CS-Note* 36

Bill J. Ellis
January 9, 2003

Raising NuSPADE Issues (Volume 4)

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

See CS-note 19 for the introduction to this series of notes. Note that this particular note has been delayed for quite a while, allowing time for various alternatives to be considered[1]. Primarily for my own record there are short historical summaries of the other directions that were eventually dismissed.

2 Support Tactic Language

Tactics are discussed in CS-note 9. The definition of what constitutes a tactic is sufficiently general to allow for some scope in their adoption. However, tactics are generally more high-level than the proof commands of SPADE-PC. It is considered advantageous to build a normal tactic language into NuSPADE, bringing the following main benefits:

- **Clear intermediate proof language -** Tactics will provide a language to describe proofs between proof plans and their corresponding SPADE-PC-scripts, as illustrated in Figure 1.

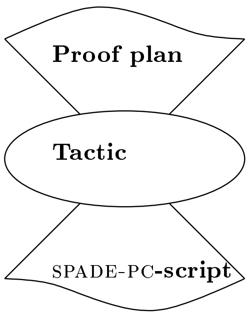

Figure 1: Tactics as intermediate proof language

The proof plans are specific to SPADE-PP, including planning information such as ripple annotations. Due to the low-level nature of the SPADE-PC-script commands, they will necessarily appear in large numbers. The tactic language is isolated from the behaviour of both SPADE-PP and SPADE-PC, allowing for a more concise and intuitive alternative.

- **Is essential for proof planning -** The methods of proof planing are tightly connected to the tactics of the object-level prover. The low-level commands of SPADE-PC, though technically covered under the large definition of tactics, are not the kind of higher level tactics envisaged for proof planning.

- **Neater conversion to SPADE-PC-scripts -** Regardless of the adoption of tactics, their will need to be some mechanism inside NuSPADE that converts from proof plans to SPADE-PC-scripts. Through adopting tactics this problem breaks down into converting each tactic into a SPADE-PC-script. Without such decomposition the conversion process will be more difficult to manage.

- **More user commands -** If individual tactics can be converted into SPADE-PC-scripts then they become new commands of SPADE-PC, implemented in terms of the old commands. Within a NuSPADE interaction environment it should be straight forward to allow a user access to these tactics.

The main drawbacks of tactics for NuSPADE are:

- **No tight coupling -** Without an intermediate tactic language proof plans are directly converted into SPADE-PC-scripts. Such tight coupling may initially be quicker to implement[2].

[1] Either this means I am indecisive or open minded...

[2] If considering a few kinds of proof plans and hence a few kinds of possible SPADE-PC-scripts a tightly coupled conversion process might be constructed quicker than if a clean intermediate tactic language were developed.

2.1 Current Tactics in CLAM

After completing a proof plan, CLAM generates a specific-tactic. This is a tactic, potentiality composed of several nested tactics, which performs a specific task, such as proving a particular conclusion. For example, the tactic in Figure 2 is generated to prove associtivity of multiplication (ASSM) (See CS-note 5). Each of the tactic components of this specific-tactic map to actions that can be performed in a theorem prover. These actions are encoded in a suitable language. For example, the Prolog code in Figure 3 is used to execute the `wave` tactic inside OYSTER.

The tactic language of CLAM works on known conclusions using named rules and hypothesis. To avoid any coercion between the names used in tactics and the names used in object-level proofs, the same names are used. A good example is `fertilize(weak(v0))` as seen in Figure 2. The `v0` here is recorded and used during proof planning to keep track of this hypothesis. This naming convention has been carefully designed to exactly mirror that of OYSTER, so that during the execution of this tactic `v0` will refer to the correct OYSTER hypothesis.

This implicit information sharing between proof plans and tactic execution makes the process of converting from proof plans to object-level commands more straight forward[3]. Both CLAM and OYSTER employ the same general proof framework, making it possible to think OYSTER inside CLAM. This is not a coincidence, as the names suggest CLAM was built as an extension to OYSTER. A further advantage is that OYSTER is very small, well understood, and at the time of building CLAM unlikely to change.

