In International Journal of Expert Systems, 9(4), 1996.

Conceptual and Formal Specifications of Problem-Solving Methods

Dieter Fensel[1], Henrik Eriksson[2], Mark A. Musen[3], and Rudi Studer[4]

[1]Department SWI, University of Amsterdam, Roetersstraat 15, 1018 WB Amsterdam, The Netherlands; e-mail: fensel@swi.psy.uva.nl

[2]Application Systems Laboratory, Department of Computer and Information Science, Linköping University, S-581 83 Linköping, Sweden; e-mail: her@ida.liu.se

[3]Medical Computer Science Group, Knowledge Systems Laboratory, Stanford University School of Medicine, Stanford, California 94305-5479, U.S.A.; e-mail: musen@camis.stanford.edu

[4]Institute AIFB, University of Karlsruhe, D-76128 Karlsruhe, Germany; e-mail: studer@aifb.uni-karlsruhe.de

Abstract Reusable problem-solving methods as provided by the PROTÉGÉ-II improve knowledge engineering by allowing developers to design reasoners quickly from pre-existing components. The PROTÉGÉ-II approach allows developers to select methods from a library, and to map the methods to a domain ontology. Still, these methods lack a clear conceptual and formal description that would enable their reuse through matching their competence and assumptions with the available domain knowledge and the given task. KARL is a conceptual and formal knowledge-specification language that provides modeling primitives for specifying problem-solving methods. In this paper, we show how the code and informal descriptions of problem-solving methods in PROTÉGÉ-II can be complemented with the conceptual and formal method definitions in KARL. For our case study we choose two methods from the PROTÉGÉ-II framework: chronological backtracking and a task-specific refinement, the board-game method. In addition to the conceptual and formal specification of these methods, we provide insights in the refinement of general-purpose methods to task-specific (i.e., strong) problem-solving methods. We further show how a task-specific method can be adapted to a given domain and application. In the case of both methods, we achieve this adaptation by introducing ontological commitments over the terminological structure of the entities used to describe the states of the reasoning process, and by using these terminological structure to define state transitions of larger grainsize.

1 Introduction

Developers are designing knowledge-based systems for a wide range of application domains. Although many application tasks have similar features, from the computer scientist's perspective, few solutions are reused across domains and tasks. Chandrasekaran (1983, 1986) recognized the potential for reusable design components in knowledge engineering. McDermott (1988) proposed the use of role-limiting problem-solving methods as the basis for the development of knowledge-based systems. Currently, several research groups are working on knowledge-engineering approaches that incorporate reusable problem-solving methods (Klinker et al., 1991; Musen, 1992; Puppe 1993; Steels 1990; Wielinga et al., 1992; Chandrasekaran & Johnson, 1993; Breuker & Van de Velde, 1994; Schreiber et al., 1994a; Benjamins, 1995). An underlying assumption in most of these approaches is that reusable methods can be stored in libraries. Developers can retrieve, specialize, and combine these methods to from the reasoning part of a knowledge-based system.

The PROTÉGÉ-II system (Puerta et al., 1992; Eriksson et al., 1995) allows developers to build knowledge-based systems from reusable problem-solving methods. The PROTÉGÉ-II approach distinguishes between *tasks* that the final knowledge-based system should perform and *problem-solving methods* that carry out such tasks. In the PROTÉGÉ-II approach, the developer performs a *task analysis*, and uses the PROTÉGÉ-II environment to select from a library of reusable methods a method appropriate for the task. Examples of such methods are state-space search by chronological backtracking, propose and revise, skeletal-plan refinement, and temporal abstraction. Because such library methods are designed to be applicable to many domains, the developer must *configure* the methods to perform new tasks by mapping the methods to the domain ontology. Moreover, methods can delegate problems as *subtasks*, that other methods or *mechanisms* must perform. In PROTÉGÉ-II, mechanisms are methods that cannot be decomposed into subtasks. Because mechanisms accomplish their task without delegation, the mechanisms are black boxes from the developer's point of view. In addition to assisting the developer in the design of the performance element of the target knowledge-based system, the PROTÉGÉ-II environment supports knowledge acquisition from domain experts by generating a domain-specific knowledge-acquisition tool, which elicits the expertise required by the problem-solving methods (Eriksson et al., 1995). PROTÉGÉ-II thus supports two important aspects of the design of knowledge-based systems: the selection and configuration of problem-solving methods for the task, and the acquisition of the domain knowledge required by those methods to accomplish their task.

Unfortunately, there are many obstacles to the reuse of problem-solving methods. Workers in software engineering have studied software reuse for several decades. Although software reuse promises to reduce the cost of software development, practical software reuse has proved to be difficult (Krueger, 1992). Currently, the most successful examples of software reuse are mathematical and statistical libraries, where developers can retrieve by name library functions that have a well-defined and well-documented meaning. Problem-solving methods, unlike these mathematical software components, are often designed to perform complex inferences using knowledge bases. Their description and adaptation to specific tasks and domains therefore require much more effort. Each method defines a whole family of variants that depend on the precise assumptions about the task and the required domain knowledge (see Fensel, 1995a). Support in refining problem-solving methods to a given task and domain is an essential precondition for successful method reuse. Other preconditions are a notion for the competence of methods (Akkermans, 1993), and appropriate indexing of methods.

To identify the requirements for formal method representations and the relative benefits of formal languages versus source code in method reuse, we have respecified two of the problem-solving methods used in the PROTÉGÉ-II framework in the knowledge-specification language KARL (Fensel, 1995; Fensel et al., 1996a), developed as part of the MIKE project. MIKE defines an engineering framework for interpreting, formalizing, and implementing knowledge to build knowledge-based systems (see Angele et al., 1996). The formal and operational specification language, KARL, is provided for specifying a model of expertise. A hypertext-based model (Neubert, 1993) mediates between natural-language protocols and the formally specified model of expertise. Thus, a communication basis is established between the knowledge engineer and the expert. A design model is provided for capturing both nonfunctional requirements and design decisions that are influenced by those requirements (Landes and Studer, 1995).

Because KARL combines logic with a conceptual model to describe a knowledge-based system it affords two advantages. First, the system (or the problem-solving method) is described at a conceptual level according to the KADS models of expertise (Wielinga et al., 1992; Schreiber et al., 1994b). The specification therefore can be used for a conceptual analysis of the method. The smooth transition from a semiformal to a formal specification improves the understandability of these specifications. Second, the system (or the problem-solving method) is described at a formal level avoiding the shortcomings of natural language descriptions, such as ambiguity or imprecision. KARL combines logic programming, object-orientation, and dynamic logic for this purpose (Kifer et al., 1995; Harel, 1984).

In this paper, we shall discuss lessons learned from specifying reusable building blocks of the PROTÉGÉ-II framework with KARL. We show the usefulness of a formal and conceptual specification of these buildings blocks. Understandability and preciseness of the description can be achieved without reference to implementation details. These implementation details neither are necessary nor improve understanding of a problem-solving method. Part of implementing a method is modifying the methods conceptual description to gain an efficient implementation. Therefore, it is not possible to directly abstract the conceptual description of a method (i.e., its main rationale) from its implementation.

We describe how we used KARL to specify different variants of a method. We started by specifying the classic search method of chronological backtracking. We then defined the board-game method as a task-specific refinement of chronological backtracking, and mapped these methods to the Sisyphus room-assignment task (cf. Linster, 1994).[1] We show how such a task-specific and domain-specific refinement of a general-purpose problem-solving method can be achieved. We did not achieve this refinement by introducing additional inference actions, by using additional levels of hierarchical refinement at the inference layer, or by changing the task layer. Instead we added terminological knowledge and new logical relationships to the specification of roles and elementary inference actions. That is, we achieved the task- and domain-specific refinement of problem-solving methods by introducing ontological commitments over the entities used to describe the input, intermediate results, and output of the reasoning process of the method.

This paper is organized as follows. In Section 2, we give the main rational underlying specification languages for knowledge-based systems, and briefly characterize the specification language KARL. In Section 3, we discuss two problem-solving methods: the chronological-backtracking method and the board-game method. Section 4 provides a partial KARL specification of the methods. In Section 5, we discuss the mapping of both methods to the Sisyphus room-assignment problem (Linster, 1994). We thus illustrate the task-, domain-, and application-specific refinements of both methods. Section 6 concludes by comparing our results with those of related work, and by pointing out lines of future research.

2 Formal Specifications at the Knowledge Level

First, we discuss the need of formal specifications of problem-solving methods that abstracts

1. We used the Sisyphus room-assignment problem (Linster, 1994) to compare different approaches in knowledge engineering. It defines an assignment problem, in which employees are assigned to offices under several contraints (see section 5).

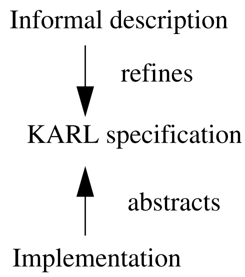


Fig. 1. The intermediate level between informal descriptions and implementations.

from the details of implementation but that still allow us to define the reasoning behavior (i.e., the *how*) of the problem-solving process. Second, we sketch the language KARL, which enables such specifications.

