A Model of the Acquisition and Improvement of Domain Knowledge for Functional Programming

CLAUS MÖBUS, OLAF SCHRÖDER, AND HEINZ-JÜRGEN THOLE
Department of Computational Science, P.O. Box 2503
University of Oldenburg, D-2900 Oldenburg, Germany

Reprinted from
Journal of Artificial Intelligence in Education
(1992) 3(4), 449-476

Jl. of Artificial Intelligence in Education (1992) **3**(4), 449-476

A Model of the Acquisition and Improvement of Domain Knowledge for Functional Programming

CLAUS MÖBUS, OLAF SCHRÖDER, AND HEINZ-JÜRGEN THOLE
Department of Computational Science, P.O. Box 2503
University of Oldenburg, D-2900 Oldenburg, Germany

This paper describes a model of students' knowledge growth from novice to expert within a theoretical framework of impasse-driven learning, success-driven learning, and problem solving. The model represents the actual state of domain knowledge of a learner. It is designed to be part of a help system, ABSYNT, which provides user-centered help in the domain of functional programming. The model is continuously updated based on the learner's programming actions. There is a distinction within the model between newly acquired and improved knowledge. *Newly acquired knowledge* is represented by augmenting the model with rules from the expert knowledge base. *Knowledge improvement* is represented by rule composition. In this way, the knowledge contained in the model is partially ordered from general rules to more specific schemas for solution fragments to specific cases (= example solutions for specific programming tasks). The model is implemented but not yet actually used for help generation within the help system. This paper describes the theoretical framework, the ABSYNT help system, the model, a preliminary study addressing some of its empirical predictions, and the significance of the model for the help system.

Introduction

The problem of student modelling is an important research topic especially within the context of help and tutoring systems (Anderson, Boyle, Farrell, & Reiser, 1987; Brown & Burton, 1982; Frasson & Gauthier, 1990; Kearsley, 1988; Sleeman, 1984; Sleeman & Brown, 1982; Wenger, 1987). Advance in designing such systems seems to be possible only if the actual knowledge state of the learner can be diagnosed *online* in an efficient and valid way. This is difficult (Self, 1990,1991) but necessary for a system in order to react adequately to the student's activities. Furthermore, it has been well recognized that progress in student modelling depends much on understanding what the student is doing and why. Thus,detailed assumptions about probem solving, knowledge representation, and acquisition processes are needed.

We face the student modelling problem within the context of a help system in the domain of functional programming: the ABSYNT Problem Solving Monitor. ABSYNT ("Abstract Syntax Trees") is a functional visual programming language designed to support beginners acquiring basic functional programming concepts. The ABSYNT Problem Solving Monitor provides help for the student constructing ABSYNT programs to given tasks. *Adaptive* help requires a student model. Our approach to model the student's knowledge rests on three principles:

- To try to understand what the student is doing and why. This amounts to constructing a *theoretical framework* which is powerful enough to describe the continuous stream of hypothetical problem solving, knowledge acquisition, and utilization events, and to describe and explain the stream of observable actions and verbalizations of the student.

- To use a subset of this theoretical framework in order to construct a student model containing the actual hypothetical state of domain knowledge of the student. This *state model* must be (and can be) simpler than the theoretical framework because its job is *efficient online diagnosis of domain knowledge* based on the computer-assessable data provided by the student's interactions with the system.

- To fill the gap between the theoretical framework and the state model by constructing an offline model of knowledge acquisition, knowledge modification, and problem solving processes. This *process model* provides hypothetical *reasons* for the changing knowledge states as represented in the state model.

In accordance with these principles, we pursue a three-level approach:

- A theoretical framework of problem solving and learning serves as a base for interpreting and understanding the student's actions and verbalizations. We call this framework *ISP-DL Theory* (Impasse - Success - Problem - Solving - Driven Learning Theory).

- An *internal model* (*IM*) diagnoses the actual domain knowledge of the learner at different states in the knowledge acquisition process (*state model*). It is designed to be an integrated part of the help system ("internal" to it) in order to provide user-centered feedback.

- An *external model* (*EM*) is designed to simulate the knowledge acquisition *processes* of learners on a level of detail not available to the IM (e.g., including verbalizations). Thus, the EM is not part of the help system ("external" to it) but supports the design of the IM.

