Using Prolog with Smalltalk-based Problem Solvers for Planning and Design Tasks

Thomas Roth-Berghofer, Werner Schirp and Frank Weberskirch

University of Kaiserslautern, Dept. of Computer Science
P.O. Box 3049, D-67653 Kaiserslautern, Germany
E-mail: {roth,schirp,weberski}@informatik.uni-kl.de

Abstract

Rules are an important knowledge representation formalism in constructive problem solving. On the other hand, object orientation is an essential key technology for maintaining large knowledge bases as well as software applications. Trying to take advantage of the benefits of both paradigms, we integrated PROLOG and SMALLTALK to build a common base architecture for problem solving. This approach has proven to be useful in the development of two knowledge-based systems for planning and configuration design (CAPLAN and IDAX). Both applications use PROLOG as an efficient computational source for the evaluation of knowledge represented as rules.

LSA-Report LSA-97-05E

Centre for Learning Systems and Applications
Kaiserslautern, Germany

Contents

1 Motivation – Why PROLOG and SMALLTALK?

Neither completely object-oriented systems nor completely rule-based systems cover all the requirements that are necessary to develop systems for planning and configuration design tasks. Of course, each of the programming paradigms is complete with respect to computability, but the considered problem solving domains have properties which should not be ignored when choosing the implementation environment.

Planning and configuration design tasks most often take place in technical domains which are well structured and can be characterized by objects in a natural way. The four major elements of the object model *abstraction*, *encapsulation*, *modularity* and *hierarchy* as defined in (Booch, 1991) can be identified at once. Following these principles can be expected to lead to major advantages in code reuseability, rapid prototyping, user interface development, and maintenance of the system. All these advantages are good reasons for developing a problem solver for planning or design tasks using an object-oriented approach (e.g., SMALLTALK).

On the other hand, rules are essential for the representation of inferential knowledge and PROLOG, especially, offers the capability to process this kind of knowledge efficiently. As a wide-spread programming language, PROLOG provides a commonly accepted standard for the representation and evaluation of rules. But we believe that it would not be a good idea to build a complete problem solver for planning or configuration design in PROLOG because we would also have to code user interfaces in a declarative way with rules. In principle this surely might be possible, but it doesn't seem to be a natural way to do it. There is much more evidence that the object-oriented approach is better suited for such parts of software projects. Of course, the opposite is at least questionable too: building another PROLOG interpreter within the object-oriented framework (e.g., (Aoki et al., 1995; Pachet, 1991)). Our two main reasons for not following this idea are, that we are not interested in research with repect to PROLOG specific topics (we just want to apply PROLOG) and that using an existing PROLOG interpreter (written in C) should be a faster way for reasoning with backward chaining rules.

Trying to profit from the advantages of logic as well as object-oriented programming, we have developed a problem solving architecture which consists of an object-oriented knowledge base, a SMALLTALK-based problem solver using TMS techniques and a PROLOG interpreter as external knowledge source. We decided to use SWI-PROLOG (Wielemaker, 1993). For the object-oriented core and user interface of the planning and the configuration design system we took VISUALWORKS. PROLOG and SMALLTALK have been connected by a communications protocol based on Unix-Sockets. PROLOG can be accessed from SMALLTALK for prooving queries based on a set of given rules and facts, SMALLTALK and its object-oriented knowledge bases can act as an external database of facts for PROLOG during prooving.

Using this environment we have developed a common architecture for problem solving. The integration of PROLOG turned out to be especially useful for two knowledge-based systems described in more detail: CAPLAN[1] (Weberskirch, 1995) is a generative planning system that offers several possibilities to control the planning process, e.g., it can act as a planning assistant in which the user makes all relevant control decisions. CAPLAN/EBL

[1]Computer Assisted Planning

is an extension with respect to controlling the planning process. It acquires search control rules through explanation-based learning techniques and uses these rules in future planning episodes to reject futile alternatives. The configuration design system IDAX[2] (Paulokat, 1995) uses preference and control rules to make its decisions.

The following pages organized as follows: Section 2 summarizes the common base architecture of the two problem solvers. Section 3 presents an overview of the problem solvers CAPLAN/EBL and IDAX. Section 4 describes the interface we implemented between SMALLTALK and PROLOG, the representation of rules in SMALLTALK and objects in PROLOG and explains the rule evaluation process. The last section discusses some advantages and disadvantages of our approach and makes some concluding remarks.

2 Base Architecture for Decision-Based Problem Solvers

Our generic model of problem solving is based on the REDUX-architecture (Petrie, 1991; Petrie, 1992), a generic architecture to represent knowledge about plans, contingencies, and for solving Constraint Decision Problems. It has been successfully reused for several knowledge-based systems in different areas of AI research including action planning (Weberskirch, 1995), configuration design (Paulokat, 1995) and workflow-management (Maurer and Paulokat, 1996; Maurer and Pews, 1996).

A *problem* is described by an initial situation and a set of open goals. For each specific application there is a language for describing initial situations and goals. Solving the problem means to transform the initial situation according to the laws of the respective domain into a situation in which all the given goals are satisfied. During this process all open goals are collected on a central *agenda* from which the problem solver can choose the next one to work on following the principle of blackboard architecture (Hayes-Roth, 1985). Usually there will be various ways to achieve a single goal. Due to interactions between the different goals not all of these ways will lead to a solution for the problem. Thus, an extensive search process is necessary in most cases. We see this search process as a sequence of *decisions*, where each decision can make assignments, which form part of the evolving solution, and either solves a goal immediately or replaces the goal by one or more simpler goals. Therefore, every (correct) decision represents a step towards a solution of the complete problem.

