An Integrated Approach to High Integrity Software Verification

Andrew Ireland, Bill J. Ellis and Andrew Cook
School of Mathematical and Computer Sciences
Heriot-Watt University
Riccarton, Edinburgh, Scotland, EH14 4AS
(a.ireland@hw.ac.uk)

Roderick Chapman and Janet Barnes
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.

Keywords: Program proof, proof planning, static analysis, SPARK

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 [45].

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] to control the search for proofs and a program analysis oracle 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. Our approach is implemented in a system called NuSPADE.

Background material is presented in §2 while in §3 we relate NuSPADE to the SPARK Approach. The techniques that are used within NuSPADE are described 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. SPARK is defined as a subset of Ada [37]. 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 of a SPARK subprogram, called Filter, is shown in Figure 1.

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 in each loop, conveying limited type information. These default invariants may be strengthened by providing explicit invariant annotations. The Examiner includes a verification condition generator, 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.

The process of discharging VCs is supported by two proof tools. Firstly, the SPADE Simplifier (henceforth Simplifier), is 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. 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 strengthening the hypotheses that are available during a proof attempt.

- **Identify Inconsistency** - Show that there is an inconsistency between the program and the specification.
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.

Figure 1. SPARK code: Filter subprogram

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 high integrity system will often give rise to 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

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 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. The Examiner generates exception freedom VCs (EFVCs) that ensure freedom from the run-time exceptions highlighted above.

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 $C_3$ and $C_4$. 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. $\text{Integer}$. This corresponds to proving conclusions $C_1$ and $C_2$. While $C_1$, $C_3$ and $C_4$ are proved by the Simplifier, $C_2$ is 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
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 \( \text{element}(a, [\text{loop}_1]) \) denotes accessing array \( a \) at index \( \text{loop}_1 \). 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.

Figure 3. An exception freedom verification condition (EFVC)

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 generalisation and lemma discovery [30, 31], loop invariant discovery [35, 50], and refining faulty conjectures [44]. 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 it has been
used in guiding proof search within the context of program synthesis [36, 41].

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 [46]. 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 [39]. As noted by Cousot [17], the use of recurrence relations in this manner fits within the general methodology of abstract interpretation. The limitations 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 practise.

3. NuSPADE and the SPARK Approach

NuSPADE\textsuperscript{1} supports proof automation and fits within the SPARK Approach, 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 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. The proof script is then used to control proof construction within the Proof Checker. If proof planning fails, proof-failure analysis may identify missing proof context. The form of this proof context

\textsuperscript{1} The name ‘NuSPADE’ emphasises that we are extending the capabilities of the SPADE proof tools.
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. For each VC the Simplifier fails to discharge, NuSPADE will attempt to:

- **Perform Proof** - Where proof planning succeeds, a proof script is generated and used to control 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.

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. If NuSPADE fails, the programmer will still need to intervene.

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.

4.1.1. 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.

4.1.2. 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 symbolic constants with concrete values. In particular, simplify aims to transform goals so that the elementary method becomes applicable.

4.1.3. 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.

4.1.4. 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, C2 in Figure 3

\[ r + \text{element}(a, [i]) \leq \text{integer}_\text{last} \]  

(1)
Preconditions for fertilise method:

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

Figure 5. Preconditions for the fertilise 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.

Figure 6. Preconditions for the transitivity method

An application of the transitivity method to (1) gives

\[(r + \text{element}(a, [i]) \leq X_1) \land (X_1 \leq \text{integer}_{\text{last}})\] (2)

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

4.1.5. Decomposition Method

The decomposition method (preconditions given in Figure 7) is applicable to a subterm of a conclusion 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

\[(r \leq X_2) \land (\text{element}(a, [i]) \leq X_3) \land (X_2 + X_3 \leq \text{integer}_{\text{last}})\]

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.

Figure 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.

4.2.1. 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 [34, 35, 50]. 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

$$f(i + 1) \ . \ (3)$$

The annotated portion of the term, represented by shading, is known 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 the termination of rippling.

