Adventures with PolySpace and friends

Contents

*Hums are short notes intended for distribution between those involved with EPSRC grant GR/T12289/01. Hums describe ϵ-baked ideas, where $1 \geq \epsilon \geq 0$. (Hum refers to both the Praxis Humming bird and Winnie-the-Pooh's indirect approach to writing: "Poetry and Hums aren't things which you get, they're things which get you.").

1 Introduction

Modern realisations about the potential for automated program reasoning tools have led to several niche systems being developed. These systems tend to target a subset of the verification problem, employing cutting edge verification techniques. While the tools tend to tackle similar verification problems, the techniques employed can vary substantially. This raises various questions: What are the similarities between these tools? What are differences? What advantages to the different styles bring? Can these be combined?

Here we focus on the practicalities on answering these questions, attempting to establish what comparisons and observations can be reasonably made without any overtly tenuous reasoning. Note that the original brief was to focus on SPARK and PolySpace. Although this has been followed, some short discussions are also made of other related systems. This helps to give a more accurate picture of the true context and the diverse nature of the systems in this field.

2 A series of observations

2.1 Features of static analysis systems

Drawing inspiration from table 1 in [1] general features seen of various static analysis systems are described:

Symbolic	Employs lexical analysis to find common patterns of code that tend to reflect problems.
Inter-procedural	Analysis descends inside procedures that are called from within procedures.
Control flow	The analysis takes control flow structures into consideration. This involves processing conditional statements (`if` statements, etc.) and taking this structure into account during the analysis.
Data flow	The analysis takes into account the flow of data through the program. This involves taking into account the nature of statements and the transformations they entail to data variables.
Abstract interpretation	The abstract interpretation framework is employed.
Theorem prover	A theorem prover is employed.
Model checking	A model checker is employed.
Constraint solving	A constraint solving system is employed.
Programming language	The programming language being analysed. This greatly affects the performance of a static analysis system, yet is often not considered a choice and hence may not generally be classed as a feature[1].

The various different systems seen take a pick and mix of these features, balancing the desire for automation with the conflicting desire for efficiency and practicality.

2.2 Its all been done before

Rod Chapman pointed out a particularly useful paper [1]. This makes a serious comparison between a selection of the better static analysis tools for detecting buffer overflows - essentially exception freedom with an emphasis on array and other memory block accesses. There is a careful classification of real world examples, considering both finding errors in code and not finding these errors once the code has been patched, all tied together with various metrics and statistics. This paper cites a few other other papers that have conducted similar, but less thorough, studies. Various ideas from this paper (and some others) will be considered within this note.

2.3 A fundamental problem of language

SPARK has been designed for static analysis. It removes complex constructs that naturally impede static analysis and provides a formal semantics allowing for an accurate interpretation of the meaning of the code. SPARK is, arguably, the most amenable language to static analysis that exists.

[1] After all, everybody writes code in C...

Thus, were an Abstract Interpretation tool targeted at SPARK, what would it achieve? Improved algorithms could be formulated, exploiting the cleaner semantics and reduced complexity. In particular, an Abstract Interpretation tool designed for Ada, but working in the SPARK subset, is unlikely to be as strong as a tool designed solely for SPARK. This observation is unfortunate. it means that, for any comparison to be fair, both tools should be built for targeting the same language.

It can, however, be successfully argued (assuming SPARK outperforms PolySpace) that working in a more constrained language leads to improved static analysis. Concluding, that, unlike any previous study[2], the key factor in improving static analysis is minimising the complexity and formalising the semantics of the underlying language[3].

2.4 The semantic gap

The semantic gap between the model and the code is significant in affecting the nature of the analysis and the results that can be gained.

Discussions of the BOON system note that it is not always obvious where a discovered problem lies in the code. The results are described in terms of solved constraint satisfaction conjectures, retrieved from the code ignoring all flow structures. To help address this problem the system related program variables to constraint variables. Discussions of ITS4, which adopts a lexical approach, suggest that the analysis will provide the exact line number of a suspected fault.

It seems natural that the more distant the model being analysed is from the source code the more difficult it is to relate issues in the model to the source code. This semantic gap tends to be greater for proof based approaches (such as SPARK) than for model based approaches (such as PolySpace).

Note that this may not be immediately apparent. SPARK propagates properties in VCs under the assumption that earlier VCs must be proved correct. A consequence is that all VCs must be proved correct to say anything categoric. For example, several problems may be masked by a single problem, such as a faulty invariant of the form:

```
--# assert false;.
```

2.5 Proof is stronger - but is this needed?

An advantage of proof is simultaneously supporting error discovery and demonstrating correctness. Thus, proof based approaches are stronger than model based approaches. However, outside the domain of high integrity software, a proof of correctness may not be desired. If debugging is the goal, a proof of an error is just as valuable as a model checker discovery of an error.

If demonstrating correctness is not important, proof might be seen as too strong a result. If this stronger result comes at additional cost, proof based approaches may reasonably be seen as inappropriate for such projects.

2.6 Some novel angles

Some angels on static analysis system comparisons system that appear not to have been considered thus far are highlighted:

- **Compare proof and model checking? -** A static analysis comparison paper has not been found that makes a comparison between proof based and model based approaches to exception freedom. References to SLAM can be found in related work sections of model based static analysis tools, but nothing deeper.
- **Compare dynamic and static? -** The static papers dismiss dynamic on the grounds of limited path coverage and need for test cases. The dynamic papers dismiss static on the grounds of unrealistic execution times and the common need for code manipulation to begin the analysis. There does not seem to be a study of the merits of these approaches, making a stance on which of the evils are least troublesome for practical use.
- **Compare machines with people? -** Code reviews are a form of static analysis that is apparently excluded from study on the grounds of not being automated. However, in practise, none of the automated tools are actually automated. Often, getting the tool to run, and interpreting the result involves a fair amount of interaction. Thus, it is possible that a user might be more productive in finding errors by avoiding static analysis tools altogether. At the very least, some indication of what can be achieved without the automated tools is required to describe productivity *improvements* where adopting static analysis tools.

[2]That i have seen.

[3]Which is, perhaps, rather obvious.

2.7 Advertise as automated (Used as user-guidance)

In [1] after quite a substantial amount of analysis effort has been made, it is noted that PolySpace and Splint (arguably the best model checking static analysis tools) returned a 'discrimination rate' ('the probability of not warning about a buffer overflow in a properly patched line of code') of 40%. In other words, it is noted with some satisfaction, they are worse at this task than flipping a coin. Nevertheless, the tools *do* narrow search. In this paper PolySpace (and only PolySpace) comes out as offering statistically significant positive results.

