CS-Note* 20

Bill J. Ellis
September 12, 2002

Raising NuSPADE Issues (Volume 2)

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 introduction.

2 Rule Organisation

A more structured form of organisation is required from CLAM to manage the different rule categories and forms envisaged for NuSPADE.

2.1 CLAM Rule Organisation

Currently CLAM supports many different forms of theorems. These include equations, definitions, schemes and lemmas. However, all of these are organised in a single rule category. They are all first order, CLAM provided rules. Note that lemmas might have been seen as another rule category. However, where a lemma is generated, it is stored as a provided rule and subsequently loaded into the CLAM environment like any other provided rule.

2.2 Rules in NuSPADE

Initial proof ideas in the development of NuSPADE suggest that the following categories of rule will be introduced. Although not all of these may be expected for the first prototype, they might be expected in future prototypes:

- **SPADE rules -** During a SPADE session a number of rules will be available. These include the built-in base rules and any problem specific user loaded rules. This category holds the formula equivalent of the rules available inside SPADE. A discussion of these rules is in CS-note 8.
- **Lemmas -** During a planning effort inside CLAM new rules may be proposed and subsequently proved. These new rules are lemmas. This category holds the formula of these generated lemmas.
- **First order rules -** Unlike SPADE, CLAM can deal more effectively with badly behaved rules. Thus, there is a potential collection of useful rules that are legal in SPADE but not made available for practical reasons. This category holds the formula equivalent of these rules, chosen to provide a richer but sound logic in CLAM.

  These rules are likely to have been previously proved true using the SPADE base rules. Note that the difference between these rules and lemmas is that the lemmas are proposed and proved dynamically while these are always available with no further proof required.
- **Higher order rules -** Rules that are defined in terms of a higher logic are very difficult to use directly in CLAM or SPADE. However, rules defined in this manner can be quite powerful and lead to a more elegant natural proof. Instead of directly using the higher order rules, it is possible to detect that a higher order rule might be applicable and generate a suitable first order rule. This category holds the formula equivalent of these higher order rules.

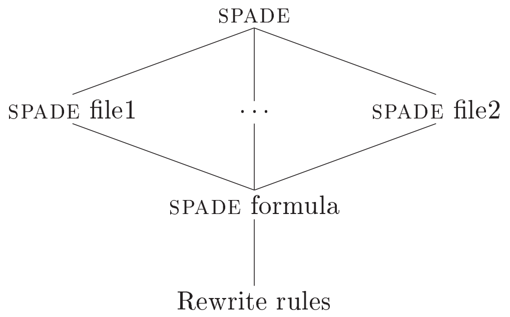
  For each higher order rule a higher order proof is expected to exist. By meta-variable instantiation this proof can be used to justify the first order rules generated.

These rules can be used as the basis for generating rewrite rules. The only exception is higher order rules which are the basis for first order rules before rewrite rules. These can be also be placed in different categories:

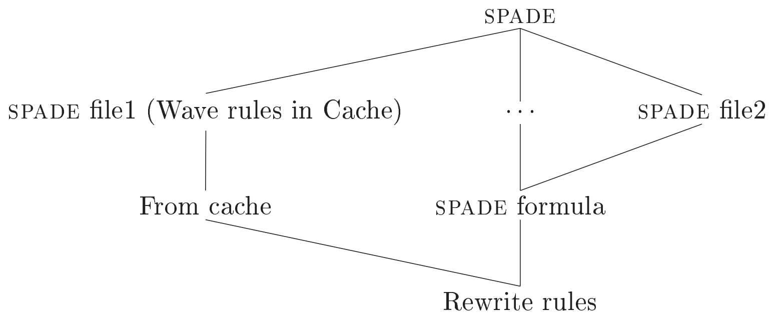
- **Plain rewrite rules -** Plain rewrite rules can be thought of as annotated rewrite rules that happen to have no annotation. This view means that plain rewrite rules are an alternative kind of annotated rule, rather than the parent of all annotated rules.
- **Wave rules -** Wave rules are an annotated from of rewrite rules, used in rippling.
- **Other annotated rules -** There is scope for alternative annotated rewrite rules to be introduced. It is not the case that plain rewrite rules or strict wave rules cater for all situations expected to be encountered in NuSPADE.

The categories considered above represent initial proposals, closely linked to the theorem proving problems of NuSPADE. However, further categories may be proposed to incorporate more pragmatic information.