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\textsuperscript{2}
\[ f(X + 1) \Rightarrow f(X) \land g(X) \]
may be annotated as
\[ f(X + 1) \Rightarrow f(X) \land g(X) \]
Wave-rules target syntactic differences by only manipulating annotated terms in the conclusion. The \textit{wave} method controls the application of wave-rules. For example, applying wave-rule (4) to (3) gives
\[ f(i) \land g(i) \]

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 \textit{fertilise} method applies, leaving the simplified conclusion \( g(i) \).

4.2.2. \textit{Induction Method}
Like rippling, the \textit{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, \textit{i.e.} situations where none of the SPADE axioms or rules are applicable.

4.2.3. \textit{Generalise Method}
The \textit{generalise} method is strongly linked with the \textit{induction} method, since a generalisation step may be required in order to obtain a stronger induction hypothesis. Our \textit{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.

\textsuperscript{2} We use \( \Rightarrow \) to denote rewrite rules and \( \rightarrow \) to denote logical implication.
5.1. Elementary Critic

The elementary critic (see 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 C2 in Figure 3

\[ r + \text{element}(a, [i]) \leq \text{integer}_{\text{last}}. \] (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

\[ (\text{element}(a, [i]) \geq 0) \land (\text{element}(a, [i]) \leq 100) \]

and that

\[ (r \geq \text{integer}_{\text{first}}) \land (r \leq \text{integer}_{\text{last}}). \]

It then follows that \( r + \text{element}(a, [i]) \) can raise an overflow exception if \( r \) is in the range \( \text{integer}_{\text{last}} - 99 \ldots \text{integer}_{\text{last}}, \) i.e. 32668 \ldots 32767. This reasoning is achieved automatically by the constraint solver. The counter-example identifies that \( r \) or \( \text{element}(a, [i]) \) 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

\[ (r \geq A) \land (r \leq B). \]

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.
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.

Figure 8. Preconditions and patch for the elementary critic

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.

Figure 9. Preconditions and patch for the transitivity critic

5.2. Transitivity Critic

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

5.3. Fertilise Critic

The fertilise critic (see Figure 10) extends the applicability of the fertilise method by recognising a near match and transforming the goal
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 $H'$.

---

*Figure 10.* Preconditions and patch for the fertilise critic

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 at the proof planning level, and involves forward reasoning from the given hypotheses. The second patch involves strengthening the hypotheses via program analysis.

5.4. Decomposition Critic

The decomposition critic (see 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 dependent upon the SPARK toolset. The program analysis oracle has strong similarities with abstract interpretation. The source code is translated into a flowgraph. Each variable at each program point is associated with an abstract state. This state aims to describe the possible values that its corresponding variable
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.

*Figure 11.* Preconditions and patch for the decomposition critic

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

package body PolishFlagPackage is
  procedure PolishFlag(A: in out A_T)
  is
    subtype ARPO_T is Integer range A'First..A'Last+1;
    I,J: ARPO_T;
    T: AC_T;
  begin
    I:=ARPO_T'First; J:=ARPO_T'Last;
    loop
      --# 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;

Figure 12. SPARK code: PolishFlag subprogram

its type once it has been assigned a value. For each variable, this method retrieves its type and where it is assigned a value within the code.

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 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;

Figure 13. SPARK code: Search 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);

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 program variable \( X \) will either have been assigned some unknown initial value (\( X^- \)) or will be undefined (\( \text{undefined} \)). The abstract state for \( X \), at program point \( p \), is denoted by \( X^p \). To minimise complexity the abstract states are restricted to a particular class of formula, \textit{i.e.} formula that only contain simple variables, constants, and regular arithmetic relations and functions. Our mechanism for propagating abstract states through non-looping code uses abstract state combinators. To remain within our restricted representation for abstract states, it is often necessary to apply approximations. For instance, as only simple 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 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 \( V^0 = V^- \), as \( V \) is an import (\texttt{in}) variable, and \( R^0 = \text{undefined} \) as \( R \) is an export (\texttt{out}) variable. The first path involves entering the outermost \texttt{then} branch yielding the property \( V < I_T'\text{First} \) and an assignment giving \( R^1 = I_T'\text{First} \). The second path involves failing the outermost \texttt{then} branch and entering the innermost \texttt{then} branch. This yields the properties \( \neg(V < I_T'\text{First}) \) and \( V > I_T'\text{Last} \) and an assignment giving \( R^2 = I_T'\text{Last} \). The final path enters the innermost \texttt{else} branch. This yields the properties \( \neg(V < I_T'\text{First}) \) and \( \neg(V > I_T'\text{Last}) \) and an assignment giving \( R^3 = V^- \). As the value \( V^- \) is unknown it offers a weak constraint. However, by exploiting the type information gathered by earlier methods, it can be found that \( R^3 \geq I_T'\text{First} \land R^3 \leq I_T'\text{Last} \). At this stage the three separate paths merge to give

