Andrew J. Cook
May 14, 2004

Onwards and Upwards

1 Introduction

The SPARK system produces verification conditions that, if shown to be true, will ensure that a program will never raise an exception at run time. Most of these conditions are trivial and can be easily removed by the SPADE simplifier however, some are more difficult and require the addition of loop invariants and the use of more complex proof systems. The NuSPADE/CLAMSPADE project uses program analysis and the CLAM proof planner to help prove these conjectures.

In order for the proof planner to plan the spade verification conditions it is useful for it to have access to the logical rules which the SPADE checker uses in user guided proof search. So a translator has been built on top of Bill Ellis' nuspade rule translator to translate these rules for CLAM.

2 Foundation

Bill's translator takes rule definition files from standard directories within a project and translates them into prolog files encoding all the information needed for each rule in a general form.

Bill separates the rules from their SPADE names, giving each a unique numerical id in his new rule system. He keeps the SPADE names separately so that they can be restored if necessary. He also explicitly adds the quantification implicit in the original rules and encodes conditional rules using a new type of implication operator.

The SPADE rule:

```
abs(1): abs(X)      may_be_replaced_by X if [ X>=0 ].
```

Becomes:

```
getPageRulesRule(ruleId(447), implicit_for_all('A','A'>=0-+->abs('A')='A')).
```

Much of the work done in interpreting this form of rule is done by Bill's own code for translating the rules for nuspade.

The predicate `ruleDefinitionToFoundation` takes the rule definition output by the translator and processes it into a more useful form. It strips the quantification, both implicit and explicit and places it in a list and splits the rule into conditions, kind, left hand side and right hand side.

My additions to Bill's code start with some modifications to the `ruleDefinitionToFoundation` predicate. Now the predicate will separate explicit quantifiers from implicit quantifiers, output the rule type in a form closer to that used in CLAM, and fetch the file name from which the rule came. I use another predicate to reunite the rule with it's SPADE name.

```
getPageRulesRule(ruleId(447), implicit_for_all('A','A'>=0-+->abs('A')='A')).
```

Becomes:

```
spade_rule(abs(1), 'A'>=0, abs('A'), 'A', equ, ['A'], [], builtin).
```

Where `spade_rule` takes has components for name, conditions, left hand side, right hand side, rule type, implicit quantification, explicit quantification, and origin file[1].

3 Upwards

The main section of code in `upwards.pl` consists of predicates for reading in Bill's files and outputting the resulting rules to the desired places.

The main predicates are `program_rules` for converting program specific rules and `builtin_rules` for converting the core rules. `program_rules` this comes in various arities.

*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] A recent change to `upwards.pl` changes `'SPECIAL.RUL'`[2] to `builtin`

[2] The file `'SPECIAL.RUL'` is not necessarily the file that the a core rule originates from but is simply the first file the predicate comes across when looking at the files used to generate the built in rule set.'SPECIAL.RUL' should be thought of as a place holder file name indicating that the rule is one of SPADE's built in rules.

3.1 program_rules/0

program_rules/0 takes Bill's output files(project.pl and subprogram.pl) from the current directory. It puts the resulting rules, each in a separate file, into separate directories called output_rul and output_rls respectively.

3.2 program_rules/1

program_rules/1 takes the name of the output files to use or user_out for outputting to the standard output stream. Like program_rules/0 it reads from files in the current directory but will output all the rules to the same file for each of the input files. The rules from project.pl will have _rul appended to the name of the file and subprogram.pl will have _rls appended. This is to prevent the rules from subprogram.pl from overwriting the rules from project.pl by outputting them to the same file. There is no such problem with builtin_rules/1 so it's output is put into the file named without anything appended.

3.3 program_rules/2

program_rules/2 Takes an input directory and an output directory and will output the rules to the same file for each of the input files project.pl and subprogram.pl. Like program_rules/1, program_rules/2 will append a suffix to the filenames to distinguish them.

3.4 program_rules/3

program_rules/3 Takes an input directory, an output directory, and a file suffix. The files project.pl and subprogram.pl in the input directory will be read and each rule output to it's own file in the destination directory. The file names will be the name of the rule family it's number within the family and the file suffix.

3.5 builtin_rules

The builtin_rules predicates, builtin_rules/0, builtin_rules/1, builtin_rules/2, and builtin_rules/3, are similar in operation to the corresponding predicates for program_rules. The exceptions to this are builtin_rules/1 and builtin_rules/2 which, unlike program_rules, do not append any suffix to the end of the output file before saving the rules.

4 Uplifting

The uplift script combines Bill's translation program and upwards.pl. Unfortunately Bill's translator requires it to be run from a directory containing specific programs and files, I have bundled these files together in a directory called getter. So uplift in turn must be executed from a directory which contains the getter directory.

uplift performs project specific and subprogram specific rule translation[3] using program_rules/3 and takes two arguments, a project directory, such as examples/filters, and a destination directory, such as nuspade/lib/srule/ using '' as the file extension argument.

Because Bill's translator looks for rules in specific places in the project directory the project directory must contain a directory for the subprograms (examples/filters/filter_sum) and a directory for user defined rules (examples/filters/rules). Also every file in the subprogram directory needs to be of the form *Name.Ext.* Aside from the necessary addition of the rules directory, because the subprogram directory is generated automatically by SPARK these conditions will be satisfied as long as nothing tampers with it afterwards.

Because uplift changes directory during it's operation if a relative path to the files if a relative path is used for the input directory then it is necessary to add ../ to the path before calling Bill's translator. However, if an absolute path is used it is necessary that this addition not be done. So the uplift script comes in two flavours, uplift and uplifter:

Script	Input path	Output path
uplift	Absolute	Either
uplifter	Relative	Either

[3] uplift does not do built in rule translation.

5 Conclusion

The `uplift` scripts will convert spade rules to a representation that CLAM can interpret. However, the spade rules need to be in specific places for the translator to find them and the scripts need to be run from their containing directory to work properly.