First) + 1);
  end Average;
end AveragePackage;
```

D.4 Example Code: Power

```
package PowerPackage is
  subtype KZ_Type is Integer;
  subtype YX_Type is Integer range 0..5;
  procedure Power(X, Y: in YX_Type; Z: out KZ_Type);
  --# derives Z from X, Y;
end PowerPackage;
```

```
package body PowerPackage
is
  procedure Power(X, Y: in YX_Type; Z: out KZ_Type)
  is
    K: Integer;
  begin
    K:= 0;
    Z:= 1;
    while K /= Y
    loop
      --# assert true;
      Z:= Z*X;
      K:= K+1;
    end loop;
  end Pow1;
end PowerPackage;
```

D.5 Example Code: Find

```
package FindPackage is
  subtype KI_Type is Integer range 0..9;
  subtype Ac_Type is Integer range 0..100;
  type A_Type is array (KI_Type) of Ac_Type;
  function Find(A: A_Type) return KI_Type;
end FindPackage;
```

```
package body FindPackage is
  function Find(A: A_Type) return KI_Type
  is
    K: KI_Type;
  begin
    K := 0;
    for I in KI_Type loop
      if A(I) < A(K) then
        K := I;
      end if;
    end loop;
    return K;
  end Find;
end FindPackage;
```

D.6 Example Code: Matrix Multiplication

```
package MatMultiplicationPackage is
   subtype I_Type is Integer range 0 .. 3;
   subtype E_Type is Integer range -9 .. 9;
   subtype R_Type is Integer;
   type OneA_Type is array (I_Type) of E_Type;
   type TwoA_Type is array (I_Type) of OneA_Type;
   type ROneA_Type is array (I_Type) of R_Type;
   type RTwoA_Type is array (I_Type) of ROneA_Type;
   procedure MatMultiplication(A: in TwoA_Type;
                               B: in TwoA_Type;
                               C: out RTwoA_Type);
   --# derives C from A, B;
end MatMultiplicationPackage;
```

```
package body MatMultiplicationPackage is
  procedure MatMultiplication(A: in TwoA_Type;
                              B: in TwoA_Type;
                              C: out RTwoA_Type)
  is
    M: Integer;
  begin
    for I in I_Type loop
      --#assert true;
      for J in I_Type loop
        M:=0;
        for K in I_Type loop
          --#assert true;
          M:=M+A(I)(K)*B(K)(J);
        end loop;
        C(I)(J):=M;
      end loop;
    end loop;
  end MatMultiplication;
end MatMultiplicationPackage;
```

D.7 Example Code: Matrix Filter

```
package body MatFilterPackage is
  procedure MatFilter(A: in TwoA_Type; R: out Integer)
  is
  begin
    R:=0;
    for I in I_Type loop
      for J in I_Type loop
        if A(I)(J) >= 0 and A(I)(J)<=10 then
          R:=R+A(I)(J);
        end if;
        --#assert true;
      end loop;
      --#assert true;
    end loop;
  end MatFilter;
end MatFilterPackage;
```

```
package body MatFilterPackage is
  procedure MatFilter(A: in TwoA_Type; R: out Integer)
  is
  begin
    R:=0;
    for I in I_Type loop
      for J in I_Type loop
        if A(I)(J) >= 0 and A(I)(J)<=10 then
          R:=R+A(I)(J);
        end if;
        --#assert true;
      end loop;
      --#assert true;
    end loop;
  end MatFilter;
end MatFilterPackage;
```

E Non-industrial Example Summary

Name	Abstract predicate	Candidate invariant
Filter	$r \geq \mathcal{A} \wedge r \leq \mathcal{B}$	`R>=0 and` `R<=I*100`
Filter2	$p \geq \mathcal{A} \wedge p \leq \mathcal{B}$	`P>=0 and` `P<=I*400`
Average	$s \geq \mathcal{A} \wedge s \leq \mathcal{B}$	`S>=(I-1)*(-100) and` `S<=(I-1)*100`

Power	$k \geq \mathcal{A} \wedge k \leq \mathcal{B}$
	$z \geq \mathcal{C} \wedge z \leq \mathcal{D}$
		`Z>=1 and`
		`Z<=(5**K)*1 and`
		`K>=0 and`
		`K<=Y`

Find	$k \geq \mathcal{A} \wedge k \leq \mathcal{B}$	`K>=0 and` `K<=I`

Matrix Multiplication	$m \geq \mathcal{A} \wedge m \leq \mathcal{B}$	`(most inner loop)`
		`M>=K*(-81) and`
		`M<=K*(81)`

Matrix Filter	$r \geq \mathcal{A} \wedge r \leq \mathcal{B}$	`(outer loop)`
		`R>=0 and`
		`R<=I*40`
		`(inner loop)`
		`R>=0 and`
		`R<=(I*40)+((J+1)*10)`