CS-Note* 13

Bill J. Ellis
May 9, 2002

Meeting Praxis Critical Systems

Contents

1 Introduction

The following is a collection of interesting points and observations taken from the meeting with Praxis Critical Systems. Some of these points are quite significant, and may potentially result in changes to the nature of NuSPADE. Note that some of these discussions are potentially more ϵ-baked than normal. Undoubtedly, the more interesting points will be considered in greater detail in future notes.

*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 Spark Observations

2.1 Still Support Pascal

The Spark tools are currently used to operate with a subset of Ada. However, some backwards compatibility remains with the original use of a subset of Pascal.

2.2 Repeating SHOLIS Proof

It was noted during the construction of the Corpus[1] that the SPADE scripts are very fragile. They rely on a particular numbering and ordering of hypothesis and conclusions. As further observed by Praxis they even rely on the number and kind of rules available in the SPADE database, as this affects the volume of search and hence nature of user interaction and hence the generated script. These problems are not only potential, they have been encountered in practise while repeating parts of the SHOLIS proof from previously recorded scripts. Note that Praxis call this repeating of proof effort a *regression proof*.

This issue is an indication of a larger problem. The commands of SPADE are very focused on the exact state of the current problem. It is helpful to compare these commands to the tactics seen in theorem provers such as HOL and CLAM (see CS-note 9 for an overview of the role of tactics in proof planning). Tactics provide larger, less constrained, proof steps. They abstract away from details like hypothesis numbering to provide a more general action.

A number of points can be raised from these observations. While these may be very difficult to implement, they still warrant a mention:

- **Record tactic level proofs -** Coercion from a CLAM proof plan to SPADE-scripts will be necessary. However, this coercion need not be performed only once. Instead the CLAM proof plan could be recorded and used to create a specific SPADE-script for the current state of the problem. If the state of the problem were to undergo minor changes, generating a new and correct SPADE-script from the original CLAM proof plan could be possible. Note that this does imply an amount of search in performing the coercion.

- **Tactic level interaction -** Implementing an explicit coercion from CLAM tactics to SPADE-scripts may allow for tactic-level user interaction in NuSPADE.

- **External proof checker -** Isolating the coercion to SPADE-scripts makes it possible to consider coercion to other systems. There is value in checking SPADE proofs in a separate system.

2.3 Efficiency and Speed

Speed is a less obvious, but still very important, metric with which to evaluate an automatic theorem prover. As an extension to SPADE, NuSPADE will also have to take into consideration matters of performance. Efficiency is concerned with how well the software makes use of recourses. As might be expected, NuSPADE has not been designed to be inefficient. However, more emphasis is generally placed on clarity than raw efficiency.

In the release versions of NuSPADE it may be useful to provide settings which favour a faster execution over less automation. Such facilitys already exist in SPADE through flags like `simplify_in_infer`.

As an aside, a clever Prolog trick has been proposed to improve performance. Part of the NuSPADE database will describe the SPADE built-in rules. As these rules remain unchanged they could be brought into NuSPADE as static predicates. Most Prologs are faster at accessing static predicates over dynamic ones. This kind of software enhancement is not particularly clear. However, such measures taken carefully might help make the more computational aspects of NuSPADE viable.

3 Information Exchange

3.1 Justified Changes To SPADE

As noted in CS-note 8 section 3, there does exist points where relatively minor changes to SPADE will save a large amount of effort in NuSPADE. Where these changes to SPADE can be fully justified, there is no major objections to making such changes to SPADE.

Thus the onus is on us to justify, implement and show to a good level of confidence that the desired changes to SPADE can be implemented safely. It is suggested that this can only be done through a fairly formal and well documented process.

[1] A number of bash scripts designed to manage a collection of NuSPADE problems.