- SPADE **Files -** Each SPADE rule will come from a SPADE rule file. It my be helpful to also record this SPADE rule hierarchy information. For example, the following categorys might be used:



- **Cache -** The process of generating wave rules from rule formulas is expensive enough to warrant caching and reusing previously generated wave rules. This is only possible where it is known that the rules generated previously are valid for this session. This is likely to always be true for the SPADE built-in rules. However, identifying these rules will require knowing the SPADE rule file information. This is provided through the extra classification structure proposed above. Instead of hiding that caching has taken place, it could be explicitly recorded by adding new classifications. This helps avoid inconsistencies if caching is ever used to also avoid the transportation of SPADE formulas.



2.3 Locating Rule Proofs

Associated with each rule their should eventually be a corresponding proof. Forcing that each rule must be proved prior to its use is unnecessarily restrictive inside the PLANNER. Of course, this is not the same for SPADE, where all rules are proved prior to entering the system.

This proof gives sufficient information such that a rule can be used soundly within SPADE. The location and nature of this proof can vary depending on the kind of rules being considered:

- SPADE **rules -** Rules imported from SPADE are assumed to be true. As the rule already exists in SPADE the only location information required to use the rule is its SPADE name.

- **Lemmas -** Lemmas are planned within the proof planner. To ensure flexibility, lemmas can be used prior to finding a proof plan for the lemmas. As proof planning is strictly not sound, even lemmas with plans may not be true. The location information for each lemma is the lemmas proof plan. This plan must be successfully executed within SPADE prior to using the lemma.

- **First order rules -** First order rules should have a supporting proof. The location information will show where to find this proof. The proof must still be successfully executed within SPADE prior to using the rule. However, unlike for planed lemmas, these are always expected to be successful.

- **Higher order rules -** Higher order rules are similar to first order rules. The rule used in SPADE must be first order, thus the location information should refer to a first order proof. This will probably be generated alongside the first order instantiation of the higher order rule. The proof must still be successfully executed within SPADE prior to using the rule. However, unlike for planed lemmas, these are always expected to be successful.

2.4 Propose: Entity Based Rule Organisation

The current CLAM rule organisation is not sufficient to deal with this rule information in an organised manner. A flexible rule organisation structure based on discrete rule entities is proposed. Each rule related item is placed in an entry. These entities are arranged to express all of the rule related information in a flexible tree structure. The general form of an entity is shown in Figure 1.

