Bill J. Ellis & Tommy Ingulfsen
February 19, 2003

Notes on Parsing SPARK for Property Extraction

Contents

*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).

1 Introduction

The SPARK code contains information that is not present or is difficult to access in generated verification conditions (VCs). It is advantageous to explore the proof of VCs alongside the information in the code. Fundamentally, this requires parsing the SPARK code. As part of the NuSPADE project two different approaches to using the SPARK code are being considered:

- **Automatic generation of algorithmic properties (AUTOGAP) -** The AUTOGAP behaves in isolation to NuSPADE. It uses heuristics to analyse a SPARK program and finds algorithmic properties that might be useful as an aid in program proof.
- **Constant path analysis (CPA) -** If proof planning fails in NuSPADE, coloured rippling can often be used to identify the point of failure. CPA may be started with the details of this point of failure. It will analysis the program, targeting extreme points using the constant path approach. The point of failure can then be used to aid in the generalisation of the extreme points and the discovery of a suitable program property.

Both of these approaches will require access to a SPARK parser. The intention is to exploit this shared problem by implementing a single parser rather than two. This note explores some of issues that might arise in parsing SPARK. It decomposes the problem into two tightly interacting systems. Some strong hints at implementation details are provided for the low level parser component.

1.1 Create new light-weight SPARK parser (LW-SPARK-PARSER)

The EXAMINER already parses the complete SPARK language. It may be possible to augment or transform the EXAMINER to provide the functionality required. However, the EXAMINER is a large system with many additional features beyond parsing. It is expected that it would be much more difficult to modify the EXAMINER rather than build a new parser from scratch.

Thus the intention is to build a brand new light-weight SPARK parser (LW-SPARK-PARSER). This will perform the single task of reading in SPARK code and outputting a structure describing this code.

1.2 Help from the EXAMINER

The full SPARK syntax is presented in [1] Appendix A1.1 and A1.2. Further, the EXAMINER grammar file `SPARK.LLA` has been provided by Peter Amey. Together, these two sources of information provide a strong foundation upon which LW-SPARK-PARSER can be built.

2 Requirements of LW-SPARK-PARSER

2.1 Work within SPARK subset

In general, all of SPARK has the potential to be useful for analysis. However, in practise, their are key parts of the SPARK language that reveal the most useful information. These key parts can be targeted rather than the entire SPARK language. It is difficult to preemptively pin down the key parts of SPARK as the choice is based on heuristics. Only through experimentation will the key parts become apparent.

Currently the following aspects of SPARK are identified as being important. These are broken down into functional code and information code with different priority levels. Early systems would need to implement everything classified as high priority to be useful. Later systems may implement the lower priority features to provide a larger coverage of SPARK.

2.1.1 Functional code

The functional code actually performs operations. For a subprogram the functional code is taken to be the body of the subprogram. Note that this definition means that the functional code for a subprogram will always exist in a single SPARK code file.

High priority

- **If statement -** If statements are the lowest level unit required to allow the encoding of all conditional statements.
- **Base loops -** Base loops are the lowest level unit required to allow the encoding of all loops.
- **Assignments -** Assignments are necessary to change the state.
- **Expressions -** Assignments must include expressions. Expressions can be quite sophisticated in SPARK. However, only the fundamental system functions, such as addition (+) subtraction (-) modulus (mod), should be required to formulate interesting examples.

Lower priority

- **Subprogram calls -** Subprogram calls are very common in SPARK code. Not dealing with subprogram calls would exclude a large amount of SPARK code from analysis.
- **Case statements -** Case statements can be expressed in terms of if statements. However, case statements are a useful construct that should be eventually supported.
- **Complicated expressions -** It is expected that a large amount of SPARK expressions will eventually want to be parsed. More important aspects include the tick attributes and user declared functions.

2.1.2 Information code

Information code provides various details that do not directly describe a functional operation. Unlike functional code this is not always attached to a single subprogram. The information code for a subprogram may exist in many SPARK code files.