\[
\begin{align*}
R^4 &= I_T'\text{First} \\
R^4 &= I_T'\text{Last} \\
(R^4 \geq I_T'\text{First} \land R^4 \leq I_T'\text{Last})
\end{align*}
\]
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;

Figure 14. SPARK code: Clip subprogram

which can be simplified as

\[(R \geq I_T'\text{First}) \land (R \leq I_T'\text{Last})\]

and expressed as the following candidate assertion.

--# assert (R\geq I_T'\text{First}) and (R\leq I_T'\text{Last});

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 submethods. The first submethod generates and solves recurrence relations for each basic
loop. Nested loops are dealt with by the second submethod. The third submethod simplifies the solutions to the recurrence relations, while the fourth submethod completes the invariant generation process by combining the simplified solutions.

6.5.1. Submethod: Generating and Solving Recurrence Relations
This method identifies and solves recurrence relations. To represent an arbitrary iteration \( n \), every variable’s abstract state \( V^p \) is treated as \( V^p(n) \). To begin, every variable is initialised to its value on the previous iteration as \( V^p(n) = V^p(n-1) \). With this initialisation our non-looping method for one iteration of the loop generates abstract states that can be extracted as recurrence relations.

To illustrate, consider again the Filter subprogram 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 \( I(n) = I(n-1) + 1 \). This can then be solved as \( I(n) = I(0) + n \), i.e. the value of \( I \) on iteration \( n \) is equal to the initial value of \( I \) (\( I(0) \)) 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 \( R(n) = R(n-1) \), which is solved as \( R(n) = R(0) \). The second path involves entering the if statement. The condition \( A(I) >= 0 \) and \( A(I) <= 100 \) reveals the property \( \text{element}(A, I) \geq 0 \land \text{element}(A, I) \leq 100 \), and the assignment statement \( R := R + A(I) \) leads to the abstract state \( R^1(n) = R(n-1) + \text{element}(A, I) \). 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

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

Note that \([c_1, \ldots, c_n]\) 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

\[
\left[ R_{(n)} = R_{(0)}, R_{(n)} = R_{(0)} + 100 * n \right].
\]

Note that these subsume the case where the then branch is not entered.
6.5.2. **Submethod: Nested Loops**
The submethod described above is applicable to single loops. To support the analysis of nested loops, the modifications made to a variable within a loop are 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 for an outer loop depends strongly on the quality of the abstracted assignment. This technique is most effective where the execution of nested loops exhibit a uniform pattern, *e.g.* using nested `for` loops to iterate over a two dimensional array.

6.5.3. **Submethod: Simplifying Solutions**
All of the solved recurrence relations above are general, referencing undefined initial values of the form $V_{(0)}$. Once the outermost loops have been solved the actual initial values may be inserted. These instantiations introduce specific values, often supporting additional simplifications.

Returning again to the Filter subprogram (see Figure 1), at entry to the loop the abstract state found for both $I$ and $R$ will be 0. Thus, both $I_{(0)}$ and $R_{(0)}$ may be replaced with 0 giving

$$I_{(n)} = n$$
$$\left[ R_{(n)} = 0, R_{(n)} = 100 \times n \right].$$

These may be simplified further by translating the collection into inequalities

$$I_{(n)} = n$$
$$(R_{(n)} \geq 0) \land (R_{(n)} \leq n \times 100).$$

6.5.4. **Submethod: Combining Solutions**
Solved recurrence relations cannot be directly expressed as candidate invariants. The problem occurs due to the presence of the artificial loop iteration variable $n$. This variable can be eliminated by deriving an expression for $n$ in terms of the actual program variables. In practise, this can often be achieved by simple equality reasoning. For example, in the case of the Filter subprogram (see Figure 1), we know that $I_{(n)} = n$, therefore 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 represent this distinction, 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 $v(n) = v(n-1)$. 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.