Some of the static analysis papers make reference to primarily manual techniques, perhaps involving the use of `grep`. This is considered the worse case scenario, where no automated tools are being used. In [2] it is noted that the new system being discussed significantly narrows the search space over such naive (`grep` like) techniques.

Perhaps such tools should place greater emphasis on ease of use and highlight that their intended application is in assisting user inspection, rather than automation. They need to demonstrate that a decent filtering (beyond just using `grep`) can be achieved with minor configuration, to make the system actually useful.

3 Notes on systems

3.1 SPARK

Classic Floyd-Hoare verification condition (VC) generator for both partial correctness and exception freedom. Many VCs are discharged automatically using the Simplifier.

3.1.1 Simplifier

The Simplifier is a strange beast. It is not built upon any fundamental decision procedure. It has been gradually extended, incorporating new styles of reasoning in reaction to the kinds of problems being seen. The best analogy is perhaps a super tactic as might be implemented in HOL. The Simplifier rapidly executes a sequence of operations, not paying much attention to the nature of the goal, hoping to eventually reduce the goal to true.

In terms of logical implementation, the Simplifier can be thought of as containing a core logic engine and the description of a super tactic that is executed on this logic engine. In terms of the actual implementation, the logical aspects and the control aspects are fundamentally intertwined. Interestingly, this lack of separation may explain the fast speed of the Simplifier. By directly manipulating logic at the control level there is no time consuming translation and checking phase, as might be seen in a traditional proof engine and tactic implementation.

The Simplifier does not generate a sequence of proof steps. However, unlike real decision procedures, there is no fundamental reason why it should not.

3.2 ESC Java

Converts from Java to conjectures which are dispatched to an automated theorem prover. Results are dependent on the success or otherwise of the prover. ESC places automation above soundness. There is no suggestion that the user should undertake to discharge the proofs that are not automatically proven.

3.3 RunCheck

RunCheck is the classic, possibly seminal, work on detecting run-time errors. RunCheck formalised a section of Pascal, employed a few carefully selected program analysis heuristics, and dispatched conjectures using an automated theorem prover. Significantly, RunCheck eased the task of analysis by curtailing the semantics of the programming language.

3.4 PolySpace

PolySpace follows the Abstract Interpretation model to implement static program analysis. PolySpace targets the detection of run-time errors in Ada, C and C++. Unlike cutting edge variants of Abstract Interpretation which target particular programs, PolySpace is applicable in general.

PolySpace is a model checker. It converts a computer program into a model of a computer program and then analyses this model. It is essentially impossible to verify that the model accurately reflects the original program, thus it is flawed to use the results from such tools to make categorical claims about the behaviour of the original program. Fundamentally, model checkers improve confidence of correctness and detect bugs - they do not perform proof.

3.4.1 Green, orange and red (and grey)

These three colours and one shade are used throughout by PolySpace to convey results to the user. These describe the intersection between a variables expected values and the values that would cause a particular run-time error.

- **green** - There is zero overlap between a variables expected values and the error values. Thus this run-time error can not occur.
- **orange** - There is a partial overlap between a variables expected values and the error values. It can not be guaranteed that there is not an error. It can not be guaranteed that there is an error.
- **red** - The variables expected values are entirely subsumed by the error values. A run-time error will occur.
- **grey** - Gray is a special case used to denote unreachable code.

3.4.2 Red in detail

A programmer will prefer to see red over orange. red denotes extremely severe errors! Once PolySpace hits a red error it is unable to reliably enter its next 'integration pass'. Further, the PolySpace methodology strongly encourages the initial targeting of red. Thus, while red denotes knowledge of the definite presence of an error, this is not expected to happen often as such absolute errors will be detected and pruned as early as possible.

This is an interesting observation, as it means PolySpace really does the exact opposite of locating run-time errors. Rather than detecting errors, it locates definite non-errors allowing the user to focus on the smaller subset of undecided cases.

3.4.3 Green in detail

A programmer wants to see as much green as possible. This indicates that PolySpace did not detect an error. In practise, green can occur (often) providing useful guidance to the user.

3.4.4 Orange in detail

A programmer does not want to see any orange. This indicates cases that can not be categorised as red or green. The PolySpace methodology outlines a number of heuristics and strategies to better address the presence of orange. For example, it is suggested that a user spend five minutes investigating an orange - if no definite error is found after five minutes then it is fairly likely not to be an error.

3.4.5 Picky PolySpace

PolySpace is especially pedantic about the hardware it should be executed on. There is no suggestion that the functionality of PolySpace depends on the hardware. However, of course, PolySpace will give poorer performance on slower hardware. Essentially, the PolySpace requirements entail using the best hardware possible. The better the hardware, the quicker the performance.

3.4.6 Installing PolySpace

PolySpace is supported under Linux, but only for particular versions of Red Hat. Despite some successful tricks, it was found to be too difficult to make PolySpace believe that our laptop (running Fedora Core 2 Linux) was Red Hat. PolySpace is also supported under Windows. PolySpace installed directly on a desktop running Windows 2000. However, for reasons still unsolved, the viewer component failed to display. PolySpace was then placed on a third computer, this time running Windows XP, hoping to resolve the viewer problem. This was successful. However, unfortunately, the verifier component does not operate on Windows XP, possibly as a consequence of requiring access to key kernel features of Windows 2000 and Windows NT. Finally, PolySpace was installed on another, more powerful, computer running Windows 2000. Both the viewer component and verifier component work correctly. The desktop component does not operate, causing a moment of brief panic, however this facility is intended for managing real applications, and its activation has been, quite reasonably, removed from our licence file.

Thus, we have PolySpace, fully patched and working as intended, on a fairly powerful (still underspecified from PolySpace's perspective) computer running Windows 2000.

3.5 MOPS

3.6 ITS4

Calculates results based on lexical analysis. Unlike many other static analysis tools this can be applied directly to the original code base. However, the lightweight lexical analysis tends to produce weaker results.

3.7 RATS

Similar to ITS4.

3.8 BOON

BOON[4] does not consider program flow, collects relevant statements, and calculates results based on performing constraint solving.

3.9 Caveat

Ties together abstract interpretation, program analysis and theorem proving.