2.1 Abstract on the Implementation

> The knowledge level "is characterized by knowledge as the medium and the principle of rationality as the law of behavior" (Newell, 1982).

We shall discuss first the need to abstract from concerns that are related to the efficiency of the implementation. We shall then show that the efficiency of the problem-solving process remains an important issue during knowledge-level modeling.

2.1.1 Abstracting from Efficiency

A specification of a problem-solving method refines the informal description but is still more abstract then its implementation. A specification can neglect implementation details, such as storage allocation or other issues that are related to the fact that such functionality is realized by a program running on a computer. In addition, a specification also ignores a *specific purpose* of an implementation: realizing an *efficient* computation of the specified behavior. A formal specification in a language such as KARL defines an *intermediate level* between a description of a problem-solving method at the conceptual level and an implementation (Figure 1). It refines the description at the conceptual level without clouding it with additional knowledge about the effective and efficient implementation of the problem-solving method in a given software environment. An implementation normally not only provides too much detail in describing a method, but also does not provide an adequate description of the method. The basis for the respecification of the chronological-backtracking and board-game methods in KARL was an informal model of the methods. Initially, we attempted to model the methods from the implementation of the methods in CLIPS code. However, we realized quickly that the basic problem-solving structures of these methods cannot be found in the same manner in the implementation. For reasons of efficiency, the different main reasoning steps of a method are partially merged together. An implementation not only provides unnecessary information, but also hides information. Its purpose is not the adequate conceptual description of a problem-solving method (i.e., the main rationale), but rather an efficient realization by a computer program. Therefore, it was appropriate to identify the basic problem-solving structures of these methods from their informal descriptions. The ambiguity and incompleteness of the informal description had to be resolved by intensive discussions with the method developer.

2.1.2 Modeling Efficiency Concerns at the Knowledge Level

Distinguishing between the specification and the implementation in knowledge engineering reflects the distinguishing between specification and design or implementation in software engineering. In software engineering, the distinction between a functional specification and the design or implementation of a system is often discussed as a separation of *what* and *how*: "The generation of system-level requirements is, to the extent possible, a pure *what*, addressing the desired characteristics of the complete system. The next steps, determining the next level of the hierarchy and allocating system requirements to the elements, are in fact a *how*" (Dorfman, 1990). During the specification phase, what the system should do is established in interaction with the users. During design and implementation, how the system does its tasks (i.e., which algorithmic solution can be applied) is defined. This separation does *not* work in the same way in the domain of knowledge-based systems, because the efficiency of the reasoning process must be considered during the specification phase. In general, most problems tackled with knowledge-based systems are inherently complex and intractable, see Bylander (1991), Bylander et al. (1991), and Nebel (1996). A problem-solving method has to describe not just a realization of the functionality, but rather one that takes into account the constraints of the reasoning process and the complexity of the task. At the knowledge level, the knowledge required by a knowledge-based system to solve the task efficiently is described in an implementation-independent manner. This knowledge is not made up of efficient algorithms and data structures, rather, exists as domain- and task-specific heuristics developed by an expert as a result of experience. "In simple terms this means analysis is not simply interested in *what* happens, as in conventional systems, but also with *how* and *why"* (Brooking, 1986). One must acquire not only knowledge about what a solution for a given problem is but also knowledge about how to derive such a solution in an efficient manner.

As a consequence, we must distinguish two kinds of knowledge of *how* to solve the problem in knowledge engineering. "There is a difference between what we would call respectively *knowledge-level control* and *symbol-level control*" (Schreiber, 1992). At the knowledge level, there is a description of the domain knowledge and the problem-solving method that an agent requires to solve the problem effectively and efficiently. This knowledge must have been modeled during the specification phase. At the symbol level, there is a description of efficient algorithmic solutions and data structures for implementing an efficient computer program (i.e., a specific agent). As it can be in software engineering, this type of knowledge can be added during the design and implementation of the system. Therefore, a requirement for languages that specify knowledge-based systems is that they must *combine nonfunctional and functional specification techniques*: On the one hand, it must be possible to express nonfunctional algorithmic control over the dynamic reasoning process of the substeps. On the other hand, it must be possible to characterize substeps of the reasoning process only functionally, without making commitments to their algorithmic realization.

2.2 The Knowledge Specification Language KARL

KARL is a specification language for knowledge-based systems. It provides language primitives to represent knowledge according to the layers of a KADS model of expertise (Wielinga et al., 1992; Schreiber et al., 1994b). It distinguishes different types of knowledge, and defines different language primitives for them. It provides a *formal* semantics that is operationalized and implemented, thus allowing prototyping as a means to evaluate a specification.

The conceptual model of KARL distinguishes three types of knowledge. The *domain layer* contains domain-specific knowledge about concepts, their features, and the relationships among them. The *inference layer* defines the inferences of the problem-solving process, and describes the role of the domain knowledge for inferences. This layer describes the logical inferences and their data dependencies. The *task layer* defines the control flow of the problem-solving method. This layer specifies the sequence, alternatives, and loops of the inferences. KARL provides two sublanguages, P-KARL and L-KARL, for specifying such a conceptual model.

2.2.1 The Domain Layer and the Inference Layer

KARL uses concepts of object-oriented databases and logic for specifying the domain layer, the inference layer, and their connections. KARL provides the sublanguage Logical-KARL (L-KARL) for this purpose. L-KARL is derived from Frame-logic (F-logic) of Kifer, Lausen and Wu (1995). Terminological knowledge can be described by a taxonomy of classes. Attributes can be defined for each class, and are inherited according to the taxonomy. Additional knowledge can be described with logical formulae. A domain layer is structured and ordered hierarchically by the is-a hierarchy between classes and by a module hierarchy.

In addition to its use at the domain layer, L-KARL is used to specify the logical relationships defined by inference actions at the inference layer. Extending KADS, L-KARL can be used to define a terminological structure of a knowledge role. In KADS, such roles are flat containers, whereas, in KARL, they can be used to define a terminology that is specific to a problem-solving-method-specific independent of the domain-specific terminology. The need for such a terminology is one of the most significant results of the role-limiting method approach (Marcus, 1988; Puppe, 1993).[2]

An inference layer in KARL can be *refined hierarchically,* similar to leveled dataflow diagrams (Yourdon, 1989). KARL distinguishes between roles that model the dataflow between inference actions, and roles that define a connection between the domain and inference layers. Roles of the first type are called *stores*. Roles of the second type are called *views* when they define an upward mapping from the domain to the inference layer, and are called *terminators* when they are used to write the results of the inference layer back on the domain layer. The inference diagrams that are used in this paper were inspired by Wielinga et al. (1992), but have been modified to incorporate these additional features of KARL.

2.2.2 The Task Layer

The sublanguage Procedural-KARL (P-KARL) is used for specifying the control flow of a problem-solving method by sequence, branch, loop, and procedure call. The elementary program statements are the execution of an inference action. Conditions can be specified via logical statements about the contents of knowledge roles. The syntax of P-KARL is a customization of *dynamic logic* (cf. Harel, 1984; Kozen, 1990). We chose dynamic logic because it incorporates the notions of states and state transitions into a logical framework.

2.2.3 Graphical Representations

KARL provides graphical representations of most modeling primitives (Figures 2 and 3). Every graphical symbol has a defined meaning given by the semantics of the corresponding language primitive. It combines the formal semantics with the conceptual model used to describe the system: enhanced Entity Relationship (EER) diagrams (Hull and King, 1987) are used to represent terminological knowledge at the domain and inference layers (see Figure 2),

---

2. The lack of such a task- and problem-solving-specific terminology in KADS was criticized by Causse (1994), who provided a different approach for specifying the reasoning process of knowledge-based systems. The entire state of the reasoning process is described by an object, and the roles in KADS are subobjects (i.e., values of complex attributes) of this object. The roles are therefore embedded in a terminological structure, and inferences are modeled as methods of objects. KARL defines a mid-position between the inference-oriented approach of KADS and the role-oriented approach of Causse. It provides additional terminological structure compared to KADS, but keeps the inferences as independent modeling primitives.