Basically, there are two different kinds of decisions that have to be distinguished:

- *Selecting* one out of several possible operations, where each operation is represented by a so-called *operator*,
- *Rejecting* a single operation, i.e., reducing the number of operators to select from.

Decisions are not necessarily correct with respect to an overall solution of a problem, so the problem solver has to perform search process. As a consequence of this, both kinds of decisions can be undone, typically, if the search got stuck and backtracking is unavoidable.

[2] Intelligent Design Assistant based on REDUX

Decision context and rationales. Each decision is made in a certain context which consists of the problem description and the current state of all prior decisions. Usually, a decision will depend only on a subset of this context and we need to identify the relevant aspects as exactly as possible in order to create a useful *rationale* for a decision. Due to the permanent incompleteness of information every part of the present context may change in time. Monitoring the dependencies between a decision and the relevant aspects of its surrounding context allows to detect the necessity or possibility to update this decision. Therefore, all decisions are embedded in a framework which allows the representation of decision rationales and other dependencies between decisions. This framework is based on Redux (Petrie, 1991; Petrie, 1992), a generic architecture to represent knowledge about plans, contingencies, and for solving Constraint Decision Problems. The decision dependencies are permanently monitored by a Truth Maintenance System (TMS) (Doyle, 1979) to keep all decisions in a consistent state. Using Truth Maintenance, we can effectively determine which decisions need to be reconsidered after the problem description or another decision has been changed.

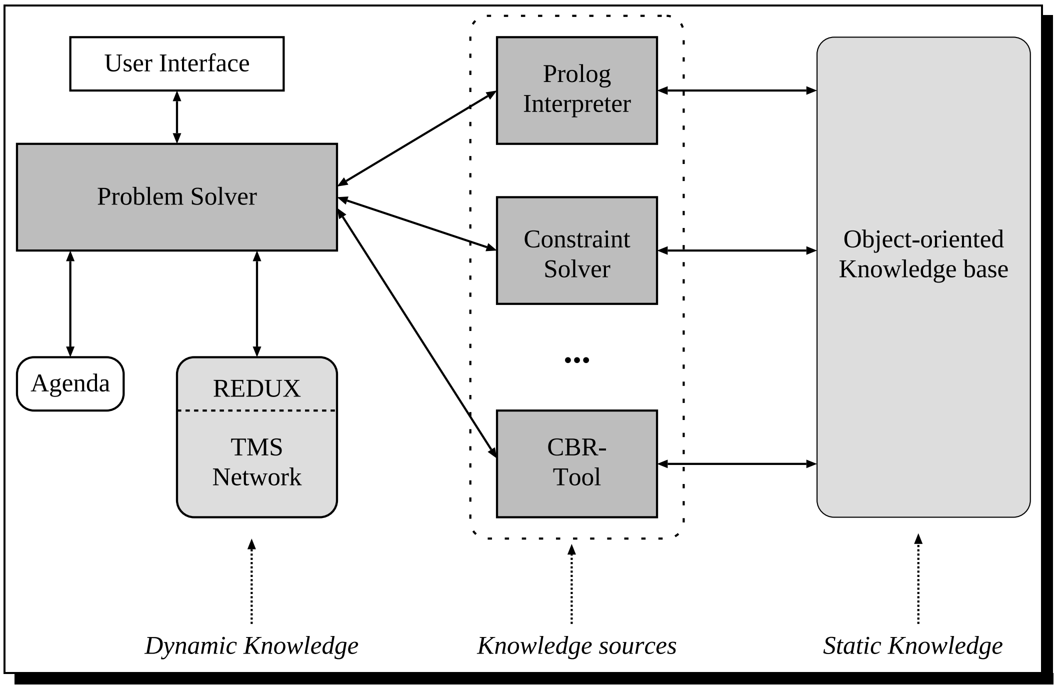

Figure 1: Base architecture of the problem solver

Knowledge base and knowledge sources. In order to minimize the risk of making a wrong decision, we try to exploit as much knowledge as possible. Therefore, the problem solver has access to a number of knowledge sources, which sometimes are computational elements inferring knowledge from the static and dynamic knowledge base. Sometimes such knowledge sources are highly domain-dependent, because the underlying knowledge representation has to be carefully chosen according to each specific aspect of domain knowledge. In order to simplify modification and extension of the knowledge base for a certain domain, all the different knowledge representations are embedded into an object-oriented model of the domain, which supports modularity and encapsulation. The knowledge sources are used to transform the static knowledge into dynamic knowledge represented within the TMS network (see Figure 1) or to infer knowledge from existing knowledge. Among the knowledge sources that currently are in use are constraint solvers, rule interpreters and case-based reasoning systems as well as interactive inferfaces for human experts.

If a knowledge source is activated, it should not only suggest a decision but also give a rationale for this decision. For example, if a constraint solver determines a certain set of constraints to be inconsistent with repect to a some new constraints, it is exspected to be able to give a rationale telling about the reasons for which this inconsistency occurred. This rationale forms a sufficient precondition, but not a necessary one, assuming that most rationales are not logically complete.