$$\text{--# assert } u = u\%;$$

7. Filter Subprogram Revisited

To describe the overall behaviour of our program analysis we return to the Filter subprogram (see Figure 1). For this example, all but one of the VCs generated by the Examiner are proved by the Simplifier. The remaining VC triggers the first iteration of NuSPADE. While the proof planning for the remaining VC fails, the elementary critic (see §5.1) generates an abstract predicate of the form

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

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

$$\text{--# assert } (R \geq 0) \text{ and } (R \leq (I \times 100));$$

The addition of this invariant revises the subprogram specification, requiring the Examiner to regenerate the VCs. Two goals arising from the new VCs are not proved by the Simplifier, triggering a second
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: r + element(a, [loop_1_i]) >= integer_first.
C2: r + element(a, [loop_1_i]) <= integer_last.
C3: loop_1_i >= ar_t_first.
C4: loop_1_i <= ar_t_last.

Figure 15. Revised exception freedom verification condition (EFVC)

iteration of NuSPADE. Below we describe the details of the remaining goals and the associated proof planning.

7.1. Exception Freedom Proofs

The Examiner generates the EFVC shown in Figure 15. The Simulator is unable to prove the goal corresponding to conclusion C2, i.e.

\[ r + element(a, [i]) \leq integer_last. \]  

Note that the proof context now includes the hypothesis

\[ r \leq i \times 100 \]  

as well as the hypothesis

\[ element(a, [i]) \leq 100. \]  

The proof planning of conclusion (7) begins with an application of the transitivity method, giving rise to a conjunction of the form

\[ (r + element(a, [i]) \leq X_1) \land (X_1 \leq integer_last). \]  

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

\[(W \leq Y) \land (X \leq Z) \rightarrow (W + X) \leq (Y + Z)\].

(11)

Given that we are performing a backward style of proof, the application of (11) to the left-hand conjunct of (10) gives rise to

\[(r \leq X_2) \land (\text{element}(a, [i]) \leq X_3) \land (X_2 + X_3 \leq \text{integer}_\text{last})\].

(12)

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 (8) and (9), fertilisation simplifies (12), instantiating \(X_2\) to be \(i \times 100\) and \(X_3\) to be 100 in the process. This leaves a proof residue of the form

\[(i \times 100) + 100 \leq \text{integer}_\text{last}\].

(13)

Assuming that \(\text{integer}_\text{last}\) is the constant 32767, the remaining goal (13), is trivial and is discharged by the simplify and elementary methods.

7.2. LOOP INVARIANT PROOFS

The Examiner generates the loop invariant VC shown in Figure 16. The goal corresponding to conclusion C2 is not proved by the Simplifier. Here we describe the proof planning of this goal, i.e. given the hypotheses

\[r \leq i \times 100\]  

(14)

\[\text{element}(a, [i]) \leq 100\]  

(15)

we focus on proving the conclusion

\[r + \text{element}(a, [i]) \leq (i + 1) \times 100\].

(16)

The annotate method identifies the difference between conclusion (16) and hypothesis (14), giving

\[r + \text{element}(a, [i]) \leq (i + 1) \times 100\].

(17)

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

\[(X + Y) \times Z \Rightarrow (X \times Z) + (Y \times Z)\]

(18)

\[(W + X) \Rightarrow (Y + Z) \Rightarrow W \leq Y \land X \leq Z\]

(19)

1 \times X \Rightarrow X\]  

(20)
The wave method applies (18) to the right-hand side of (17) to give
\[ r + \text{element}(a, [i]) \leq (i \times 100) + (1 \times 100). \] (21)

Using wave-rule (19), the wave method further ripples (21) to give
\[ (r \leq i \times 100) \land (\text{element}(a, [i]) \leq 1 \times 100). \] (22)

Rippling is complete and the fertilise method applies, matching with hypothesis (14), leaving a proof residue of the form
\[ \text{element}(a, [i]) \leq 1 \times 100. \] (23)

The simplify method, using (20), reduces (23) to give
\[ \text{element}(a, [i]) \leq 100. \] (24)
Matching against hypothesis (15) the fertilise method reduces (24) 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. 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 $C_2$ in Figure 15 is shown in Figure 17 and the tactic for conclusion $C_2$ 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