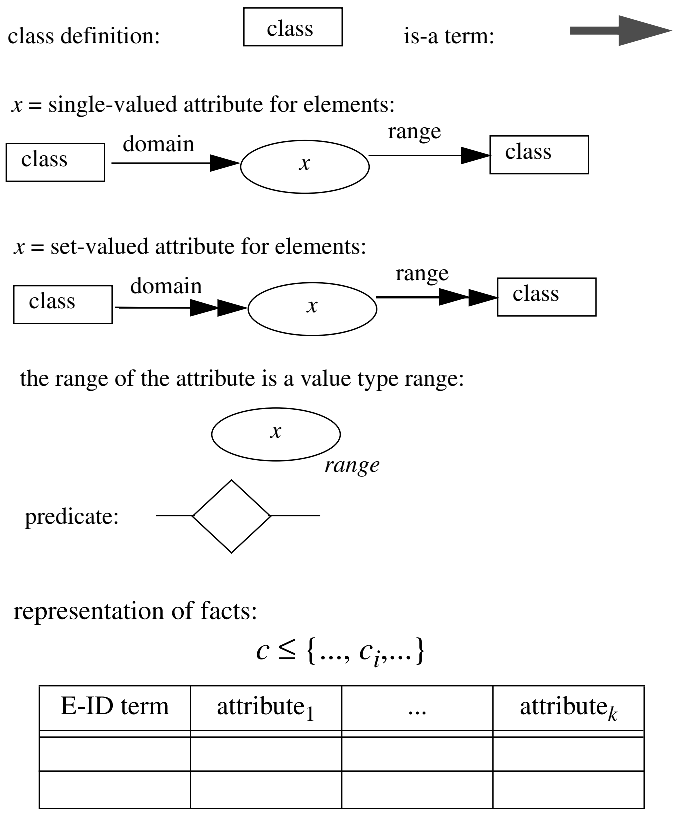


Fig. 2. Graphical modeling primitives of L-KARL.

leveled dataflow diagrams are used at the inference layer (see Figure 3), and program-flow diagrams are used at the task layer.

2.2.4 Comparison of KARL with Other Languages

The terminological modeling primitives, provided to express the structure of the static knowledge, constitute a significant distinction between KARL and other formal specification languages for knowledge-based systems such as DESIRE (van Langevelde, Philipsen and Treur, 1993), $K_{BS}SF$ (Spee & in 't Veld, 1994), $(ML)^2$ (van Harmelen & Balder, 1992), and, from software engineering, specification languages such as VDM-SL (Jones, 1990) and Z (Spivey, 1992). The rich terminological structure that is provided by KARL is based on

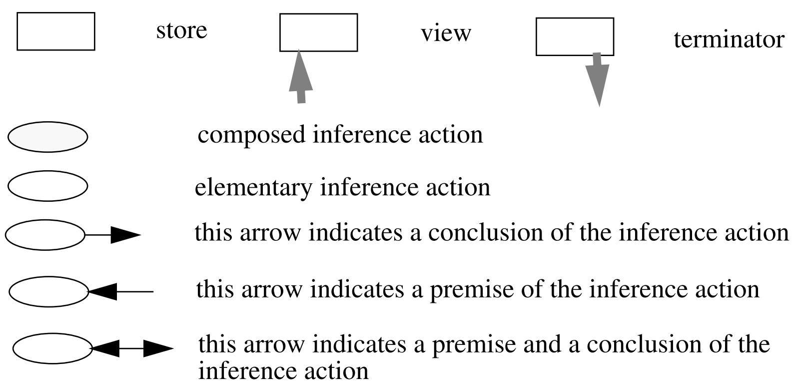


Fig. 3. Graphical modeling primitives at the inference layer.

specification languages for information systems and databases, such as F-logic (Kifer et al., 1995), TROLL (Jungclaus, 1993), and INCOME (Oberweis et al., 1994). The main advantages of these modeling primitives are the following:

- There is a close relationship between the semiformal and formal models. In the case of KADS, the rich conceptual models that can be specified semiformally with the conceptual modeling lLanguage CML (Schreiber et al., 1994a) must be translated to $(ML)^2$, which provides the usual means of first-order logic for specification (i.e., constants, functions, predicates, and types). As a consequence, epistemological distinctions of the conceptual model disappear in the formal model, and the two models have to be discussed separately. In the case of KARL, the semiformal and the formal models use the same primitives; the formal model is simply a refinement of the semiformal one (cf. Fensel & Neubert, 1994; Angele et al., 1996).
- The terminological structure can be used for the automatic creation of the knowledge acquisition editors that are part of the PROTÉGÉ-II approach (Eriksson et al., 1995).
- The terminological modeling primitives of KARL can characterize the task- and domain-specific refinement process of problem-solving methods. We achieve such refinement by adding terminological structure and logical axioms to a given specification (Section 5).

More detailed comparisons of KARL with other specification languages that also cover other aspects are provided by Fensel & Harmelen (1994) and by Fensel (1995c).

3 Chronological Backtracking and the Board-Game Method

We use two problem-solving methods to illustrate method specification in KARL. In this section, we provide intuitive descriptions of the methods, and describe briefly how these methods are defined in the PROTÉGÉ-II framework. Section 4 discusses the KARL models of these methods.

State-space search is a fundamental artificial-intelligence technique for problem solving. We can model many tasks in terms of states, and we can encode many problem-solving behaviors as a search through a state space. The *chronological-backtracking* method searches for a sequence of states, where the initial and goal states are given, and where two consecutive states in the sequence satisfy constraints on how states can follow one another. It backtracks to the final state that provides successor states that have not already been selected if a death-end of the search appears. The concept of state transitions is important for state-space search. For the chronological-backtracking method, PROTÉGÉ-II uses a *transition function* (*T-function*) that produces a set of subsequent states given the current state. To reuse the chronological-backtracking method for new tasks, the developer must define the initial and goal states (or a goal predicate) and the T-function.

The *board-game* method (Eriksson et al., 1995) supports tasks that the developer can model as a one-player board game. The method adopts a cognitive model where game *pieces* move between board *locations* under certain *constraints*. The method assumes that the game has a fixed number of pieces and locations. Operationally, the board-game method searches through the space of game states by performing legal *move operations* on the current state to generate subsequent states.

To configure the board-game method to support a new board-game task, the developer must identify the pieces and locations of the game, and must define the constraints under which pieces can move. For example, the rules of the game might stipulate that the target location for a move must be empty. Also, the board-game method requires as input the initial and goal states of the game (or a goal predicate that is satisfied when the goal is reached). In addition to constraints on moves, certain games define constraints in terms of illegal game states. Consider, for instance, the cannibals-and-missionaries problem. A legal move in terms of transportation may result in a forbidden state where missionaries are cannibalized (i.e., there are more cannibals than missionaries at one shore). The developer must configure the board-game method to avoid such illegal states in the game.

The board-game method is generic in the sense that the developer can use the method to perform many game like tasks. For instance, we have configured the board-game method to perform the following tasks: towers of Hanoi, cannibals and missionaries, and Sisyphus room assignment (Eriksson et al., 1995). We can model the Sisyphus room-assignment task as a board game by viewing the rooms as locations, and the persons as pieces. Initially, all persons are located outside the building (i.e., at the unassigned location), and the method moves one person at the time to an appropriate room. If illegal states occur, the board-game method backtracks to a legal state and examines a different search path.

4 The Models in KARL

This section presents highlights from the KARL specifications of the methods introduced in Section 3. In Section 4.1, we specify chronological backtracking. In Section 4.2, we discuss the refinements and extensions that are required for developing the board-game method specification from the chronological backtracking specification.

4.1 Chronological Backtracking: A Weak Method

In the following, we illustrate how chronological backtracking is modeled in KARL. The chronological-backtracking method can use *depth-first search*, *beam search*, or *breadth-first search*[3] (Bundy, 1990). Whether the specification of the method selects one or several successor states depends on a *preference* predicate (called an *S-Function* in Eriksson et al., 1995). The search collapses to depth-first search if the domain knowledge used to define the preference predicate defines a total ordering. In domains where we do not have sufficient knowledge to identify a single best state, all subsequent states that cannot be distinguished are considered in parallel. In this case, beam search is used. Finally, if no domain knowledge is provided, the search becomes full breadth-first search. Therefore, which type of search is done

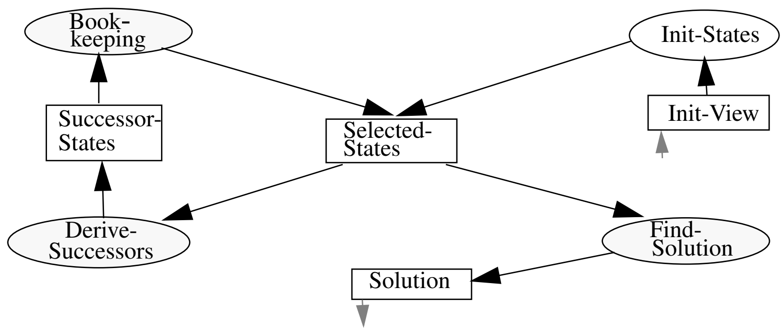


Fig. 4. Inference structure of *chronological backtracking*.

[3] In that case, the backtracking disappears, because all possible successor states of earlier states have already been checked.