3.10 SLAM

3.11 BLAST

4 Some foundations

4.1 Problem, error, and bug

Here it is attempted to introduce some terminology to support describing the subtle differences between proof based and model based approaches to static analysis:

- **Problem -** A static analysis tool reports a problem when it is unable to automatically resolve the situation. The static analysis tool is answering: 'i don't know'.
- **Error -** A static analysis tool, automatically (or via user guidance), concludes that an error exists. The static analysis tool is answering: 'there is something wrong here'.
- **Specification and/or Code Bug -** A static analysis tool, automatically or via user guidance, concludes that a bug exists somewhere in the specification or in the code. The static analysis tool is answering: 'the specification or code is flawed here'.
- **Code Bug -** A static analysis tool, automatically or via user guidance, concludes that a bug exists in the code. The static analysis tool is answering: 'the code is flawed here'.

Using these definitions the proof based and model based styles are compared:

- **Proof based -** In a proof based approach it is a problem if a conjecture can not be automatically proved. It is an error if the conjecture can not be proved. Every error indicates a bug in the specification and/or code. There is no way to identify the true cause of the bug without employing informal analysis.
- **Model based -** In a model based approach it is a problem if the flows through a code point are not all outside or not all inside the legal bounds. The interpretation of problems requires informal analysis.

  It is an error if the flows through a code point are all outside the legal bounds. Every error indicates a code bug.

These subtle differences are important to gain an appreciation for the behaviour is these tools. However, from the users perspective, the goal is to eliminate bugs from the code. The tools should be measured on their ability to achieve this task.

[4] Also referred to as 'the system built by Wagner, *et all*' until they gave it a name.

4.2 Relating SPARK and PolySpace

Proof techniques tend to focus on correctness over error discovery. To prove a program correct every VC must be proved correct. From this it can be argued that any automated program proof that does not discharge 100% of the VCs is returing an undecided result. Thus, the vast majority of the time, proof does not make any categoric claims. In practise, this is not a problem as there is a definite relationship between the VCs and the code. The unproved VCs direct the user to inspect particular areas of the code. Thus, even where nothing categoric is reported, the user is still gaining useful insight. Thus, the following mapping can be (essentially) made between SPARK and PolySpace:

Assume the colours are used to refer to individual VCs:

- **red -** The VC is automatically proved to be false.

  (SPARK: There is a specification and/or code bug.) (PolySpace: There is a code bug.)

- **orange -** The VC is undischarged. The VC may or may not be provable.

  (SPARK: There is a problem.) (PolySpace: There is a problem.)

- **green -** The VC is discharged. The VC does not warrant analysis. Note this is not strictly true unless all of the VCs are discharged. The slightly stronger relationship to the code means that when PolySpace reports **green** it is safe - regardless of the status of other colours.

  (SPARK: Essentially no problem.) (PolySpace: No problem.)

- **grey -** SPARK supports the detection of an infeasible path by finding contradictory hypotheses. If every path to a section of code contains contradictory hypotheses then it is unreachable code. However, such an approach to detecting unreachable code is not directly supported in SPARK.

  (SPARK: Infeasible path.) (PolySpace: Unreachable code.)

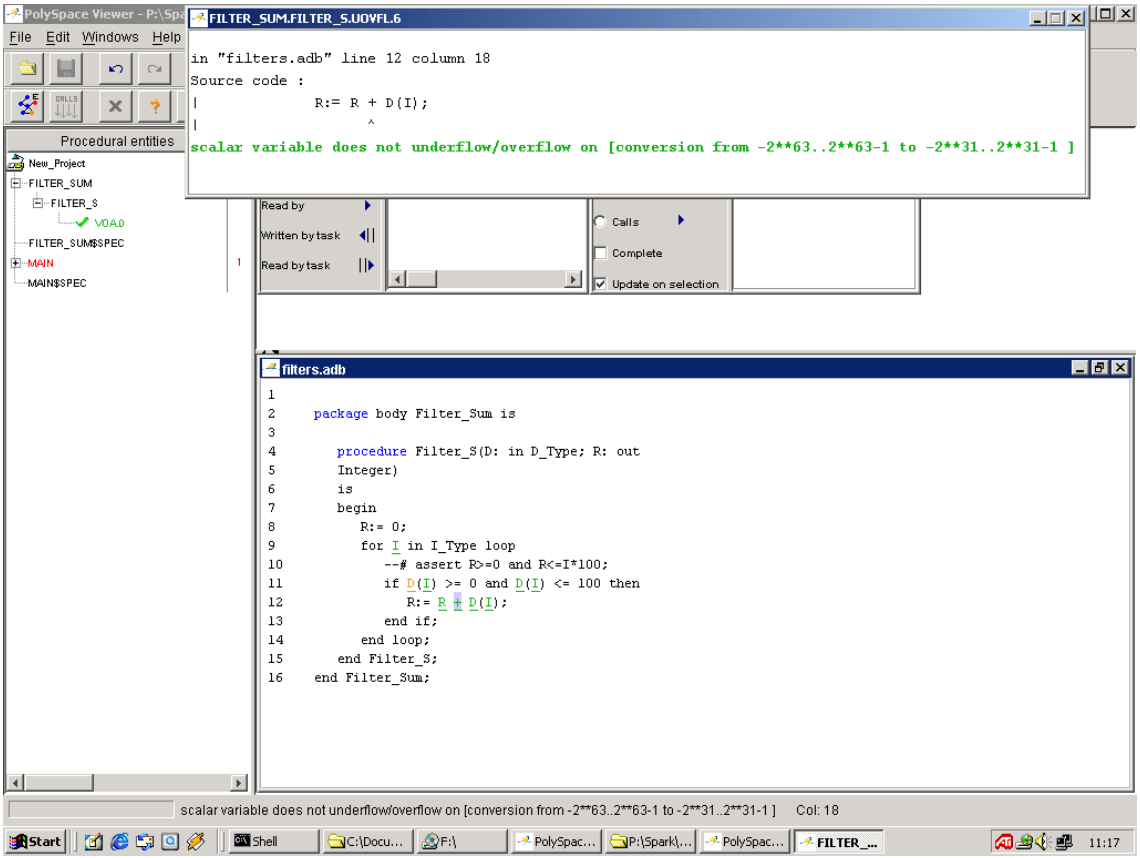
5 Playing with PolySpace

5.1 Filters - Experiment 1

5.1.1 Code

```
package Filter_Sum is

   subtype I_Type is Integer range 0 .. 9;
   type D_Type is array (I_Type) of Integer;

   procedure Filter_S(D: in D_Type; R: out
   Integer);
   --# derives R from D;

end Filter_Sum;


package body Filter_Sum is

   procedure Filter_S(D: in D_Type; R: out
   Integer)
   is
   begin
      R:= 0;
      for I in I_Type loop
         --# assert R>=0 and R<=I*100;
         if D(I) >= 0 and D(I) <= 100 then
            R:= R + D(I);
         end if;
      end loop;
   end Filter_S;
end Filter_Sum;


with Filter_Sum;
--#inherit Filter_Sum;

--# main_program;
procedure main
    --# derives ;
    is
    Whatever: Integer;

    D: Filter_Sum.D_Type;
    R: Integer;
    pragma volatile (Whatever);

    pragma volatile (D);
    pragma volatile (R);
    begin
    loop
        if (Whatever = 1) then
            Filter_Sum.Filter_S(D, R);
        end if;
    end loop;
end main;
```

