Bill J. Ellis
April 19, 2004

Automatically Enriching SPARK Specifications

1 Introduction

Currently we proof plan the SPARK VCs. The proof plan may fail, with a critic suggesting that a stronger invariant is required. Our program analysis may find such a stronger invariant. However, the user must then intervene to add this invariant back into the code.

In general, we may want to strengthen any specification information, such as assertions outside loops, preconditions and postconditions. It would be ideal if this strengthening process could be achieved without user interaction.

2 The Problem

2.1 Specification in SPARK code

Specification information is embedded inside the SPARK source. The SPARK code is just a long sequence of characters. Relating the specification information to a particular loop, subprogram and package requires deep knowledge of the SPARK code. The SPARK code needs to be parsed.

2.2 But we already parse SPARK?

For our program analysis we already parse the SPARK code. However, this gradually turns SPARK into a flowgraph and associated data structures. This could be enriched with the new specification information and translated back to SPARK. Although this might be technically possible it would be quite difficult.

2.3 We could parse the SPARK again?

The problem is retrieving all of the extra information about the SPARK code (such as indentation, comments, ordering, etc.) that is not really interesting for program analysis. Besides, parsing all of SPARK is not very trivial.

2.4 Find a lightweight text processing solution

The SPARK assertions all exist on a line of their own. The program analysis may want to strengthen an existing specification or add a new specification.

Thus the problem is, given a line number L, a property P and some spark code S, enrich or create a specification at line L in code S with property P. It may be possible to also provide information N, indicating if the specification is new or not. This more targeted problem can be dealt with in a decent text processing language (such as Awk or Perl).

2.5 The default invariant caveat

In some cases the EXAMINER adds default invariants. These invariants exist, and are discussed in the generated VCs, but are hidden to the program analysis.

Technically, it is possible for the program analysis to determine that a default invariant must have been inserted (as no supplied invariant was found). It should be possible to determine where the default invariant would have been inserted, and progress accordingly. This is somewhat fiddly.

Default invariants are confusing. It is possible to eliminate default invariants, by explicitly cutting every loop. This will involve adding an invariant for every loop that does not already have an invariant.

*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).

2.6 Finding the default invariants

If the program analysis avoids finding the location of the default invariants then they must be found by other means. When using the Examiner to generate the VCs it can be asked to generate listing files:

```
spark -e -l=ls_ <Files>
```

These listing files can be lightly parsed by a text processing system (Awk, Perl, possibly Sed or even Grep) to determine the point of default invariants.

2.7 But the Examiner lies?!

The Examiner does not report the line that the default invariant would be inserted, rather the line where it decided that one should be added. When the Examiner reports:

```
  35  I

---        Warning           :402: Default assertion planted to cut loop.
```

The invariant should actually be placed on a separate line just following the next word 'loop' found at at some point after the word 'I' starting at line 35. Again, this is text processing (Awk or Perl).

2.8 And non-invariants

These are quite complicated. Preconditions and postconditions are added at the specification of a subprogram. Or its body for nested or refined subprograms. Preconditions and postconditions are typically omitted, so there is no hook to insert the annotation. However, it would be untidy to add explicit preconditions and postconditions. Further, it would be difficult to add explicit preconditions and postconditions without a fairly deep understanding of the code.

2.9 The whole problem

To do the whole problem will require looking at the whole SPARK project. Looking at individual files is not sufficient due to refinement. Would need a system that analysis the SPARK project and generates a collection of tags, pointing out exactly were in the code different properties should be added. This might have the same kind of behaviour as something like `exuberant-ctags`. Then the program analysis could exploit these tags to control an automatic specification strengthening program.

This might be feasible. However, issues remain in adding specification properties that merely cut the code. The tag system would need to be strongly coupled with the program analysis. *This is a tricky problem!*

2.10 The little problem

Only consider invariants. Need to add explicit invariants were default invariants are used, have a light program for enriching an invariant at a line number and need to (somehow) make sure the program analysis is aware of each invariants line number.

2.11 Problem in locating line numbers

There are complexities in making the program analysis aware of which line number each invariant is associated with. There seems to be some some good tool support for this behaviour (`pt-support-bundle-1.0`) where working with ATerms and SDF2, the parsing level of our program analysis. Unfortunately, proper integration of these tools would likely impose quite a few difficulties. In particular, we do not actually parse the SPARK code, instead we parse the result of the SPARK tokeniser. This means that any line numbers will refer to the token stream. It would be possible to relate the token stream to the SPARK source or re-implement the tokeniser directly in SDF. Either solution would be valid but not straight forward.

An alternative is to enrich invariants with line number information. Write a program which turns invariants of the form:

```
--# assert true and true;
```

Into:

```
--# assert lineNumber(320) and (true and true);
```

This line number can then be extracted directly from the invariant during program analysis. This is not very elegant, but is an easy way of making the program analysis aware of invariant line numbers.