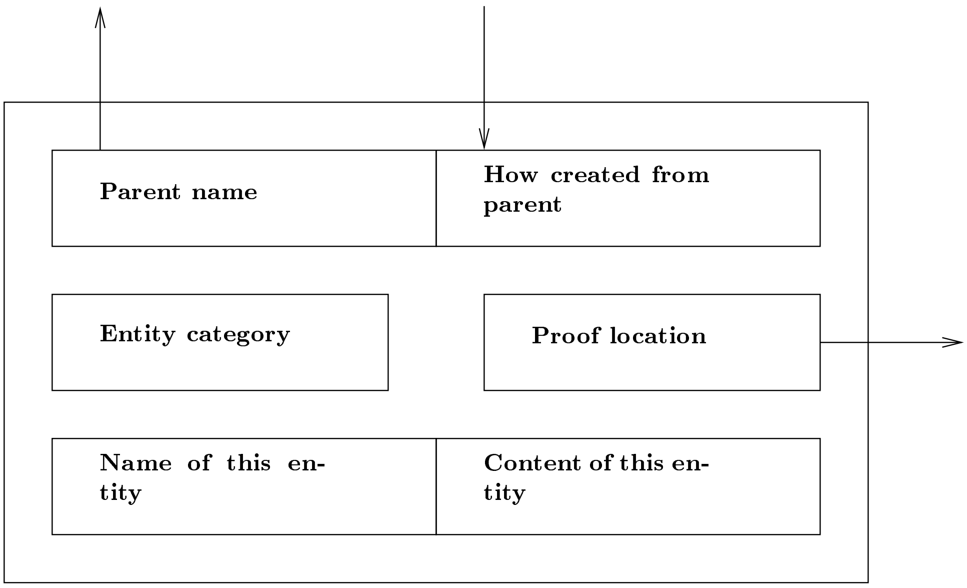


Figure 1: CLAM rule organisation structure

- **Parent name -** The name of the item above which was used to generate this item.
- **How created from parent -** How an item from the level above was transformed to generate this item. For example, in the case of rewrite rules this may specify the direction in which the rule was used. In the case of first order formula based on higher order formula, this may specify how the higher order formula was instanciated.
- **Entity category -** The name for this category of entity. For example, "Formula" may be the category name for all entities holding rule formulas.
- **Proof location -** The location of a proof for this entity. Significantly, the proof may be completed long after the introduction of the rule entity. For example, for lemmas this will contain the location of the lemmas plan.
- **Name of this entity -** The name for a particular item of this entity category. For example, if the entity category is "Formula", then the name for this entity might be "generated-lemma-1".
- **Content of this entity -** A Prolog term that describes or links to the content for this entity. Describing a higher order rule using first order Prolog may be unnatural. For these cases the Prolog term can link to (via a predicate or file name) where the actual information is held.

Note that not all entities logically have all of these entries. In general where an entity can not have a value it should be assigned a null value. The identified special cases are listed below:

- **Root has no parent -** The top most entity does not have a parent. Its parent name and how it was created entries should be null.
- **Entities not based on parent -** There are entities which are not based on its parent in any manner. In these cases its parent name entry should be null.
- **Entities with no content -** There are entities which exist solely to structure the rule classification. These do not have any content, so their content entries should be null.

- **Entities with no proof -** There are entities which do not have an associated proof, for example structural entities such as SPADE-base-rules. For these the proof location entry should be null.
- **Entities with inherited proof -** There are entities which are based on previously proved components. For example, formulas have a proof, yet may be decomposed eventually into several wave rules. The wave rules are not proved, they assume the formula was originally true. Rather than having null these entities might specify proof-by-parent.

2.4.1 Demonstration

To further clarify the nature of the rule organisation a short practical demonstration shall be considered. Each entity will be stored in a predicate of the form:

```
ruleEntity(EntityName, EntityCategory, ParentName, HowCreated, ProofLocation, EntityContent).
```

For completeness, introduce a root entity:

```
ruleEntity(root-of-all-entities, root, none, none, none, none).
```

Add an entity for all SPADE files:

```
ruleEntity(all-spade-files, spade-files, root, none, none, none).
```

Add an entity for the SPADE file `ARITH.RUL`:

```
ruleEntity(spade-file-arith, spade-file, all-spade-files, none, none, none).
```

Now the structure is available to add the formula in `ARITH.RUL`. Adding the first entry `arith(1): X*1 may_be_replaced_by X.` gives:

```
ruleEntity(arith-1, formula, spade-file-arith, none, location(in-spade,[arith(1)]), X*1=X).
```

From this formula two rewrite rules can be added:

```
ruleEntity(rewrite-rule-1, rewrite-rule, arith-1, left-to-right, see-parent, X*1=>X).
```

```
ruleEntity(rewrite-rule-2, rewrite-rule, arith-1, right-to-left, see-parent, X=>X*1).
```

From these rewrite rules wave rules can be added:

```
ruleEntity(wave-1, wave-rule, rewrite-rule-1, wave-annotated, see-parent, ‘‘{X}*1’’=>X).
```

2.4.2 Note on Implementation

CLAM currently loads all external information, including rules, methods and critics, through the same library access mechanism. This is primarily implemented in `library.pl`. This imposes a light organisational structure which is quite different to the hierarchy structure presented above.

In the short term it should be possible to continue using the existing CLAM structure. The exact details of such a short term solution are not formalised here. It is likely this will include include some mechanism to generate files containing SPADE rules inside the CLAM library directory.

However, as integration with SPADE increases it will become necessary to adopt something more structured as discussed above. Implementation of this will essentially be replacing the existing CLAM library mechanisms. While some of the low level parts of the CLAM library code may be salvaged, it is not expected that any of the rest will.

3 Planing Organisation

The planning of VCs is seen as being different to the planning of regular mathematical conjectures.

3.1 CLAM Planning Weaknesses

As noted in CS-note 9 CLAM plans a single theorem in isolation. There are a number of weaknesses in this approach:

- **Many individual conclusions -** The nature of VCs is slightly different to the normal mathematical inspired conjectures. Each VC has a list of hypothesis and a list of conclusions ($[H^1, \ldots H^n] \overset{?}{\vdash} [C^1, \ldots C^m]$). These can be thought of as a single hypothesis and conclusion as a conjunction of the different parts ($(H^1 \wedge \ldots H^n) \overset{?}{\vdash} (C^1 \wedge \ldots C^m)$).