5.1.2 Results

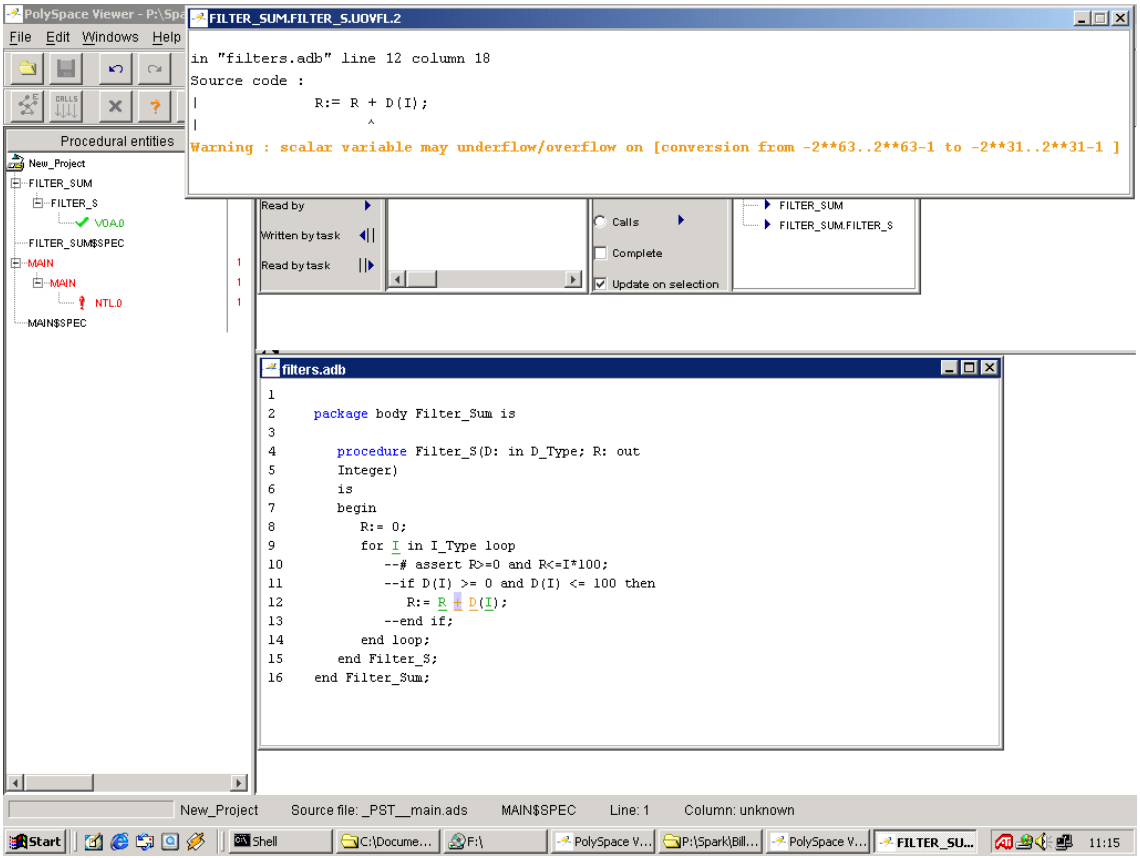

PolySpace Viewer - P:\Spa
File Edit Windows Help
Procedural entities
New_Project
FILTER_SUM
FILTER_S
VOA.0
FILTER_SUM$SPEC
MAIN
MAIN$SPEC
FILTER_SUM.FILTER_S.UOVFL.6
in "filters.adb" line 12 column 18
Source code :
| R:= R + D(I);
| ^
scalar variable does not underflow/overflow on [conversion from -2**63..2**63-1 to -2**31..2**31-1]
Read by
Written by task
Read by task
Calls
Complete
Update on selection
filters.adb
1
2 package body Filter_Sum is
3
4 procedure Filter_S(D: in D_Type; R: out
5 Integer)
6 is
7 begin
8 R:= 0;
9 for I in I_Type loop
10 --# assert R>=0 and R<=I*100;
11 if D(I) >= 0 and D(I) <= 100 then
12 R:= R + D(I);
13 end if;
14 end loop;
15 end Filter_S;
16 end Filter_Sum;
scalar variable does not underflow/overflow on [conversion from -2**63..2**63-1 to -2**31..2**31-1] Col: 18
Start
Shell
C:\Docu...
F:\
PolySpac...
P:\Spark\...
PolySpac...
FILTER_...
11:17

5.2 Filters - Experiment 2

5.2.1 Code

```
package Filter_Sum is

   subtype I_Type is Integer range 0 .. 9;
   type D_Type is array (I_Type) of Integer;

   procedure Filter_S(D: in D_Type; R: out
   Integer);
   --# derives R from D;

end Filter_Sum;


package body Filter_Sum is

   procedure Filter_S(D: in D_Type; R: out
   Integer)
   is
   begin
      R:= 0;
      for I in I_Type loop
         --# assert R>=0 and R<=I*100;
         --if D(I) >= 0 and D(I) <= 100 then
            R:= R + D(I);
         --end if;
      end loop;
   end Filter_S;
end Filter_Sum;


with Filter_Sum;
--#inherit Filter_Sum;

--# main_program;
procedure main
    --# derives ;
    is
    Whatever: Integer;

    D: Filter_Sum.D_Type;
    R: Integer;
    pragma volatile (Whatever);

    pragma volatile (D);
    pragma volatile (R);
    begin
    loop
        if (Whatever = 1) then
            Filter_Sum.Filter_S(D, R);
        end if;
    end loop;
end main;
```