A Prolog interpreter as a knowledge source. Within the variety of knowledge representations used in different domains rules are playing an important role because they are often both comprehensive and expressive. Therefore, the evaluation of rule sets is a frequently used knowledge source for both, selecting and rejecting operations. This knowledge source has been realized using a Prolog interpreter as an efficient engine for rule evaluation (see Section 4 for details). Successful evaluation of a rule set results in the application of one top-level selection- or rejection-rule. From the facts used to prove the precondition of that rule, a rationale can be constructed.

In the following sections we describe in brief two applications based on this problem solving architecture. Both problem solvers make use of Prolog-based rule evaluation as an important knowledge source.

3 Applications

In this section we describe two knowledge-based systems, CAPlan and Idax, from different areas of AI research (action planning and configuration design) that are built upon the base architecture for problem solving summarized in the last section.

3.1 Planning with CAPlan

First, we give an overview of the CAPlan planning architecture. We will focus on the explanation-based learning module of CAPlan/EBL (Roth-Berghofer, 1996). It learns operator rejection rules from failures during planning episodes and uses a Prolog system to evaluate these rules in order to avoid the same failures in the future.

So far, CAPlan is an AI planning prototype, but we evaluate it using a complex and large application domain from the area of mechanical engineering – manufacturing process planning for rotary symmetrical workpieces (Muñoz-Avila and Weberskirch, 1996b). More information about CAPlan is present in the World-Wide Web at location `http://wwwagr.informatik.uni-kl.de/~caplan`.

3.1.1 The CAPlan Architecture for Domain-Independent Planning

CAPLAN (Weberskirch, 1995) is an SNLP-like (McAllester and Rosenblitt, 1991; Barrett and Weld, 1994) domain-independent planner that is built upon the generic REDUX architecture (Petrie, 1991) for dependency maintenance that already has been subject of Section 2. CAPLAN is given a specification of the planning domain that consists of a language to describe world states and the allowed actions that change world states. The allowed *actions* are represented using a STRIPS-like (Fikes and Nilsson, 1971) formalism and consist of preconditions, effects, and constraints, i.e., if the preconditions and the constraints are satisfied in a world state, the action will change the world as defined by the effects.[3] The task of CAPLAN is to find a *plan*, i.e., a sequence of actions (called *plan steps*) that transform a given initial world state into a state in which the specified goals are achieved. The initial situation in this search process is an initial plan representing the initial world state and the goals with two dummy plan steps, s_0 and s_∞.[4] During the planning process the initial plan is modified (refined) gradually until it becomes a solution plan. A *solution* plan must satisfy the correctness condition that for all plan steps (actions) all preconditions must be established and interactions between parallel steps must be solved.

As described in Section 2, the process of refining the initial plan can be seen a sequence of decisions. The process proceeds by selecting open preconditions of plan steps or unsolved interactions (both are open goals on the problem solvers agenda) and by making decisions about the *rejection* and *selection* of operators that refine the plan. The decision to select a certain operator is equivalent to the decision to select a certain plan refinement method (Kambhampati et al., 1995) for the current partial plan.

CAPLAN uses so-called *control components* (Weberskirch, 1995) as a common interface to control the search process. Each control component relies on a suitable knowledge source and encapsulates a strategy for problem solving, loosely spoken, the concrete behaviour of the system at goal selection and operator selection points. In general, a control component has to do the following things:

- It has to select the *goal* which has be be processed next from the agenda of open goals,
- for the selected goal all inconsistent operators (plan refinement operators) have to be *rejected*, and
- a consistent operator has to be *selected* and applied.

Different control strategies are available for CAPLAN, e.g., a case-based control strategy (Muñoz-Avila and Weberskirch, 1996a) that uses cases as knowledge source to guide the planner. In CAPLAN/EBL a control component has been added that learns rules from

[3] As explained in (Weberskirch and Muñoz-Avila, 1997) in more detail, CAPLAN extends the standard SNLP planning paradigm by allowing to define type hierarchies and types constraints in addition to simple codesignation constraints.

[4] In each plan s_0 is the first plan step, s_∞ is the last plan step. s_0 has no preconditions but its effects create the initial world state, s_∞ has no effects but its preconditions are the goals that have to be achieved. All other plan steps are ordered between s_0 and s_∞.

operator failures during planning and evaluates this knowledge using a PROLOG system to avoid similar failures in other problem solving episodes (Roth-Berghofer, 1996).

3.1.2 Explanation-Based Learning of Control Information

Research in the field of machine learning developed mechanisms to learn the correct behaviour at decision points of a search algorithm (to some degree). In planning, especially the method of *Explanation-Based Learning (EBL)* (DeJong and Mooney, 1986; Mitchell et al., 1986) has been applied to improve planning performance for state-space planners by learning control rules (Minton, 1988). Recently, (Katukam and Kambhampati, 1994; Kambhampati et al., 1996) presented an approach for applying EBL techniques to plan-space planners[5] like SNLP which serves as basis for CAPLAN.

In CAPLAN/EBL (Roth-Berghofer, 1996) we extended the learning mechanism described in (Kambhampati et al., 1996) with respect to the more powerful domain specification possibilities of our base level planner CAPLAN. Basically, similar to (Kambhampati et al., 1996) learning concentrates on *failures in selecting the right operator.* Whenever the planner reaches a dead end, the local reason for this is that there exists a goal for which all possible operators are inconsistent with the current plan. Learning from such failures consists of three steps:

Analysis: Learning starts with *analyzing* the inconsistencies that led to the failure and giving an initial failure explanation. These initial explanations can easily be extracted from what CAPLAN's consistency tester (Weberskirch, 1995) automatically generates for inconsistent operators: it determines a set of prior planning decisions that had effects that led to the inconsistency. Exactly these justifications build the initial failure explanations.