High priority

- **Type declarations -** Type information directly provides constraints on variables. This is very valuable information with direct applications for advancing exception freedom proofs.
- **Subprogram calling -** The details of a subprogram call reveals the access mode and type of every parameter. The access mode constrains the variables which can change, reducing search space in many expected analyses. The type name is required to link to the type declarations.
- **Global annotation -** The global annotation is an extension to subprogram call details and will be necessary for the reasons given above.
- **Local variable declarations -** The type names of local variables are required to link to the type declarations.

- **Invariant location in loop -** The positioning of an invariant in a loop has a significant effect on the properties to be generated. For example, consider a loop with the statement `I:=I+1;` if it is invariant that `I=P` just before this statement then it is invariant that `(I-1)=P` just after this statement. Thus it is essential to know the location of the invariant with respect to the surrounding code to generate accurate properties.

Lower priority

- **Preconditions, postconditions and invariants -** The proof annotations are effectively repeated in the VCs. However, from a practical perspective, it is helpful to know what these are to allow comparisons with any generated properties to avoid adding the same information again.

- **Initialises -** Where global own variables are initialised this knowledge and the initial value provides additional information in any subprogram which uses the global variable.

2.2 Cope with all of SPARK

The finished system should be able to operate on the entire SPARK language, or at least a very large subset. This does not mean that structures will be generated for all of SPARK. Rather it requires having the ability to filter out the SPARK not considered while still collecting the structures desired. Two options are proposed:

- **Prune SPARK then collect the key parts -** A higher level program filters out all of the SPARK that is not considered presenting the parser with the remainder. For example, it would be possible to use AWK to find the text for subprograms rather than actually parse the enclosing package details.

- **Recognise SPARK but collect only the key parts -** The parser can recognise a large subset of SPARK but only generate structures for the key parts.

The first approach should be more straight forward to implement initially, but will become increasingly difficult to pursue as the amount of SPARK being parsed increases and the granularity required of the pruning becomes tighter. The latter approach scales up much more favourably as the entire parser will be captured in precise grammar rules alone. The additional initial implementation overhead is not expected to be large as the grammar rule construction this requires is already provided through `SPARK.LLA`[1].

2.3 Code to structure and structure to code?

It would be a minor niggle if, once properties are generated based on the output of LW-SPARK-PARSER, there was no method for automatically adding these properties back in the source. It might be thought that LW-SPARK-PARSER should collect the line number of key proof annotations to facilitate this feature. Although a cosmetic detail it may be worthwhile including a line counter in the parser to support this feature.

[1] The intention is to implement LW-SPARK-PARSER without any pruning step. Early prototypes suggests this is quite feasible and the most straight forward solution.

3 The SPARK subprogram spider (SUBPROG-SPIDER)

As subprograms are the main unit in SPARK it is desirable to build the analysis around these. Consider the case of requesting details on a subprogram. The functional code can be directly extracted from a single file and stored in a structure. However, other details may not be found in this file:

- **Other subprograms -** The subprogram considered may call subprograms declared in other files. The details of these subprograms may be desired up to some calling depth.
- **Type -** Type information is especially valuable for the property analysis being considered. However, any given subprogram may exploit the types of any number of packages in any number of separate files. Further, types are often defined in terms of other types, again potentiality declared in separate files in separate packages.

In almost every realistic case the details desired of a subprogram will exist in the nested structures of code held in many SPARK code files. It would be unsuitable and quite difficult to incorporate this heuristic based wide spread property extraction inside LW-SPARK-PARSER.

Thus another system is proposed, called the SPARK subprogram spider[2] (SUBPROG-SPIDER). This system is responsible for calling LW-SPARK-PARSER multiple times and filtering its output based on the subprogram that details are requested on. A high level view of the behaviour of SUBPROG-SPIDER and its interactions with LW-SPARK-PARSER is presented in Figure 1.

The result of this extra system is that the actual parsing process is kept quite straight forward. LW-SPARK-PARSER will take as input a SPARK file generating an bare structure that describes the files contents.

3.1 Output from SUBPROG-SPIDER

The output from SUBPROG-SPIDER will describe the details of a root subprogram. This root subprogram may call other subprograms which may also have their details added. Each subprogram in this tree of subprograms will have various information attached, including functional code, its interface (the modes of variables), all local variables and annotations. Some details will be global to the tree of subprograms, such as type information or initialisation of global variables.