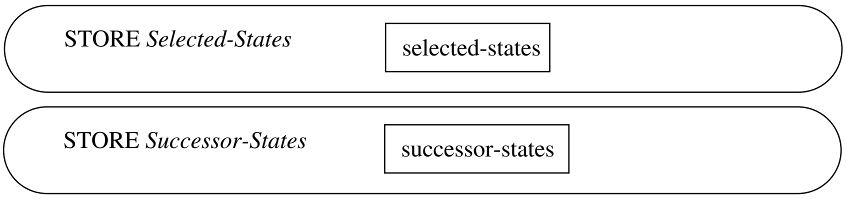


Fig. 5. The definitions of the stores *Selected-States* and *Successor-States*.

depends on the properties of the domain knowledge provided to define the preference relation for a given application. Figure 4 shows the inference structure of chronological backtracking. It consists of three subtasks (i.e., composed inference actions) and one elementary inference action:

- The elementary inference action *Init-States* creates all initial states. *Init-States* uses the view *Init-View* to read the necessary knowledge from a domain layer.
- *Find-Solution* checks to see whether the store *Selected-States* contains a goal state. If it finds a goal state, it stores the best state (according to the *preference* predicate) in the domain layer via the terminator *Solution*.
- *Derive-Successor* derives the successors of all selected states, and checks whether they are legal states.
- *Bookkeeping* checks whether these successors are different from already-considered states. If new states are found, the best of them are put into the store *Selected-States*. As mentioned previously, if the domain preference knowledge defines a total ordering on all states, one state is chosen. If the preference knowledge defines only a partial ordering, the *Bookkeeping* inference action chooses several states. Bookkeeping provides an earlier state according to backtracking if no new states could be derived.

Figure 5 shows the definition of the stores *Selected-States* and *Successor-States*. Each store contains only one class definition without any attributes, because chronological backtracking does not make any assumptions about the internal structure of a state. This simplicity is shown in Figure 5. We shall demonstrate how the board-game method adds (compare Figure 5 to Figure 10).

The control flow of the method is defined at the task layer. Figure 6 shows the control flow at the top-level of the method specification. After initialization (by *Init-States*), the iteration of the three composed inference actions is repeated as long as no solution has been found and a further successor that is different from an already-considered state can be found.

```
Selected-States := Init-States(Init-View)
while ¬∃x ∈ Solution ∧ ∃x ∈ Selected-States
    Solution := Find-Solution(Selected-States);
    if no Solution
    then
        Successor-States := Derive-Successors(Selected-States);
        Selected-States := Bookkeeping(Successor-States )
    endif
endwhile
```

Fig. 6. The task layer of *chronological backtracking*.

4.1.1 Refinement of Derive-Successors

In the following, we show the refinement of the composed inference action *Derive-Successors*. Due to space limitation, we omit the refinement of the other two composed inference actions.[4] The composed inference action *Derive-Successor* (Figure 7) derives the successors of all selected states, and checks whether these successors are legal states.

The store *Possible-Successors* contains the current successor states of the states in the store *Selected-States*. The definition of the store *Possible-Successors* is similar to those of the stores *Selected-States* and *Successor-States*. We use the view *Successor-Relation* (Figure 8) to define the successor relationship (i.e., the T-function) of states.

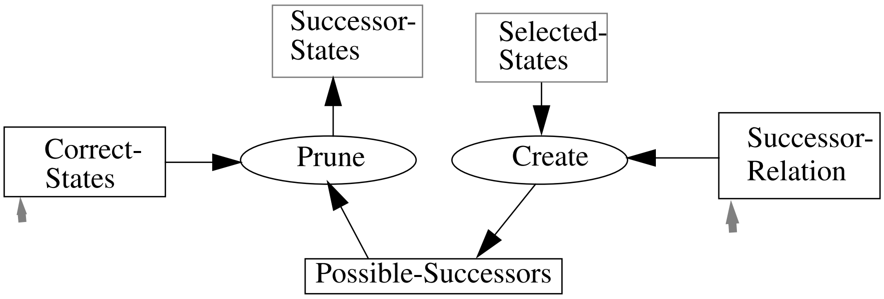


Fig. 7. Refinement of *Derive-Successors*.

The elementary inference action *Create* (Figure 9) creates all successor states from the states in the store *Selected-States*. The successors are written to the store *Possible-Successors*. *Create* uses the view *Successor-Relation* to read the required knowledge from the domain layer. The elementary inference action *Prune* (Figure 9) checks whether these states are legal. States that are legal are considered to be new successor states, and are written to the store *Successor-States*. The view *Correct-States* defines the class *correct-states*.

Create and *Prune* use different types of knowledge. *Create* uses general knowledge about the search space. In a board-game application, *Create* generates all possible successor states that a player can achieve by moving one piece. *Prune* tests whether these moves lead to a situation that is legal according to the rules of the game. Conceptually, this is, most of the time, a useful way to describe the different inferences and their different knowledge types. On the other hand, this generate & test strategy clearly will be changed during implementation. For efficiency reasons ones tries to compile into the generation step as much as possible of the knowledge of correct states (cf. Fensel & Straatman, 1996). Actually, the two different reasoning steps and knowledge types became merged in the implementation of the board-game method, to improve their efficiency.

4.2 Task-Specific Refinement: The Board-Game Method

The specification of chronological backtracking is trivial. It defines only a search strategy. We now show how we can refine such a search strategy to a strong problem-solving method by making assumptions about the task that it can solve. We introduce a KARL specification for the board-game method. We assume that the game has a fixed number of pieces and locations. Multiple pieces can be moved to the same location simultaneously, and, if required, the pieces at each location can be ordered.

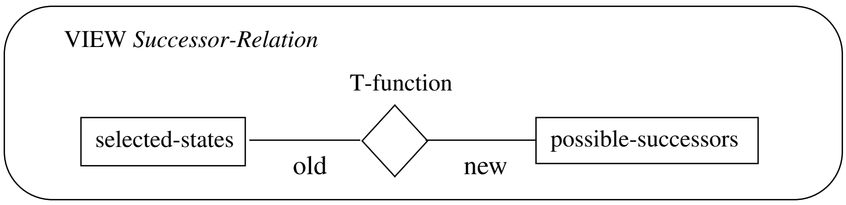


Fig. 8. The definition of the view *Successor-Relation*.

---

[4] See Fensel (1993) for further details.

ELEMENTARY INFERENCE ACTION *Create*
  PREMISES
    Selected-States, Successor-Relation;
  CONCLUSIONS
    Possible-Successors;
  RULES
    $\forall x_{St}, \forall y_{St}$
      $y_{St} \in$ *possible-successors* $\leftarrow$
        $x_{St} \in$ *selected_states* $\wedge$ *T-function(old :* x_{St}*, new :*y_{St}*).*
END

ELEMENTARY INFERENCE ACTION *Prune*
  PREMISES
    Correct-States, Possible-Successors;
  CONCLUSIONS
    Successor-States;
  RULES
    /* Every possible successor state that is correct is assumed to be a new element of the class *successor-states*. */
    $\forall x_{St}$ ($x_{St} \in$ *successor-states* $\leftarrow x_{St} \in$ *possible-successors* $\wedge$ *correct-states*(x_{St})*.*
END;[1]

1. For a definition of the logical symbols see Figure 11.

Fig. 9. The elementary inference actions *Create* and *Prune*.

We can define the board-game method as a refinement of chronological backtracking. We do not achieve this refinement by introducing additional inference actions, by using additional levels of refinement, or by changing the task layer; rather, we do so *by adding (problem-solving-method-specific) terminological knowledge and new clauses to roles and elementary inference actions*.

The specification of the board-game method introduces definitions of new classes and predicates, as well as new attributes in the existing class definitions.

- We introduce three new class definitions in all stores. An assignment is now a triple consisting of a *location*, a *piece*, and an *ordering* of pieces (Figure 10a). The order information is relevant in domains such as the towers-of-Hanoi game, where several pieces share the same location, but differ in their sequencing.
- All other original class definitions in roles and elementary inference actions of chronological backtracking are extended by the set-valued attribute *assign*. For example, a *selected-state* is now defined as shown in Figure 10b. The attribute *assign* assigns a set of *assignments* to each element of *selected-states*.

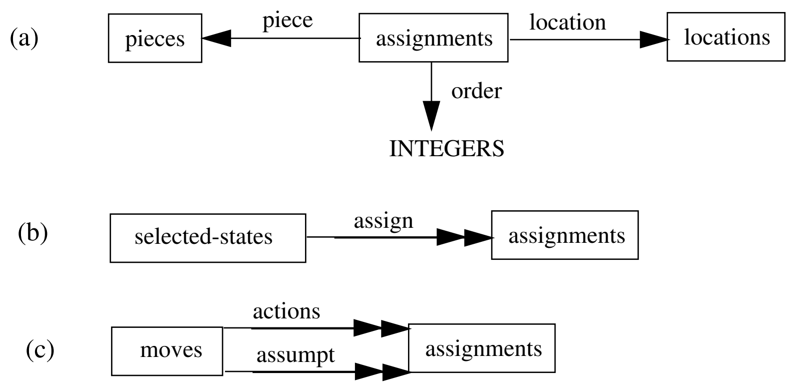


Fig. 10. Additional terminological structure of the board-game method.