The situation is not the same for NuSPADE. The general proof framework of SPADE-PC is different to that seen in CLAM. In particular, subgoaling (newgoaling) in SPADE-PC is slightly different (See CS-note 8 and CS-note 4). The execution of some tactics in SPADE-PC, for example rewriting, is likely to require further proof structure manipulating temporary hypothesis (see CS-note 8). While not preventing the mirroring of SPADE-PC behaviour in the proof plan, it seems that this would be rather difficult.

```
induction([s(a)],[a:pnat]) then
   [eval_def([1,1],times1) then
      eval_def([1,2,1],times1) then
        eval_def([2,1],times1) then
          elementary(...),
    wave([1,2,1],[times2,equ(left)]) then
      wave([1,1],[times2,equ(left)]) then
        wave([2,1],[lemma1,equ(left)]) then
          fertilize(weak(v0)) then
            elementary(...)
   ]
```

Figure 2: CLAM tactic for associtivity of multiplication (ASSM)

```
wave(Pos,[Rule,Dir]) :-
    dequantify_all(Vars,Frees)
       then rewrite_at_pos(Pos,Rule,Dir)
          then requantify(Vars,Frees).

dequantify_all(Vars,Frees) :-
    goal(G), matrix(Vars,_,G), dequantify(Vars,Frees).

rewrite_at_pos(Pos,L=R in T,equ(left)) :-
    goal(G),hyp_list(Hl), (hypothesis(Eq:L=R in T) ; hypothesis(Eq:R=L in T)),
    genvar(V), free([V],Hl), \+ appears(V,G), \+ appears(V,R),
    replace(Pos,V,G,Gv),
    top_univ_level(G,I),II is I+1,
    subst(at(II),over(V,Gv),L=R in T)
       then [univ_elim(Eq), idtac,wfftacs].
rewrite_at_pos(Pos,H,Dir) :-
    hypothesis(H:X=Y in T),!,
    rewrite_at_pos(Pos,X=Y in T,Dir).
rewrite_at_pos(Pos,H,Dir) :-
    hypothesis(H:_=>X=Y in T), !,
    hyp_list(HL), free([EqH],HL),
    elim(H,new[EqH])
     then [elementary,
           rewrite_at_pos(Pos,X=Y in T,Dir) then thin([EqH])].

requantify([],[]) :- !.
requantify(Vars,Frees) :-
    goal(G), untype(Vars,Vs), s(G,Vs,Frees,NewG), quantify(Vars,NewG,QNewG),
    hyp_list(H), free([Seq],H),
    seq(QNewG,new[Seq])
        then [thin(Frees),
              elim_on(Seq,Frees,Last)
                 then hyp(Last)].
```

Figure 3: CLAM tactic for associtivity of multiplication (ASSM)

[3]Probably. Getting the planning process to exactly copy the behaviour of the object-level prover appears fiddly.

2.2 Current Tactics in HOL

The HOL theorem prover refers to hypothesis and conclusions via their structures, rather than employing a labelling system.

The disadvantage of holding complete structures is loss of efficiency. The structures themselves will occupy a reasonable amount of space and be more costly to manipulate. The advantage is that their is no need to model the object-level prover labelling mechanism in describing tactics.

2.3 Propose: Expansive Tactics

For implementation purposes, an intermediate tactic language would be helpful. The problem lies in supporting this higher level descriptive language while still holding enough information in each tactic so that they can be unpacked into corresponding SPADE-PC-scripts.

As seen in CLAM the problem is overcome by using implicit information about the exact behaviour of the object-level prover. This would be quite difficult for SPADE-PC. Instead, the HOL approach is more favourable.

To execute these tactics it will be necessary to convert into SPADE-PC-scripts and hence take account of labels. This will involve a search during tactic execution to identify the corresponding labels for each structure. As different hypothesis can hold the same structure there may be more than one label that a structure matches with. This is a not a problem, as the duplicate hypotheses are describing exactly the same thing. It does not matter which label is eventually selected, so long as it matches the structure.

This conversion process is performed after proof planning has finished. Certainly the operation should still be efficient, but it is less critical than the main process of proof planning.

A historical note...
A tactic language emerged as a good idea, primarily because it would give some organisation to the process of converting from proof plans to a SPADE-PC-script.
However, the intention for quite a while was to copy the CLAM approach and try to think SPADE-PC in SPADE-PP, gathering labels as appropriate. This involved a hypothesis label dispenser, which would be used alongside the global and local hypotheses management outlined in CS-note 21. As this was considered in more detail, it became apparent that more and more special rules would need to be built into the label dispenser and proof planning. It was eventually decided that the crude method of returning complete structures would be much more straight forward, and probably a fairly insignificant efficiency loss.

2.3.1 Note on Implementation

The tactic language will emerge naturally from the development of methods for use during proof planing. It is likely that many of the tactics already established in CLAM will be repeated inside NuSPADE.

The conversion from tactics to SPADE-PC-scripts is most naturally implemented in a library attached to NuSPADE. As the behaviour of SPADE-PC changes across newer versions, hopefully only this library need be modified to accommodate the changes.