5.2.2 Results

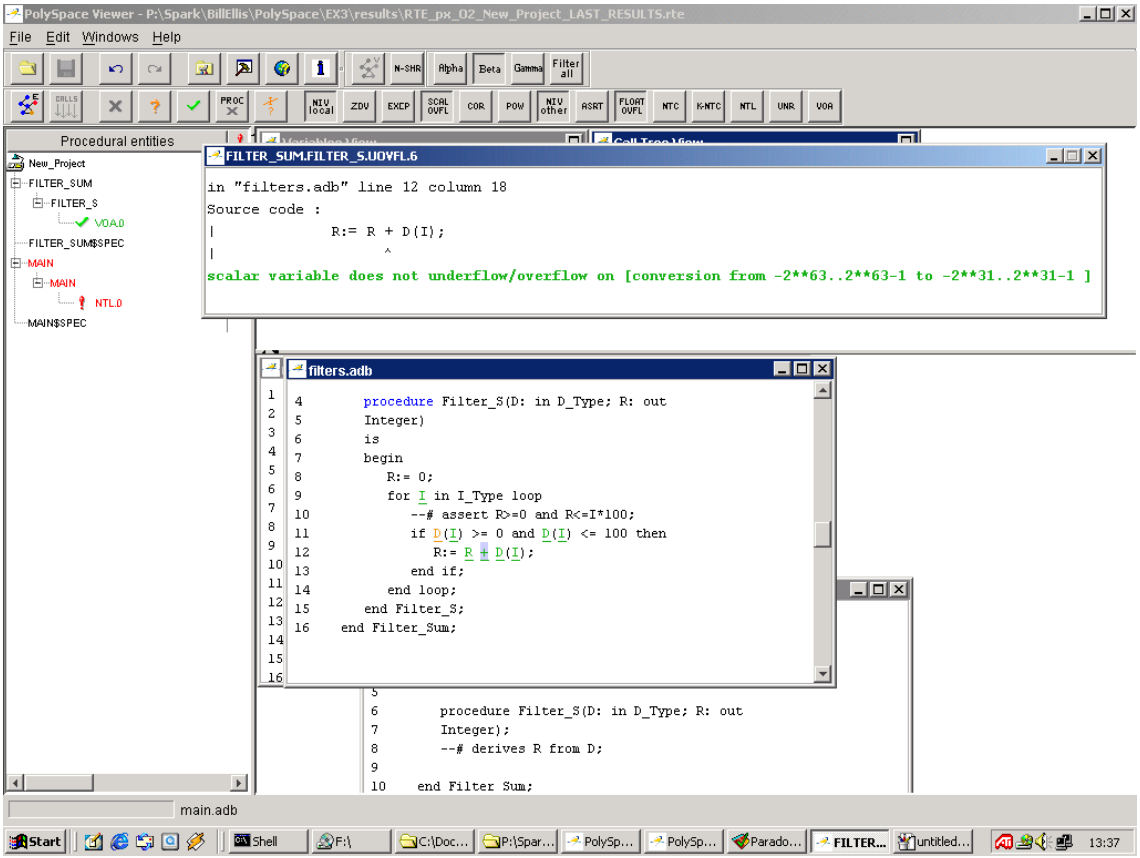

PolySpace Viewer - P:\Sp
File Edit Windows Help
Procedural entities
New_Project
FILTER_SUM
FILTER_S
VOA.0
FILTER_SUM$SPEC
MAIN
MAIN
NTL.0
MAIN$SPEC
FILTER_SUM.FILTER_S.UOVFL.2
in "filters.adb" line 12 column 18
Source code :
| R:= R + D(I);
| ^
Warning : scalar variable may underflow/overflow on [conversion from -2**63..2**63-1 to -2**31..2**31-1]
Read by
Written by task
Read by task
Calls
Complete
Update on selection
FILTER_SUM
FILTER_SUM.FILTER_S
filters.adb
1
2 package body Filter_Sum is
3
4 procedure Filter_S(D: in D_Type; R: out
5 Integer)
6 is
7 begin
8 R:= 0;
9 for I in I_Type loop
10 --# assert R>=0 and R<=I*100;
11 --if D(I) >= 0 and D(I) <= 100 then
12 R:= R + D(I);
13 --end if;
14 end loop;
15 end Filter_S;
16 end Filter_Sum;
New_Project
Source file: _PST__main.ads
MAIN$SPEC
Line: 1
Column: unknown
Start
Shell
C:\Docume...
F:\
PolySpace V...
P:\Spark\Bill...
PolySpace V...
FILTER_SU...
11:15

5.3 Filters - Experiment 3

5.3.1 Code

```
package Filter_Sum is

   subtype I_Type is Integer range 0 .. 1000;
   type D_Type is array (I_Type) of Integer;

   procedure Filter_S(D: in D_Type; R: out
   Integer);
   --# derives R from D;

end Filter_Sum;


package body Filter_Sum is

   procedure Filter_S(D: in D_Type; R: out
   Integer)
   is
   begin
      R:= 0;
      for I in I_Type loop
         --# assert R>=0 and R<=I*100;
         if D(I) >= 0 and D(I) <= 100 then
            R:= R + D(I);
         end if;
      end loop;
   end Filter_S;
end Filter_Sum;


with Filter_Sum;
--#inherit Filter_Sum;

--# main_program;
procedure main
    --# derives ;
    is
    Whatever: Integer;

    D: Filter_Sum.D_Type;
    R: Integer;
    pragma volatile (Whatever);

    pragma volatile (D);
    pragma volatile (R);
    begin
    loop
        if (Whatever = 1) then
            Filter_Sum.Filter_S(D, R);
        end if;
    end loop;
end main;
```