- We introduce the class *moves* in the view *Successor-Relation*. We use *moves* to specify the predicate *T-function* used by the elementary inference action *Create*. The attribute *assumpt* describes the assumptions that states must fulfill for a move to be applied to the state. The attribute *actions* describes the modified assignment. That is, the result of applying the move to a state. It is a set of triples of pieces, locations, and ordering of the pieces that describes the new assignments of the state (Figure 10c).

New knowledge about the inferences is added primarily to the elementary inference action *Create*.[5] That is, the inference action *Prune* remains unchanged. We use the newly introduced terminology to define in detail how a successor state is derived. In that refined version of

ELEMENTARY INFERENCE ACTION *Create*
PREMISES
Selected-States, Successor-Relation;
CONCLUSIONS
Possible-Successors;
RULES
$\forall x_{St}, \forall y_{St}$
$y_{St} \in possible\text{-}successors \leftarrow$
$x_{St} \in selected\_states \wedge T\text{-}function(old : x_{St}, new : y_{St}).$
/* A move defines a successor of a selected state if its assumptions are a subset of the assignments of this state. */
$\forall x_m, \forall x_{St}$
$T\text{-}function(old : x_{St}, new : NewSt(x_{St}, x_m))^1 \leftarrow$
$x_m \in moves \wedge$
$x_{St} \in selected\text{-}states \wedge$
$apply(state : x_{St}, move : x_m).$
$\forall x_m, \forall x_{St}$
$apply(state : x_{St}, move : x_m) \leftarrow$
$x_m \in moves \wedge x_{St} \in selected\text{-}states \wedge$
$\neg\exists x_A\ (x_m[assumpt :: \{x_A\}] \wedge \neg x_{St}[assign :: \{x_A\}]).$
/* A successor gets the *changed* assignments. */
$\forall x_A, \forall x_l, \forall x_m, \forall x_p, \forall x_O, \forall x_{St}$
$NewSt(x_{St}, x_m)[assign :: \{x_A[piece : x_p, order : x_O, location : x_l]\}] \leftarrow$
$x_m[actions :: \{x_A[piece : x_p, order : x_O, location : x_l]\}] \in moves \wedge$
$T\text{-}function(old : x_{St}, new : NewSt(x_{St}, x_m)).$
/* A successor gets the *unchanged* assignments. An assignment of a piece remains unchanged if the actions of a move do not contain this piece. */
$\forall x_A, \forall x_m, \forall x_p, \forall x_{St}$
$NewSt(x_{St}, x_m)[assign :: \{x_A\}] \leftarrow$
$x_m \in moves \wedge$
$x_{St}[assign :: \{x_A[piece : x_p]\}] \in selected\text{-}states \wedge$
$T\text{-}function(old : x_{St}, new : NewSt(x_{St}, x_m)) \wedge$
$not\text{-}changed\text{-}pieces(state : x_{St}, move : x_m, pieces :: \{x_p\}) \wedge$
$\neg\exists y_A\ (x_m[actions :: \{y_A[piece : x_p]\}]).$
END

1. The object-id of the new state depends functionally on the object-id of the old state and on the object-id of the *move*. Further features of L-KARL are the following:
- $x \in y$ means that x is an element of the class y;
- $x \cong y$ means that x and y are equal;
- x[*attribute* : y] means that x has the value y for the attribute *attribute*;
- x[*attribute* :: $\{y\}$] means that x has the set of values y for the attribute *attribute*;
- p(*name* : x, ...) means that p is a predicate, *name* is the argument name, and x is an argument.

Fig. 11. Extended definition of the elementary inference action *Create*.

5. One new clause must also be introduced in the inference actions *Init-States* and *Bookkeeping*.

Create, we define a successor state by the elements of the class *moves*. A move is a tuple of two sets: a set of actions and a set of assumptions. An *action* derives a new location (or a new ordering) for a piece. When we configure the board-game method to a specific domain, we must define a move, instead of defining the general *T-function* in the upward mapping of the view *Successor-Relation*. The *assumptions* of a *move* model the conditions necessary for applying it in a given situation such as that a piece can move to only a location that is not already occupied by another piece. Figure 11 shows the extended definition of *Create*.

5 Domain and Task-Specific Refinements of Both Methods

In Section 4, we discussed parts of the formal specifications of the chronological backtracking and the board-game method. In this section, we discuss how these generic problem-solving methods can be adapted to a specific domain and task—namely, the Sisyphus room-assignment problem (Linster, 1994), an assignment problem in which employees are assigned to office places under several requirements. The *domain* is described by a set of employees, their names and roles (such as researcher), and their projects. Also it is indicated whether the employees smoke tobacco, whether they are hackers, and who are their immediate coworkers. In addition, there is a list of rooms that includes a description of how the rooms are situated with respect to the building and to the other rooms, and how many people the rooms can accommodate. The *task* is to find an assignment of employees and places that fulfils given requirements.

In Section 5.1, we discuss the modeling of the Sisyphus problem using the board-game method; in Section 5.2, we examine this mapping for the weak problem-solving method chronological backtracking. In Section 5.3, we identify different levels of task- and domain-specifity, and compare these results with another problem-solving method. In Section 5.4., we draw some general concluions on the differences between weak and strong problem-solving methods.

5.1 The Board-Game Method: Domain Mapping

Because the *board-game method* is designed for tasks that can be defined by pieces that are arranged at locations, the method can be applied easily to tasks where components are arranged in slots. Therefore, we can apply the board-game method to the Sisyphus room-assignment task without changing the inference or task layer of the method definition. We have to define the mapping from the Sisyphus domain to the inference layer of the board-game method by defining the mapping parts of the views and terminators. Figure 12 shows the mapping of the domain view *Correct-States* to the room-assignment domain. This domain view is used by the elementary inference action *Prune* (see Figure 9). The mapping rules use the terms of the inference layer and of the domain layer. Similar mappings must be defined for the other views of the method.

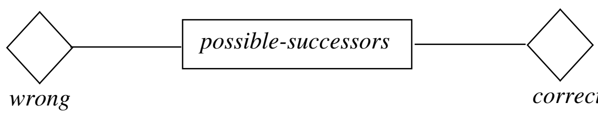
5.2 Chronological Backtracking: Domain and Task Mappings

The mapping problem is different for the chronological backtracking method than it is for the board-game method. Specifications for both the elementary inference actions and the definition parts of the stores are smaller for chronological backtracking. The formal specification of the elementary inference actions requires only one simple clause. The terminological knowledge also is trivial (see Section 4.1). This structure implies that, if we want to apply chronological backtracking to the room-assignment task, we must introduce new definitions and clauses *at the inference layer* (i.e., in *stores* and *elementary inference actions*). We must define the *task-specific terminological knowledge* of the internal structure of states: That is, we have to specify that states are sets of assignments involving slots (i.e., locations) and components (i.e., pieces). In addition, we must introduce new clauses, such as clauses that define how a successor state is derived from a prior state, and how the successor state inherits assignments from the old state.

The need for this modeling indicates that chronological backtracking not only requires a mapping on the given domain knowledge, but also requires adaptation to the given task if it is to be applicable to the Sisyphus problem. The interpretation that slots represent domain locations (e.g., rooms) is domain specific. The board-game and chronological-backtracking methods have this type of mappings in common. That states have this internal structure is, however, not merely a domain-dependent circumstance, rather, it is related to the *task* that should be solved by the problem-solving method. The adaptation of the chronological-backtracking method to the Sisyphus problem therefore includes two different activities:

1) We must specialize the generic problem-solving method chronological backtracking to a relatively *strong* problem-solving method (McDermott, 1988), such as the board-game method, that makes specific (i.e., strong) assumptions about a task. Such a *task-specific* method can solve only those tasks where a state is characterized by a set of assignments of pieces and locations.

VIEW *Correct-States*
DEFINITIONS



UPWARD MAPPING

/* Every possible successor state that is not a wrong state is a correct state. */

$\forall x_{St}\ (correct(x_{St}) \leftarrow x_{St}\ \varepsilon\ possible\text{-}successors \wedge \neg wrong(x_{St}))$.

/* Every state that assigns to the same room two employees who do not fit together is a wrong state. */

$\forall x_A, \forall x_C, \forall x_N, \forall x_S, \forall x_{St}, \forall y_A, \forall y_C, \forall y_S$

$wrong(x_{St}) \leftarrow$

$x_{St}[assign :: \{x_A[c : x_C, s : x_S]\}] \in possible\text{-}successors \wedge$

$x_{St}[assign :: \{y_A[c : y_C, s : y_S]\}] \in possible\text{-}successors \wedge$

$\neg(x_C \cong y_C) \wedge$

$x_S[room : x_N] \in place \wedge y_S[room : x_N] \in place \wedge$

$\neg\, fit\text{-}together\ (c_1 : x_C, c_2 : y_C)$.