3 Recording Proofs

The main document produced within a theorem prover is a proof of the given problem. Once produced, this document is recorded as evidence that the problem was proved. It is often desired to store the document in a form such that the proof of the problem can be automatically repeated. This is similar to regression testing, and is termed by SPARK *regression proof*. It is important that NuSPADE facilities the creation of executable proof documents. This is not a trivial exercise for various reasons:

- **Partial plans -** Where using SPADE-PP the result may be a partial plan. Such an incomplete plan might decompose the current problem into a number of smaller more manageable problems, some of which could not be proven automatically. Certainly, partial plans indicate a need for better automation. However, total automation is unlikely and partial automation is better than none at all. Therefore it is important that partial plans are supported by SPADE-PP.

  The CLAM-HOL project [2] added proof planing facilities to HOL by communicating with CLAM. This took a very practical view on partial plans. They viewed a partial plan in CLAM as an opportunity to exploit their communication mechanism and attempt the unsolved problems in HOL. SPADE-PP may similarly make use of external systems. However, regardless of the power of SPADE-PP a partial plan is always a possibility.

  Where automatically finding a partial plan the remaining problems should be presented to the user to discharge. The partial plans might be converted to a SPADE-PC-script and executed within SPADE-PP as normal. As the proof is partial, the SPADE-PC-script will eventually stop placing the SPADE-PC user right where the planner got stuck.

  Unfortunately, such an approach is not viable in general. As a feature[4] SPADE-PC only has one problem open at a time (see CS-note 4). If the partial plan leads to more than one open subgoal then it is necessary to delay executing parts of the partial proof document until these branches are explored. Further, as planning is a proof command it is possible that the user may tackle one of the open subgoals of a partial proof plan by performing another proof plan, which may again have multiple partial branches. Coping with such nested partial plans in an organised manner appears problematic.

- **Locating middle plans -** It is possible that the proof of one VC is achieved through more than one plan. For example, a typical proof might have the following interaction. A user loads a problem and immediately executes the planner. The planner discharges everything bar one conclusion. The user identifies a transformation that might assist proof (perhaps a well placed case split) and successfully launches another plan to discharge the remainder of the problem.

  The first planning attempt for a goal is expected to be performed just after the goal is loaded. It is straight forward to identify that the generated proof plan maps to the first commands that should be applied to the goal. However, further proof plans may be requested deeper in a proof attempt. The current goal might be a result of numerous plans and user interactions. Although it should be possible to execute the proof plan on the correct goal it is no longer clear where in the proof tree the current goal actually is. This might not be a problem if all the intermediate steps were recorded. However, such recording must take into account all of the operations available to the user, including backtracking. Coupled with considerations for multiple partial proof plans this is evidently not a trivial exercise.

- **Resume -** A useful facility in SPADE-PC is the ability to save the current problem, and resume on this problem in a future SPADE-PC session. This is implemented by saving the SPADE-PC database and loading this again for resuming[5].

  Resuming in NuSPADE could be achieved by using the existing SPADE-PC resume mechanism. This would involve populating the SPADE-PC database by invoking the appropriate SPADE-PC predicate. Unfortunately, not even SPADE-PC maintains SPADE-PC-scripts across saves. It becomes difficult to relate the resumed problem to the original goal.

3.1 Options

There are numerous options available to record proof documents in NuSPADE:

- **Ignore (its a SPADE-PC problem) -** Currently SPADE-PC records almost everything typed during each session, making up a proof document. Resuming is supported by saving and restoring the essential aspects of

[4] An integral part of the operation of SPADE-PC.

[5] Note that this makes the soundness of proof depend on the SPADE-PC save and resume mechanism. As noted in CS-note 13 supporting external proof checkers (diverse proof) would eliminate these concerns.

the system state. Although not particularly ideal, this approach works and, as NuSPADE contains SPADE-PC, will continue to be available.

- **Record the proof plans -** The proof plans provide a high level description of how to solve a goal. It might be possible to use proof plans as executable proof documents if they always fully automatically discharged a goal. However, as considered above, it is difficult to consider multiple partial proof plans and correctly tie these together. It will also be necessary to find another mechanism to record those proof steps taken by the user and not the planner.

- **Record the tactics -** As described above, it is helpful to create an intermediate tactic language to describe proof operations. These might be recorded by NuSPADE.

  Executing the proof document would mean executing the tactics. The disadvantage is the additional calculations required to get from the tactics to the low-level SPADE-PC commands. This process will need to be repeated each time the proof is conducted. However, this operation will involve little search and is not expected to be time consuming.

