Bill J. Ellis
March 18, 2002

Proof Planning ($\begin{smallmatrix}\textbf{Bluffers guide}\\ \textbf{Architectures}\end{smallmatrix}$) and Prototyping NuSPADE

1 Introduction

In CS-note 6 an architecture was presented for building NuSPADE. Since writing this note the architecture has been elaborated in view of actually building the system. During this process it was gradually accepted that CLAM in its current form may not bring all of the proof planning power sought. However, extending, modifying or otherwise making significant changes to CLAM is a large project in itself. Thus it is proposed that NuSPADE should be prototyped, making gradual changes over time.

2 Proof Planing

CLAM is a proof planner with support for critics. It is helpful to consider the key components of such a proof planner before considering the architecture of CLAM.

2.1 Components Of A Proof Planner

- **Goal (Verification Condition (VC))**
  The goal is the problem being tackled. This is a theorem, specified as a number of hypotheses and a single conclusion. The desired result is to prove that, assuming the hypotheses are true, the conclusion is true.

  Note that a goal with multiple named conclusions $(H1, H2, \ldots Hi \vdash C1, C2, \ldots, Cj)$ can be expressed as a goal with a single conclusion by presenting the conclusions as a single conjunction $(H1, H2, \ldots Hi \vdash C1 \wedge C2 \ldots \wedge Cj)$. Alternatively, the goal may be expressed as many separate goals, one for each conclusion $((H1, H2, \ldots Hi \vdash C1), (H1, H2, \ldots Hi \vdash C2), \ldots, (H1, H2, \ldots Hi \vdash Cn))$.

  > The goal is an unproved theorem. It is a specification of something to be built, without any indication of how to do so. A customer may contact a builder, requesting a house to be built, specifying key details such as the price and number of rooms. They do not describe detailed architecture plans and how to mix the concrete.

- **Rules (Theorems and Axioms)**
  Rules are formula that are true inside the application environment in which a goal is described. For example, if the application environment allows for integer arithmetic then the rule $1 + 1 = 2$ might be expected to be available. Note that rules are specific to a given application, not a particular goal. For example, another application may use the number 1 to represent true, and + for boolean or. In this environment the arithmetic rule is invalid.

  Rules provide additional information that can be used in proving a goal. They may allow a goal to be incrementally transformed to a point where the truth of the goal can be determined.

  Note that a goal's hypotheses will take the same form as rules. These may be assumed true inside the corresponding goal. However, these are not necessarily true inside the entire application environment.

  > The rules provide the logic allowing the gradual transformation of the goal. These are true for the application environment. Assuming the application environment is Earth, and not say Outer Space, they may be thought of as complete manufacture guides for building regular tools such as hammers, screwdrivers and cement mixers. The hypotheses of the goal may introduce other rules available for this particular project. These need not be true for the whole application environment, and might included tools such as magic wands and flying carpets.

*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 CLAM-SPARK project (EPSRC GR/R24081).

- **Rewrite rules**
  Rules are often expressed in a general sense using meta-variables. For example, the rule $A + (B + C) = (A + B) + C$ can be applied to an infinite number of arithmetic expressions. The rules may also be expressed using other logical constructs, such as implication or equivalence.

  To uniformly cater for these different rules a single general form is introduced as rewrite rules. Every rewrite rule is based on a rule. They capture the logic of the rule in a practical mechanism that can be applied directly to the goal.

  > Rules are the logical basis for rewrite rules. While rules are the manufacture guides for building tools, rewrite rules are the actual physical tools that can be used.

- **Annotated Rules**
  Rewrite rules can often be applied to a goal in many different ways. The shear number of ways makes an exhaustive search very expensive and often impossible. It is necessary to constrain the application of rewrite rules and thus minimise the search space.

  This constraining is achieved by adding annotation to the rewrite rules. The annotation encodes a heuristic, expressing the context where the rules are expected to be more useful. For example, wave rules are annotated rewrite rules that ensure the rules are only applied where they are structure preserving. This is based on a heuristic that keeping the structure of expressions the same is more likely to lead to a proof in certain kind of problems.

