ERICA MELIS AND JÖRG H. SIEKMANN

CONCEPTS IN PROOF PLANNING

1. INTRODUCTION: THE PROOF PLANNING PARADIGM

Knowledge-based proof planning is a new paradigm in automated theorem proving. Its motivation partly comes from an increasing disillusionment by many researchers in the field of automated deduction, who feel that traditional techniques have been pushed to their limit and – as important as these developments have been in the past – will not be sufficient to reach the goal of a mathematical assistant system. Human mathematicians – among other differences – have a long term plan for a complex proof which is clearly not represented at the calculus-level of individual inference steps. Moreover a mathematician in a well established field has a whole battery of proof strategies, techniques, and tricks of the trade at his disposal.

How can these abilities be simulated on a machine? The idea is as follows: As opposed to traditional automated theorem proving (see, e.g., (Loveland, 1978; Chang and Lee, 1973) for standard monographs) which searches for a proof at the *object-level* of the underlying logic calculus, proof planning constructs a plan that represents a proof at a more abstract level. Here the representation of a proof, at least while it is developed, consists of a sequence of macro-operators, such as the application of a homomorphism, the expansion of a definition, the application of a lemma, some simplification, or the differentiation of a function. Each of these operators can in principle be expanded into a sequence of inference steps, say, in natural deduction (ND), and is therefore, in accord with tactical theorem proving (Gordon et al., 1979), called a tactic. Now if an individual tactic of this kind, which may also be a call to a traditional automated theorem prover, a fast decision procedure, or a computer algebra system, is extended by pre- and postconditions, then it is called a method and one can *plan* such a sequence of methods in the classical AI-planning paradigm. This was the central idea when proof planning was introduced in (Bundy, 1988).

A plan is a sequence of such methods, where the precondition of a method *matches* the postcondition of its predecessor in that sequence. This well-known view of a plan leads naturally to a new engine for automated theorem proving: a STRIPS-like planner (Fikes and Nilsson, 1971) – or any other planner – can be used to *plan* such a

sequence of methods that transforms the proof assumptions into the theorem. Standard heuristics and techniques from AI-planning can now be employed.

Moreover, proof planning provides means of *global search control* that correspond to mathematical intuition, as opposed to the local and syntactic search heuristics which are commonly used for search control in automated theorem proving.
The purpose of this paper is primarily a methodological discussion of *representational issues* in proof planning. This appears to be timely as in the different research groups different representations emerged from the actual practice of particular application domains. For instance, in $CLAM$ (Bundy et al., 1991) the control knowledge that consists of a single difference reduction heuristic is represented in the preconditions of methods, in ΩMEGA (Benzmueller et al., 1997) too, some control knowledge was represented in particular methods thereby precluding further attempts to easily modify or extend, to learn or to generalize the control knowledge. We shall suggest some standard notions for proof planning and their representation based on today's AI-planning methodology. In particular, we define plans, methods, control-rules, and strategies.

2. BASICS OF PLANNING

A planning problem consists of an *initial state* that describes some initial situation and of *goals* to be achieved. A planning domain is defined by *operators* that represent actions. The operators have specifications to be used in the planning process. In STRIPS notation (Fikes and Nilsson, 1971), the specifications are called preconditions and effects. Preconditions specify the conditions of the planning state that have to be satisfied for the operator to be applicable, whereas effects describe the potential changes of the planning state caused by an operator application. Effects can be represented (in STRIPS notation) by add-lists ($\oplus$) and delete-lists ($\ominus$). Note that preconditions and effects usually are literals formulated in an *object-level* language such as `(on A B)` or `(arm-holds A)`. However, some planners, e.g., Prodigy (Minton et al., 1989) allow for additional preconditions formulated in a *meta-level* language that restrict the instantiations of parameters.

A partial plan is a partially ordered set of operators with additional instantiation constraints and auxiliary constraints. A partial plan can be seen as an implicit representation for sets of instantiated operator sequences (potential solutions) that are consistent with the ordering, instantiation, and auxiliary constraints. A solution of a problem, a complete plan, is a linearization of a partial plan (a sequence of actions) that transforms the initial state into a goal state, i.e., a state in which the goals hold.