3.2 Learn From Real Examples

There is certainly value in exploring text book examples for NuSPADE. However, to help build NuSPADE to tackle realistic real world problems, it is necessary to explore the nature of such real world problems. While it would be most beneficial to gain access to the source code of realistic problems, a host of alternatives might be explored:

- **High level descriptions -** Exchange information about the kind of problems that are encountered, including coding constructs used, level of annotations and complexity of proof. This can convey useful information that might lead to more careful selection of textbook examples.
- **VC problem only -** Supply real world *.VCG files without their corresponding code. Without the code, any potential bottom-up techniques[2] can not be explored. However, the top-down techniques can be tested and perhaps refined in view of this information.
- **Older systems -** Legacy support for Pascal may allow some older systems to be considered as examples.
- **Other Spark systems -** There are a number of systems known to be implemented in Spark, ranging from the huge EXAMINER to the small Central Heating Example. Every Spark system, regardless of size, has the potential for gaining useful insights.
- **Praxis In-House Visit -** A few days working with the real examples may be sufficient to begin abstracting common patters of code. From these abstractions our own examples can be derived, in the knowledge they mirror some important aspect of the real examples.

4 Thoughts on Proof

4.1 Arrays as Infinite With Constraints

In SPADE arrays are thought of as infinite entities, where the index is constrained to a particular range. This should mean that wherever an array occurs in SPADE there will be information constraining its range.

4.2 Cases Using Ranges

A proof by cases attempt in SPADE can use range cases. For example, it is be possible to do a proof by cases that looks like `x<=4 or x=5`. Such a mechanism can be used to mimic the behaviour of the proposed range splitting method (see CS-note 10). While the explicit use of a rule is more principled for range splitting, it is interesting to note that SPADE already has a facility to cater for this.

4.3 Diverse Proof Checker

Proof Planning can be conducted in a quite free-form manner. It is thus necessary to check all proof plans through a proof checker. Only once this has been performed is the plan shown to be correct. Current thought is to have the planner of NuSPADE relied on SPADE as this proof checker.

However, SPADE relies on itself being correct. Further, the SIMPLIFIER may be used prior to any proof effort, giving it the same correctness requirements as SPADE. Again the SIMPLIFIER relies on itself being correct.

Currently both SPADE and the SIMPLIFIER demonstrate a high degree of correctness through empirical evidence. This is valuable, but far from ideal. As SPADE and the SIMPLIFIER both do much more than merely proof checking, there is a valid concern that a subtle error is lurking somewhere. While it has not been shown that the current versions of SPADE or the SIMPLIFIER can prove false[3] it remains a distinct possibility.

It would be ideal if the SIMPLIFIER and SPADE relied on a further, dedicated proof checker. Such an external proof checker is likely to require all of the actions of SPADE and the SIMPLIFIER to be recorded in a logical fashion. This sequence of actions can then be played back on the proof checker to check their soundness.

The advantage of this technique is that it forces the theorem provers to justify every action taken in a formal manner. Further, as the proof checker is likely to be fairly light-weight, it can be subjected to rigours analysis. Once expressed in a formal manner it might also be possible to export proofs to other systems, providing further evidence of soundness.

Unfortunately, making such a large architectural change may require a great deal of effort. Both SPADE and the SIMPLIFIER have a number of reasoning built-in that is not formally justified. For example, embedded in the

[2]Extracting proof information from the code, rather than the VCs.

[3]Such a demonstration might help highlight this issue. Of course, this would have to be done without using known SPADE loopholes such as `callpro` or adding false rules.

definition of `deduction` is a number of built in rules (For example, `deduce(X+A<>X+B,T) :- deduce(A<>B,T).`), none of which have an associated formal proof. Providing the suitable formal grounding for such features would be a time consuming, but hopefully possible, task.

It is perhaps a little unreasonable for NuSPADE to propose such sweeping changes to SPADE or the SIMPLIFIER. However, where NuSPADE can deal with these issues, it is worthwhile considering them, rather than pretending that SPADE is a genuine proof checker.

5 Using Proof as a Practical Tool

5.1 Highlight False Hypothesises

Under a false hypothesis, any conclusion can be shown to hold. For example the following is a theorem ($[\Gamma, false] \vdash false$). It might be thought unlikely that a hypothesis will directly contain false. However, it is conceivable that through reasoning about the hypotheses, false can be derived.