  An advantage is that the higher level tactics will be more robust in light of small changes to the goal. For example, changing the hypothesis ordering would not cause a tactic proof to break. The process of executing the tactics would find the correct labels from the full structures recorded in the tactics.

  Perhaps more interesting is the possible pursuit of checking proofs in a system other than SPADE-PC. The tactics do not use any SPADE-PC specific labels or proof commands[6]. It is not entirely unreasonable that, with a little effort, another proof checker might be able to interpret these executable proof documents.

  A problem with recording tactics is where the user employs a low-level SPADE-PC command. Including this command and its effects amongst the tactics is not viable due to the differences between the two kinds of command. It may be possible to shadow all of the SPADE-PC commands with tactics that perform essentially the same operation. However, such tactics may erode the look and feel of SPADE-PC.

  A side-effect of recording every proof operation is that proof branches can be automatically repeated. Undoing the last step should be the same as restarting the proof and repeating every step but the last step. By explicitly recording tactics and their point within a proof tree it becomes possible to implement the appearance of features such as undo or having multiple goals open[7]. Unfortunately the scope of such features is limited by SPADE-PC newgoaling. It is not possible to introduce a new hypothesis until it has been proved. Thus it is not possible to assume a hypothesis within a goal and tackle the goal, with this assumption, before proving the assumed hypothesis is true.

- **Record the SPADE-PC commands -** Although SPADE-PC does record every proof command, it does not do so in a particularly organised manner. Recording the structure of the proof tree would give slightly more readable proof scripts and possibly allow for the improved interaction side-effect noted above if tactics were recorded.

  A key benefit of using the SPADE-PC commands is that they *are* the commands of SPADE-PC. The proofs should be more familiar to a SPADE-PC user. However, this is less likely to be true after executing a proof plan. Reading the low-level generated proof commands might compare to reading the assembly mnemonics for the code generated by a high level language compiler[8].

  As described above SPADE-PC already records the command files. This is sufficient to record proofs within NuSPADE such that they can be repeated on a regular version of SPADE-PC. There is little extra to be gained from recording the SPADE-PC commands over the tactics.

3.2 Proof Recording in CLAM

Where CLAM finds a proof plan, this is converted into a tactic and displayed on screen, as shown in Figure 2. However, this specific tactic is not recorded. It is executed, using the tactic predicates, to generate the object-level OYSTER proof steps. These proof steps are then saved alongside the original problem. See Figure 4 for the start of of the recorded OYSTER commands for ASSM.

The OYSTER commands are difficult to read due to dense and deeply nested syntax. As the commands make reference to explicit labels (`v1`, $\cdots$ `vn`) contextual information about the behaviour of OYSTER must be known to

[6]They will refer to SPADE-PC rules. However referring to rules is quite common in proofs. The tactic proofs are not reliant on any special propertys of SPADE-PC.

[7]Switching between subgoals translates to restarting the proof and descending down alternative recorded proof paths.

[8]The gap is probably far smaller. However, as explored in CS-note 8, straight forward rewrite operations might be reasonably unpacked into 4 or 5 SPADE-PC proof commands. The gap certainly exists.

```
problem([]==>a:pnat=>b:pnat=>c:pnat=>times(a,times(b,c))=times(times(a,b),c)in pnat,ind_strat_I(induction(s(v0
),a:pnat)then[try base_rewrites([base([1,1],times1),base([1,2,1],times1),base([2,1],times1)]),ripple_out([wave
([1,1],[times2,left]),wave([1,2,1],[times2,left])])then try weak_fertilize(left,[weak_fertilize(left,in,[1],v1
)])]),lambda(a,p_ind(a,lambda(v0,lambda(v1,su(su(lambda(v2,su(su(v4,[v3 of v1],[v4]),[v2 of v0],[v3]))of lambd
a(v0,lambda(v1,su(su(lambda(v2,su(su(v4,[v3 of v1],[v4]),[v2 of v0],[v3]))of lambda(v0,lambda(v1,su(su(lambda(
v2,su(su(v4,[v3 of v1],[v4]),[v2 of v0],[v3]))of _385,[times1 of v1],[v2]),[term_of(times1)],[times1]))),[time
s1 of v0],[v2]),[term_of(times1)],[times1]))),[times1 of times(v0,v1)],[v2]),[term_of(times1)],[times1]))),[v0
,v1,lambda(v2,lambda(v3,su(su(su(lambda(v4,su(su(v6,[v5 of v3],[v6]),[v4 of v2],[v5]))of lambda(v2,lambda(v3,s
u(su(su(lambda(v4,su(su(v6,[v5 of v3],[v6]),[v4 of v2],[v5]))of lambda(v2,lambda(v3,su(su(lambda(v4,su(su(v6,[
v5 of v3],[v6]),[v4 of v2],[v5]))of _567,[v4 of v3],[v5]),[v1 of v2],[v4]))),[v4 of v2],[v5]),[times2 of v0],[
v4]),[term_of(times2)],[times2]))),[v4 of times(v2,v3)],[v5]),[times2 of v0],[v4]),[term_of(times2)],[times2])
))])),[problem([a:pnat]==>b:pnat=>c:pnat=>0=0 in pnat, tautology(dequantify_once then dequantify_once then ide
ntity),lambda(b,lambda(c,axiom)), [ ]) ext _385, problem([a:pnat,v0:pnat,v1:b:pnat=>c:pnat=>times(v0,times(b,c
))=times(times(v0,b),c)in pnat]==>b:pnat=>c:pnat=>plus(times(times(v0,b),c),times(b,c))=times(plus(times(v0,b)
,b),c)in pnat,...
```