The operation of a (partial order) planner repeatedly *refines* a partial plan, i.e., it adds steps and constraints and thus resticts the set of potential solutions until a complete plan can be picked from its set of potential solutions. Table 1.1 shows a simplified backward planning algorithm (that does not handle goal interactions). It starts with a partial plan π_0 defined by the problem to be solved. π_0 consists of steps t_0 and t_∞ that are instantiations of the dummy operators `start` and `finish`, respectively. They have the initial state as $\oplus$-effects and the goals as preconditions, respectively. π_0 also

represents the order constraint $t_0 \prec t_\infty$. The introduction of a step into a partial plan removes an open goal g from the goal agenda G (it satisfies g) and may introduce new open subgoals and constraints and this is continued until no open goals are left and a solution is found or until no operator is applicable anymore. Note that while most planners plan backward, the execution of a plan works forward.

`Backwards-Refine-Plan`(π, G)

Termination: **if** goal agenda G empty, **then** `Solution`.
if no operator applicable, **then** `Fail`.

Goal Selection: Choose open goal $g \in G$.

Operator Selection:

- For each operator M
  for each $\oplus$-effect e of M
  let σ be the matcher $\sigma(e) = g$
  if application-conditions($\sigma(M)) = true$
  then M is applicable.
- Choose one applicable M (backtracking point)
  insert M into π
  insert constraints into π
  update G.

Recursion: Call `Backwards-Refine-Plan` on the refined partial plan π.

Table 1.1 Outline for backward planning

A diversity of refinement algorithms has been developed in AI (see, e.g. (Weld, 1994; Kambhampati, 1996)), among them two kinds of hierarchical planning: precondition abstraction (Sacerdoti, 1974) and operator abstraction planning (Tate, 1977), also called hierarchical task network (HTN)-planning. Precondition abstraction first searches in an abstract space by ignoring certain object-level preconditions of operators. These ignored goals are then considered later at a lower hierarchical level. Operator abstraction employs complex (i.e. non-primitive) operators that may represent complex actions. A complex operator can be expanded into a partial (sub)plan. Note that the expansion of non-primitive operators refines a partial plan by introducing new steps and constraints. Since only primitive actions can be executed, all complex operators have to be expanded before a plan can be executed.

Search is involved in each of the planning algorithms, independent of which space they search (state-space, plan-space, abstract-space). Hence, the process of planning can be viewed as a sequence of heuristically guided decisions, i.e., as a choice from a set of candidates, such that the choice eventually leads to a complete plan. These decisions include choosing which open goal to satisfy next, which planning operator to introduce in order to satisfy this goal, and which instantiations to choose for a parameter. In any case, the decisions which goal to satisfy next and which operator to choose influence the way the search space is traversed and thus the efficiency of the search. Therefore, many search strategies that fix a particular preference have

been described in the planning literature, e.g., in (Pollack et al., 1997; Koehler, 1998). For instance, one of these strategies always prefers the goal g that produces the least number of alternative applicable methods.

Some advanced planners such as Prodigy (Veloso et al., 1995) use explicit control-rules to guide the search. This employment of control-rules changes the basic planning algorithm of Table 1.1 into the following algorithm in Table 1.2.

`Backwards-Refine-PlanC`(π, G)

Termination: **if** goal agenda G empty, **then** `Solution`.
if no operator applicable, **then** `Fail`.

Goal Selection: Evaluating control-rules (for goals) returns G' with $G' \subseteq G$.
Choose open goal $g \in G'$.

Operator Selection:

- Evaluating control-rules (for operators) returns O'.
- For each operator M $\in O'$
  for each $\oplus$-effect e of M
  let σ be the matcher $\sigma(e) = g$
  if application-conditions($\sigma(M)) = true$
  then M is applicable.
- Choose one applicable M (backtracking point)
  insert M into π
  insert constraints into π
  update G.

Recursion: Call `Backwards-Refine-PlanC` on the refined partial plan π.

Table 1.2 Outline for controlled backward planning

3. CONCEPTS IN PROOF PLANNING

Now we introduce the basic concepts of proof planning and briefly describe how they are realized in the ΩMEGA system. The initial state in proof planning is a collection of sequents, the proof assumptions, and the goal is the sequent to be proven. A proof planning domain consists of operators, called *methods*, of control-rules and maybe theory-specific solvers.