The exact details of the output remain to be formulated. The most natural form would be a number of many small searchable Prolog facts, similar to the internal data structures of NuSPADE.

[2]Spiders are programs that crawl over the Internet collecting lots of information for search engines. Small subsets of this information are presented depending on a users search query. SUBPROG-SPIDER crawls over the many SPARK code files in a project, collecting details on a user targeted subprogram.

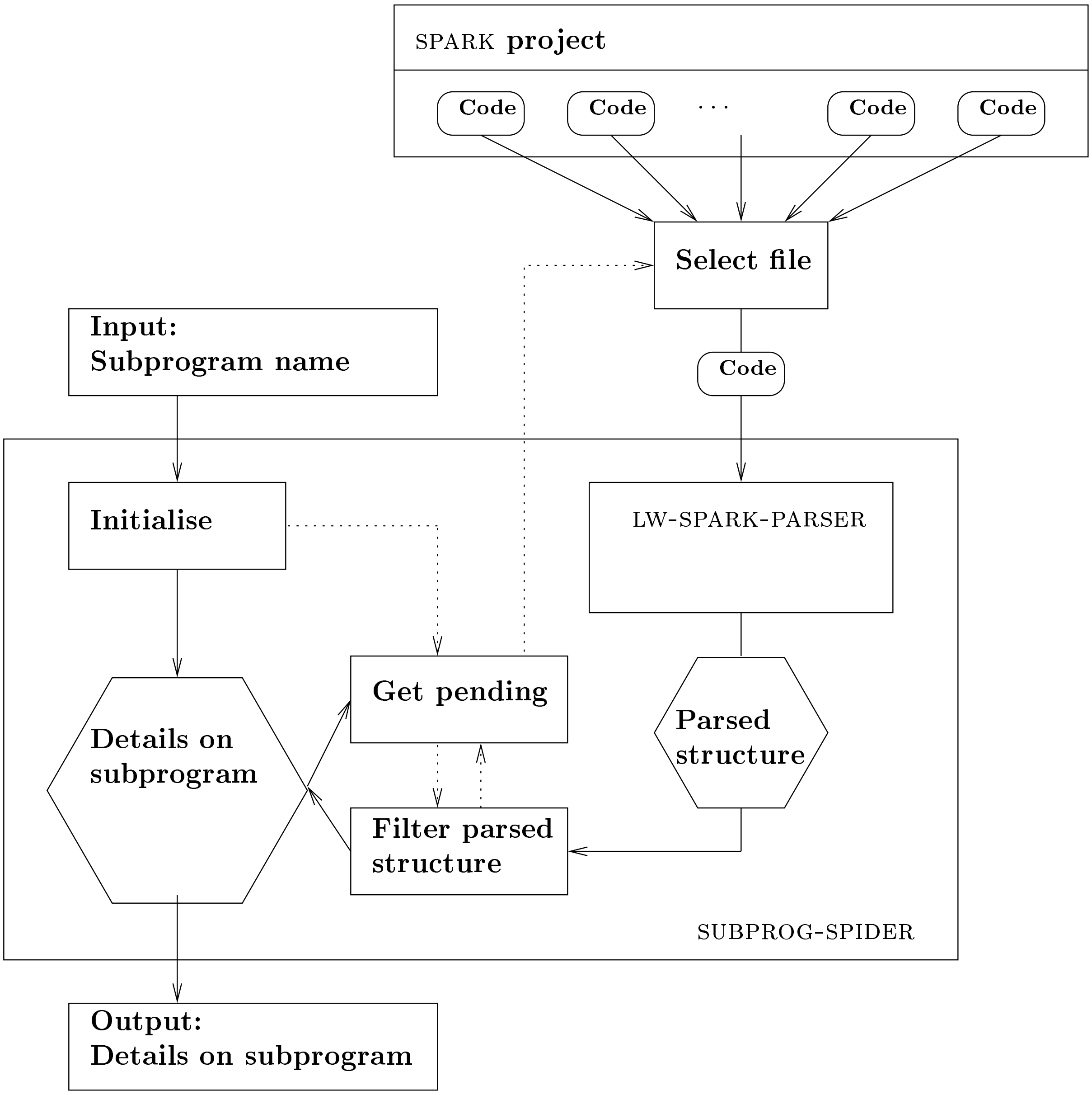

Figure 1: High level view of SUBPROG-SPIDER interacting with LW-SPARK-PARSER

3.2 Some issues for SUBPROG-SPIDER

Some issues previously considered for LW-SPARK-PARSER are now in the scope of SUBPROG-SPIDER.

3.2.1 Representation language

The structures returned by LW-SPARK-PARSER will describe the SPARK program in an accessible form. However, these need not be the same structures returned from SUBPROG-SPIDER.

The systems AUTOGAP and CPA will reason with the structures outputted by SUBPROG-SPIDER. The most appropriate form will be closer to formulae rather than the raw structures of LW-SPARK-PARSER. A reasonable solution would be, where possible, to make the structures returned by SUBPROG-SPIDER mirror the SPADE-PC Functional Description Language (FDL). However, FDL will be suitable for expressions only. The remaining structure can be a refined version of that returned from LW-SPARK-PARSER. For example, a possible[3] mapping from a SPARK assignment to LW-SPARK-PARSER structure and finally the SUBPROG-SPIDER structure is below:

<table>
<tr><td>Code</td><td><pre>I:=Arr(P);</pre></td></tr>
<tr><td>LW-SPARK-PARSER</td><td><pre>example(assignment_statement(
 name(
  simple_name(i)),
 expression(
  relation(
   simple_expression(
    simple_expression_opt(
     term(
      factor(
       primary(
        name(
         name(
          simple_name(arr)),
         name_argument_list(
          positional_argument_association(
           expression(
            relation(
             simple_expression(