Regression: A *regression* mechanism (Kambhampati et al., 1996) allows to combine and propagate failure explanations backwards in the search tree. The result is a set of expressions that must have been valid before the failure has been detected by the planner but that were responsible for it.

Generalization: Following the EBL approach, these explanations are further *generalized* to be applicable to a bigger set of examples later. The generalization mechanism had to be extended in CAPLAN/EBL because operators in CAPLAN may add type constraints. Thus, generalization will not only replace constants by variables but also add predicates that check type constraints on these variables (Roth-Berghofer, 1996).

The learning module of CAPLAN/EBL then generates control rules for the rejection of operators from these regressed and generalized expressions and stores them in its rule base.

[5]See, for example, (Barrett and Weld, 1994), (Minton et al., 1994) or (Kambhampati et al., 1995) for a more detailed discussion of difference between these two planning paradigms.

3.1.3 Learning and Using Operator Rejection Rules

Within CAPLAN/EBL we can distinguish between a *learning mode* in which the system analyzes operator selection failures and generates control rules and a *rule application mode* in which the planner solves a planning problem using the learned rules.

Learning mode: As described in Section 3.1.2, the learning module of CAPLAN/EBL analyses operator failures. Rules from regressed and generalized explanations can be expected to recognize a potential failure of a sequence of operators in advance as regression eliminates the specific influence of an operator in a failure explanation. The generated rules have the form `reject-op(op, ...) :-` c_1`(..),` c_2`(..), ...` where the c_i`(..)` are expressions that refer to the current state of the planning process, i.e., elements of the plan or the planning problem. Examples for such predicates are `hasStep(plan,step)` or `hasOrdering(plan,step1,step2)`.

Rule application mode: To use the learned rules we built a control component (RbC, Rule-based Control) that checks potential operators using the generated rejection rules. As summarized in Section 3.1.1, a control component has to reject inconsistent operators. The standard consistency tester only can detect obvious inconsistencies among constraints that have been added to the plan. Our rule-based control component additionally tries to *prove* that an operator has to be rejected based on the learned rejection rules and the facts about the current plan state. For the process of proving an operator rejection this control component uses a PROLOG system (see Section 4). If PROLOG can prove the necessity to reject an operator, this operator will be rejected without having been selected. The used PROLOG interface additionally is capable of giving a rationale for a successful proof, so the results of each successful proof can be stored in the dependency network of the problem solver in exactly the same way as the results from simple consistency tests.

(Kambhampati et al., 1996) has shown that rejections found by rule evaluation in the way described above are correct. So, if an operator is rejected that would have been selected otherwise, we definitely cut away a branch of the search tree without loosing a solution. Unfortunately, the process of proving needs time. In our experience, however, the benefits of rule application are higher than the costs. Especially in non-trivial planning domains with many different operators that might be chosen for a certain goal, we observed an increase of the overall planning performance although the time per inference step increased because of the additional rule evaluation.

3.2 Configuration Design with IDAX

The task of configuration design is to satisfy a functional requirement by assembling an artefact from a given set of basic components without violating any constraint imposed on the connection of those components. As the number of available components and possible assemblies tends to be very large, configuration design is a search-intensive task which cannot be solved without further knowledge (Mittal and Frayman, 1989).

Based on the problem solving architecture described in Section 2 we have developed a tool to support configuration design called IDAX[6] (Barth et al., 1994; Paulokat, 1995). By now IDAX is mainly used as a prototype for AI-research, but it forms also part of CYCLOPS (von Wangenheim, 1996), a system for knowledge-based image understanding which was exhibited during the CeBIT fair in 1996.

3.2.1 Object-centered Representation of Knowledge

Following the idea that there is a continuum of concepts between the mainly functional problem description and the resulting assembly of basic components (Mittal and Frayman, 1989), the complete domain knowledge is organized in an abstraction hierarchy of objects with the basic components at its lowest level. Our approach is truely object-centered as there does not exist any kind of global knowledge. Instead there are five categories of domain knowledge associated with each conceptual object:

- **Attributes** describe properties of an object and can be used to express requirements.
- **Relations** show possible refinements of an object using specialization or decomposition.
- **Constraints** represent restrictions between attributes of an object and its components.
- **Phases** describe different strategies used during a configuration process.
- **Rules** give local, context dependent heuristics for making a good decision.

Attributes, relations, and contraints can be summarized as domain specification aspects, whereas phases and rules encode (domain-dependent) strategies and heurististics for solving problems in a certain domain.

3.2.2 The Use of Different Knowledge Sources

The configuration process follows the principle of stepwise top-down refinement along a structural hierarchy (defined by the specialization relation). Basically each step consists of three subsequent stages:

1. Choose the next *open goal* according to the current strategy.
2. Make one or more *decisions* (rejections and/or selection) concerning the chosen goal.
3. Determine the appropriate *strategy* for the next step.

In each of theses stages specific knowledge sources are exploited:

[6] Intelligent Design Assistant based on REDUX

- The evaluation of the *filtering and ordering conditions* of the active phase reduces the number of goals to choose from.
- Destructive *constraint propagation* leads to the rejection of inconsistent values and reduces the number of selectable operations. If all but one are rejected, the remaining operation can instantly be selected.
- The evaluation of the *preference rules* determines which of the still unrejected operations should be selected. In addition it returns a rationale for this decision.
- The evaluation of the *control rules* of the active phase possibly results in entering a new phase or returning to a previous one.