  Typically the hypothesis are seen as being quite distinct, and thus represented in the former list form. Conclusions on the other hand are typically seen as being a single expression, forcing the later conjunction form. This is the view adopted by CLAM.

  However, the conclusions of VCs are individual. They are generated from user annotations that are likely to combine a number of related but quite different statements. For example, consider the invariant of the polish flag problem discussed in CS-note 10:

  ```
  --# assert Flag'First<=I and
  --#        J<=(Flag'Last+1) and
  --#        I<=J and
  --# (for all Q in Integer range Flag'First..(I-1) => (Flag(Q)=Red)) and
  --# (for all R in Integer range J..Flag'Last => (Flag(R)=White));
  ```

  Although this is a single conjunction expression, the individual conjuncts are quite distinct. That `I<=J` is different to `(for all R in Integer range J..Flag'Last => (Flag(R)=White))`.

- **Shared conclusion insight -** As noted above, although the individual conjuncts of a conclusion may be quite distinct they are likely to be related. Conducting isolated plans of each conclusion would prevent exploiting shared conclusion information.

  Finally, though not strictly planning related, SPADE deals with conclusions separately. Mirroring the general behaviour of SPADE in the PLANNER is likely to make integration more straight forward.

- **Shared VC insight -** As noted in CS-note 9 section 3.1.2, the individual VCs of a procedure are distinct yet related to each other. Conducting isolated plans of each VC prevents exploiting the possible gains from information present in other VCs.

- **Proved conclusions as new hypothesis -** Once a conclusion is proved in a particular VC, it can be made available as a hypothesis in any remaining conclusions of the VC. This extra hypothesis has the potential to advance otherwise stuck conclusion proofs.

  What may be less apparent is that conjoining all of the conclusions of a VC together into a single conclusion makes this less straight forward to implement. The various parts of the single large conclusion will become scattered across a proof plan tree. Detecting the proof of a top level conjunct and making this available as a hypothesis to all other branches of the proof plan is likely to involve editing the hypothesis available at each node in the proof tree.

  By tackling one conclusion at a time CLAM does not allow for using proved conclusions in this manner.

- **Generated lemmas as global rules -** Lemmas are generated as a result of patching the proof of a conclusion in a VC. However, once proved, the generated lemma can be used as rule on any conclusion in any VC.

  This requires that lemmas are proved outside the context of any VC specific hypothesis. As conditional lemmas are supported, it may still be the case that a lemma is fairly focused for a given VC. However, the VCs generated by CLAM are rarely so constrained.

  Currently CLAM plans lemmas external to the current problem, exactly as prescribed above. Once generated the lemmas are directly added to the available rules. Thus even though tackling one conclusion at a time, CLAM does allow generated lemmas to be used as rules in future conclusion proofs.

- **About shared procedure insight -** Tackling the conclusions of a VC together brings benefits of shared insight and the ability to use proved conclusions as new hypothesis. Tackling the VCs of a procedure together brings benefits of shared insight.

  It considered unlikely that there are similarly strong benefits in tackling the VCs from different procedures together. Nevertheless, absence of strong benefits does not rule out the possibility of any benefits. Although not contemplated for the current implementation, shared procedure insight is a potential future refinement.

  Note that the isolation of each procedure proof attempt is key to supporting parallel proof effort in a project. This could still be supported by tackling one procedure at a time and allowing read only access to other procedures.

3.2 Propose: Many VC and Many Conclusion Planning

Many VCs from a procedure should be planned together. Many of the conclusions within each planned VC should be separated into individual plans and tackled together. Each lemma should be planned outside the context of any VC hypothesis. This leads to the planning structure presented in Figure 2.