              simple_expression_opt(
               term(
                factor(
                 primary(
                  name(
                   simple_name(p)))))))))))))))))))))).</pre></td></tr>
<tr><td>SUBPROG-SPIDER</td><td><pre>assignment_statement(i,element(arr,p))</pre></td></tr>
</table>

[3]This presentation is realistic, based on the representation given in §5 and the grammar in `SPARK.LLA`. The transformation from the LW-SPARK-PARSER input to the SUBPROG-SPIDER output was performed automatically by an early prototype of SUBPROG-SPIDER.

3.2.2 Mapping variables in code to variables in VCs

The variable names used in SPARK code are often not the same as their corresponding names in VCs. For example:

Code	`For I in ... loop ... end loop;`
Variables in VC	`I becomes loop__1__i`
Reason	In Ada for loops the variable iterator is local to the body of the for loop. SPARK avoids potential name clashes with other variables by renaming these variables in the VCs.

Code	`Double(X);` `Double(X);`
Variables in VC	`X becomes x__1` `X becomes x__2`
Reason	SPARK models procedures as a series of assignments to new variables. If a variable is assigned to at least once via a procedure it will have a an extra counter subscript in the VCs. This counter will increase each time the variable is assigned to via a procedure.

Some mechanism will need to be implemented that allows variables in SPARK code to be related to their corresponding variables in VCs and vice-versa.

3.2.3 Code transformations

In CS-note 39 it is shown how it is convenient to reason with SPARK programs expressed in a normal form, breaking down loops and conditional statements into their elementary components. These transformation will also be performed by SUBPROG-SPIDER.

3.2.4 From LW-SPARK-PARSER to SUBPROG-SPIDER in practise

The conversion process from LW-SPARK-PARSER structures into SUBPROG-SPIDER structures will not be trivial. In `SPARK.LLA` their are 336 grammar rules. Quite a large number of these are expected to have corresponding structures in LW-SPARK-PARSER.