The last two of these knowledge sources, the evaulation of preference and control rules, rely on backward-chaining rules that are evaluated by a PROLOG interpreter.

3.2.3 Preference Rules

If a selection among several possible operations is to be made, we need a rating of the alternatives to determine the optimal choice, which is to be preferred among all others. The preference order on a set of alternatives is not static, but depends on the context of the decision. Therefore we use a set of rules to specify the preconditions for the preference of a certain alternative.

In order to describe the preconditional context for a preference, we need to specify properties of our configuration problem and its partial solution found so far. We defined a vocabulary of logical predicates to describe basic properties of the solution, like values or ranges of attributes, included or excluded parts, chosen components and so on. These predicates (called *product predicates*) allow a rather natural description of a certain context, but this description tends to become quiet long if a lot of basic properties are involved. To shorten the context description, we derive more abstract characteristics (called *aspects*) of the context from a set of basic properties. This derivation is done with domain-specific *aspect rules*, which take several product predicates as preconditions. As the existence of an aspect is tested by another product predicate, we can build an abstraction hierarchy of aspects. The PROLOG interpreter can prove the existence of a certain aspect by evaluation of the given aspect rules and the preconditional product predicates.

To evaluate a product predicate, the PROLOG interpreter submits a query to the SMALLTALK system, where the current context is represented using REDUX (see Section 2). REDUX does not only return the value of the specified property but also a set of TMS nodes, whose current labeling represents this property. These TMS nodes are collected during the proof of the preconditions of a preference rule. A successful evaluation of a preference rule results in returning the preferred operation along with a collection of TMS nodes justifying its selection. These nodes are used to generate a sufficient rationale for the decision.

The preference rules for all tasks concerning a single conceptual object are collected in a rule set and stored in the knowledge base as part of the object's specification. Use of inheritance and aggregation within the object model keeps the rule sets as small as possible.

A rule set is transmitted to the PROLOG system as soon as the respective object becomes part of the so far generated solution. As most rules are applicable only for a single task, they are not evaluated more than once during a configuration process, unless backtracking forces retraction of prior decisions.

Preference rules are used as a heuristic for a local optimization. They are not intended to increase the speed of the configuration process but the quality of its solution. To ensure this quality, the decision rationales retrieved during a PROLOG proof are continuously monitored by REDUX. As soon as such a justification becomes invalid, the concerning decision can be revised. As the justification only represents a single out of possibly many different proofs for a preference selection, we first try to find another proof by re-evaluating the previously used preference rule. Only if this attempt fails, we retract the decision and select another operation according to the evaluation of the preference rules using the current context.

If part of a configuration process is done by a human expert, he will also have to give rationales for his decisions. Instead of explicitly enumerating prior decisions or a set of TMS nodes, the expert can describe this rationale in a much more abstract way using product predicates (Schirp, 1996). The translation of such a rationale into a set of TMS nodes is once again done by the PROLOG interpreter. It is also possible to transform a rationale given by the expert into a preference rule. This provides an easy and direct way for the acquisition of additional heuristic knowledge, which could be difficult to formalize otherwise.

3.2.4 Control Rules

Strategic knowledge about the importance of unsolved subproblems and the order they are to be reduced in, is encapsulated in *phases* (Günter, 1991). To increase flexibility, the sequence of phases is not statically predefined, but individually determined during each configuration process. Therefore, each phase has to provide additional knowledge about the conditions that will terminate or suspend the current phase and switch to another one. This knowledge is expressed using *control rules* that are constructed in a similiar way like the preference rules described above.

Instead of a preferred operation, the evaluation of control rules returns the phase that should be used next. As the decision which phase to follow does not only depend on properties of the so far developed solution but also on the history and current state of the configuration process itself, the product predicates are not sufficient to express all possible conditions. Therefore, additional *process predicates* have been invented, that describe properties of the configuration process like a history of used phases, currently open subproblems and known inconsistencies.

Control rules that are associated with a certain phase remain valid as long as this phase is not terminated and reside within the rule base of the PROLOG system during this time. Evaluated after each configuration step, the control rules work as a trigger for the change of the active phase. As the choice of a phase does not immediately affect the solution under developement it is not represented in REDUX. Thus, the evaluation of the control rules does not have to deliver justifying TMS nodes.

4 Implementation Issues

As explained before, our base architecture and the two prototypical applications make use of an object-oriented environment for the problem solver as well as of an PROLOG interpreter as an engine for efficient rule evaluation. The last section explained how these two applications, CAPLAN/EBL and IDAX, use rule evaluation to speed-up problem solving or to increase the quality of the solution.

In this section we will describe the communications structure of the underlying combined SMALLTALK-PROLOG system in more detail. We will present an overview of the system components and explain how rule evaluation is done in the context of problem solving.

4.1 Interface between SMALLTALK and PROLOG

The PROLOG interpreter is an external knowledge source for SMALLTALK. It can be seen as a kind of server that is only activated on request by the SMALLTALK client (problem solver). So far this is, however, not a real client/server architecture, as there can be only one client at a time. In addition, the so-called PROLOG server can submit queries back to the SMALLTALK system. Thus, during a query is processed by PROLOG, SMALLTALK is a kind of server for information about the current state of the problem solving process for PROLOG, an extended external database for facts. The possibility of queries from PROLOG to SMALLTALK was added to avoid a complete replication of the knowledge bases residing in SMALLTALK.