  > Annotated rules describe when rewrite rules can be more effectively used. They are the manuals and training that say it is not good practise to hammer in nails with a screwdriver.

- **Tactic**
  There are only a few low level logical operations that may be performed on a goal. From experience with these low level operations common patterns begin to emerge. Tactics group together these patterns to provide more powerful blocks of logical operations. Where a tactic can be applied it is always logically sound.

  > The low level logic operations are fiddly little bits of sand and clay. The tactics group these together to form more practical bricks.

- **Tacticals**
  From experience with the use of tactics common patterns of tactic application begin to emerge. In the same manner that programming languages evolved the `case` statement to capture a common control structure, appropriate commands for tactic application have been created. For example, the tactical `THEN` can be used to combine a chain of tactics that are to be applied one after the other (`Do-A THEN Do-B THEN Do-C`). The tactical `ORELSE` can be used to provide a list of tactics to try until one is successful (`Try-A ORELSE Try-B ORELSE Try-C`).

  > The tactics are the practical bricks. These may be arranged in suitable patterns for building different parts of walls. Tacticals provide a clear mechanism for recording and repeating these patterns.

- **Methods**
  In tackling a proof particular strategies begin to emerge. These strategies may not be applicable for all proofs or even the majority of proofs. Therefore they are strictly heuristics. Methods capture the essence of each of these strategies. They record where the strategy is applicable, what effects it should have and when its application has been successful.

  Every method is strongly tied to tactics. It need not be the case that each method has a single corresponding tactic, however, for practical implementation reasons, this is often the case. This relationship between methods and tactics usually allows proof plans constructed using methods to be implemented using tactics.

  > A method captures rules of thumb in building a proof. These rules of thumb are strongly tied to real actions that can be taken. Building the walls before the ceiling is generally a good idea. Putting the television in a corner of a room is generally a good idea. Putting carpet in the kitchen is generally a really bad idea.

- **Methodicals**
  Methodicals are to methods what tacticals are to tactics. They provide a language for describing patterns of method applications. For example, the methodicals `THEN` and `OR` have similar behaviour to the tacticals `THEN` and `ORELSE` respectively.

  The benefits of methodicals are not as clear as for tacticals. Tacticals are beneficial as they provide a clean notation for presenting a tactic level proof to a theorem prover. Proof plans are created by searching through possible method applications. Once the proof plan is complete it will be converted to tacticals for execution in a theorem prover. There is no particular benefit in using methodicals in this process.

  Methodicals do have potential searching benefits, in acting as constrained mini-planners. For example, a tightly constrained search may be launched to prove a certain kind of goal by executing a previously thought out and tested methodical. Here methodicals are essentially providing a mechanism to allow the creation of new methods that are implemented in terms of existing methods.

  > A method captures rules of thumb in building a proof. Methodicals express patterns of method applications that can be performed. A section of a methodical for building a kitchen might take the form: paint walls `THEN` fit cupboards `THEN` (put in linoleum `ORELSE` put in floor tiles) THEN ... .

- **Critics**
  Given the heuristic nature of methods, it is not surprising that they may only be partially successful. However, where a method is partially successful another collection of strategies present themselves to complete the method application. These are captured in further heuristics called critics. Each critic will be associated with a particular method. They record a partially successful method state and a proposal for patching up the problem such that the remaining parts of the method can be applied.

  Although critics are method helpers, they are not light weight in operation. Critics can propose fairly radical solutions to complete a method. For example, critics may propose a new goal (lemma) that when shown true would introduce rewrite rules that would allow the method to be fully applied. A critic may even propose abandoning the current plan, returning to the original goal to perform a well placed generalisation.

  > A critic captures rules of thumb used to complete the application of a method. When putting the television in the corner means the cables are not long enough, try the other corners of the room. If this does not fix the problem, get an extension cord. If this still does not work, sell up, and move to another house.