DEFINITION: A *partial proof plan* is a partially ordered set of instantiated methods and a *complete proof plan* (a solution of a problem) is a sequence of instantiated methods that transfers the initial state into a goal state. ■

ΩMEGA 's Planner employs both, operator abstraction and precondition abstraction. As a generalization of the standard notion of operator abstraction planning, ΩMEGA's complex methods can be expanded *recursively* into primitive methods that correspond to inference rules of the ND-calculus. A calculus-level proof of a theorem is a complete plan consisting of *instantiated primitive* methods. Each expansion of the non-

primitive methods is stored in the hierarchically structured proof plan data structure (PDS) (Benzmueller et al., 1997).

3.1. Methods

What are methods? Methods are data structures that encode mathematical proof methods. They represent inferences that are typically more complex than the single inference steps in a logical calculus. The methods should capture

(i) common patterns in proofs, e.g. the common structure of an induction proof or

(ii) common proof techniques such as simplification or the computation of a derivative and similar macrosteps that are called tactics in tactical theorem proving.

How can we discover methods? One heuristic for knowledge acquisition not only from mathematical (text)books is the following: More often than not the importance of a method is indicated by *naming* it. Named mathematical methods are, for instance, a proof by diagonalization or by induction, the application of an important theorem such as the Hauptsatz of Linear Algebra (each n-dimensional vector space has a basis of n linearly independent vectors), the Hauptsatz of Number Theory (each natural number can be uniquely represented as the product of prime numbers), etc. The mathematical monograph *Introduction to Real Analysis* (Bartle and Sherbert, 1982) introduces mathematical methods by referring, for example, to the Supremum Property, to the Monotone Convergence Theorem, etc. It states as a didactic help or hint for the reader:"The next two results will be used later as methods of proof"(p.32), "We shall make frequent and essential use of this property (the Supremum property of $\mathcal{R}$)" (p.45), or "the method we introduce in this section (associated with the Monotone Convergence Theorem) ... applies to sequences that are monotone..." (p.88).

How to represent methods? Starting from tactical theorem proving (Gordon et al., 1979), Alan Bundy's key idea (Bundy, 1988) was to represent methods as tactic + specification, where a tactic produces a calculus-level sequence of proof steps when it is executed and the specification part is used for proof planning. As we use a wider spectrum of methods now, our definition of methods is somewhat more general: we do not necessarily require the full expansion of a method to be generated by one tactic (Huang et al., 1994).

DEFINITION: *Methods* are planning operators, i.e., the building blocks of a proof plan. They consist of declarative specifications at the object- and meta-level and of an expansion function that realizes a schematic or a procedural expansion of the method to a partial proof plan. ■

The object-level specifications correspond to the usual preconditions and effects in planning. That is, they specify the sequents that are open goals to be planned for by the method, the sequents that have to be in the planning state when the method is applied (i.e. the subgoals produced by the method), the sequents that have to be assumptions,

and those that are produced as new assumptions when the method is applied in forward planning.

The meta-level specifications declaratively specify in a meta-language *local* conditions of the method's application, i.e., properties and relations of the sequents processed by the method.[1] This is in contrast to global properties, such as relations between methods, or how a method is used to prove a theorem in a particular theory or how the planning history including failed attempts and resources influence the choice of that method. The meta-level specifications preferably express the *legal* conditions for the method's application, in particular, the restrictions of the instantiation of the method's parameters. They can be encoded in application-conditions.

Depending on whether a method encodes a particular proof structure or a proof technique/procedure, the expansion of a non-primitive method can be realized schematically or procedurally: A schematic expansion defines a schematic proof tree that may contain meta-variables.[2]

In ΩMEGA, methods are represented as frame-like data structures with slots, slot names, and fillers. The slot names are *premises*, *conclusions*, *application conditions* (*appl.cond*), and *proof schema*. The *premises* and *conclusions* constitute a logical specification, in the sense that the conclusions are supposed to follow logically from the premises. Note that the sequents in the *premises* and *conclusions* are abbreviated by the labels of their proof lines. The $\oplus$- and $\ominus$-annotations indicate object-level specifications.

These specifications are used in planning for chaining methods together. The respective object-level specifications are matched with the current planning goal and with assumptions. Then the new planning state is computed from the previous state and the method's specifications.

In ΩMEGA the annotations are interpreted as follows: A $\ominus$-conclusion is deleted as an open goal, a $\oplus$-premise is added as a new open goal, a $\ominus$-premise is deleted as an assumption, and a $\oplus$-conclusion is added as an assumption. This representation of the object-level specifications is somewhat more involved than the usual preconditions and effects because both, forward planning (planning new assumptions from existing ones) and backward planning (reducing a goal to subgoals), are possible and, therefore, assumptions and open goals are both included into the planning state and the potential changes of the goals *and* the assumptions have to be represented.