Figure 4: Start of CLAM recorded proof as OYSTER commands for ASSM

establish what these labels will mean at this point in the proof. Essentially, by storing proofs in this manner they are suitable for consumption only by OYSTER.

3.3 Propose: Record Tactics

The tactics shall be recorded to make up the executable proof document of a NuSPADE session. As NuSPADE contains SPADE-PC, the SPADE-PC-scripts will continue to be generated as usual. Thus the traditional behaviour of SPADE-PC will continue to be supported, while allowing more powerful behaviour to be explored in NuSPADE.

A key problem in recording tactics is continuing to allow the user access to SPADE-PC commands. It should be possible to shadow the SPADE-PC commands with tactics, ensuring that any proof can be fully described using tactics alone. If the user still choses to employ the low level SPADE-PC commands then the tactic recorded proof will not be complete, and only the SPADE-PC-scripts will record the complete proof.

3.3.1 Note on Implementation

As each tactic is executed it is recorded within a proof tree. The leaves of the proof tree will be marked either open or closed to indicate that the subgoal is either unproven or proved, respectively. For example, see Figure 5.

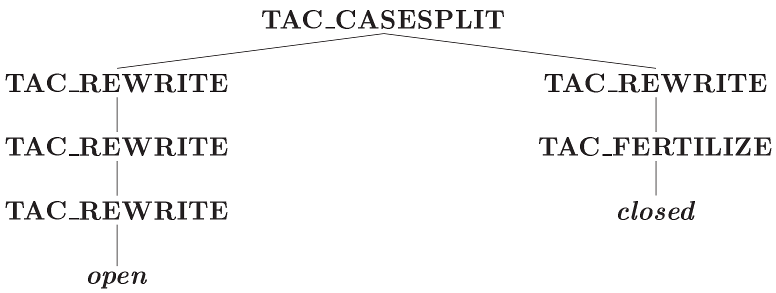

Figure 5: Recorded tactic tree

It is probably most straight forward to hold the proof tree as a collection of asserted predicates. The executable proof document will be a save of these predicates. Note that until a proof is completed, there may be more than one partially complete path, open at the leaf, in the executable proof document. In this manner the proof document supports both the recording of completed proofs and the ability to resume partially completed proofs.

Converting from proof plans to a specific tactic should be relatively straight forward given the direct relationship between methods and tactics. This specific tactic can then be inserted into the current point in the proof tree. Note that these inserted tactics remain to be executed. Such an approach will allow the supporting of partial proofs. The tactics inserted will be executed as the user reaches their branch in the proof. If the proof plan is complete then all branches will be automatically explored as there will be no point at which to wait for user interaction.

A historical note...
For some time it was desired to store plans and their association to tactics or SPADE-PC proof commands. The ideas centred around a separate directory for each plan and its generated proof document. Some over-arching structure would be responsible for joining the plans together, possibly with some user interactions, to prove the original goal. This process would be quite difficult. Further, on reflection, the value of correctly archived plans is fairly minimum, especially if the proofs are also recorded in another form.
These issues were eventually overcome with the decision to record the SPADE-PC commands. Following the current practise of SPADE-PC this was originally thought of as a collection of separate SPADE-PC-script files tied together by an over-arching proof makefile. This approach was gradually refined until all the information was asserted in Prolog and saved much like resuming is supported in SPADE-PC. This eliminated may of the problems associated with recording proof plans.
However, recording the SPADE-PC commands still left open a few problems. Primarily it does not fit well with the use of an explicit tactic language. Recording tactics may be problematic if the user still has direct access to the SPADE-PC commands. However, through appropriately packaging these commands, it should be possible to support both the modern tactic view of proof and the historic operation of SPADE-PC.
Note that the possibility of supporting diverse proof via recorded tactics is somewhat accidental. The tactics do not include SPADE-PC specific labels, as doing so would be difficult and unnecessary. The tactics are recorded rather than SPADE-PC-scripts as they are more natural. Together these independent choices allow for a outputting a proof that has no special reliance on SPADE-PC.