5.3.2 Results

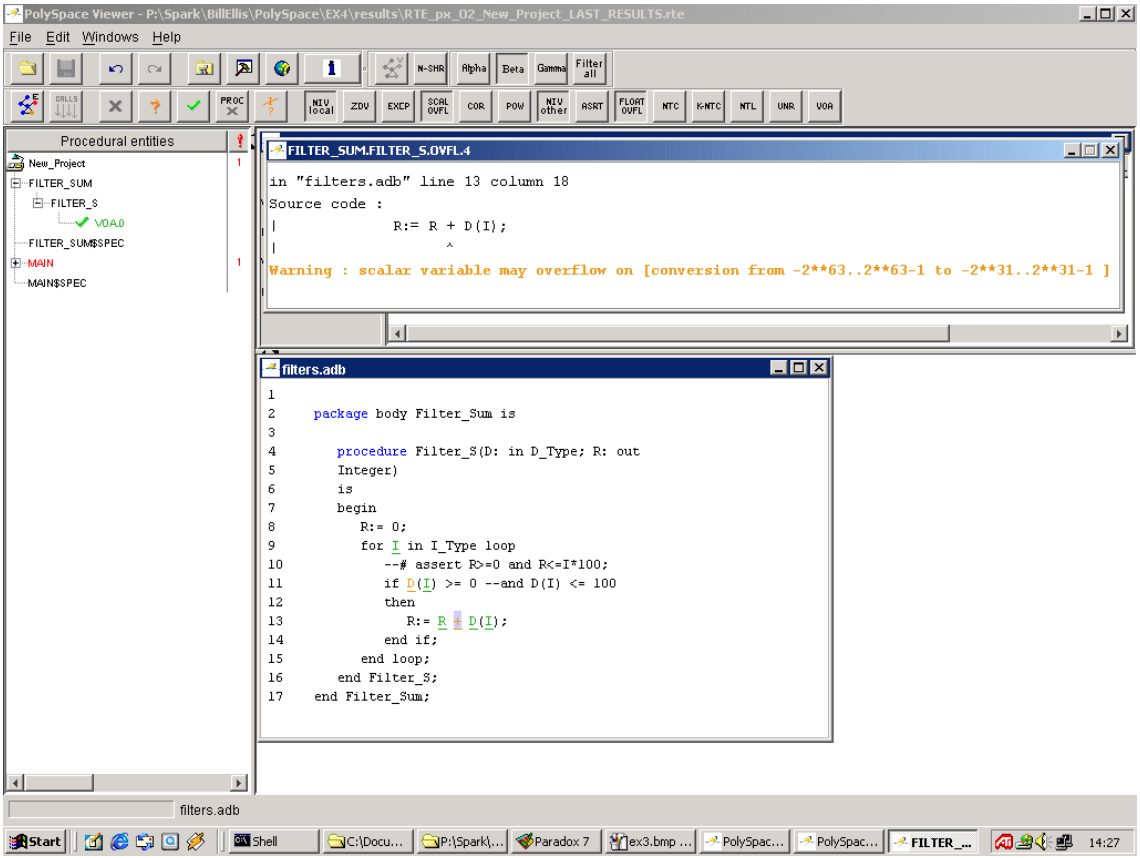

PolySpace Viewer - P:\Spark\BillEllis\PolySpace\EX3\results\RTE_px_02_New_Project_LAST_RESULTS.rte
File Edit Windows Help
Procedural entities
New_Project
FILTER_SUM
FILTER_S
VOA.0
FILTER_SUM$SPEC
MAIN
MAIN
NTL.0
MAIN$SPEC
FILTER_SUM.FILTER_S.UOVFL.6
in "filters.adb" line 12 column 18
Source code :
| R:= R + D(I);
| ^
scalar variable does not underflow/overflow on [conversion from -2**63..2**63-1 to -2**31..2**31-1]
filters.adb
4 procedure Filter_S(D: in D_Type; R: out
5 Integer)
6 is
7 begin
8 R:= 0;
9 for I in I_Type loop
10 --# assert R>=0 and R<=I*100;
11 if D(I) >= 0 and D(I) <= 100 then
12 R:= R + D(I);
13 end if;
14 end loop;
15 end Filter_S;
16 end Filter_Sum;
6 procedure Filter_S(D: in D_Type; R: out
7 Integer);
8 --# derives R from D;
9
10 end Filter_Sum;
main.adb
Start
Shell
F:\
C:\Doc...
P:\Spar...
PolySp...
PolySp...
Parado...
FILTER...
untitled...
13:37

5.4 Filters - Experiment 4

5.4.1 Code

```
package Filter_Sum is

   subtype I_Type is Integer range 0 .. 1000;
   type D_Type is array (I_Type) of Integer;

   procedure Filter_S(D: in D_Type; R: out
   Integer);
   --# derives R from D;

end Filter_Sum;


package body Filter_Sum is

   procedure Filter_S(D: in D_Type; R: out
   Integer)
   is
   begin
      R:= 0;
      for I in I_Type loop
         --# assert R>=0 and R<=I*100;
         if D(I) >= 0 --and D(I) <= 100
         then
            R:= R + D(I);
         end if;
      end loop;
   end Filter_S;
end Filter_Sum;


with Filter_Sum;
--#inherit Filter_Sum;

--# main_program;
procedure main
    --# derives ;
    is
    Whatever: Integer;

    D: Filter_Sum.D_Type;
    R: Integer;
    pragma volatile (Whatever);

    pragma volatile (D);
    pragma volatile (R);
    begin
    loop
        if (Whatever = 1) then
            Filter_Sum.Filter_S(D, R);
        end if;
    end loop;
end main;
```