Traditional theorem provers do not highlight the occurrence of false in hypotheses. They are only concerned with proving that a conclusion leads from the hypotheses. No consideration is given to the relevance of the theorem or how sensible it might be.

The theorems tackled by SPADE are the VCs. These are extracted from a potentiality flawed specification and corresponding source code. The hypotheses of a VC may potentially contain contradictions that allows false to be derived. Significantly, the SPADE user is interested to know if this is the case, as it is likely to indicate a problem somewhere (Such as a section of unreachable code or flawed specification).

It is worth noting that the SIMPLIFIER already searches for contradictory hypotheses as part of its search (the `CONTRADICTION-HUNTER`). If the SIMPLIFIER detects contradictory hypothesis it discharges the VC, recording the proof as:

```
*** true .   /* contradiction within hypotheses. */
```

Further note that the SIMPLIFIER may prove a VC without noticing that there are contradictory hypotheses. This is the likely reason for not seeing the above message when conducting a suitable example in CS-note 3 section 4.4. Detecting contradictory hypotheses is still a theorem proving operation, and hence may not necessarily be solved automatically.

It may be beneficial to incorporate the search for contradictory hypotheses into NuSPADE. A complete implementation might formally prove that the initial hypotheses of a VC are *not* contradictory.

5.2 Show Non-Theorems

Theorems are conjectures that are proved to be true ($[\Gamma] \vdash true$). Non-theorems are conjectures that are shown to be false ($[\Gamma] \overset{?}{\vdash} false$). Assuming that the hypotheses do not reduce to false, searching for a theorem can be viewed as a search for a series of rule applications that reduces the conclusion to true. Conversely searching for a non-theorem can be viewed as reducing the conclusion to false.

Most theorem provers occupy themselves with searching for theorems. The reasons for such a bias toward showing true might include:

- **Expect true if conducting proof -** Where a theorem prover is being used it is likely that the person conducting the proof believes that their conjecture is true. A search for truth mirrors the task the human user is likely to be trying to perform.

  Where a user is trying to show that a conjecture is not true, the natural way to express the problem is as a goal which when proved true would demonstrate that the conjecture is false. Again a search for truth is expected.

- **True theorems are more useful -** Knowing what is true is generally more useful than knowing what is false. Thus theorem provers aim to prove the more useful conjectures.

However, in each of these cases there is an argument that they do not hold for VC proofs in SPADE:

- **Do not expect true if conducting proof in SPADE -** The conjectures are VCs, generated from the specification and source code. An error in the specification or the code is likely to result in the VC being false. While the user might want the VCs to be true, they are conducting the proof because they are unsure if they are.

  The important observation is that VCs are considered more likely to be false than the typical conjectures presented to a regular theorem prover.

- **Not the case that true theorems are more useful in SPADE -** A false VC indicates an error in the code or the specification. Detecting such errors is the primary reason for conducting a VC proof. Therefore showing non-theorems is quite valuable to the SPADE user.

It is considered that NuSPADE should be able to show non-theorems as well as prove theorems. There are some interesting theorem proving points that can be made here:

- **Negated logic of for all is favourable -** To show that $([\Gamma] \overset{?}{\vdash} \forall a.x)$ is true requires a proof that for every possible value of a, x is true. However, to show that this is false requires a proof that there exists at least one possible value of a for which x is false.

  It is considered more straight forward to find a single witness to demonstrate false, than the exhaustive proof to show true.

- **Negated logic of Exists is less favourable -** To show that $([\Gamma] \overset{?}{\vdash} \exists a.x)$ is true requires a proof that there exists at least one possible value of a for which x is true. However, to show that this is false requires a proof that for every possible value of a, x is false.

  It is considered less straight forward to find an exhaustive proof to show false, than a single witness to demonstrate true.

- **Exploit underlying bugs -** A false VC stems from either an error in the specification or the code. It indicates that the specification and the code are inconsistent.