In ΩMEGA the slot *proof schema* may provide a schema for the schematic expansion of the method and thereby capture the semantics of the method.

Examples of methods that encode a common proof structure are the `Diagonalization` method (Cheikhrouhou and Siekmann, 1998), `Induction` (Bundy et al., 1991), or the following method `ComplexEstimate` (Melis, 1998b) which is an estimation method used for planning limit theorems.

method: `ComplexEstimate`(a, b, e_1, ϵ)			
premises	L0, $\oplus$L1,$\oplus$ L2, $\oplus$L3		
conclusions	$\ominus L5$		
appl.cond	$\exists k, l, \sigma(Cas(a, b) = (k, l, \sigma))$		
proof schema	L0. Δ	$\vdash$ $\lvert a\rvert < e_1$	(j)
	L1. Δ	$\vdash$ $\lvert k\rvert \leq \mathbf{M}$	(OPEN)
	L2.	$\vdash$ $\lvert a_\sigma\rvert < \epsilon/2 * \mathbf{M}$	(OPEN)
	L3. Δ	$\vdash$ $\lvert l\rvert < \epsilon/2$	(OPEN)
	L4.	$\vdash$ $b = k * a_\sigma + l$	(CAS)
	$L5$. Δ	$\vdash$ $\lvert b\rvert < \epsilon$	(fi x;L4,L1,L2,L3)

The task of `ComplexEstimate` in planning is to reduce the goal of estimating a complicated term b to simpler estimation goals. It suggests the existence of a real number $\mathbf{M}$ that satisfies these simpler goals. $\mathbf{M}$ has to be instantiated during planning. `ComplexEstimate` removes an open goal that matches L5 and introduces the three correspondingly instantiated new subgoals L1, L2, and L3. The *application condition* requires that a computer algebra system is called (`Cas`) that computes terms k and l such that b can be represented as a linear combination $(k * a_\sigma + l)$ of a_σ.[3]

The very essence of `ComplexEstimate` is represented in the *proof schema* whose right-most entries are justifications. They can name ND-rules, methods, or tactics (e.g., `CAS` for a computer algebra tactic or `ATP` for calling a classical theorem prover). For the purpose of a short presentation of `ComplexEstimate`, 'fix' abbreviates a whole subproof of $|b| < \epsilon$ from the lines L4, L1, L2, and L3. The schematic expansion of `ComplexEstimate` introduces a subplan into the PDS that is obtained from the *proof schema*. This subplan may contain meta-variables.

3.2. Control Knowledge

Although proof plans are abstract representations of proofs and therefore plans are usually much shorter than calculus-level proofs, the search space is still exponential and potentially infinite. Therefore, means for controlling and restricting the search space are needed. Control knowledge can be distinguished along several dimensions, for instance, legal vs. heuristic control knowledge and local vs. global control.

In the following, we shall substantiate, how the different types of control-knowledge should be represented. Essentially, our suggestion is to represent legal and local knowledge in *application conditions* of methods and – most importantly – to represent heuristic knowledge in control-rules. Implicit control knowledge is encoded in tactics and strategies.

3.2.1. Control-Rules

Informed search can advantageously replace blind local search in domains, where enough control knowledge exists. Mathematics surely is a domain, where control knowledge, i.e. ways of attacking a problem, has been accumulated even over hundreds of years. The problem is, however, to extract this knowledge and to represent it appropriately.

Since methods should ideally correspond to mathematically meaningful steps, the control knowledge in proof planning should correspond to some mathematical 'intuition' on how to proceed in a proof.

Some of this control knowledge may be represented locally within a particular method, however, not all control aspects can be represented in this local manner. In accordance with a common principle in AI, we advocate an explicit representation of control knowledge in proof planning by declarative control-rules and its separation from the factual knowledge as well as its modular implementation.

As known from AI-research (Laird et al., 1987; Minton et al., 1989), separately implemented and declaratively represented control-rules ease the modification and extension of the control knowledge considerably. When new control information comes up in new cases of problem solving, then the control-rules can be modified and new control-rules can be introduced easily, as opposed to a re-implemention of an built-in procedurally implemented control. For instance, when the control is changed or generalized or when new methods are introduced into the domain, a re-implementation of the methods is not necessary. Rather, the set of control-rules can be extended or modified.