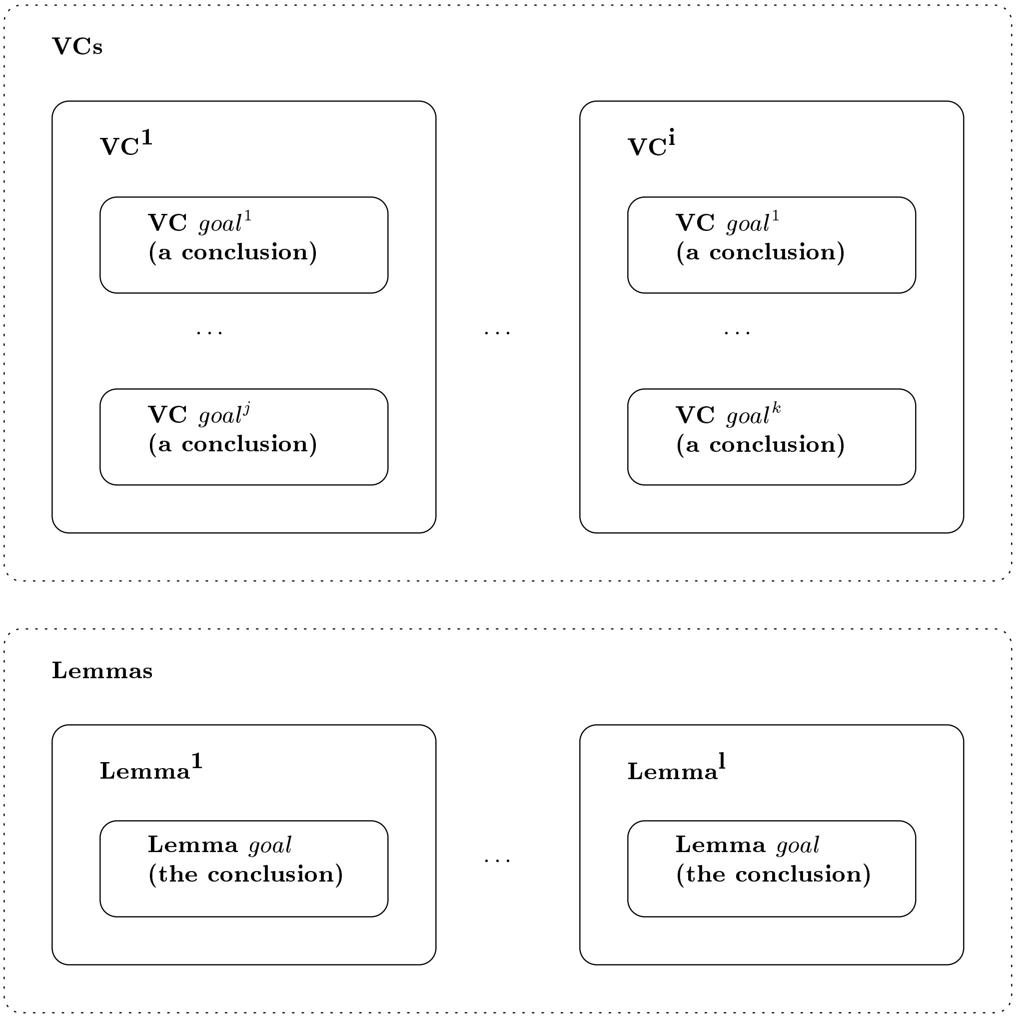


Figure 2: CLAM planning Organisation

From this proposal the following points are made:

- **Many is not necessarily all -** There may be occasions where tackling all of the VCs in a procedure or all of the conclusions in a VC is unlikely to bring shared insight. Further, there may be practical resource constraints preventing the PLANNER from performing such complete plans. Finally, the user may prefer to use the PLANNER on a particular VC or even a particular conclusion in this VC.

It is therefore advantageous to allow some flexibility in the numbers of VCs and the numbers of their conclusions that are tackled in the PLANNER.

- **Require names -** It is required to identify the plans inside the PLANNER for their eventual export into SPADE. Every plan must be associated with a SPADE VC number and SPADE conclusion number.

3.2.1 Note on Implementation

Although CLAM tackles one problem at a time, it does allows for multiple plans as seen for generated lemmas. The main problem is that CLAM implements this using a recursive planning attempt. As outlined in CS-note 9 a more blackboard oriented architecture with non-recursive planners is a more useful foundation on which to build the PLANNER.

4 Integrating with Other Systems

Existing systems may be integrated within NuSPADE. Systems that might be considered for integration include constraint solvers, algebraic tools and resolution theorem provers. Significantly, existing implementations are generally highly tuned for their given problem domain. Integrating these in NuSPADE will give access to state of the art implementations. Further, as these systems might otherwise have been implemented from scratch in NuSPADE, it allows NuSPADE development to focus on more unique SPARK based concepts.