A historical note...
An idea, later dropped, was the explicit recording of the factors influencing confidence in a proof. In HOL the special type `thm` is reserved for theorems. Everything marked `thm` has been shown to be a theorem entirely within the internal logic of HOL. A similar approach might be considered in NuSPADE. For example:

`GOAL1 is_thm_if SIMPLIFIER`

`GOAL1` was proved automatically by the SIMPLIFIER. It is a theorem, so long as the `SIMPLIFIER` is correct.

`GOAL2 is_thm_if (RULE1 and SPADE-PC)`

`GOAL2` was proved inside SPADE-PC making use of rule `RULE1`. Thus `GOAL` is a theorem, so long as `RULE1` is a theorem and SPADE-PC is correct.

`RULE1 is_thm_if (SIMPLIFIER)`

`RULE1` is proved by the SIMPLIFIER.

`GOAL2 is_thm_if ((SIMPLIFIER or RULE1) and SPADE-PC)`

The above knowledge of `RULE1` can now be included in the discussion of `GOAL2`. This is a theorem if the SIMPLIFIER or `RULE1` is correct and SPADE-PC is correct. Note that `RULE1` remains as it might be proved through means other than the SIMPLIFIER.

Note that SPADE-PP is absent from these confidence descriptions. SPADE-PP makes no soundness claims, instead deferring this to SPADE-PC which executes the plans that SPADE-PP generates. Thus no confidence can be gained in the correctness of a theorem by the correctness of SPADE-PP.
The original source of these ideas came from a desire to document diverse proof checking. Where a goal has been shown to be a theorem in many systems its confidence that it is a theorem will be the disjunction of the correctness of these systems. It was thought that the relationship between NuSPADE and CLAM might allow for proofs to be checked in OYSTER as well as SPADE-PC. Hence there was some grounds for considering a method that explicitly recorded which systems a theorem has been proved within. The relationship between NuSPADE and CLAM is likely to be much wider than first thought. However, as discussed above, the possibility of diverse proof remains through an accidental elimination of SPADE-PC specific details in recorded proof documents.
There are some merits in recording confidence in this manner. It makes explicit the conditions under which each theorem is stated. This is especially useful in NuSPADE where the list of conditions will change depending on the systems used in proving VCs.
However, although presenting an interesting extension NuSPADE, it is far from being a vital component. Further, the implementation of this would require many involved changes to the SPARK tools.

4 Project Management

Using NuSPADE will involve accessing and generating a lot of information. There are already a number of external files used by SPADE-PC and it is likely that NuSPADE will contribute some more. It is considered beneficial if NuSPADE actively manages this information.

4.1 Users Perspective

In [3] a users perspective of using the SPARK tools, in particular SPADE-PC, for proving exception freedom is discussed. This concludes with a number of final thoughts including a recognition of the need for 'Management of proofs as code evolves'.

4.1.1 Separate Analysis and Proof Directories

There is no prescribed method for managing proofs within the SPARK approach. The directory structure illustrated in figure 6 was employed for this particular project.

```
|-Subsystem_1\
|      |-package_1\
|      |      |-Analysis\
|      |      |-Proof\
|      |
|      | ...
|      |
|      |-package_A\
|             |-Analysis\
|             |-Proof\
|
| ...
|
|-Subsystem_B\
```

Figure 6: Directory structure for recording proof process

The main feature of the directory is separating the user effort in a proof directory and the automated effort in an analysis directory. Such an approach is taken to store the valuable user effort away from the frequently modified analysis directories.

4.1.2 Good Proof Document Replaces Proof Log

It was commented that 'the command file [SPADE-PC-script] is just as important as the proof log'. The proof log is a human readable account of the actions taken during a SPADE-PC session.

As considered in detail above, the executable proof document is important, and should be generated and recorded by NuSPADE. However, little thought was given to also recording a similar proof log file. It is hoped that the executable proof document generated should also be suitable for human reading. If it is desired to pretty print a particular proof then perhaps a new program might take as input an executable proof document and output the same information in a more readable form.