Because of their level of abstraction, control-rules are often well-suited for producing explanations on how a proof is found, and thus they can be naturally communicated and devised. Indeed, it is an important aspect of mathematical teaching to explain and teach control knowledge. For instance, the authors of the introduction to a computer-based calculus course (Davis et al., 1994) state that:

> "Today's popular texts present most of the tools (derivatives, differential equations, definite integrals, series expansion and approximation, data fitting, linearizations, vectors, dot products, line integrals, gradients, curls, and divergences) but end up focusing too much on the tools and not enough on their *uses*. The successful calculus course produces students who recognize a calculus situation and who know what calculus tools to bring to bear on it. This is the goal of Calculus & Mathematica."

Meta-reasoning about resources, the state of the partial proof, about failed proof attempts, etc. can be captured in control-rules and last but not least, the declarative representation by rules can be a basis for automated learning of control-rules, as realized, e.g., in (Minton, 1989; Leckie and Zukerman, 1998).

For all these reasons, in ΩMEGA global (heuristic) control knowledge is represented by declarative control-rules rather than in methods as it is done in the proof planner $CLAM$. Even if the implementation format of a method allows the direct encoding of this knowledge, it might be disadvantageous. For example, the order of goals to work on next, could be encoded into a method. However, if the situation in planning

becomes resource-bounded, then the control has to be changed and should prefer goals that are simple to satisfy within a given amount of time/space but the evaluation of resources is not local to methods and in any case the direct coding would be too inflexible.

In ΩMEGA, the control-rules guide the planner in its decision making. Control-rules have an antecedent and a consequent. The antecedent encodes the global context of the decision. It is used by the planner to recognize whether the rule fires at a given node in a plan search tree. The consequent of a rule encodes the advice that can be used to *select* a specified subset of the available candidates, to *reject*, or to *prefer* one candidate over another. The first two types of advice prune the search space, while prefer-rules change the default order without excluding other alternatives.

Corresponding to the choices the planner has to make, we have the following classes of control-rules in ΩMEGA:

- `operator` for choosing among several methods,
- `sequent` for choosing among goals (in backward planning) or assumptions (in forward planning),
- `binding` for choosing instantiations of method parameters, and
- `strategy` for choosing a planning strategy.

Here is a simple example for an `operator` control-rule used in planning limit theorems. This rule expresses the heuristic that, in case the goal is an inequality, `Solve, SOLVE*, UnwrapHyp` should be preferred candidate methods in the given order, where `Solve` passes an inequality goal to the constraint solver, `SOLVE*` reduces a more complicated inequality to a simpler one that can be passed to the constraint solver, in case an appropriate assumption exists in the planning state, and `UnwrapHyp` extracts a particular subformula from an assumption in order to prepare the application of `ComplexEstimate`. This control-rule is then represented as follows:

```
(control-rule prove-inequalities
        (kind operator)
        (if (goal-matches (?goal (?x < ?y))))
        (then (prefer ((Solve ?goal)
                       (SOLVE* ?goal)
                       (UnwrapHyp ?goal)))))
```

In its antecedent, the control-rule may use decidable meta-predicates to inspect the current planning state, the planning history, the current partial proof plan, the constraint state (in case a constraint solver is employed in proof planning as in ΩMEGA), resources, the theory in which to plan, typical models of the theory, etc. The meta-predicates in the following rules inspect the planning history and failed planning attempts (`failed-method`) and the current planning state (`most-similar-subterm-in`).

```
(control-rule case-analysis-critic
        (kind operator)
        (if (and (failed-method (wave))
                 (failure-condition (trivial ?C))))
        (then (prefer (CaseSplit (?C or not ?C)))))
```

The `case-analysis-critic` rule expresses the heuristic that if the `wave` method is not applicable at some stage because a formula C is not trivially provable, then it may be useful to introduce a `CaseSplit` method into the plan. This rule represents a rational reconstruction of one of Ireland's 'critics' that we shall discuss below.

3.3. Strategies

The process of finding a plan can be seen as a repeated refinement and modification. Refinement operations take a partial plan and add steps and constraints to it, whereas modification operations take a partial plan and modify, i.e., change it. We call such operations *strategies*.