/* Two employees may share a room (1) if the roles of both are the same, (2) if either both are smokers or both are not, (3) if neither of the two is a boss, and (4) if both work on different projects. */

$\forall x_C \forall x_{V1}, \forall x_{V2}, \forall x_{V3}, \forall y_C, \forall y_V$

$fit\text{-}together\ (c_1 : x_C, c_2 : y_C) \leftarrow$

$\neg(x_C \cong y_C) \wedge$

$x_C[project : x_{V1}, role : x_{V2}, smoker : x_{V3}] \in employees \wedge$

$y_C[project : y_V, role : x_{V2}, smoker : x_{V3}] \in employees \wedge$

$\neg(x_C\ \varepsilon\ boss) \wedge \neg(y_C\ \varepsilon\ boss) \wedge \neg(x_{V1} \cong y_V)$.

END;[1]

1. For a definition of the logical symbols see Figure 11.

Fig. 12. Definition of the view *Correct-States*.

2) We must map this task-specific instantiation of chronological backtracking to the domain of the room-assignment problem.

The board-game method specializes chronological backtracking for assignment tasks, where a state describes a set of assignments (i.e., a set of tuples of assignments of components and slots). Because the board-game method can model directly the Sisyphus room-assignment task, we need to define only the mapping parts of views and terminators to adapt this method to the domain of the Sisyphus problem.

5.3 Different Levels of Specifity

Eriksson et al. (1995) report that 280 lines of CLIPS code were required to apply chronological backtracking to the Sisyphus task. The board-game method required only 40 lines. Our experience with the use of KARL for modeling the same tasks is consistent with this result in that the relative benefit of the board-game method was the same. By using the conceptual model of KARL to specify these two methods, we make visible the *different types of knowledge*, rather than only their size) that are required for realizing this application:

- The board-game method requires a mapping from the domain-specific knowledge onto the generic terminology of the problem-solving method. For example, the developer must map *rooms* to *locations*, and *persons* to *pieces*.
- Chronological backtracking requires both this mapping and additional knowledge about the task that should be solved with this method. We must define what a state is (i.e., a set of assignments) and how we derive a successor state from it (i.e., by changing some of the assignments). For example, we must define that a state has a task-specific internal structure. Chronological backtracking requires therefore not only an excessive mapping, but also the specification of terminological and inference knowledge at the inference layer (i.e., in stores and elementary inference actions) if this method is to be applied to the Sisyphus problem.

It is the *dual* character of KARL that is, the formal character and the knowledge-level primitives, that gave us these insights into the task-specific refinement of problem-solving methods. Because the language primitives of KARL distinguish explicitly among different kinds of knowledge, epistemological differences became visible. In addition, it is the formal character of KARL that is, the precise semantics, that made it a necessary result. Using KARL forced us to model the two methodsprecisely. Neither a formal language, which would result in an amorphous set of clauses, nor an informal knowledge-level language would provide this strong guiding feature.

In a related study, Angele et al. (1992) provide a straightforward KARL specification for the Sisyphus problem. They developed the problem-solving method bottom up, using the Sisyphus problem statement as a guideline. The mapping between the inference and domain layers in the Angele method is therefore straightforward. The mapping specifies that slots correspond to places and components correspond to employees. In addition, the place requirements for employees, as well as the constraints for correct assignments, must be defined. Naturally, this naive problem-solving method is highly task specific, and hardwires application- and domain-specific assumptions. Examples of these assumptions follow:

- The method assumes that a goal state is an assignment where every component has a location and
- the preference for assignments can be defined independently for components and slots. That is, one preference is used to select the next most important component that is placed

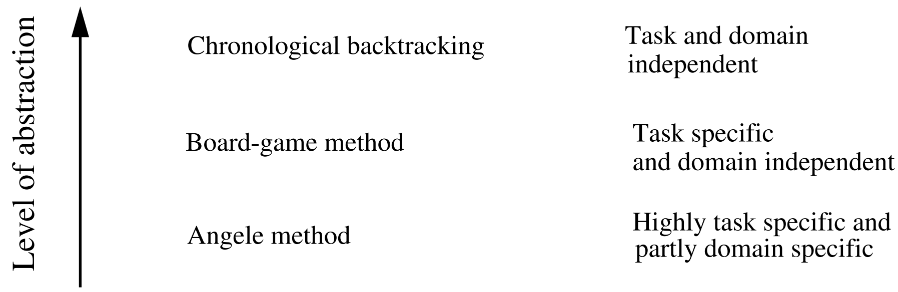


Fig. 13. Different levels of generality.

into a location of highest preference that is still available and that does not cause constraint violations.

It is clear that both assumptions are not fulfilled by every assignment problem. They reflect properties of the domain and of the application, as provided in Linster (1994).

If we compare the KARL definition of the board-game method with the naive Angele method, we see that the main difference lies in the inference action *Create* and its mapping by the view *Successor-Relation*. This view is defined in terms of *moves* for the board-game method, whereas the Angele method makes stronger assumptions about the relation. The relatively complex mapping of the board-game method must therefore define assumptions and actions of moves.[6] Parts of the knowledge, which must be defined in this mapping of the board-game method, are hard-wired at the inference layer in the Angele method. The corresponding mapping of the Angele method therefore requires only a preference on components and slots. The Angele method uses these preferences to define the derivation of the next "move" in the body of the inference action *Create*.[7]

Figure 13 illustrates the different levels of abstraction of the three methods. Chronological backtracking can be reused for arbitrary tasks and domains. However significant effort is required to apply it to any given task and domain. Such application requires not only the definition of complicated mappings, but also the introduction of new terminological and inference knowledge at the inference layer. The board-game method requires only the definition of domain mappings, because it hardwires stronger task-specific assumptions that fit the Sisyphus task.

The board-game method is not domain-specific and is less task-specific than is the Angele method. Thus, more work is required to define the mappings of the board-game method for the Sisyphus problem than for the Angele method. However, in addition to beeing used to solve Sisyphus problem, the board-game method can be reused for other tasks that can be modeled as one-player games, such as Towers of Hanoi (Eriksson et al., 1995). That is, it has a broader scope of reuse than does the Angele method.

5.4 Weak and Strong Problem-Solving Methods

The main distinction between chronological backtracking and the board-game method results from the richer ontological commitments that are introduced by the latter to characterize a state. As a consequence, the transition between states can be defined in more detail. These commitments restrict the number of tasks that can be tackled by the method, but they improve the *efficiency of the acquisition process for domain knowledge, and of the adaptation process of the method for a given application* (Eriksson et al., 1995). The method is already adapted to

---

6. Since this mapping is complex (approximately one page of KARL clauses), we do not introduce it in this paper.
7. The Angele method is specified in a generic, rather than in a domain-specific ,terminology. This terminology, however, incorporates domain- and task-specific assumptions in its definition and in the inferences that are drawn from it.

the task, and the domain knowledge can be provided in terms of moves of pieces between locations, instead of generic state transitions. In the case of the Angele method, knowledge about state transitions does not need to be provided at all, because the method hardwires these transitions. Only preferences on slots and components have to be provided.

Fensel and Straatman (1996) refined a version of *generate & test* to *propose & revise* to solve a parametric design task. The terminological structures of both methods were the same, in contrast to those in our analysis. The main distinction between generate & test and propose & revise was that the latter introduces heuristic assumptions over the provided domain knowledge and the task to improve the *efficiency of the problem-solving process*. Generate & test does a complete search that finds the optimal design. On the other hand, even for simple design problems, such a strategy is intractable. *Propose & revise* uses a local search technique that requires either strong assumptions on the provided search control knowledge or assumptions that weaken the goal of the tasks (assumptions on local optimums). Based on these assumptions, it can solve the problem with reasonable efficiency. From that perspective, chronological backtracking and the board-game method behave with equal efficiency as the latter introduces neither any kind of assumptions on domain knowledge that should improve the search process nor assumptions that weaken the task.

IN general, one can distinguish at least two dimensions of refining weak to strong problem-solving methods (cf. O'Hara & Shadbolt, 1996):

- Improving the efficiency of the knowledge acquisition and adaptation process by strengthening ontological commitments,
- improving the efficiency of the problem-solving process by introducing assumptions over the provided domain knowledge or precise characterisations of the task.

To develop methods for deriving task-specific problem-solving methods, we must integrate both aspects.

6 Conclusions

PROTÉGÉ-II, like many other approaches to reusable problem-solving methods, such as generic tasks (Chandrasekaran & Johnson, 1993) and role-limiting methods (Marcus, 1988), is close in spirit to the source-code libraries available in conventional software reuse (Krueger, 1992). Just as certain source-code libraries are designed to provide implementations of mathematical functions, problem-solving methods are designed to perform operations that are meaningful in terms of knowledge-based systems. Problem-solving methods, however, must perform tasks that are more complex than are those performed by these mathematical software libraries. An additional obstacle for the design of knowledge-based systems from reusable methods is that there is no underlying theory of models for problem-solving behavior in the same sense that mathematics is the underlying theory for numerical-calculation packages.