4.1.3 Proof Review Files

Proof review files (`*.PRV`) are a relatively new extension to SPADE-PC. They allow the explicit recording of informal proofs so that they are detected and appropriately processed by the SPARK tools.

4.2 Observations

A number of important observations can be made about the operation of SPADE-PC in view of managing proofs:

- **VC directory tree as working area -** The EXAMINER typically generates VCs inside a a directory tree, with a separate directory for each layer of program scope and a separate file for each subprogram. A SPADE-PC session involves trying to discharge the VCs in a `*.VCG` file (Or `*.SIV` file after simplification). During this session SPADE-PC will save proof information, such as the proof log (`*.PLG`) or SPADE-PC-script (`*.CMD`), into the current directory within the VC directory tree. The POGS tool gathers its proof status information from the various files held within the VC directory tree.

Thus the VC directory tree is the working area for performing SPARK proofs. Further, this working area is constantly changing as proof progresses. This is essentially the analysis directory described in the user perspective.

A historical note...
Prior to hearing the users perspective a very similar separation between analysis and user proof had already been formulated for THE-CORPUS and within NuSPADE. The driving force behind such plans was to record proofs, including generated lemmas and proofs of these lemmas, such that they could be automatically repeated.
A feature of SPARK is a multitude of systems interacting via numerous files. However, there is no SPARK Integrated Development Environment (IDE) program that neatly combines these tools. In the meantime manually creating directory structures and formulating some consistent use of these directories is a natural alternative.

- **VCG isolation allows for parallel proof -** The analysis of a typical program will result in a number of `*.VCG` files to prove. Each SPADE-PC session is specific to a particular `*.VCG` file. Thus it is possible for multiple SPADE-PC sessions to be launched in parallel to tackle different `*.VCG` files of a program proof.

  This might be implemented by giving users shared access to the VC directory tree. These users could launch separate SPADE-PC sessions to tackle different `*.VCG` files at the same time.

  This becomes more problematic if the code is likely to be modified. Code or annotation modifications, especially to pre or post conditions, can affect a number of `*.VCG` files. Thus, while the `*.VCG` files can be proved in parallel, they must be generated in serial. Note that a similar concept was raised in CS-note 34, to preserve the consistency of VCs after code changes.

- **Definitions, properties and lemmas -** SPADE-PC has no explicit notion of which rules are definitions and which are properties. However, it is common to see this separation presented in SPADE-PC rule files, and other theorem provers in general. Lemmas are foreign to SPADE-PC, but can be thought of as properties which are automatically generated.

- **Proof of rules -** As automatically generated by NuSPADE, lemmas should have supporting proofs. However, as provided by SPADE-PC users, definitions and properties need not have supporting proofs.

- **Using the source -** Currently there is no direct link between the VCs in the VC directory tree and their corresponding source file. The structure of the VC directory tree is based on the code and each group of VCs have comments describing the paths they take between assertions and at which line numbers. However, there is no straight forward way to identify the source code for the whole project or which source links to which VC.

  This is important as the source code can provide useful information in tackling a proof. In the weakest use a user can gain insights about what a VC means from scanning its corresponding source code. In a stronger use, the code can subjected to automated analyses to generate information that later assists in an automated proof.

- **Allow global lemmas -** There is the potential for a proof of one VC to inspire a lemma that helps prove another VC. By adopting centralised proof management lemmas can become available across entire project, allowing for any potential reuse to be exploited.

- **Be SPADE-PC and NuSPADE -** To support POGS the SPADE-PC part of NuSPADE must act within the appropriate directory of the VC directory tree for the `*.VCG` file being tackled. Supporting POGS is an example of how SPADE-PC must not just be kept intact, but how the environment within which SPADE-PC operates must also be kept intact. Such behaviour can be achieved by ensuring that SPADE-PC is started in the appropriate directory.

4.3 Outline a Project Directory

A possible project directory is outlined in 7. Its various components are summarised below:

- **System code -** The complete source code of the system. This may be a large directory tree of code.