DEFINITION: A *strategy* is an operation on partial plans that returns a partial plan. This operation determines its search space and restricts the way to traverse it. ■

Strategies can be very diverse. Examples for planning strategies are forward planning, backward planning, difference-reducing backward planning (Hutter, 1990; Bundy et al., 1990), case-based planning, precondition abstraction, instantiation of meta-variables, and expansion of abstract methods.

Conceptually, strategies differ from methods which are building blocks of a plan. Therefore we represent and implement them differently form methods. Some control knowledge is procedurally encoded into the strategies' algorithms. The representation of strategies may be parametrized, e.g., by termination conditions, a set of methods, and by a set of control-rules which make some of the control in the strategy explicit.

A multi-strategy planner (Melis, 1998c) can employ several strategies in one planning process. Now, picking an appropriate strategy requires *strategic control knowledge* which can again be declaratively represented in strategic control-rules.

An example for a strategic control-rule formalizes the heuristic that if a complex method `M` is rendered 'unreliable' in the partial plan and if enough time resources are available, then the expansion strategy is called with parameter `M`.

```
(control-rule expand-method
        (kind strategy)
        (if (and (complex-op(?M)
                 and (unreliable(?M)
                     and (less (minimum time))))))
        (then (prefer expand(?M))))
```

The purpose of this multi-strategy proof planning is a flexible use of different strategies and the capability to switch between strategies. Moreover, the strategy concept is a

means to structure the vast amount of mathematical knowledge that is available in real application scenarios.

4. CRITICS RECONSIDERED

Using the conceptual understanding of proof planning as outlined so far, in particular of the role of methods, control knowledge, and strategies, we shall now analyze some of the *critics* introduced by Ireland and Bundy in (Ireland and Bundy, 1996). We shall show how some of them can be reconstructed by methods, strategies, and control-knowledge.

Proof critics are a means to cope productively with failures in planning proofs based on mathematical induction. They are an *exception handling mechanism* which exploits the partial success of a proof plan, although the planning failed at some particular stage of the search for a proof. Critics are based upon the partial satisfaction of the application conditions of the `wave` method (Ireland and Bundy, 1996) in CL^AM. The `wave` method rewrites an (annotated) subterm t of the current planning goal g using an (annotated) conditional rewrite $(Cond \rightarrow LHS \Rightarrow RHS)$. Roughly, in CL^AM the application conditions of `wave` are

1. there is an appropriate subterm t
2. LHS matches t and the matcher respects the annotation
3. $Cond$ is trivially derivable from the hypotheses of g.

Several critics analyze the reasons why the `wave` method was not applicable although it should have been applicable for the planning to succeed. Reasons can be that there is no appropriate term t, that there is only a *partially* matching rewrite, that no rewrite rule matches at all, or that $Cond$ is not easily derivable from the hypotheses of g. Depending on these reasons, different critics are applied in order to automatically patch the proof attempt by speculating a lemma, by goal generalization, by introducing a case-split, or by revising the induction scheme.

In the following, we briefly describe, when these critics are called and how they work. We analyze them in terms of the concepts introduced earlier and re-describe them in our conceptual framework. This way we can distinguish between different impacts of the different critics on proof plans and generalize some of them.

4.0.1. Lemma Critics

The lemma speculation and lemma calculation critics deal with a failure of the second condition of the `wave` method since no (annotated) rewrite matches a subterm t. As a reaction to this failure, the lemma critic constructs a rewrite and validates it in order to continue the rewriting successfully.

From our conceptual point of view, meta-level reasoning about the failure and the heuristic of constructing a rewrite can be represented by control-rules. The constructed rewrite becomes an open goal in planning and, when proved, can be used for continuing the proof planning. This allows for a generalization of the critics in the sense that meta-level reasoning with control-rules may react to failed *application conditions* of

methods – even different from `wave` – that require the existence of a certain lemma in the planning state.

4.0.2. Case Analysis

The case analysis critic deals with a failure of the third application condition of `wave`. As a reaction to this failure, the case analysis critic computes a tautologous disjunction D (e.g., $Cond \vee \neg Cond$) and immediately inserts the step `case-split`(D) into the partial plan such that the `wave` method is now applicable in the $Cond$ branch of the plan. From our point of view, this case analysis critic can be replaced by a control-rule that chooses an instantiation of the `case-split` method as shown in the `case-analysis-critic` control-rule.