  As has been highlighted by the testing community, there are a number of key points in code and specifications where bugs are more likely to occur. A common example is the 'off by one' error in dealing with arrays. This leads to the idea that false VCs may be more frequently derived by exploiting the logical shortcomings expected from common coding errors. For example, the 'off by one' error might inspire a more intensive exploration of the extreme ends of any quantified array in the VC during a search for false.

Dealing with non-theorems in NuSPADE is likely to require introducing an explicit structure to record both proofs of theorems and showing non-theorems. The planner part of NuSPADE may be kept more or less intact by transforming a search for false into a search for true by negating the conclusion. Alternatively, the planner could be extended to accept the search for false as a valid planning effort. This might then also be used to support a proof by contradiction.

Note that non-theorems can not be dealt with as distinct logical objects as the proof checker (SPADE) does not support these. While it is possible to demonstrate and record theorems to a standard the SPADE user should accept, the same is not necessarily the case for non-theorems. A few possible solutions to this are proposed:

- **Negate conclusion and record proof -** The negation of the conclusion could be presented to SPADE inside a specially generated VCG file. The necessary FDL and RLS context files could be borrowed from the original conclusion. The proof of this negated conclusion can then be demonstrated as if for any other SPADE VC. The correct interpretation of these negated proofs could be ensured within the NuSPADE interface.

- **Reduced recording and proof checking -** A non-theorem indicates an error. Therefore, where encountered, the SPADE user will want to modify the source or specification. They are not interested in recording the proof of a non-theorem to successfully demonstrate the existence of an error. This observation is important as it indicates that non-theorem proofs need not be recorded in full detail, or even proof checked in full detail. From this observation two approaches are proposed:

  - **No record, no proof checking: Accept the plan directly -** The planner could be used independent from SPADE in detecting and highlighting non-theorems. The drawback here is that the planner is not a proof checker. It is possible for the planner to generate a proof plan that can not be executed in practise. Therefore it is possible for the planner to generate a non-theorem proof plan of a conjecture, even where the conjecture is true. Such behaviour is generally not useful and hence uncommon in planners. However the potential for this will always remain.
  - **Light record, with proof checking: Execute plan to false -** This is as above, only now the proof plan is executed as far as possible inside SPADE. Reducing a conclusion to a subgoal of the form $([\Gamma] \vdash false)$ should be sufficient to convince the SPADE user that the conclusion is indeed a non-theorem. Note that as SPADE supports proof by contradiction it will also be necessary to show that the subgoal has the correct polarity. The `status` command should provide this information.

6 The SIMPLIFIER

6.1 Why the SIMPLIFIER

Most of the problems associated with the SIMPLIFIER stem more from why there is a SIMPLIFIER in the first place, rather than the SIMPLIFIER itself. Thus the weaker `simplify` command of SPADE does not present similar problems, not because it does not simplify, but because it does not do so outside the scope of SPADE!

The SIMPLIFIER is isolated from SPADE because it can operate in batch mode. It can be run overnight with zero human interaction. It compares with the EXAMINER in operation, giving the user valuable information fully automatically.

6.2 Issues for NuSPADE

There are a number of potential issues in dealing with the SIMPLIFIER as part of NuSPADE:

- **Structural changes to problem -** The SIMPLIFIER takes a problem, tries its very best to solve it, and where unsuccessful gives up and stores its current progress ready for a human user to continue work on. Unfortunately, while this half-worked-on example may be quite straight forward for humans to use, the same is not necessarily the case for NuSPADE.
- **Is a collection of valuable components -** The SIMPLIFIER performs a valuable amount of theorem proving. Thus the SIMPLIFIER can not be ignored by NuSPADE. Further, the SIMPLIFIER is really a collection of theorem proving components. Each of these may have the potential to provide some valuable theorem proving. Again these can not be ignored by NuSPADE. Finally, although the current practise, there is no reason that the SIMPLIFIER or its components should only be used once in a proof.
- **Is also human-simplification -** The SIMPLIFIER also simplifies the problem to make it clearer for humans to read. People generally prefer to work with the output from the SIMPLIFIER than the original problem[4]. This is a conflict of interests for NuSPADE. On the one hand it wants to perform automatic proofs using the original output. On the other hand it wants to subsume the operation of SPADE and hence allow users to operate with the more readable simplified output.
- **Is part of the proof checker -** The SIMPLIFIER is able to make changes to a goal. If the SIMPLIFIER is in error, there is no later proof checker that will detect this. Thus the SIMPLIFIER must meet the same levels of correctness required of a genuine proof checker. This is a closely related to the diverse proof checker issue in section 4.3.