It might be thought that LW-SPARK-PARSER should directly return cleaner structures that are closer to the desired output of SUBPROG-SPIDER. However, this would not remove the problem, merely make it part of the operation of LW-SPARK-PARSER. Rather than taking the grammar from `SPARK.LLA`, LW-SPARK-PARSER would have custom grammar rules describing the final structures desired, including the treatment of expressions as FDL and reformulating high level structures into their elementary parts. It is considered more straight forward to write a parser for regular SPARK and implement the more heuristic based post processing of these separately[4].

Thus SUBPROG-SPIDER will include a a post processing facility capable of building polished structures from the low level structures of LW-SPARK-PARSER. This post processing facility might be thought of[5] as applying a large number of rewrite rules to the structure returned from LW-SPARK-PARSER.

[4]This is especially true if writing LW-SPARK-PARSER in a language other than Prolog.

[5]This used to say: "Or even implemented?". This is no longer a question, as a SUBPROG-SPIDER prototype has been successfully implemented using this very approach.

3.3 Isolated SPARK is ambiguous

Isolated SPARK syntax, as might be found within a subprogram, is not sufficient to determine the meaning of the syntax. Consider the assignment:

```
A:=B(C);
```

Depending on the details of `A`, `B` and `C` this could mean any one of:

- **Array access -** Assigning to variable `A` the contents of array `B` at index `C`.
- **Function call -** Assigning to variable `A` the result of calling the function `B` with parameter `C`.
- **Casting -** Assigning to variable `A` the value of variable `C` casted to fit in type `B`.

Another source of ambiguity exists where distinguishing between variables and procedure calls with zero arguments. The key to distinguishing between the different semantic operations is the declared types of each name. However, this type information will not necessarily be available within the syntax of a subprogram.

This is not a large issue, as SUBPROG-SPIDER will collect type information to aid in property analysis. The same information can be used within SUBPROG-SPIDER to extract the semantically correct structures from the LW-SPARK-PARSER parsed syntax[6].

Note that the SPARK syntax is not ambiguous! The issue identified here is that SPARK syntax may be ambiguous if only considering the syntax encountered within a subprogram. To correctly find the meaning of every SPARK subprogram it is necessary to have the capability to search all of the SPARK source within the same scope.

Note that these isolated ambiguities are probably a result of sensible decisions by the ADA designers. Accessing elements of an array or performing casting can be thought of as a function call, so there is a good argument that they should all look the same[7].

[6] This issue was discovered long after the initial plans for SUBPROG-SPIDER. However, this is a good example of the more involved parsing that originally inspired the twin systems of LW-SPARK-PARSER and SUBPROG-SPIDER.

[7] Although, it can be frustrating having to refer to types to work out what `A:=B(C);` actually means. Arrays could have been indexed by square brackets (`A:=B[C];`) and the compiler could check that `B` is actually an array...

4 Rapid prototyping of LW-SPARK-PARSER

Two alternative approaches to an implementation of LW-SPARK-PARSER were considered. Some rapid prototyping was undertaken to identify the key features of these approaches.

4.1 Prolog Definite Clause Grammar (DCG)

Definite Clause Grammar (DCG) is a means for parsing top-down non left recursive grammars in Prolog.

Advantages

- Prolog has strong term construction features, allowing the straight forward construction of internal structures.

Disadvantages

- The SPARK grammar presented in `SPARK.LLA` is left recursive in a few cases. Some minor tweaks are required to parse this with DCG.
- The SPARK grammar contains a few rules that can apply zero or more times. These also need to be reformulated in DCG.
- The DCG becomes quite large as each grammar rule translates into several lines of code.
- The DCG describes the entire parser. Their is no separate easily accessible file like `SPARK.LLA`.

4.2 Flex and Yacc

LEX (FLEX on Linux) automatically generates a C tokeniser from a concise description of tokens using regular expressions. YACC automatically generates a C parser from a concise description of grammar rules and associated action commands. Combined together FLEX and YACC can be used to implement a parser.

Advantages

- The concise descriptions of FLEX and YACC are readable and accessible documents.
- The SPARK grammar presented in `SPARK.LLA` is naturally transferable to FLEX and YACC.

Disadvantages

- Building the internal parse structures will require some C code. C is not well suited for term manipulation.