Thus ISP-DL Theory, IM, and EM are designed to be mutually consistent but serve different purposes. This paper is concerned with the IM. It is organized as follows: First, we will describe the ISP-DL Theory and our help system, the ABSYNT problem-solving monitor. Then, the IM is described and illustrated. Empirical predictions and a first evaluation are presented. Finally, we will show how the IM enables adaptive help.

The ISP-DL Knowledge Acquisition Theory

The ISP-DL Theory is intended to describe the continuous flow of problem solving and learning of the student as it occurs in a sequence of, for example, programming sessions. In our view, existing approaches touch upon main aspects of this process but do not cover all of them. Consequently, the ISP-DL Theory is an attempt to integrate several approaches. Before describing it, we will briefly discuss three theoretical approaches relevant here:

(1) In VanLehn's (1988, 1990, 1991b) theory of Impasse Driven Learning, the concept of an impasse is of central importance to the acquisition of new knowledge. Roughly, an impasse is a situation where "the architecture cannot decide what to do next given the knowledge and the situation that are its current focus of attention" (VanLehn, 1991b, p. 19). Impasses trigger problem-solving processes which may lead to new information.

Impasses are also situations where the learner is likely to actively look for and to accept *help* (VanLehn, 1988). But problem solving or trying to understand remedial information might as well lead to secondary impasses (Brown & VanLehn, 1980). Impasse Driven Learning Theory is concerned about conditions for problem solving, using help, and thereby acquiring new knowledge. It is not concerned about optimizing knowledge already acquired. "Knowledge compilation . . . is not the kind of learning that the theory describes" (VanLehn, 1988, p. 32). Thus, with respect to our purposes, the theory seems incomplete.

(2) In SOAR (Laird, Rosenbloom, & Newell, 1986, 1987; Rosenbloom, Laird, Newell, & McCarl 1991), the concept of impasse-driven learning is elaborated by different types of impasses and weak heuristics performed in response to them. Impasses trigger the creation of subgoals and heuristic search in corresponding problem spaces. If a solution is found, a chunk is created acting as a new operator in the original problem space. As in Impasse Driven Learning Theory, all learning is triggered by impasses. But in our view it seems questionable whether all knowledge acquisition events can reasonably be described as resulting from impasses (VanLehn, 1991b). In SOAR there is no "success-driven" change of knowledge staying within *one* problem space (i.e., as the result of the successful application of *existing* knowledge).

(3) ACT* (Anderson, 1983, 1986, 1989) focuses on the success-driven optimization of already existing knowledge by knowledge compilation but pays less attention to the problem where new knowledge comes from.

We think that for our purposes it is necessary to cover problem solving, impasse-driven learning, and success-driven learning as well (see also Schröder, 1990). Thus, ISP-DL Theory incorporates

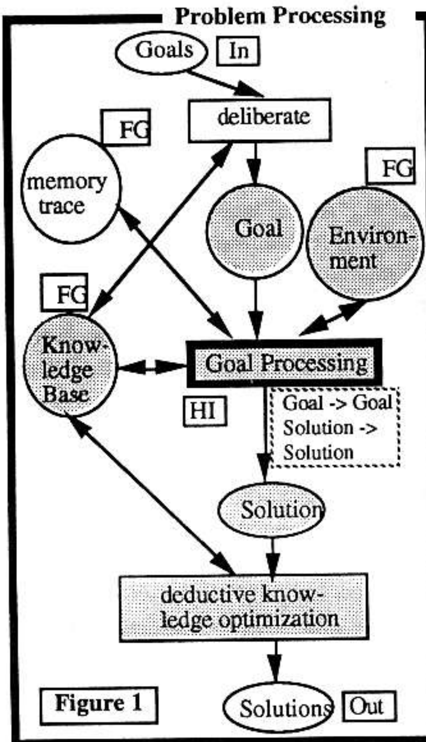
- the distinction of different problem-solving phases (according to Gollwitzer, 1990): *Deliberating* with the result of choosing a goal, *planning* a solution to it, *executing* the plan, and *evaluating* the result.