6.3 Solutions for NuSPADE

A few solutions are presented for integrating the SIMPLIFIER within the NuSPADE architecture:

- **All-or-nothing** SIMPLIFIER **-** The SIMPLIFIER could have an additional switch forcing it to either discharge conclusions, or not make any changes. After this use of the SIMPLIFIER NuSPADE could pick up the remaining problems, in the knowledge that they have been untouched by the SIMPLIFIER.

  It might be argued that there are certain operations performed by the SIMPLIFIER that would always want to be performed before presenting problems to NuSPADE. For example, the replacement of constants with their actual values. However, even such minor changes have the potential to destroy valuable structure information. For example, replacing constants with their values will result in structure (variable names) being lost.

  Note that the SIMPLIFIER can affect hypothesis as well as conclusions. Similar to the treatment of conclusions the all-or-nothing SIMPLIFIER should leave all hypothesis alone, unless they are completely removed. This case should only occur where there are exact duplicate hypothesis in the problem. Thus the only operation available to the all-or-nothing SIMPLIFIER is the complete removal of hypothesis or conclusions.

  This is a reasonable solution allowing much of the current situation to remain the same. However, it does not address many of the issues associated with the SIMPLIFIER.

- **Include the** SIMPLIFIER **as part of** NuSPADE **-** The SIMPLIFIER could be broken down into its constituent parts and integrated as part of NuSPADE. A new command might be made available in NuSPADE to perform the operation usually done by the SIMPLIFIER on the current problem.

[4] For example, amongst many other more significant changes, the SIMPLIFIER simplifies `not(x=y)` into the clearer expression `x<>y`.

To continue to support the batch mode behaviour, NuSPADE could include a batch mode setting. Such a mode could now combine the previous automation provided by the SIMPLIFIER plus the new automation offered by NuSPADE.

- **Rationally reconstruct[5] the** SIMPLIFIER **as part of** NuSPADE **-** This is essentially the same idea as above, only it is no longer implied that the SIMPLIFIER code is used. As discussed in section 4.3, the SIMPLIFIER must meet the correctness of a proof checker. Adding the SIMPLIFIER into NuSPADE would mean making NuSPADE a proof checker also. Justifying the complete correctness of NuSPADE (a research system) is unlikely.

  In essence, it might be more straight forward to extract the ideas from the SIMPLIFIER rather than its actual code. These ideas can then be built into NuSPADE in a manner suitable for NuSPADE. This need not be the daunting task it sounds. Simplification components used for automatic proof may translate into methods available for proof planning in NuSPADE. Simplification for the purposes of making the problems easier for humans to read may be viewed as a special kind of interaction-support proof planning. In both of these cases it should be possible to mix in new NuSPADE components with the traditional ideas of simplification.

- **All of the above -** The previous proposal makes NuSPADE stronger by extracting good ideas found in the SIMPLIFIER. There is an assumption that this is done to the extent that NuSPADE will eventually do everything that the SIMPLIFIER can do, removing the need for a separate SIMPLIFIER. On the grounds of speed alone, this is probably flawed.

  Thus, it may be beneficial to use the first idea of an all-or-nothing SIMPLIFIER in combination with NuSPADE. Where this version of NuSPADE has incorporated, and probably refined, some of the more powerful and novel aspects of the SIMPLIFIER. Note that this incorporation may be done by copying sections of code out of the SIMPLIFIER or by rational reconstruction in NuSPADE.