KARL enables us to formalize problem-solving methods, and it bridges the gap between conceptual model and implementations of problem-solving methods. A specification of the chronological-backtracking and the board-game methods using the formal specification language KARL improves the developer's understanding of these methods. We have made explicit the degree of task and domain dependency of these methods by respecifying the methods in KARL. Chronological backtracking can be applied to all tasks that can be solved by search through a space of states. The board-game method makes stronger assumptions about a state and about transitions between states. We do not accomplish the task-specific refinement of chronological backtracking into the board-game method by modifying the task layer or by refining the inference layer hierarchically; rather, we add clauses and terminological expressions to the definitions of knowledge roles and elementary inference actions. That is, we introduce new ontological commitments during task- and domain-specific refinements of problem-solving methods.

6.1 Comparison with Related Work

Source code libraries, such as those in SFB (Klinker et al., 1991), KREST (Steels 1990), and PROTÉGÉ-II (Puerta et al., 1992), combine implemented building blocks with semiformal descriptions. Problems arise when a precise understanding of the competence or the assumptions of such a building block are required. Code inspection is the only way to gain such an understanding. The other extreme is provided by the CommonKADS library of problem-solving methods of Breuker and Van de Velde (1994). Only semiformal descriptions at a high level of abstraction are provided.

Code descriptions and informal descriptions of a problem-solving method could be supplemented with formal high-level descriptions that abstract from implementation details. These descriptions provide two main advantages:

- The method is defined at the conceptual level in terms of the different types of knowledge required to specify problem solving. The transition of the informal description of a problem-solving method in terms of the KADS model of expertise to its formal definition is structure preserving (see also van Harmelen & Aben, 1996).
- The method is defined in a language that has a formal semantics, and abstracts from implementation issues. The problem-solving method is described precisely, and we can use the semantics to derive characteristic features of the problem-solving method.

Our formal specification of different method variants supplements the analysis of Schreiber et al. (1992). They compared the formal specification of the problem-solving methods *cover and differentiate* and *heuristic classification,* and found significant differences between these methods. Whereas Schreiber et al. (1992) compared two problem-solving methods developed for the same type of tasks, we have compared two problem-solving methods that differ in their task specificity. Akkermans et al. (1993) and Wielinga et al. (1995) sketched an approach that views the construction process of problem-solving methods for knowledge-based systems as an assumption-driven activity. One derives a formal specification of a task from informal requirements by introducing assumptions about the problem and the about problem space. One refines this task specification into a functional specification of the problem-solving method by making assumptions about the problem-solving paradigm and about the available domain theory. Still, their results remain abstract, and they do not relate their work to implemented libraries of problem-solving methods as provided by PROTÉGÉ-II or operational specifications of methods that define the dynamics of the reasoning process of a method. In addition, the discussion of the terminological refinement of methods is vague because no conceptual modeling primitives for static knowledge are used.

6.2 Future Work

An important task for future research is the analysis of problem-solving methods based on their formal and conceptual descriptions (see Fensel (1995a) for an analysis of the problem-solving method *propose and revise*). To reuse problem-solving methods, we must be aware of the assumptions underlying such methods. Each method defines requirements on the available domain knowledge and the given task. These requirements can be necessary for the effect of the method (i.e., its correctness), and for its efficiency. We can use them either to weaken the task definition or to strengthen the competence of a problem-solving method by increasing its demands on domain knowledge (see Benjamins et al., 1996, for details). Making these assumptions explicit has several advantages (see Fensel & Straatman, 1996):

1) They can be used to prove properties of a specification. The formal specification of a method, with the specification of the method´s assumptions, should enable us to prove formally the correctness and efficiency of a method. This formal proofs are important, because only reliable components can be reused.
2) These assumptions must be proved to hold in a given domain to which the method is to be applied. Otherwise, reuse of given methods is questionable, because it is not clear whether the methods will produce a correct result (in an efficient manner).
3) These assumptions can be used to index the reusable components, and thus to support the selection, combination, and adaptation of methods for a given domain and task.

Progress in these areas requires two different research activities. A conceptual framework for describing the functionality, efficiency, dynamic behavior, and assumptions of reusable building blocks is necessary. In addition, as it is for validation and verification of formal specifications, a proof calculus is required.[8] Recently, Fensel & Groenboom (1996) introduced the *Modal Logic for Predicate Modification (MLPM)* for this purpose. Its axiomatic semantics enable the automatization of such proofs. Fensel et al. (1996b) proposed a conceptual and formal framework for the specification of problem-solving methods. The *Karlsruhe Interactive Verifier (KIV)* (Reif, 1995) is used to verify such a specification and to detect hidden assumptions of a problem-solving method.

Further support for the development of problem-solving methods should be provided by application of ideas and tools of program transformation. Currently, we are examining the usefulness of tools such as KIDS (Smith, 1990), which supports the transformation process from a declarative specification to an operational (i.e., algorithmic) specification. This tool could be used to transform a desired competence theory of a problem-solving method into an operational specification.

Acknowledgments

We thank Richard Benjamins, John Gennari, Gertjan van Heijst, Angel Puerta, Remco Straatman, and two anonymous reviewers for helpful comments on an earlier drafts of the paper. Lyn Dupré provided editorial assistance.

This work was supported in part by grants LM05157 and LM05208 from the National Library of Medicine, and by gifts from Digital Equipment Corporation. Dr. Musen is recipient of National Science Foundation Young Investigator Award IRI-9257578.

References

J. M. Akkermans, B. Wielinga, and A. TH. Schreiber (1993): Steps in Constructing Problem-Solving Methods. In N. Aussenac et al. (eds.), *Knowledge-Acquisition for Knowledge-Based Systems*, Lecture Notes in AI (LNAI), no 723, Springer-Verlag, Berlin.

J. Angele, D. Fensel and D. Landes (1992). An Executable Model at the Knowledge Level for the Office-assignment Task. In M. Linster (ed.), Sisyphus ´92: Models of Problem Solving, report no 663, Gesellschaft für Mathematik und Datenverarbeitung (GMD), Bonn, Germany.

---

8. The substantial work that is required for formal proofs is justified if the building blocks will be reused several times.

J. Angele, D. Fensel, and R. Studer (1996). Domain and Task Modelling in MIKE. In *Proceedings of the IFIP TC8/WG8.2 Conference on Domain Knowledge in Interactive System Design*, Geneva, Switzerland, May 8-10, 1996.

R. Benjamins (1995): Problem-Solving Methods for Diagnosis and Their Role in Knowledge Acquisition, *International Journal of Expert Systems: Research and Applications*, 8(2):93—120.

R. Benjamins, D. Fensel, and R. Straatman (1996): Assumptions of Problem-Solving Methods and Their Role in Knowledge Engineering. In *Proceedings of the 12th European Conference on Artificial Intelligence (ECAI-96)*, Budapest, August 12-16.

J. Breuker and W. Van de Velde (eds.) (1994). *The CommonKADS Library for Expertise Modelling*, IOS Press, Amsterdam, The Netherlands, 1994.

A. G. Brooking (1986): The Analysis Phase in Development of Knowledge-Based Systems. In W. A. Gale (ed.), *AI and Statistic*, Addison-Wesley Publishing Company, Reading, Massachusetts.

A. Bundy (ed.) (1990). *Catalogue of Artificial Intelligence Techniques*, 3rd ed., Springer-Verlag, Berlin.

T. Bylander (1991). Complexity Results for Planning. In *Proceedings of the 12th International Joint Conference on Artificial Intelligence (IJCAI-91)*, Sydney, Australia, August.

T. Bylander, D. Allemang, M. C. Tanner, and J. R. Josephson (1991): The Computational Complexity of Abduction, *Artificial Intelligence*, 49:25—60.

K. Causse (1994): A Model for Control Knowledge. In *Proceedings of the 8th Banff Knowledge Acquisition for Knowledge-Based System Workshop (KAW´94)*, Banff, Canada, Januar 30 - February 4.

B. Chandrasekaran (1983).Towards a Taxonomy of Problem-solving Types, *AI Magazine*, 4(1):9—17.

B. Chandrasekaran (1986). Generic Tasks in Knowledge-based Reasoning: High-level Building Blocks for Expert System Design, *IEEE Expert*, 1(3):23—30.

B. Chandrasekaran, T. R. Johnson, and J. W. Smith (1992). Task-Structure Analysis for Knowledge Modeling, *Communications of the ACM*, 35(9):124—137.

B. Chandrasekaran and T. R. Johnson (1993). Generic Tasks and Task Structures: History, Critique and New Directions. In J.-M. David et al. (eds.), *Second Generation Expert Systems*, Springer Verlag, Berlin.

M. Dorfman (1990): System and Software Requierements Engineering. In R. H. Thayer and M. Dorfman (eds.), *System and Software Requierements Engineering*, IEEE Computer Society Press, Los Alamitos, California.