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

The proof script segment generated for this single rewrite step is shown in Figure 19. The complexity of the proof script segment 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.
plan(vc6_2,c2,
  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(h7) then
                simplify(logical_and(5),[1],equ) then
                  simplify(logical_and(2),[],equ) then
                    simplify(filter_rules(4),[2]) then
                      elementary)

Figure 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)

Figure 18. Loop invariant verification condition tactic

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 flowgraph. Praxis provided the SPARK grammar and a tokeniser
... 

dosubgoalproperty \( r \leq \text{loop}_1 \cdot i \cdot 100 \) and 
\( \text{element}(a, [\text{loop}_1 \cdot i]) \leq 1 \cdot 100 \).
...

done c#1.
alldone.

dosubgoalproperty 
\( r + \text{element}(a, [\text{loop}_1 \cdot i]) \leq \text{loop}_1 \cdot i \cdot 100 + 1 \cdot 100 \).

dosubgoalproperty \( r \leq \text{loop}_1 \cdot i \cdot 100 \) and 
\( \text{element}(a, [\text{loop}_1 \cdot i]) \leq 1 \cdot 100 \rightarrow 
\( r + \text{element}(a, [\text{loop}_1 \cdot i]) \leq 
\text{loop}_1 \cdot i \cdot 100 + 1 \cdot 100 \).

prove c#1 by implication.
dosimplifyhypsconj.
infer \( r + \text{element}(a, [\text{loop}_1 \cdot i]) \leq \text{loop}_1 \cdot i \cdot 100 + 1 \cdot 100 
\text{using } \text{inequals}(80) \).

done c#1.
alldone.
done c#1.
alldone.

forwardchain h#21.
done c#1.
alldone.
...

Figure 19. Proof script extract

that was extracted from the Examiner. Exploiting these components 
the Stratego [51] system was used to translate a core subset of SPARK 
into our flowgraph representation. The program analyser itself is 
implemented in SICStus Prolog. During execution the program analysis 
exploits results from the PURRS [48] 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) [40]. SHOLIS was the first system developed to meet the UK Ministry of Defence Interim Defence Standards 00-55 [43] and 00-56 [42]. 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
In the above table, we separate our 21 examples depending on the number of NuSPADE iterations involved. On each iteration, we list the number of goals that the Simplifier fails to prove. We partition these unproven goals into those for which NuSPADE performs proof-failure analysis (PF) and those for which proof planning is successful (PP).

<table>
<thead>
<tr>
<th>Number of examples</th>
<th>Goals at iteration</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>PF</td>
</tr>
<tr>
<td>14</td>
<td>PF</td>
</tr>
<tr>
<td>1</td>
<td>PF</td>
</tr>
</tbody>
</table>

Examples are summarised in Figure 20. For a more detailed analysis of these results, see [32].

On each iteration, NuSPADE typically generates an additional annotation. Note that the number of remaining goals tends to increase after the first iteration of NuSPADE. This can be explained by the fact that the process of loop invariant strengthening typically gives rise to loop invariant VCs which the Simplifier fails to prove.

Our evaluation identified goals which NuSPADE failed to prove. NuSPADE does not support the generation of preconditions, which gave rise to three such failures. Due to resource constraints, the Examiner does not generate exhaustive rules for large constant structures. The resulting missing information gave rise to a further eight failures. In such situations NuSPADE directs the user to provide the missing rules. Once the rules are in place, NuSPADE successfully plans the remaining goals.

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 flex-
ibility 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 [49] 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 [52] and Polyspace [47]. 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 reported in [19, 20]. In particular, here we have significantly extended both the proof methods and critics. Further, we have broadened the application of our program analysis to deal with nested loops. In addition, we have 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 Simplier. 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 planning 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 linear expressions, we did not find a need for such enhancements in practise.

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 [38]. Further, there are 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 [35, 33] 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, Benjamin Gorry, Gudmund Grov, Jonathan Hammond, Tommy Ingulfsen, Ian O’Neill, Julian Richardson, Maria McCann and Phil Thornley for their feedback and
encouragement. The authors would like to thank the four anonymous JAR reviewers for their detailed and constructive feedback. The research reported in this paper is a collaboration between Heriot-Watt University and 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