Furthermore, the idea of the critic can be generalized: In mathematical theorem proving the need for a case-split is often discovered only later. For instance, often $Cond$ is not tried to be proven before using the conditional rewrite but only later. In this case, `CaseSplit` has to be introduced by a particular refinement strategy that inserts a method while keeping the whole subproof as one of its branches of the case-split. This later introduction of a case split differs from the ordinary backward planning since the subgoal s occurring while planning for a main goal g is no longer satisfied by a method that applies $Cond \rightarrow h$ which reduced s to s'.

Moreover, this case-split strategy is no longer bound to the particular `wave` method anymore and the type of non-provability of $Cond$ can be generalized. The generalized conditions for calling this strategy can be expressed in the antecedent of a strategic control-rule.

So in summary, critics as introduced by Ireland and Bundy serve several very valuable purposes. While some of these purposes can be generalized by the conceptually clearer representation as strategies or control-rules, many other critics are just as valuable the way they are and should still serve as a local exception handling mechanism.

5. CONCLUSION

Automated proof planning has several advantages over classical automated theorem proving: It can employ classical automated theorem provers as methods and at the same time search for a plan at a level that is far more abstract than the level of a logical calculus. The methods represent mathematically meaningful proof steps and the control knowledge therefore corresponds to mathematical intuition. Employing this control knowledge yields an *informed search behaviour* that is superior to the search in classical automated theorem proving (see, e.g., the empirical results in (Melis, 1998a)).

In particular, the search at a more abstract level can find proofs of a length far beyond those of classical systems. For example, the OTTER prover (McCune, 1990) searches spaces of several billion clauses and the maximal proof length that can be found that way is around several hundred resolution or paramodulation steps. With proof planning we could potentially find a *plan* of several hundred steps that could then

be expanded into an object-level proof of several thousand steps. Proofs of that length are not uncommon in program verification tasks. For example, the VSE verification system (Hutter et al., 1996), which is now routinely used for industrial applications in the German Centre for Artificial Intelligence (DFKI) at Saarbrücken, successfully synthesized proofs of up to 8,000 and 10,000 steps in the verification of a television and radio switching program. Often these proofs represent many weeks or even months of labor, where about 80% of the steps were successfully generated by the machine and the remaining 20% by manual interaction of the verification engineer. These proofs are, by their very length, one or two orders of magnitude beyond fully automated methods but may come into the range of possibilities, if the proof planning paradigm turns out to be successful in these settings as well.

Further advantages of knowledge-based proof planning not addressed in this paper are that a general-purpose machinery (the planning technology) is combined with domain-specific knowledge (methods and control-rules), and thus proofs in specific mathematical theories such as calculus can now be handled not only with special-purpose provers such as Bledsoe's IMPLY (Bledsoe et al., 1972) but in a general setting. The ΩMEGA system, for example, was the first to solve the open problem (open for computer based systems) of LIM* (see (Melis, 1998b)).

Finally, high-level plans can be communicated much more naturally to the user and can yield understandable explanations and proof presentations as we may find them typically in a mathematical paper or text book.

ACKNOWLEDGMENTS

We thank the ΩMEGA Group in Saarbrücken, in particular Lassaad Cheikhrouhou, Andreas Meier, Volker Sorge, Michael Kohlhase; Dieter Hutter from the VSE-team at the DFKI, and the Edinburgh Dream Group, in particular, Alan Bundy and Julian Richardson, for helpful discussions.

Erica Melis
Universität des Saarlandes, FB Informatik
D-66041 Saarbrücken, Germany
Jörg H. Siekmann
Universität des Saarlandes, FB Informatik and DFKI Saarbrücken
D-66041 Saarbrücken, Germany

NOTES

1. e.g. that a subformula of an employed asssumption equals the processed goal
2. Meta-variables are place holders for syntactic objects such as terms and formulae.
3. a_σ is the term resulting from applying σ to a. $||, *, +, /$ denote the absolute value function, multiplication, addition, and division in the real numbers, respectively.

REFERENCES

Bartle, R. and Sherbert, D. (1982). *Introduction to Real Analysis*. John Wiley& Sons.

Benzmueller, C., Cheikhrouhou, L., Fehrer, D., Fiedler, A., Huang, X., Kerber, M., Kohlhase, M., Konrad, K., Meier, A., Melis, E., Schaarschmidt, W., Siekmann, J., and Sorge, V. (1997). OMEGA: Towards a mathematical assistant. In McCune, W., editor, *CADE-14*, pages 252–255, Springer.