The interface between SMALLTALK and PROLOG is based on UNIX sockets. SMALLTALK and PROLOG can either run on the same machine as two different processes or on different machines that are connected via a TCP/IP network. VISUALWORKS provides classes and methods for accessing sockets. SWI-PROLOG on the other side allows to add C-functions which can be accessed through PROLOG predicates. Using this mechanisms, PROLOG establishes a server socket and listens for requests, and SMALLTALK connects to this socket to start the evaulation of queries.

4.2 Rules and Objects

Due to different representations of data in the two programming paradigms conversion functions in both directions are needed. The basic types *integer*, *real* and *string* have direct equivalents. Objects of class *Symbol* translate into *atoms* in PROLOG. This automatic translation is part of the functionality of our socket communications mechanism.

Rules in SMALLTALK. The term structure is a basic concept of PROLOG and doesn't exist in SMALLTALK. We provide a set of SMALLTALK-classes, closely representing the datastructures used in PROLOG:

- The class `PrologTerm` has been created to build up the equivalent structures in SMALLTALK.

- Predicates in PROLOG are mapped onto instances of the class `PrologPredicate`.
- `PrologPredicates` are used to construct instances of `PrologRule`.

These objects represent PROLOG rules in the object-oriented knowledge base of the problem solver.

Objects in PROLOG. In PROLOG C-functions parse a string representation of the transferred SMALLTALK objects. Such functions exist for either asserting/retracting rules and facts to/from the rule base of PROLOG, starting the evaluation of rules, sending back queries of PROLOG to the dynamic knowledge base, and sending back the results.

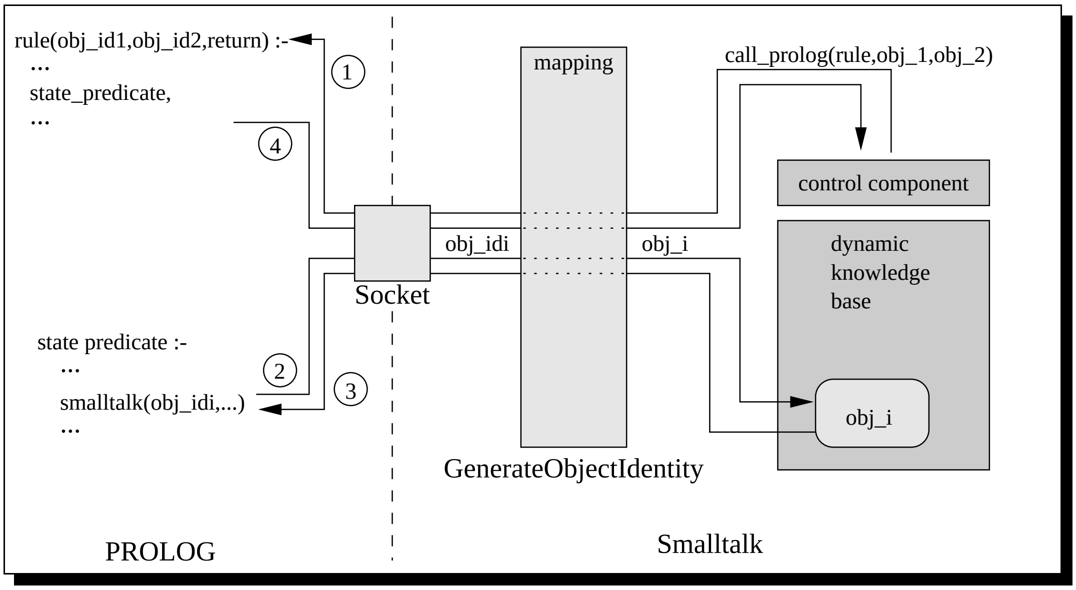

Figure 2: Object to atom mapping.

Mapping between objects and atoms. SMALLTALK objects can only be used by PROLOG in a restricted way. For our purposes, it was sufficient to determine the identity of an object because we do not need to change or access the internal structure of objects. This is accomplished by the meta class `GenerateObjectIdentity` which provides a unique identifier for every object that is sent to the rule interpreter (see Figure 2). The circled numbers show the conversion paths of objects to and from PROLOG:

1. A call to PROLOG (assert/retract/query) leads to the conversion of objects into object identifiers (atoms).
2. Predicates about the state of the problem solving process (state predicates) can send back queries to the dynamic knowledge base. The object ids of objects involved in such a query are converted back to the original objects.

3. The response of the dynamic knowledge base (including TMS nodes if necessary) is sent to PROLOG. Again all referenced objects are mapped to identifiers.

4. Given that a rule has fired, the collected TMS node identifiers are transmitted to the problem solver.

The problem solver therefore can get two results from PROLOG:

Success: If the query could be proven with the given rules and requested facts from the dynamic knowledge base, PROLOG signals success and gives a rationale that justifies the success.

Failure: Otherwise the proove failed and no justification can be given.

As the PROLOG server can give rationales for successful prooves, it is fully integrated into the problem solver based on REDUX. In this way, CAPLAN and IDAX are both able to use rule-based knowledge which is evaluated on demand by the assisting PROLOG server.

4.3 Rule Evaluation Cycle

The general mechanism for the execution of queries to PROLOG as described in the last section is used to represent and evaluate rule-based knowledge in our problem solvers CAPLAN/EBL and IDAX.