- **VC directory tree -** The VC directory tree as generated by the EXAMINER and modified by the various SPARK tools.
- **Project directory -** All of the user generated content is recorded in this directory, separate to the VC directory tree.
- **Project file -** The project file is accessed by NuSPADE. It lists the main components of the entire project.
- **Executable proof documents -** As described in §3 the intention is to record executable proof documents using tactics. For each `*.VCG` file its corresponding executable proof document[9] will be held here.
- **Definitions -** Each definition is stored in an `*.rul` file in this directory.
- **Properties -** Each unproved property is stored in an `*.rul` file in this directory.
- **Lemmas -** Each proved property is stored in this directory. The property is held in an `*.rul` for processing by SPADE-PC, with the proof being recorded separately in a suitable form[10].

```
|-system_code\
|-vc_directory_tree\
|-project_directory\
         |-project_file
         |-executable_proof_documents\
         |        |-vc_1\
         |        |
         |        | ...
         |        |
         |        |-vc_A\
         |-definitions\
         |        |-definition_1\
         |        |
         |        | ...
         |        |
         |        |-definition_B\
         |-properties\
         |        |-property_1\
         |        |
         |        | ...
         |        |
         |        |-property_C\
         |-lemmas\
                  |-lemma_1\
                  |        |-proof_of_lemma\
                  |
                  | ...
                  |
                  |-lemma_D\
                           |-proof_of_lemma\
```

Figure 7: A project directory structure

4.4 Propose: Key Features Only

There is great scope for enhancing project management within NuSPADE. However, the majority of such changes are not essential. Only where it is felt that project management will be directly advantageous for the main research should it be explored. For example, it is likely that accessing the source code will be an important factor in boosting automation. Further, to demonstrate the practical applicability of NuSPADE some project management features will be required.

[9] This may not be a single file. It may more organised to record separate executable proof documents for each individual VC.
[10] Ideally the lemmas should be proved as if a regular VC.

5 Polarity

All expressions in SPADE-PC have a logical polarity, positive (+), negative (−) or zero (0). All hypothesis have a − polarity and all conclusions a + polarity. Thus a goal has the form of (1).

$$Hypotheses^{-} \vdash Conclusions^{+} \tag{1}$$

Within each hypothesis and conclusion the polarity can change. Depending on the presence of negation ($\neg$), either explicit or within an implication ($(A \to B) \Leftrightarrow ((\neg A) \vee B)$), the polarity will switch. Polarity is significant where dealing with rewrite rules derived from implications.

For example, if it is known that (2), then this allows entire hypotheses[11] to be rewritten using (3) and entire conclusions using (4).

Given:	$(A \wedge B) \to B$	(2)
Rewrite whole hypothesis:	$(A \wedge B) \Rightarrow B$	(3)
Rewrite whole conclusion:	$B \Rightarrow (A \wedge B)$	(4)

This example is only valid as it is known that hypothesis are of − polarity and conclusions + polarity. The underlying principle is that − polarity subterms must be rewritten using implication rules left-to-right and + polarity subterms must be rewritten right-to-left.

5.1 Dealing With Polarity

Logically implication rewrite rules can be applied to subterms of either hypotheses or conclusions, so long as the appropriately directed rule is used based on polarity. This was considered in [1] to allow the full and correct use of rewrite rules derived from implications within rippling. The approach taken involved adding additional polarity annotations to wave rules and to the annotated goal. This was implemented by making slight modifications to the difference matching algorithm centre to the operation of rippling.

5.2 Propose: Deal With as Required

As noted above, the problems with polarity only arise where wanting to rewrite subterms with implication derived rewrite rules. Where rewriting whole expressions the distinction between hypothesis and conclusions conveys all the polarity information required. It is fairly uncommon to require subterm rewriting with implication derived rules. Thus the polarity issue can be resolved by accepting whole expression rewriting with implication derived rules as a limitation of SPADE-PP.

If it becomes apparent that subterm rewriting with implication derived rules is desired then the existing strategies outlined in [1] provide concrete details on how this might be implemented.

A historical note...
Polarity is an important issue very obviously missing from CS-note 8. It does not impact on this notes various schemes proposed to deal with rules within SPADE-PC. However, it does make a few of the tables in error[a].

[a] In section 2.4 the heading 'Rewriting Subterm Of' is incorrectly applied to both equation based and implication based rewrite rules.

References

[1] S. Negrete and A. Smaill. Guiding proof search in logical frameworks with rippling, 1995.

[2] K. Slind, M. Gordon, R. Boulton, and A. Bundy. System description: An interface between CLAM and HOL. volume 1421, pages 134–138, Lindau, Germany, July 1998. Springer. Earlier version available from Edinburgh as DAI Research Paper 885.

[3] P. Thornley (BAE Systems). A user's view of proof of exception freedom, October 16th 2002.

[11] Technically hypothesis are not *rewritten* using implication rewrite rules, as the old hypothesis is not deleted. Rather a new hypothesis is introduced that contains the rewritten expression. However, the distinction is generally not important at the rippling or planning level and thus the concept of rewriting remains.