- **Planning space**
  The proof space is a tree, with a goal at the root node. At every node a number of logic operations are available, each advancing the goal by a tiny amount. Like the proof space, the planning space has a goal at the root node. However, now at each node a number of methods are available, each advancing the proof by a significant amount. The planning space is a more constrained tree than the proof space. There are fewer methods than low level logic operations and each method application is subject to tighter constraints.

  Though both the planning space and proof space have infinite branching points, a reasonable amount of theorem proving can be conducted in the planning space far more efficiently than in the proof space.

  There are a few possible actions that may be taken at each node in the planning space:

  – **Subgoal -** The node may be decomposed into n new nodes through the successful application of a method.
  – **Shown true -** The node can be shown to be true through a suitable truth detector method.
  – **Shown false -** The node can be shown to be false through a suitable false detector method.

  > The problem space shows progress towards proving a goal. It is the architects plans on which a partially designed house has been drafted.

- **Proof plan**
  Where the problem space has been reduced to a collection of true nodes it is a proof plan. Each application of a method can be translated into a tactic and each tactic into a logical operation. Thus the proof plan can be used to prove the goal.

> The proof plan shows how to prove the goal. It is the completed final draft of the architects plans.

- **Planner**
  The planner conducts a search of the problem space in pursuit of a proof. There may be many different planners to perform different search strategies. A few possible search strategies are:

  - **Depth first planner -** Descend until no methods apply then backtrack to the last choice point.
  - **Breadth first planner -** Examine every node at the current depth, before descending to the next depth.
  - **Iterative deepening planner -** This is a combination of depth and breadth first. It searches a layer of nodes to some given depth using depth first, but only tackles lower depths once this layer has been fully searched as if for breadth first.
  - **Best first -** Rate each node that is encountered on the basis of some heuristic. The most promising nodes have methods applied next.

  The relationship between methods and critics must be considered by the planner. While critics themselves are never applied to nodes they may help a method be applied to a node. A few different techniques for using critics are proposed:

  - **Critic first -** As soon as a partially applicable method is detected every critic associated with a method is attempted.
  - **Critic last -** Only once every node has been fully investigated for method application are critics considered. There is likely to be many partially applicable methods to investigate, so a suitable search strategy will need to be adopted.
  - **Mixed search -** The application of critics could be mixed with the application of methods. Critics could be applied when switching layers in iterative deepening. Critic application could be placed on the same queue as method application in best first.

  The combination of a method and critic search strategy is the planner.

  > The planner controls the search for a design of a solution. It makes use of methods (televisions go in corners), critics (move the television if that corner is no good), knowledge of rewrite rules (No screwdriver available?) to help find a proof plan (completed architects plans) that can be implemented using tactics (bricks) each made up of low level logic operations (sand and clay).

- **Meta-planner**
  The meta-planner plans the planner. It helps guide the planner in its search. Unlike the previous components, the meta-planner does not have a discrete mode of operation. The meta-planner is responsible for selecting which planner should be used. It is also responsible for which methods, critics and annotated rules should be made visible to this planner.

  In many ways the meta-planner is a manager. It makes decisions about the information that should be available and prioritises proof planning efforts.

  > The meta-planner is the manager. Looking over the architects firm, setting priorities, making resources available, and assigning tasks to the architects.

3 Proof Planning Architectures

3.1 CLAM Architecture

CLAM is an implementation of a proof planner, with support for critics. Importantly, CLAM has been implemented and shown to operate. It is a design that works. The architecture of CLAM is presented in Figure 1.