4.1 External Components in Proof Planning

CLAM does not make use of any external systems. However, an interface has been constructed between CLAM and HOL [4, 1]. This raised many relevant points about the integration of external systems in proof planing. The ΩMEGA proof planner uses a math bus to communicate between different systems in a coherent manner [2]. More significantly, the ΩMEGA project has also considered integrating external systems directly into proof planning [3].

All of these systems have integrated external systems into the proof planning paradigm by treating the external system as a tactic. As usual this tactic will have a corresponding method for use during proof planning.

The main issues lie in translating the efforts of the external system (new tactic) into object level proof commands. For example, in CLAM it would be necessary to translate the actions of the external system into OYSTER proof commands and in NuSPADE it would be necessary to translate into SPADE-scripts. A number of different strategies can be taken to achieve this:

- **External system only makes suggestions -** An external system may generate a suggestion for the proof plan, rather than complete a section of the proof. For example, an external system might provide a witness for an existential proof. In these cases there is no need for translation. The external systems are adding to the information available to the planner, rather than implementing proof actions.
- **External system generates expansive proof -** An external system may generate a sequence of proof steps to describe its actions. These could be translated into planning steps (then tactics then object-level commands as usual) or directly into object-level commands. In either of these cases the proof checker need not even be aware of the external system.

  Unfortunately, such an idealised situation is unlikely. The external systems under consideration tend to solve problems in a notation and manner that is radically different to that seen in planning. Therefore converting to planning steps or object-level commands may be somewhat difficult.
- **Rely on external systems -** At the point where an external system completes a proof it is accepted at the object-level as being true. This might translate to a special completion object-level command which discharges the current goal under the knowledge that it has been successfully proved elsewhere. Such an approach should be viewed with caution. A number of potential issues are considered below:
  - **User may exploit this command -** It might be possible for the user to abuse the completion command. For example, a user presented with a trivial problem might use this command to avoid going through the obvious proof. However, it is often only through attempting such proofs that their real nature becomes apparent. It is quite possible the user could trivially see as true what is actually subtlety false. To avoid such problems the object-level system could hide the completion command from the user.
  - **External system may be in error -** Even where the completion command may only be used by external systems, the external systems themselves may be in error. For example, the HOL theorem prover only accepts as theorems that which it has proved itself. It does not trust anything provided outside of the HOL environment. Such a xenophobic approach is a key feature for ensuring soundness in HOL. This soundness issue might be overcome by formally proving the external system. Such a proof effort is likely to be quite large and is considered unlikely to be available in general.
  - **Proof planning is potentially unsound -** A proof planner may find a plan that can not be executed in practise. Even if an external system is proved correct it may be used in an unsound context during planning. Relying on the execution of an external system during proof planning is therefore unsound. This issue could be overcome by integrating the external system within the object-level proof checker. By extending the proof checker in this manner the entire plan can be executed inside the proof checker.

4.2 Propose: External Systems as Tactics and Per-system Evaluation

It is proposed that external systems are treated as tactics. This integrates well with proof planning and has been successfully adopted in other systems. Further, this allows for the wide range of potential systems to be considered in the same manner.

It is also proposed that each system integrated is considered on a per-system basis. The best strategy for each system should minimise the bypassing of the proof checker as much as possible.

References

[1] R. Boulton, K. Slind, A. Bundy, and M. Gordon. An interface between CLAM and HOL. In J. Grundy and M. Newey, editors, *Proceedings of the 11th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'98)*, volume 1479 of *Lecture Notes in Computer Science*, pages 87–104, Canberra, Australia, September/October 1998. Springer.

[2] Andreas Franke and Michael Kohlhase. System description: Mathweb, an agent-based communication layer for distributed automated theorem proving.

[3] E. Melis and V. Sorge. Employing external reasoners in proof planning, 1999.

[4] K. Slind, M. Gordon, R. Boulton, and A. Bundy. System description: An interface between CLAM and HOL. In C. Kirchner and H. Kirchner, editors, *Proceedings of the Fifteenth International Conference on Automated Deduction (CADE-15)*, volume 1421 of *Lecture Notes in Artificial Intelligence*, pages 134–138, Lindau, Germany, July 1998. Springer.