Bill J. Ellis
October 29, 2004

Praxis trip III: Evaluation Problems and Solutions

1 Introduction

A key task in the empirical evaluation of NuSPADE is the rearranging of high integrity software such that it can be analysed. This rearranging is required due to the limitations of the NuSPADE system. Essentially, NuSPADE can tackle a subset of SPARK typically not seen directly in real high integrity software systems.

2 The isolation problem

2.1 Problem description

There is no need to consider an entire high integrity software system, as it is only the unsolved aspects that are interesting. This typically accounts for 5% of verification constiions (VCs). There is a strong mapping between VCs and code size thus typically about 5% of the code is intresting. However, while VCs become independent once generated, source code is more complicated. A troublesome subprogram is likely to exist within a large context, taking type from package A, calling some subprogram in package B, etc. Although only 5% of subprograms might be interesting, they are interwoven within a much larger context making then difficult to analyse in isolation.

2.2 Proposed solutions

Different approaches are presented for solving the isolation problem:

- **Ignore the problem -** It is possible to isolate the source code by retrieving all of the dependencies leading from the central, targeted, subprogram. The result is a web of source code, dense at the targeted subprogram with branches sprawling off into other called packages. Extracting this web can be straight forward if the target subprogram is particularly isolated from the source code. Unfortunately, this is somewhat rare. There is often a couple of called subprograms, leading to a large trace of further called subprograms and related type information. Arguably all of this information is relevant to the targeted subprogram. Worse, the traces of information can include code not handled by NuSPADE at all, such as interfacing with hardware.
- **Generalise to the core functionality -** One solution to the isolation problem is to prune the trace of subprogram calls and type to the fundamental operations being seen. However, this begins to weaken the value of the empirical analysis. The targeted subprogram may become diluted to the point of no linger being relevant. Although such generalised examples are quite useful for development and are much more straight forward to extract they are unsuitable for evaluation due to their distance from the original code.

  As a ground principle it is felt that the subprogram under analysis should only be amended to transform the code into the NuSPADE subset. These transformations are slight, restricted to making enums numbers, etc. This ensures that the isolated subprograms retain the exact meaning (and nearly exact syntax) of the high integrity code. The simplifications to make the isolation possible will take place at the level of the called subprograms and type.
- **Omit what is not needed -** Rather than generalising the trace of subprogram calls and type they can instead be curtailed. Called subprograms (and especially subprograms called from these subprograms) have little bearing on the analysis of the targeted subprogram. It is reasonable to present such uninteresting subprograms as empty stubs. These will return values in their type: any deeper analysis is not expected. It is also possible to curtail some type information. However, where this can reasonably be done without diluting the targeted subprogram, the simplifications available are so slight (strip out intermediate types) as to not be worthwhile.
- **Use multiple packages -** NuSPADE likes to analyse single subprograms in single packages. However, it is quite laborious expressing examples in this form in SPARK. Further, this is not necessary. Clean multiple packages can be considered, bundling type and state accordingly. The extended fdl adopted by NuSPADE will allow this more distributed information to be described locally.

*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).

- **Distributed state and subprograms -** NuSPADE does support local variables and hence supports global variables. The problem in the past has been the parsing and comprehension of the global annotations. However, this information can now be provided through the extended fdl. NuSPADE is weak at relating called functions to their corresponding location in the code. However, this is fairly minor. Subprograms typically only reveal their type. In the rare cases where called subprogram details might constrain the situation they can be suitability positioned for access by NuSPADE.

- **Isolation becomes extended fdl -** The consequence of these observations is that the items required to do analysis within NuSPADE are:

  - **Target code -** The source code of the targeted subprogram.
  - **The extended fdl -** The extended fdl provides all type information and declares all entities seen within the targeted subprogram. The extended fdl effectively replaces several other package dependencies.
  - **Optionally, called code -** Called subprograms may be targeted themselves if it is felt that they would reveal information beyond type.

- **Extended fdl is extended -** The extended fdl, short and manageable for small examples, becomes rather long and protracted for real examples. The problem is that the extended fdl describes the full type information for the targeted subprogram. Type information is never abstracted away - if a function uses a variable that is a record of an array of records, etc. every one of these types needs to be explained. Unsurprisingly this becomes restrictively long in most real examples.

- **Use regular fdl -** The extended fdl exists to convey the context of an analysed subprogram. The SPARK tools already do this, to some extent. The extended feature provides access to the modes of variables, subtypes, constants, details of called functions, translations for procedures. Limited progress can be made with a subset of this information. However, with weaker information, weaker results will be found. This is not an effective means for analysis, thus the extended fdl is required.