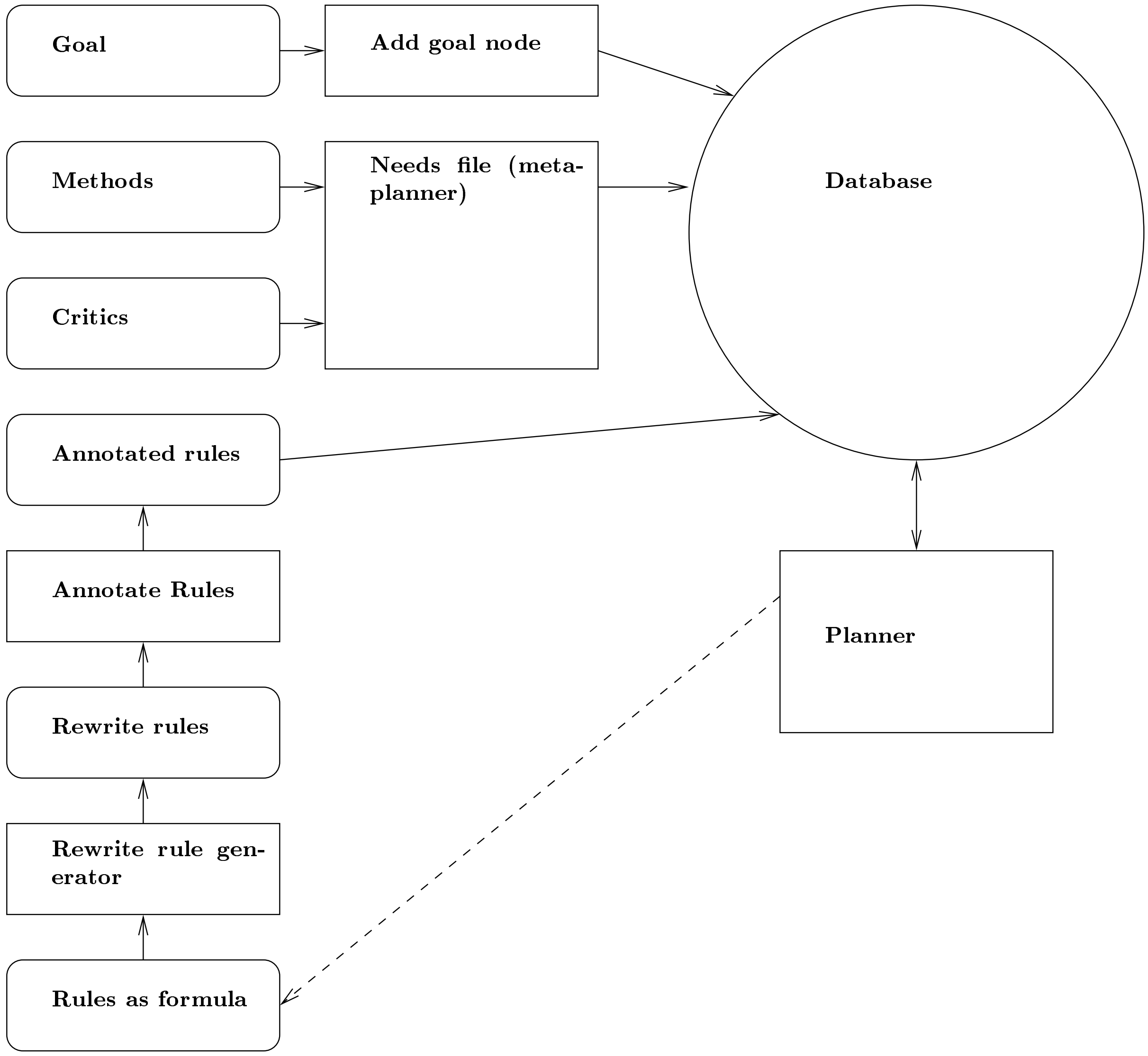

Figure 1: Current CLAM architecture

3.1.1 CLAM Architectural Observations

There are a few observations that can be made about the proof planning architecture in CLAM.