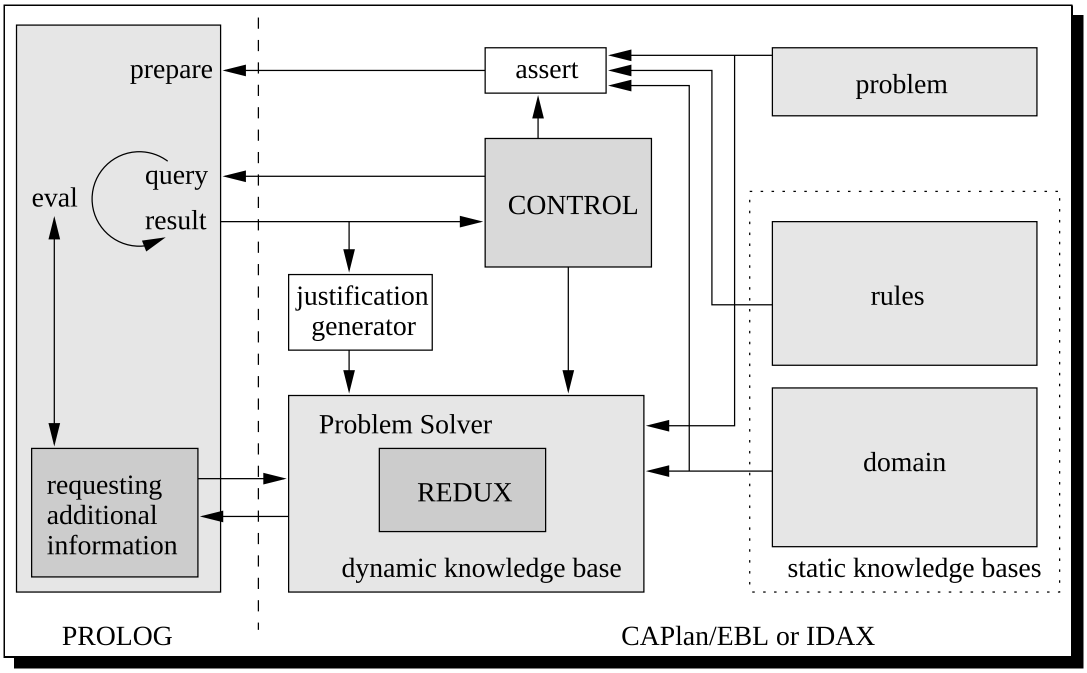

Figure 3: Rule evaluation

Rule evaluation consists of three phases:

Preparation. Facts and rules are transferred from the knowledge bases and the problem specification in SMALLTALK to PROLOG. Aspects of static knowledge may be transmitted completely before the first evaluation starts. Dynamic knowledge, on the other hand, has to be transmitted during the solving process as it is needed or becomes available.

Evaluation. During problem solving a *control component* of the problem solver uses PROLOG as an external knowledge source. It submits *queries* to the rule interpreter. As already explained in Section 4.2, PROLOG may request additional information from the dynamic knowledge base during evaluation. The dependency maintenance system REDUX at this time acts as an *external database* of facts for the rule interpreter and provides the requested information if possible.

Justification Generation. If one of the rules fired, PROLOG returns all information needed by the *justification generator* to justify the selection or rejection decision. Usually, this corresponds to all problem solving decisions (which are represented explicitly in REDUX) that are responsible for the facts in the dynamic knowledge base, that were needed for the prove to succeed. The generated justifications will be stored in the *dynamic knowledge base* and are used in the further problem solving process. Because rule evaluation results in justifications for REDUX the use of rule-based knowledge is seamlessly integrated into the problem solver.

Figure 3 illustrates the flow of information between the software components within each rule evaluation cycle.

5 Discussion and Conclusions

In the field of constructive problem solving, rules are an essential formalism to describe domain knowledge. The need to evaluate rules efficiently on the one hand, and the advantages of object-oriented programming and knowledge representation on the other hand, led to the approach of combining a SMALLTALK-based problem solver with a PROLOG system.

We consider this solution to be more efficient compared to using rule interpreters written in SMALLTALK like MeiProlog (Aoki et al., 1995) or Neopus (Pachet, 1991). The crucial point in using an external PROLOG systems is, of course, the overhead for inter-process communication. Generally spoken, our approach tends to be more efficient, if the cost for transmitting rules is small compared to the cost of their evaluation. This condition is usually fulfilled when using rejection rules (Section 3.1.3) or preference rules (Section 3.2.3), which both tend to be recursively nested. Control rules (Section 3.2.4) are rather numerous and simple, thus less efficiently handled. We try to keep the rule sets small by avoiding redundancies and to transmit each rule not more than once, in order to minimize transmission cost.

Even more time consuming than the transmission of rules are the numerous queries from the PROLOG interpreter back to the SMALLTALK system. We could avoid many of the queries if we would store facts once retrieved from REDUX in the PROLOG database as well. But this redundant knowledge representation would raise the problem of keeping both databases consistent with each other. Another possibility to increase the performance of the coupling is the use of asynchronous communication, which would reduce the time the SMALLTALK system spends waiting for the answer of a PROLOG request.

Although the problems associated with the hybrid approach presented in this paper are far from being solved, we estimate the use of PROLOG as a major improvement of our problem solving architecture. Future research topics, besides an enhancement of the communication as outlined above, include a partially automated acquisition of preference rules in IDAX and the use of heuristic selection rules in CAPLAN.