Bledsoe, W., Boyer, R., and Henneman, W. (1972). Computer proofs of limit theorems. *Artificial Intelligence*, 3(1):27–60.

Bundy, A. (1988). The use of explicit plans to guide inductive proofs. In Lusk, E. and Overbeek, R., editors, *CADE-9*, pages 111–120, Springer.

Bundy, A., van Harmelen, F., Hesketh, J., and Smaill, A. (1991). Experiments with proof plans for induction. *Journal of Automated Reasoning*, 7:303–324.

Bundy, A., van Harmelen, F., Ireland, A., and Smaill, A. (1990). Extensions to the rippling-out tactic for guiding inductive proofs. In Stickel, M., editor, *CADE-10*, Springer.

Chang, C.-L. and Lee, C.-T. (1973). *Symbolic Logic and Mechanical Theorem Proving*. Computer Science Classics. Academic Press.

Cheikhrouhou, L. and Siekmann, J. (1998). Planning diagonalization proofs. In *AIMSA-98*, Springer.

Davis, W., Porta, H., and Uhl, J. (1994). *Calculus & Mathematica*. Addison-Wesley.

Fikes, R. and Nilsson, N. (1971). STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208.

Gordon, M., Milner, R., and Wadsworth, C. (1979). *Edinburgh LCF: A Mechanized Logic of Computation*.

Huang, X., Kerber, M., Kohlhase, M., and Richts, J. (1994). Methods - the basic units for planning and verifying proofs. In *Jahrestagung für Künstliche Intelligenz KI-94*, Springer.

Hutter, D. (1990). Guiding inductive proofs. In Stickel, M., editor, *CADE-10*, Springer.

Hutter, D., Langenstein, B., Sengler, C., Siekmann, J., Stephan, W., and Wolpers, A. (1996). Deduction in the verifi cation support environment (VSE). *Third International Symposium of Formal Methods Europe*, pages 268–286.

Ireland, A. and Bundy, A. (1996). Productive use of failure in inductive proof. *Journal of Automated Reasoning*, 16(1-2):79–111.

Kambhampati, S. (1996). Refi nement planning: Status and prospectus. In *AAAI-96*, pages 1331–1336.

Koehler, J. (1998). Solving complex tasks through extraction of subproblems. In Simmons, R., Veloso, M., and Smith, S. (eds), *AIPS-98*, pages 62–69.

Laird, J., Newell, A., and Rosenbloom, P. (1987). SOAR:an architecture for general intelligence. *Artificial Intelligence*, 33(1):1–64.

Leckie, C. and Zukerman, I. (1998). Inductive learning of search control rules for planning. *Artificial Intelligence*, 101(1-2):63–98.

Loveland, D. (1978). *Automated Theorem Proving: A Logical Basis*. North Holland, New York.

McCune, W. (1990). Otter 2.0 users guide. Technical Report ANL-90/9, Argonne National Laboratory, Maths and CS Division, Argonne, Illinois.

Melis, E. (1998a). AI-techniques in proof planning. In *ECAI-98*, pages 494–498, Wiley.

Melis, E. (1998b). The 'limit" domain. In Simmons, R., Veloso, M., and Smith, S. (eds), *AIPS-98*, pages 199–206.

Melis, E. (1998c). Proof planning with multiple strategies. In *workshop: Strategies in Automated Deduction*.

Minton, S. (1989). Explanation-based learning: A problem solving perspective. *Artificial Intelligence*, 40:63–118.

Minton, S., Knoblock, C., Koukka, D., Gil, Y., Joseph, R., and Carbonell, J. (1989). *PRODIGY 2.0: The Manual and Tutorial*. Carnegie Mellon University. CMU-CS-89-146.

Pollack, M., Joslin, D., and Paolucci, M. (1997). Flaw selection strategies for partial-order planning. *Journal of Artificial Intelligence Research*, 6:223–262.

Sacerdoti, E. (1974). Planning in a hierarchy of abstraction spaces. *Artificial Intelligence*, 5(2):115–135.

Tate, A. (1977). Generating project networks. In *IJCAI-77*, pages 888–893. Morgan Kaufmann.

Veloso, M., Carbonell, J., Pérez, M. A., Borrajo, D., Fink, E., and Blythe, J. (1995). Integrating planning and learning: The PRODIGY architecture. *JETAI Journal*, pages 81–120.

Weld, D. (1994). An introduction to least committment planning. *AI magazine*, 15(4):27–61.