5.4.2 Results

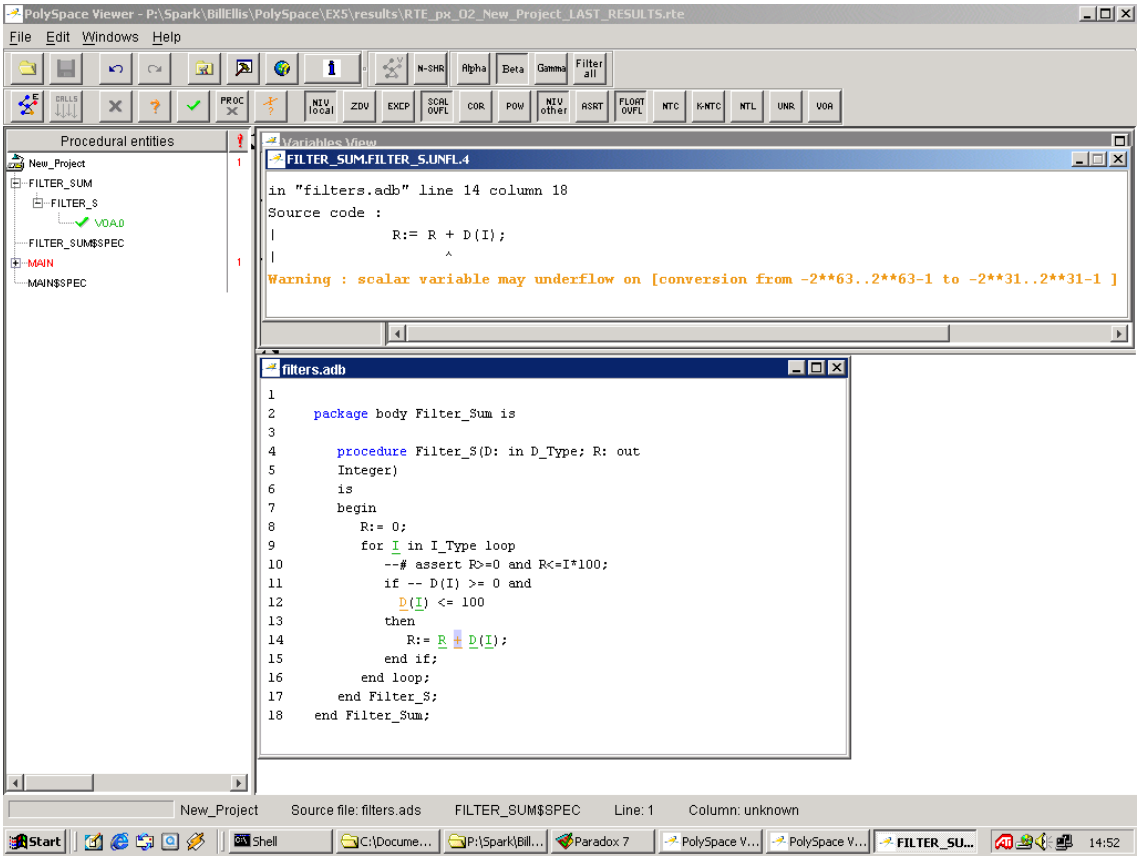

PolySpace Viewer - P:\Spark\BillEllis\PolySpace\EX4\results\RTE_px_02_New_Project_LAST_RESULTS.rte
File Edit Windows Help
Procedural entities
New_Project
FILTER_SUM
FILTER_S
VOA.0
FILTER_SUM$SPEC
MAIN
MAIN$SPEC
FILTER_SUM.FILTER_S.OVFL.4
in "filters.adb" line 13 column 18
Source code :
| R:= R + D(I);
| ^
Warning : scalar variable may overflow on [conversion from -2**63..2**63-1 to -2**31..2**31-1]
filters.adb
1
2 package body Filter_Sum is
3
4 procedure Filter_S(D: in D_Type; R: out
5 Integer)
6 is
7 begin
8 R:= 0;
9 for I in I_Type loop
10 --# assert R>=0 and R<=I*100;
11 if D(I) >= 0 --and D(I) <= 100
12 then
13 R:= R + D(I);
14 end if;
15 end loop;
16 end Filter_S;
17 end Filter_Sum;
filters.adb
Start
Shell
C:\Docu...
P:\Spark\...
Paradox 7
ex3.bmp ...
PolySpac...
PolySpac...
FILTER_...
14:27

5.5 Filters - Experiment 5

5.5.1 Code

```
package Filter_Sum is

   subtype I_Type is Integer range 0 .. 1000;
   type D_Type is array (I_Type) of Integer;

   procedure Filter_S(D: in D_Type; R: out
   Integer);
   --# derives R from D;

end Filter_Sum;


package body Filter_Sum is

   procedure Filter_S(D: in D_Type; R: out
   Integer)
   is
   begin
      R:= 0;
      for I in I_Type loop
         --# assert R>=0 and R<=I*100;
         if -- D(I) >= 0 and
           D(I) <= 100
         then
            R:= R + D(I);
         end if;
      end loop;
   end Filter_S;
end Filter_Sum;


with Filter_Sum;
--#inherit Filter_Sum;

--# main_program;
procedure main
    --# derives ;
    is
    Whatever: Integer;

    D: Filter_Sum.D_Type;
    R: Integer;
    pragma volatile (Whatever);

    pragma volatile (D);
    pragma volatile (R);
    begin
    loop
        if (Whatever = 1) then
            Filter_Sum.Filter_S(D, R);
        end if;
    end loop;
end main;
```