References

Aoki, A., Watanabe, K., Matsuoka, T., Nishinaka, Y., and Minagawa, M. (1995). Mei prolog. Available via ftp://ftp.sra.co.jp/pub/lang/smalltalk/mei/VisualWorks2.0/.

Barrett, A. and Weld, D. (1994). Partial-order planning: Evaluating possible efficiency gains. *Artificial Intelligence*, 67(1):71–112.

Barth, G., Paulokat, J., and Richter, M. (1994). Knowledge-based configuration in technical areas, applications, and problems. In Liebowitz, J., editor, *Moving toward Expert Systems Globally in the 21th Century*. Scholium International Inc.

Booch, G. (1991). *Object oriented design with applications.* The Benjamin/Cummings Publishing Company, Inc.

DeJong, G. and Mooney, R. (1986). Explanation-based learning: An alternative view. *Machine Learning*, 1:145–176.

Doyle, J. (1979). A truth maintenance system. *Artificial Intelligence*, 12(3):231–272.

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

Günter, A. (1991). Begriffshierarchie-orientierte Kontrolle. In Cunis, R., Günter, A., and Strecker, H., editors, *Das PLAKON-Buch*, Informatik-Fachberichte, pages 92 – 110. Springer.

Hayes-Roth, B. (1985). A blackboard architecture for control. *Artificial Intelligence*, 26:251 – 301.

Kambhampati, S., Katukam, S., and Qu, Y. (1996). Failure driven dynamic search control for partial order planners: An explanation-based approach. *Artificial Intelligence*, 88(1-2):253–315.

Kambhampati, S., Knoblock, C., and Yang, Q. (1995). Planning as refinement search: A unified framework for evaluating design tradeoffs in partial order planning. *Artificial Intelligence*, 76:167–238.

Katukam, S. and Kambhampati, S. (1994). Learning explanation-based search control rules for partial order planning. In *Proceedings of AAAI-94*, pages 582–587.

Maurer, F. and Paulokat, J. (1996). Operationalizing conceptual models based on a model of dependencies. In *Proc. of the 11th European Conference on Artificial Intelligence (ECAI-94)*.

Maurer, F. and Pews, G. (1996). Supporting cooperative work in urban land-use planning. In *Proc. of COOP-96*.

McAllester, D. and Rosenblitt, D. (1991). Systematic nonlinear planning. In *Proceedings of AAAI-91*, pages 634–639.

Minton, S. (1988). *Learning Search Control Knowledge: An Explanation-Based Approach*. Kluwer Academic Publishers, Boston.

Minton, S., Bresina, J., and Drummond, M. (1994). Total-order and partial-order planning: A comparative analysis. *Journal of Artificial Intelligence Research*, 2:227–262.

Mitchell, T., Keller, R., and Kedar-Cabelli, S. (1986). Explanation-based generalization: A unifying view. *Machine Learning*, 1:47–80.

Mittal, S. and Frayman, F. (1989). Towards a generic model of configuration tasks. In *Proceedings of the 11th IJCAI*. Morgan Kaufman.

Muñoz-Avila, H. and Weberskirch, F. (1996a). Planning for manufacturing workpieces by storing, indexing and replaying planning decisions. In *Proceedings of the 3rd International Conference on AI Planning Systems (AIPS-96)*. AAAI-Press.

Muñoz-Avila, H. and Weberskirch, F. (1996b). A specification of the domain of process planning: Properties, problems and solutions. Technical Report LSA-96-10E, Centre for Learning Systems and Applications, University of Kaiserslautern, Germany.

Pachet, F. (1991). Reasoning with objects: The neopus environment. Internal Report 13/91, University of Paris VI.

Paulokat, J. (1995). Entscheidungsorientierte Rechtfertigungsverwaltung zur Unterstützung des Konfigurationsprozesses in IDAX. In *XPS 95: Beiträge zur 3. Deutschen Expertensystemtagung*, pages 19–36. infix.

Petrie, C. (1991). *Planning and Replanning with Reason Maintenance*. PhD thesis, University of Texas at Austin, Computer Science Dept.

Petrie, C. (1992). Constrained decision revision. In *Proceedings of AAAI-92*, pages 393–400.

Roth-Berghofer, T. (1996). Explanation-based learning of control information from failures in planning. Masters thesis (in german), University of Kaiserslautern.

Schirp, W. (1996). Einsatz expliziter Begründungen in einem Assistenzsystem für Konfigurationsaufgaben. In Sauer, J., Günter, A., and Hertzberg, J., editors, *Planen und Konfigurieren 96*, number 3 in Proceedings in Artificial Intelligence, pages 75 – 86. infix.

von Wangenheim, A. (1996). *Cyclops: Ein Konfigurationsansatz zur Integration hybrider Systeme am Beispiel der Bildauswertung*. Dissertation, Universität Kaiserslautern.

Weberskirch, F. (1995). Combining SNLP-like planning and dependency-maintenance. Technical Report LSA-95-10E, Centre for Learning Systems and Applications, University of Kaiserslautern, Germany.

Weberskirch, F. and Muñoz-Avila, H. (1997). Advantages of types in partial-order planning. In Müller, M., Schumann, O., and Schumann, S., editors, *Beiträge zum 11. Workshop 'Planen und Konfigurieren' (PuK-97)*, number FR-1997-001 in FORWISS-REPORT. FORWISS.

Wielemaker, J. (1993). *SWI-Prolog 1.8*. University of Amsterdam, Dept. of Social Science Informatics (SWI), Roeterstraat 15, 1018 WB Amsterdam.