- **Needs File acts as part of meta-planner**
  There is a needs file that, based on the problem, loads selected methods and critics. This captures some of the behaviour expected from the meta-planner. However, the choices taken by the needs file have been hardwired for a number of known goals. Further, the needs file is only consulted once. There is no scope for refining the items loaded based on new knowledge.
- **Integrated Planners**
  There is support on CLAM for depth first and ittertive deepening planners. However, these are not loaded or configured as discrete components. They are bundled together with the rest of the system.
- **Planner mixed with control**
  The planner part of CLAM is tightly integrated with the rest of the system. There is not a separate controller component. Instead all control is bundled directly into the planner.
- **Plan once to termination**
  The planner is started on a goal and continues planning to termination. At termination either the planner finished successfully or not. During the course of planning new goals (lemmas) may be proposed and planned. However, these are tackled by a recursive call to the planner. Thus the success of the top level planning call is the success of the planning attempt. This means that it is not possible to pause the planning of a goal.
- **Planning goal in isolation**
  The nature of the planner means that each goal is planned in isolation of any other goals. In practise this means that a collection of n goals will be planned in serial.
- **Makes repeated use of conversion from formulas to annotated rules**
  There is a conversion procedure form formulas to annotated rules. This is used by the planner in initialisation to process globally true rules. However, the same procedure is also used to process any new formula (lemma) that are generated in the course of planning.

3.1.2 Limitations of this Architecture

The architecture of CLAM does bring some constraints that may limit the potential power of proof planning.

- **Cleanly adopt different search strategies**
  Although CLAM does have multiple planners they are not available as discrete components. Further, there is no explicit controller managing these planners during their search. Together, these mean that there is no clear entry point for cleanly implementing and adopting new search strategies.
- **Can not reason with multiple goals**
  CLAM tackles multiple goals by considering each is isolation, one after the other. By operating in this manner CLAM can not reason with multiple goals in the same planning attempt. The potential benefits in operating in this manner is shared insight, where the planning of one goal leads to additional knowledge that could help plan another goal.

  These potential benefits are only likely to occur in practise where the multiple goals being considered have some common structure and purpose. This occurs where a number of VCs are generated for the same section of code.

  A realistic example is a loop with a reasonable pre-condition and post-condition, but no invariant. The EXAMINER will automatically insert the weak invariant `true` when generating the VCs. This will manifest itself as trivial pre-condition to invariant and invariant to invariant VCs, but a very tricky invariant to post-condition VC. Looking at each of these VCs in isolation will probably result in a proof of the first two, and at best, a proof that the third VC can not be proved. Looking at these VCs together, the post-condition could be used to strengthen the invariant, and under these modifications a complete proof performed. The user would be prompted to use the refined invariant in their code.
- **Use external components**
  In CLAM the planner is also the main controller. While it may be possible to pause the planner and perform another task[1] it is not built for this purpose.

[1] In the same manner that recursion over the planner is "pausing" the current plan and starting a new plan.

A possible example is launching a model checker or some theorem proving component to gain extra insight during a planning attempt.

3.2 Blackboard Inspired Architecture

Blackboards are an idea from artificial intelligence. The blackboard is a container of information, that can be written to and read by a number of experts. The flexibility of the blackboard architecture and the clear separation between different experts makes it well suited to proof planning. A possible transformation of CLAM into a blackboard architecture is show in Figure 2.