5.5.2 Results

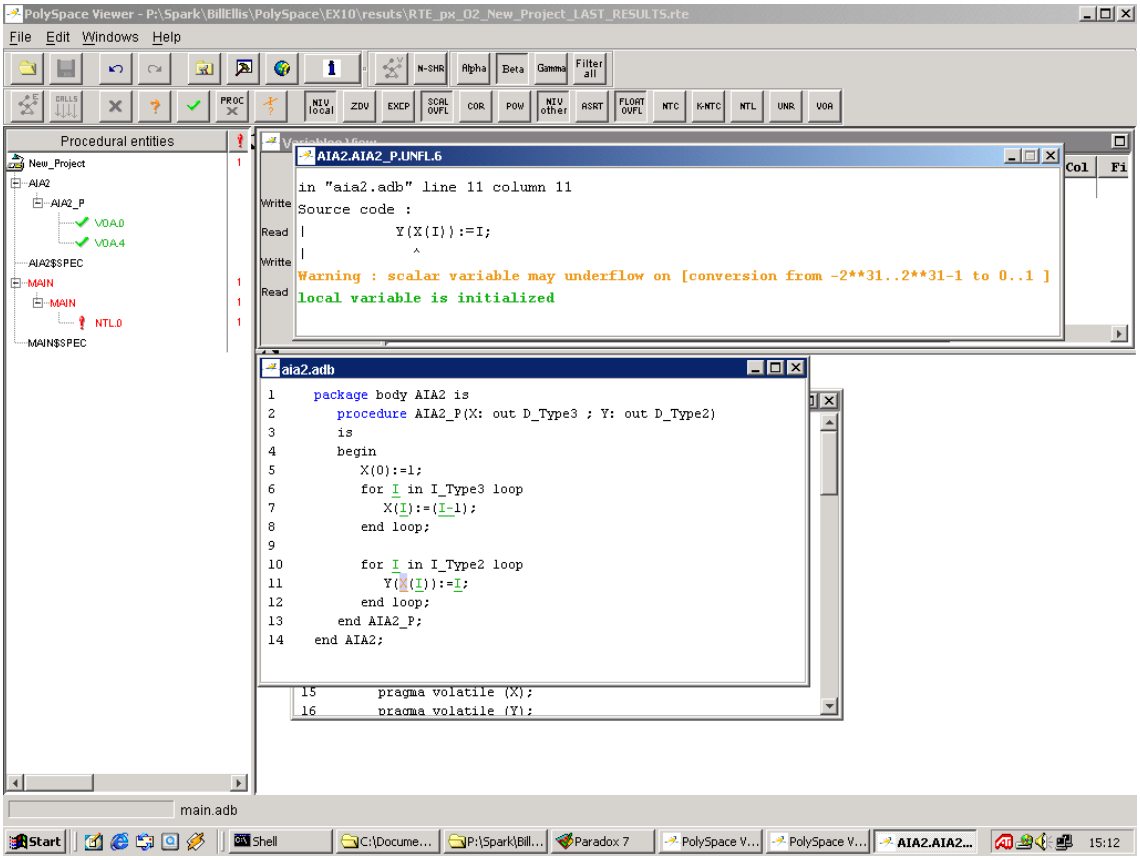

PolySpace Viewer - P:\Spark\BillEllis\PolySpace\EX5\results\RTE_px_02_New_Project_LAST_RESULTS.rte
File Edit Windows Help
Procedural entities
New_Project
FILTER_SUM
FILTER_S
VOA.0
FILTER_SUM$SPEC
MAIN
MAIN$SPEC
Variables View
FILTER_SUM.FILTER_S.UNFL.4
in "filters.adb" line 14 column 18
Source code :
| R:= R + D(I);
| ^
Warning : scalar variable may underflow on [conversion from -2**63..2**63-1 to -2**31..2**31-1]
filters.adb
1
2 package body Filter_Sum is
3
4 procedure Filter_S(D: in D_Type; R: out
5 Integer)
6 is
7 begin
8 R:= 0;
9 for I in I_Type loop
10 --# assert R>=0 and R<=I*100;
11 if -- D(I) >= 0 and
12 D(I) <= 100
13 then
14 R:= R + D(I);
15 end if;
16 end loop;
17 end Filter_S;
18 end Filter_Sum;
New_Project
Source file: filters.ads
FILTER_SUM$SPEC
Line: 1
Column: unknown
Start
Shell
C:\Docume...
P:\Spark\Bill...
Paradox 7
PolySpace V...
PolySpace V...
FILTER_SU...
14:52

5.6 AIA2 - Experiment 1

5.6.1 Code

```
package body AIA2 is
   procedure AIA2_P(X: out D_Type3 ; Y: out D_Type2)
   is
   begin
      X(0):=1;
      for I in I_Type3 loop
         X(I):=(I-1);
      end loop;

      for I in I_Type2 loop
         Y(X(I)):=I;
      end loop;
   end AIA2_P;
end AIA2;

package AIA2 is

   subtype I_Type2 is Integer range 0 .. 1;
   subtype I_Type3 is Integer range 0 .. 2;

   type D_Type2 is array (I_Type2) of Integer;
   type D_Type3 is array (I_Type3) of Integer;

   procedure AIA2_P(X: out D_Type3 ; Y: out D_Type2);
   --# derives X, Y from ;

end AIA2;


with AIA2;
--#inherit AIA2;

--# main_program;
procedure main
    --# derives ;
    is
    Whatever: Integer;

    X: AIA2.D_Type3;
    Y: AIA2.D_Type2;
    pragma volatile (Whatever);

    pragma volatile (X);
    pragma volatile (Y);
    begin
    loop
        if (Whatever = 1) then
            AIA2.AIA2_P(X, Y);
        end if;
    end loop;
end main;
```

5.6.2 Results


PolySpace Viewer - P:\Spark\BillEllis\PolySpace\EX10\resuts\RTE_px_02_New_Project_LAST_RESULTS.rte
File Edit Windows Help
Procedural entities
New_Project
AIA2
AIA2_P
VOA.0
VOA.4
AIA2$SPEC
MAIN
MAIN
NTL.0
MAIN$SPEC
AIA2.AIA2_P.UNFL.6
in "aia2.adb" line 11 column 11
Source code :
Y(X(I)):=I;
Warning : scalar variable may underflow on [conversion from -2**31..2**31-1 to 0..1]
local variable is initialized
aia2.adb
1 package body AIA2 is
2 procedure AIA2_P(X: out D_Type3 ; Y: out D_Type2)
3 is
4 begin
5 X(0):=1;
6 for I in I_Type3 loop
7 X(I):=(I-1);
8 end loop;
9
10 for I in I_Type2 loop
11 Y(X(I)):=I;
12 end loop;
13 end AIA2_P;
14 end AIA2;
15 pragma volatile (X);
16 pragma volatile (Y);
main.adb
Start
Shell
C:\Docume...
P:\Spark\Bill...
Paradox 7
PolySpace V...
PolySpace V...
AIA2.AIA2...
15:12

6 Comments about PolySpace results

6.1 Filters

PolySpace successfully tackles the filters example. This is an improvement over previous studies on an earlier version of PolySpace. Various refinements to the filters example were considered and PolySpace behaved correctly in each. Note that PolySpace does present an orange about the initialisation of the input array D. This seems elementarily true (given that the array is an input).

6.2 AIA2

PolySpace performs admirably on this tricky example taken from [1]. It essentially reports the same results as SPARK where using default invariants.

7 Rough conclusions

The following rough conclusions are made. Note that there is probably not sufficient evidence in this note to fairly reach such conclusions. Nevertheless, it is estimated that they are on the right track:

- PolySpace is actually quite good at exploring individual examples within the SPARK semantics.
- Other studies have shown that PolySpace (and friends) is weaker when dealing with complex structures, such as data pointers, functions pointers, etc. Such studies makes the unremarkable claim that static analysis on complex software returns poor results.
- Other studies have shown that PolySpace (and friends) struggle with large systems. Experience on tiny examples, even on a medium range computer, tends to suggest that this observation is quite accurate.
- PolySpace does not have any annotations. The user can not insert additional information to improve the analysis. The orange results from an analysis may be dismissed by the user, but the results will continue to be present in future analysis. It is difficult to trace these dismissed oranges across different revisions of the code. PolySpace is good for a sweep of the code, and detecting bugs, but is less practical for application in an iterative process. Such an iterative process is expected to be desired for most software development projects.
- Technically speaking, there are aspects of Ada that are compiler dependent. PolySpace appears to assume a default behaviour of the compiler in such cases, and thus may report inaccurate results. There exists a few examples that demonstrate false aliasing assumptions and order evaluation assumptions. Nevertheless, PolySpace results were never intended to faithfully reflect the code. They make assumptions to ease analysis and interaction.
- SPARK is much faster, has a sound formal basis, and can automatically (via the Simplifier) provide similar results to PolySpace. Via annotations and user provided proofs SPARK can be employed in an iterative development process. Eventually, SPARK is be able to guarantee the absence of bugs, rather than report that none were found.

References

[1] Tim Leek Misha Zitser, Richard Lippmann. Testing static analysis tools using exploitable buffer overflows from open source code. 2004.

[2] David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities. In *Networking and Distributed System Security Symposium 2000*, San Diego, California, February 2000.