- **Generalised examples -** Given the above, it is not entirely practical to perform full analysis for all considered examples. It is, however, reasonable to consider a representative collection of examples instead.

2.3 Summary

Ideally the SPARK code would be presented as several small legal SPARK programs. These could then have extended fdl added to support analysis. However, due to the interconnected nature of high integrity software this is not practical. Fortunately, the extended fdl provides a convenient means to describe the context of targeted subprograms. Alas, formulating such extended fdl can be time consuming (and error prone) for larger examples. Thus instead a small representative collection of examples are analysed.

3 The evaluation problem

3.1 Problem description

Evaluation can produce promising results on the examples being considered. However, in the smaller system T, the considered examples for program analysis amount to 2% of the overall system! The other 98% is straight forward enough to not require any user invariants for exception freedom proofs (some preconditions / postconditions may be required). Although system S has a greater proportion of potentially considered examples, these will probably be around 5%.

3.2 Proposed solutions

Different approaches are presented for solving the evaluation problem:

- **Do not count subprograms -** Relating to subprograms is code analysis specific. Further, subprograms take several sizes. Typically we target the more complex subprograms, which are far outnumbered (made statistically less relevant!) by the presence of several less complex subprograms. Thus this is not especially fair to our analysis. Further, this presents problems from a general reasoning perspective. The proof planning aspect may prove VCs not discharged by the SIMPLIFIER, irrespective of the program analysis.

- **Do not relate to full system -** Each considered example will reveal statistically relevant and promising results. If only describing the targeted considered examples, the results will be quite favourable.

  Unfortunately, this is rather cheating. It is important to establish how common these patterns are in practise.

- **Compare inside the 5% -** The considered examples will form a subset of the 5% of the 100% of excpetion freedom VCs. Relating to the 5% (which is what we actually intend to target) is fair and will, of course, provide more flattering numbers. These numbers can be gathered from Praxis internal assessments of systems S and T. Each considered example will replace a segment of these base numbers with new numbers.

  This is also more sensible from a general reasoning perspective. The proof planning aspect may prove VCs not discharged by the SIMPLIFIER. A slight problem here is distinguishing between not proved and unprovable. This is difficult to establish without substantial effort, and so must, for practical reasons, be avoided.

3.3 Summary

For each targeted subprogram gather:

- **Location -** The owning system and owning file.
- **No annotation -** Find number of VCs before and after simplification.
- **With annotation -** Find number of VCs before and after simplification.
- **With extra proof planning -** Find number of VCs before and after the additional proof effort.

Note that the third phase is generally applicable. Any additional proofs found in non-targeted subprograms will be relevant here. Note that deriving false might also be reasonably categorised as finding a proof - it has successfully proved that the VC is not provable.

4 The weak results problem

4.1 Problem description

The aim has been to focus on the program analysis side of NuSPADE and how it benefits proof of exception freedom for SPARK VCs. Unfortunately, if a VC requires additional annotations to make it provable it is particularly likely that the SIMPLIFIER will be less prone to complete the proofs, especially those introducing annotations (loop invariant VCs). Thus while the program analysis might change an unprovable VC into two provable VCs, none of these need be proved. In this case, looking an the SIMPLIFIER results alone indicates the program analysis has doubled the number of undischarged VCs! It is difficult to present such statistics favourably.

4.2 Proposed solutions

Different approaches are presented for solving the evaluation problem:

- **Show before unprovable / after provable -** Proving that the VCs before program analysis are false would successfully indicate that some changes are required. However, without showing that the program analysis VCs are true (not being able to show they are not provable is not sufficient) this still does not provide any evidence.

  Note that this is separate to being able to automatically show that these VCs are true. It is sufficient to show that before they were unprovable and now they are provable. However, in practise, manually demonstrating the correctness of several VCs would be restrictively time consuming.

  Show not proved before / proved after - The emphasis has always been on automated proof. Discussing provable VCs is not particularly relevant if they can not be proved automatically. Showing that VCs are unprovable is false is good for debugging, however, not especially valuable for proof. That it is not solved automatically before and is afterwards is relevant where focusing on automation.

  Essentially this requires involving our proof planner tools. VCs may be exported from Praxis to conduct this evaluation.

4.3 Summary

Gather VCs to describe before annotations and after annotations. Hopefully these can be exported for further analysis. Showing that previously false (not proved) VCs become true will give some positive (rather than negative) results. Relating these to the numbers of unproved VCs (rather than the total number of VCs) should give some encouraging (though not explosive) results.