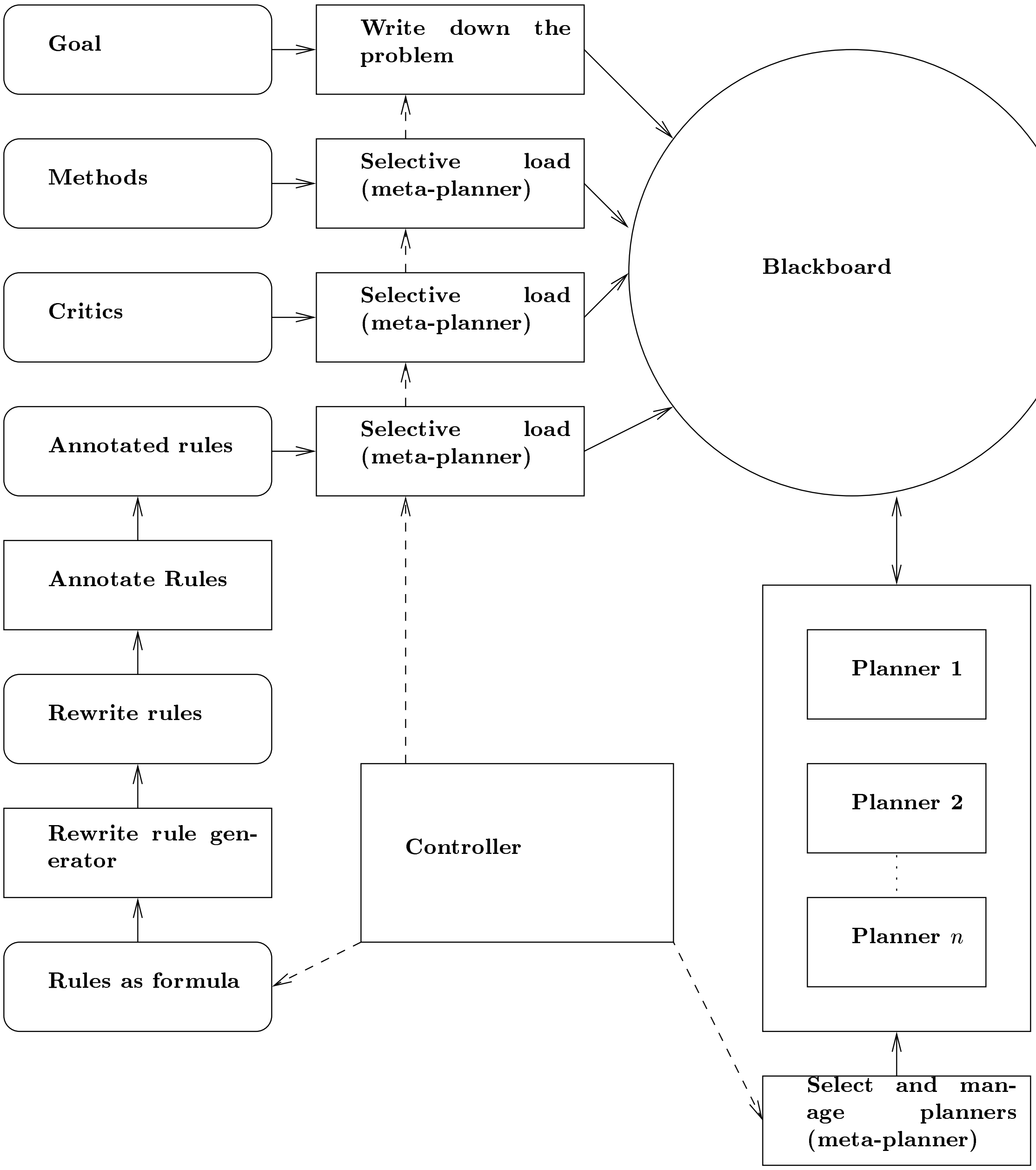

Figure 2: Blackboard proof planner with critics

3.2.1 Blackboard Architecture Observations

There are a few observations that can be made about the blackboard architecture of a proof planner.

- **There are multiple planners**

There is a number of planners, each implementing some different search strategy. Significantly, the planners will only plan.

- **There is a controller**
  There is a single top level controller for the current planning session. The success of the controller is the success of the planning attempt.

- **Meta-planning is distributed**
  The meta-planner is not resident in any one place. Instead there are a number of components that have meta-planning related skills. Together all of these components provide the meta-planning.

4 Prototyping

CLAM implements a particular search strategy for single goal planning with methods and critic patches. Although this is not the most flexible proof planning architecture, it is known to be applicable in many cases. This is quite sufficient for an early version of NuSPADE.

Thus for the first prototype only the necessary changes will be made to CLAM for integration with SPADE. The advantage of pursing this approach is reduced complexity and time in producing a working prototype. This can then be used to illustrate the direction of the project, and gain further insight, before embarking on any future developments to CLAM.

5 Conclusions

It is accepted that CLAM is not as flexible and powerful a proof planner as is possible. However, for the early stages of the project it is quite sufficient. Once the limitations of CLAM, particularly in the context of the problems expected inside NuSPADE, are better appreciated then more radical architectural restructuring may take place.