4.3 Results

It would seem possible to implement LW-SPARK-PARSER by following either approach. However it is now quite apparent that FLEX and YACC is most appropriate.

5 Output representation for LW-SPARK-PARSER

The LW-SPARK-PARSER will output some structure that describes SPARK code in a manner suitable for analysis by SUBPROG-SPIDER. This structure will be quite straight forward, encoding little beyond the structure of the code being parsed. This minimal approach is advantageous, as any more significant structural processing in LW-SPARK-PARSER will require some involved C programming.

This minimal structure will subsequently be processed by SUBPROG-SPIDER, which will likely be implemented in Prolog. Moving the more sophisticated processing into SUBPROG-SPIDER allows the parser to be kept as uncomplicated as possible.

Thus the structure to be built by LW-SPARK-PARSER should be quite uncomplicated and directly understandable by Prolog.

5.1 Example structures

The output required is a single Prolog term describing the structure of the program. The formulation of this term is often quite intuitive given the grammar rules. However, note that their remains a degree of pruning and formatting for the intended application. Generally terminal symbols can be dropped. Some examples are presented below. Note that the names starting with upper case letters denote variables that will eventually be instantiated by Prolog terms of that name.

Grammar	`function_specification :`
	`      RWfunction designator formal_part RWreturn type_mark`
	`    | RWfunction designator RWreturn type_mark ;`

Structure	`function_specification(Designator, Formal_Part, Type_Mark).` `function_specification(Designator, Type_Mark).`

Grammar	`designator :` `      identifier ;`
Structure	`designator(Concrete_Identifier).`

Grammar	`formal_part :` `      left_paren formal_part_rep right_paren ;`
Structure	`formal_part(Formal_Part_Rep).`

Grammar	`formal_part_rep :`
	`      formal_part_rep semicolon parameter_specification`
	`    | parameter_specification ;`

Structure	`formal_part_rep(Formal_Part_Rep, Parameter_Specification).` `formal_part_rep(Parameter_Specification).`

6 Note on building structures in C

The main problem with FLEX and YACC is building the parsed structures. It is helpful to explore in more concrete terms the structure building facilities required.

6.1 The structures

Each **Structure** is comprised of a single **Name** and a number of **Parts** each of which may contain a string or another **Structure**. This description can be translated into a recursive data structure:

- **Structure -**
  - **Name -** Structure name.
  - **Parts 1..N -**
    - **Structure -** A nested structure.
    - **OR**
    - **String -** A string.

6.2 The structures as nodes

Each structure is a node in a tree. The leafs of this tree will be structures that contain only string in parts. The tree can be converted into the output Prolog term by recursively traversing this tree displaying deepest nested structures first.

The problem is building this tree inside C around FLEX and YACC. Using the above template some suitable C data structures can be declared:

```
#include <stdlib.h>              /* for malloc, NULL, size_t */
#include <stdarg.h>              /* for va_ stuff */
#include <string.h>              /* for strcat et al */

struct component {
  char * tokenAsString;
  struct structure * nestedStructure;
};

struct structure {
  char * name;
  struct component ** allComponents;
};
```

The key function building the tree will take the form:

```
struct structure * node(char * name, struct component * c1, ...);
```

This can be integrated into the processing aspect of YACC rules. There will need to be some additional conversion of YACC of items into components. Further, other functions will be required to eventually display the tree once the top level grammar rule is matched.

Note that the `...` in `node` is legal C, allowing functions to have multiple arguments[8]. An alternative to using `...` is having N different versions of `node` for N parameters.

[8] A good example of this is here: http://cpptips.hyperformix.com/cpptips/varargs2

7 Conclusions

It is possible to implement a single parser that will accommodate the two systems AUTOGAP and CPA. It is reasonable to separate the process of parsing SPARK and the construction of accessible subprogram information into the two separate programs LW-SPARK-PARSER and SUBPROG-SPIDER. Although details on the outer SUBPROG-SPIDER system are sketchy, the requirements for LW-SPARK-PARSER are well defined with some fundamental implementation decisions considered.

References

[1] J. G. P. Barnes. *High Integrity Ada The SPARK Approach.* Addison-Wesley, 1997.