7 Interface and Interaction

7.1 Orientation

In some problems the number of hypothesis and conclusions and their term structures can be very large. Where this occurs it can be difficult for a human user to orient themselves within the problem to spot an insight leading to proof. To deal with these issues SPADE has hypothesis hiding commands (`delete` and `forget`), allowing the user to focus on the key details for their current problem. Further, there is an expression exploration command (`traverse`) allowing the user to gradually pull-apart a large expression to find its key details.

Much of the emphasis placed on NuSPADE has been that of automatic proof. However, there is certainly scope for catering for such interaction issues. Improving the interface of an interactive theorem prover is rarely a case of merely adding a Graphical User Interface (GUI). It is necessary to consider the way in which the prover operates. Some initial suggestions are shown below:

- **Automatic grouping of hypothesis and conclusions -** By exploiting shared structure information it may be possible to automatically narrow the visible hypothesis in view of tackling a particular conclusion. This might translate into a command of the form: `narrow c#3`.

  Note that this narrowing may also have potential efficiency benefits, if used inside the planner to constrain proof planning searches.

- **Add a graphical user interface -** Where interaction assisting information has been calculated by the system, a GUI may allow the information to be presented in a more intuitive manner. For example, the `traverse` command could be enhanced by a traversable syntax tree.

8 Changes

8.1 Version Dependent Rules In SPADE

The latest version of SPADE is able to make different rules available depending on the context in which the VCs were generated. This can be used to deal with language incompatibilities, such as Ada and Pascal having different meanings for the `mod` operator. As SPADE is being used as the VC parser of NuSPADE this change should not impact on NuSPADE at all.

[5] A term used by A. Bundy in [2], for "ignoring aspects of peripheral interest, and focusing on ... their essential contribution".

8.2 Entire New Edition of The Spark Approach

The Spark language and its tools continues to change at a fairly rapid pace, to the extent that enough changes have been made to warrant a completely new edition of the Spark Approach[1]. It was being considered that the new edition of the book may have a Linux version of the Spark demo tools. With Linux growing in popularity as an academic platform, this may help make Spark more viable in a teaching environment. It is conceivable that this is a further route to Spark examples, as considered in section 3.2.

9 Terminology

9.1 Command Log is a Script

During operation SPADE stores all user input in a *.CMD file. The Praxis term for this is a 'script', rather than a 'command log'. As the term script is a little misleading if used out of context, the term 'SPADE-script' seems appropriate.

9.2 No Run-Time Errors as Exception Freedom

In general, proving the absence of run-time errors is not necessarily sufficient to prove the absence of exceptions. Users of languages can often introduce their own exceptions, over and above those that occur when a run-time check is unsuccessful.

The Spark language ensures that users can not declare new exceptions. It also rules out the possibility of many run-time errors occurring. Thus part of the proof of no run-time errors in Spark comes from the language definition. Further, showing that there are no run-time errors in Spark is also a proof of exception freedom.

To emphasise the additional role that the Spark language plays the stronger term 'exception freedom' is preferred over the weaker 'no run-time errors'. As much of the required effort has been eliminated by the Spark language definition, 'Spark exception freedom' may be more appropriate where the language is not obvious from the context.

10 Emerging Ideas

10.1 Constraint Logic Programming (And Sicstus again)

An important step in some of the example proof work already undertaken is reasoning about constraints, in particular integer constraints. Much of the reasoning for this has already been implemented as part Constraint Logic Programming systems. A reasonable[6] implementation of such a system is provided as part of Sicstus Prolog. Further, although there are no current plans for building a GUI, it is noted that Poplog does not provide a good architecture for doing so. It would require using Poplog hooks to one of the X widget sets: athena, motif or openlook. There is no Microsoft Windows GUI support. Sicstus, on the other hand, supports both TCL/TK and Java.

As noted in CS-note 7 Sicstus is a very favourable platform for building SPADE and NuSPADE. Further weight is now added to the Sicstus proposal through support for Constraint Logic Programming.

References

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

[2] A. Bundy. *The Computer Modelling of Mathematical Reasoning.* Academic Press, 1983. Second Edition.

[6] While alternative systems have not been fully evaluated, it is apparent that the Sicstus Constraint Logic facilities are somewhat better than the average.