H. Eriksson, Y. Shahar, S. W. Tu, A. R. Puerta, and M. A. Musen (1995). Task Modeling with Reusable Problem-Solving Methods, *Artificial Intelligence*, 79(2):293—326.

D. Fensel (1993). The Knowledge Acquisition and Representation Language KARL, PhD thesis, University of Karlsruhe, Karlsruhe, Germany.

D. Fensel (1995a). Assumptions and Limitations of a Problem-Solving Method: A Case Study. In *Proceedings of the 9th Banff Knowledge Acquisition for Knowledge-Based System Workshop (KAW´95)*, Banff, Canada, February 26 - March 3.

D. Fensel (1995b). *The Knowledge Acquisition and Representation Language KARL*, Kluwer Academic, Boston, 1995.

D. Fensel (1995c). Formal Specification Languages in Knowledge and Software Engineering, *The Knowledge Engineering Review*, 10(4).

D. Fensel, J. Angele, and R. Studer (1996a). The Knowledge Acquisition and Representation Language KARL, to appear in *IEEE Transactions on Knowledge and Data Engineering*.

D. Fensel and R. Groenboom (1996). MLPM: Defing a Semantics and Axiomatization for Specifying the Reasoning Process of Knowledge-based Systems. In *Proceedings of the 12th European Conference on Artificial Intelligence (ECAI-96)*, Budapest, August 12-16.

D. Fensel and F. van Harmelen (1994). A Comparison of Languages which Operationalize and Formalize KADS Models of Expertise, *The Knowledge Engineering Review*, 9(2).

D. Fensel and S. Neubert (1994). Integration of Semiformal and Formal Methods for Specification of Knowledge-Based Systems. In B. Wolfinger (ed.), *Innovation bei Rechen- und Kommunikationssystemen*, Informatik Aktuell, Springer-Verlag, Berlin.

D. Fensel, A. Schönegge, R. Groenboom, and B. Wielinga (1996b). Specification and Verification of Knowledge-Based Systems. In *Proceedings of the ECAI'96 Workshop on Validation, Verification and Refinement of Knowledge-Based Systems at the 12th European Conference on Artificial Intelligence (ECAI-96)*, Budapest, August 12-16.

D. Fensel and R. Straatman (1996). The Essence of Problem-Solving Methods: Making Assumptions for Efficiency Reasons. In N. Shadbolt et al. (eds.), *Advances in Knowledge Acquisition*, Lecture Notes in Artificial Intelligence (LNAI), no 1076, Springer-Verlag, Berlin.

D. Harel (1984). Dynamic Logic. In D. Gabby et al. (eds.), *Handbook of Philosophical Logic, vol. II*, Extensions of Classical Logic, Publishing Company, Dordrecht (NL).

F. van Harmelen and M. Aben (1996). Structure-perserving Specification Languages for Knowledge-based Systems, *International Journal for Human-Computer Studies*, 44:187—212.

F. van Harmelen and J. Balder (1992): $(ML)^2$: A Formal Language for KADS Conceptual Models, *Knowledge Acquisition*, 4(1).

R. Hull and R. King (1987). Semantic Database Modeling: Survey, Applications, and Research Issues, *ACM Computing Surveys*, 19(3).

C. B. Jones (1990). *Systematic Software Development Using VDM*, 2nd ed., Prentice Hall.

R. Jungclaus (1993). Modeling of Dynamic Object Systems - A Logic-based Approach, *Vieweg Verlag*.

G. Klinker, C. Bhola, G. Dallemagne, D. Marques, and J. McDermott (1991). Usable and Reusable Programming Constructs, *Knowledge Acquisition*, 3(2):117—135.

M. Kifer, G. Lausen, and J. Wu (1995). Logical Foundations of Object-Oriented and Frame-Based Languages, *Journal of the ACM*, 42:741—843.

D. Kozen (1990). Logics of Programs. Van Leeuwen, J., (ed.), *Handbook of Theoretical Computer Science*, Elsevier Science, Amsterdam.

C. W. Krueger (1992). . Software Reuse, *ACM Computing Surveys*, 24(2):131—184.

D. Landes and R. Studer (1995). The Treatment of Non-Functional Requirements in MIKE. In Proceedings of the 5th European Software Engineering Conference ESEC'95, Barcelona, Spain, September 25-28, 1995, Lecture Notes in Computer Science (LNCS), no 989, Springer-Verlag.

I. van Langevelde, Philipsen, and J. Treur (1992). Formal Specification of Compositional Architectures. In *Proceedings of the 10th European Conference on Artificial Intelligence (ECAI-92)*, Vienna, Austria, August 3-7.

M. Linster (ed.) (1994). Sisyphus ´91/92: Models of Problem Solving, *International Journal of Human Computer Studies*, 40(3).

S. Marcus (ed.) (1988). *Automating Knowledge Acquisition for Experts Systems*, Kluwer Academic, Boston.

J. McDermott (1988). Preliminary Steps Toward a Taxonomy of Problem-solving Methods. In Marcus (1988), pp. 225—255.

M. Musen (1992). Overcoming the Limitations of Role-limiting Methods, *Knowledge Acquisition*, 4(2):165—170.

B. Nebel (1966). Artificial intelligence: A Computational Perspective. To appear in G. Brewka (ed.), *Essentials in Knowledge Representation*.

A. Newell (1982). The Knowledge Level, *Artificial Intelligence*, vol 18.

S. Neubert (1993). Model Construction in MIKE (Model Based and Incremental Knowledge Engineering). In N. Aussenac et al. (eds.), *Knowledge-Acquisition for Knowledge-Based Systems*, Lecture Notes in AI, LNAI, no 723, Springer-Verlag, Berlin.

A. Oberweis, G. Scherrer, and W. Stucky (1994). INCOME/STAR: Methodology and Tools for the Development of Distributed Information Systems, Information Systems, 19(8).

K. O'Hara and N. Shadbolt (1996): The Thin End of the Wedge: Efficiency and the Generalized Directive Model Methodology. In N. Shadbolt et al. (eds.), *Advances in Knowledge Acquisition*, Lecture Notes in Artificial Intelligence (LNAI), no 1076, Springer-Verlag, Berlin.

A. R. Puerta, J. W. Egar, S. W. Tu, and M. A. Musen (1992). A Multiple-method Knowledge-Acquisition Shell for the Automatic Generation of Knowledge-acquisition Tools, *Knowledge Acquisition*, 4(2):171—196.

F. Puppe (1993). *Systematic Introduction to Expert Systems: Knowledge Representation and Problem-Solving Methods*, Springer-Verlag, Berlin.

W. Reif (1995). The KIV Approach to Software Engineering. In M. Broy and S. Jähnichen (eds.), *Methods, Languages, and Tools for the Construction of Correct Software*, Lecture Notes in Computer Science (LNCS), no 1009, Springer-Verlag, Berlin.

A. Th. Schreiber (1992). Pragmatics of the Knowledge Level, PhD dissertation, University of Amsterdam, Amsterdam.

G. Schreiber, B. Wielinga, and H. Akkermans (1992). Differentiating Problem Solving Methods. In T. Wetter et al. (eds.), *Current Developments in Knowledge Acquisition*, Lecture Notes in Artificial Intelligence (LNAI) , no 599, Springer Verlag, Berlin, pp. 95—111.

G. Schreiber, B. Wielinga, H. Akkermans, W. Van de Velde, and A. Anjewierden (1994a). CML: The CommonKADS Conceptual Modelling Language. In L. Steels et al. (eds.), *A Future for Knowledge Acquisition*, Lecture Notes in Aritificial Intelligence (LNAI), no 867, Springer-Verlag, Berlin.

A. TH. Schreiber, B. Wielinga, J. M. Akkermans, W. Van De Velde, and R. de Hoog (1994b). CommonKADS. A Comprehensive Methodology for KBS Development, *IEEE Expert*, 9(6):28—37.

D. R. Smith (1990). KIDS: A Semiautomatic Program Development System, *IEEE Transactions on Software Engineering*, 16(9):1024—1043.

J. W. Spee and L. in 't Veld (1994). The Semantics of K_{BS}SF: A Language For KBS Design, Knowledge Acquisition, vol 6.

J. M. Spivey (1992): *The Z Notation. A Reference Manual*, 2nd ed., Prentice Hall, Englewood Cliffs.

L. Steels (1990). Components of Expertise, *AI Magazine*,11(2).

B.J. Wielinga, A.Th. Schreiber, and J.A. Breuker (1992). KADS: A Modelling Approach to Knowledge Engineering, *Knowledge Acquisition*, 4(1):5—53.

B. Wielinga, J. M. Akkermans, and A. Th. Schreiber (1995). A Formal Analysis of Parametric Design Problem Solving. In B. R. Gaines and M. A. Musen (eds.), *Proceedings of the 8th Banff Knowledge Acquisition for Knowledge-Based Systems Workshop (KAW-95)*, vol II, Alberta, Canada.

E. Yourdon (1989). *Modern Structured Analysis*, Prentice Hall, Englewood Cliffs.