CS-Note* 57

Bill J. Ellis
August 18, 2003

Visiting Praxis: A Praxis Shopping List

1 Introduction

The plan is to visit Praxis Critical systems from Monday July the 28$^{\text{th}}$ to Friday August the 1$^{\text{st}}$. Although there remains the possibility of staying until the following Monday, the intention is to complete the various tasks within the first week.

2 Task 1: Inspect the code

Although familiar with SPARK, seen many SPARK verification conditions and read various reports of proofs of high integrity software, the software itself remains something of a mystery. A few assumptions have been made about the nature of high integrity software:

- **Is geared for showing correctness -** Software must conform to a programming language. However, this no more dictates the style of the code than English dictates the style of a novel. High integrity software will follow a style to assist in the key objective of showing that the software is correct. This tends to result in simple elementary constructs being selected in favour of powerful complex constructs.
- **Is structured and constrained -** It is expected that high integrity software is particularly well structured into several subprograms. Further, it is expected that constructs which enhance the level of detail in the code such as types, subtypes and enumerated values, will be common.

Some questions to consider are:

- **Loops -** How frequent? ; How many? ; What kind? (for, while, general) ; What counters?
- **Iteration ranges -** Complexity? ; Dynamic (inlined expressions) or static (just use type)?
- **Case statements -** When chose over if statement? ; Always total coverage? ; Or use default?
- **Data structures -** Complexity? ; Number of array dimensions? ; Integers or floats? ; Base other than ten?
- **Initialisation -** What is the extent of code outside subprograms?
- **Conditionals -** Are they complex? ; Are double references common? (`A<B or B=2`) ; Is disjunction broken down into nested ifs? ; What tests are common? (Inequalities verses equalities) ; Is direct function evaluation common? (`A=this(X,Z,Z)`)
- **Functions -** How common? ; What do they do? ; Are return postconditions common?
- **Coding -** Are return values overloaded? (e.g. return 0 on error) ; How handle errors? ; What is the balance between using preconditions and guards?
- **Low level -** Is `--#hide` common? ; How much regular SPARK?
- **Proof -** Is there a style for partial correctness code too? ; Any extra rules used? ; Kind of extra properties added? ; Are postconditions common for exception freedom? ; How work around floats?

The aim - test the assumptions and establish the nature of high integrity software. Hopefully these insights will help support our current heuristics and assist in the development of new heuristics.

*Notes in this series are for ϵ-baked ideas, for $1 \geq \epsilon \geq 0$. Only exceptionally should they be cited or distributed outwith the NuSPADE project (EPSRC GR/R24081).

3 Task 2: Little bits of evaluation

NuSPADE currently comprises of several unfinished modules. Evaluation of the system is therefore difficult! However, it is possible to evaluate the ideas and test some bits of the modules.

- **Stratego Spider (SPIDER) -** A key component of NuSPADE is providing access to the SPARK source. The SPIDER crawls over a SPARK source tree creating accessible subprogram descriptors (see §3.1).

  The problems with this are mechanical but involved. In particular getting variable scope correct and resolving names across the whole project present some technical difficulties.

- **Smart VCs (SMART)-** Following from the SPIDER is the intention to present all data in an accessible form, rather than just the source code. Smart VCs are regular VCs pre-processed for analysis. These may also be refined to integrate more closely with the program analysis. For example, rather than describe the VC path in terms of line numbers in the original source they might be directly related to the pseudo-code used for program analysis.

- **Property Generator (PROPGEN)-** This accesses the output of the SPIDER generating properties at program points, such as invariants, preconditions and postconditions. Currently PROPGEN is built around a weak input format and an older version of SPIDER. Worse, it (nearly) employs, an old brand of analysis. The approach to property generation was refined following autoGAP[1]. This successfully demonstrated more ad-hoc program analysis while the full complexities of the original PROPGEN became apparent. This is documented in an unfinished note.

- **Clam and** NuSPADE **Proof Planner (NuSPADE)-** Our proof planner is surprisingly complete. However, it is also old and out of sync with new developments. The aim is to build up to the proof planner and continue to use Clam in the mean time.

The aim - check that the implementation continues to go in the right direction and that our existing heuristics are viable.

[1] Tommy Ingulfsen's fourth year dissertation project.

3.1 Talking about Spark

A new problem is the presentation of SPARK code for program analysis. Essentially this involves translating SPARK into something that can be easily analysis within Prolog. An example of this is below:

```
sparkVariable(name(reference([simple('PolishFlag'), simple('Partition_Section'), simple('Flag')])),
              name(reference([simple('PolishFlag'), simple('Partition_Section'), simple('ArrayOfColours')])),
              parameter(inout)).

sparkVariable(name(reference([simple('PolishFlag'), simple('Partition_Section'), simple('I')])),
              name(reference([simple('PolishFlag'), simple('JustBiggerRange')]),
              local(uninitilised)).

sparkVariable(name(reference([simple('PolishFlag'), simple('Partition_Section'), simple('J')]),
              name(reference([simple('PolishFlag'), simple('JustBiggerRange')]),
              local(uninitilised)).

sparkVariable(name(reference([simple('PolishFlag'), simple('Partition_Section'), simple('T')]),
              name(reference([simple('PolishFlag'), simple('Colour')]),
              local(uninitilised)).

sparkCode(
procedure,
name(['PolishFlag', 'Partition_Section']),
annotation(...),
parameter([name(['PolishFlag', 'Partition_Section', 'Flag'])),
declare([name(['PolishFlag', 'Partition_Section', 'I']),
         name(['PolishFlag', 'Partition_Section', 'J']),
         name(['PolishFlag', 'Partition_Section', 'T'])]),
code([becomes(name(['PolishFlag', 'Partition_Section', 'I']), integer(0)),
      becomes(name(['PolishFlag', 'Partition_Section', 'J']), integer(11)),
      loop([ifexit(name(['PolishFlag', 'Partition_Section', 'I'])=
                   name(['PolishFlag', 'Partition_Section', 'J']),
                   []),
            if(name(paramater(reference([simple('PolishFlag'), simple('Partition_Section'), simple('Flag')]),
               argument([name(reference([simple('PolishFlag'), simple('Partition_Section'), simple('I')]))])))=
               name(reference([simple('PolishFlag'), simple('Red')])),
               [assignment(name(reference([simple('PolishFlag'), simple('Partition_Section'), simple('I')])),
                          name(reference([simple('PolishFlag'), simple('Partition_Section'), simple('I')]))+
                          integer(1))],
               [assignment(name(reference([simple('PolishFlag'), simple('Partition_Section'), simple('J')])),
                          name(reference([simple('PolishFlag'), simple('Partition_Section'), simple('J')]))+
                          integer(1)),
                assignment(name(reference([simple('PolishFlag'), simple('Partition_Section'), simple('T')])),
                          name(paramater(reference([simple('PolishFlag'), simple('Partition_Section'), simple('Flag')]),
                               argument([name(reference([simple('PolishFlag'), simple('Partition_Section'), simple('I')]))])))),
                assignment(name(reference([simple('PolishFlag'), simple('Partition_Section'), simple('T')])),
                          name(paramater(reference([simple('PolishFlag'), simple('Partition_Section'), simple('Flag')]),
                               argument([name(reference([simple('PolishFlag'), simple('Partition_Section'), simple('I')]))])))),
                assignment(name(paramater(reference([simple('PolishFlag'), simple('Partition_Section'), simple('Flag')]),
                               argument([name(reference([simple('PolishFlag'), simple('Partition_Section'), simple('I')]))]))),
                          name(paramater(reference([simple('PolishFlag'), simple('Partition_Section'), simple('Flag')]),
                               argument([name(reference([simple('PolishFlag'), simple('Partition_Section'), simple('J')]))])))),
                assignment(name(paramater(reference([simple('PolishFlag'), simple('Partition_Section'), simple('Flag')]),
                               argument([name(reference([simple('PolishFlag'), simple('Partition_Section'), simple('J')]))])))
                          name(reference([simple('PolishFlag'), simple('Partition_Section'), simple('T')])))])])])).
```

This output is fine. But the implementation requires a fair amount of grammar processing. In particular the full scope rules of SPARK would be needed. Is there something in the EXAMINER, or just SPARK documentation, that might aid this process?

4 Task 3: More Phil Examples

The Phil Thornley examples are challenging and realistic while not revealing any details of the code that inspired them. This involves extracting the essence of difficult high integrity software proofs. These more general template problems directly aid the discovery of heuristics. Further, as most high integrity software is also very secure the specific details of a system can not be considered.

Although the target for these examples will be exception freedom, it would be interesting to consider some partial correctness examples also.

The aim - construct a few (10 at most) examples that reflect real and recurring problems seen in high integrity software. Note that although the emphasis is on targeting common problems, this is under the assumption that problems are solved with the same amount of effort. Some less frequent but hard problems would also be interesting. It would be appropriate to try and classify the examples with this in mind. It is hoped that some of the examples for task 3 will naturally arise through the consideration of tasks 1 and 2.