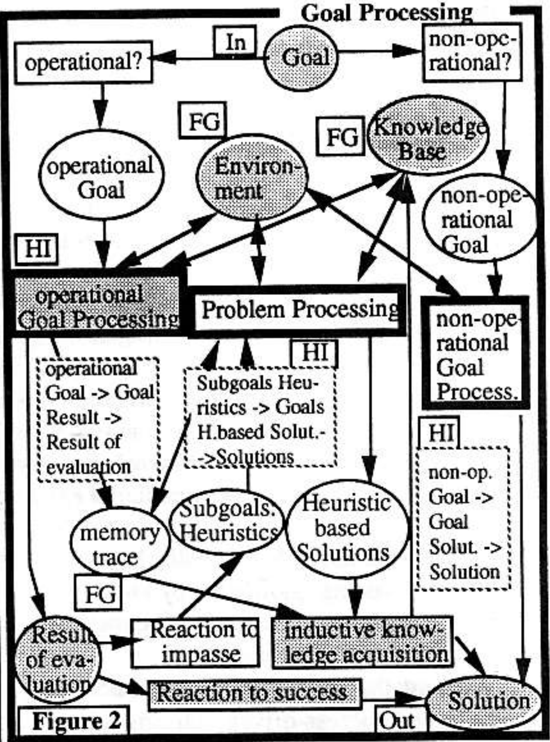
- the *impasse-driven acquisition of new knowledge*. In response to impasses, the problem solver applies weak heuristics, like asking questions, looking for help, etc.(Laird, Rosenbloom, & Newell, 1987; VanLehn, 1988, 1990, 1991b). Thus, new knowledge may be acquired.

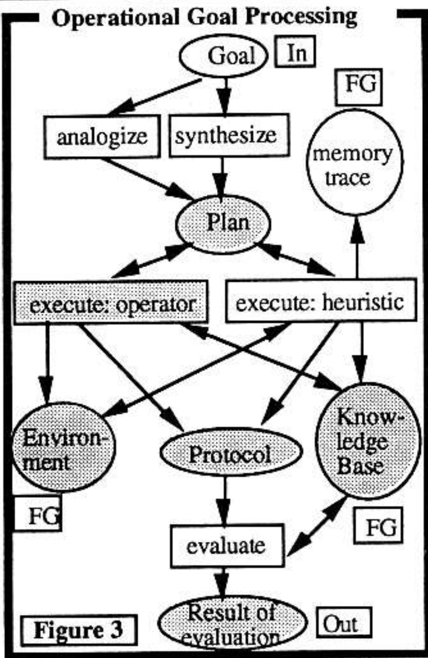
- the *success-driven improvement of acquired knowledge*. Successfully used knowledge is improved so it can be used more effectively. More specifically, by *rule composition* (Anderson, 1983, 1986; Lewis, 1987; Neves & Anderson, 1981; Vere, 1977), the number of control decisions and subgoals to be set is reduced. In our approach, composition is based on the resolution and unfolding method (Hogger, 1990).

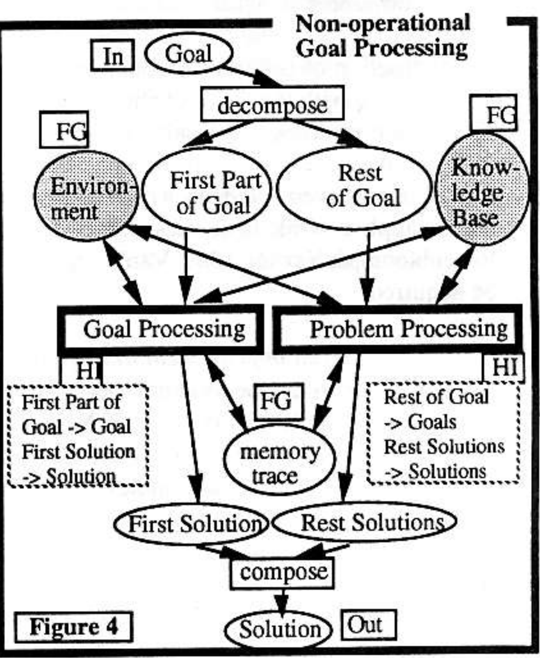
We describe the ISP-DL Theory by *hierarchical higher Petri nets* (Huber, Jensen, & Shapiro, 1990), though alternative modelling formalisms are possible, for example, *stream* communication (Gregory, 1987). Petri nets show temporal constraints on the order of processing steps more clearly than a purely verbal presentation. Thus they emphasize empirical predictions. The whole process is divided into 4 recursive subprocesses (*pages*): "Problem Processing," "Goal Processing," "Nonoperational Goal processing," and

"Operational Goal Processing" (Figures 1-4). *Places* (circles/ellipses) represent states (e.g., the content of data memories); *transitions* (rectangles) represent events or process steps.









Figures 1-4. The ISP-DL Theory of problem solving and learning

Places may contain tokens which represent mental objects (goals, memory traces, heuristics, etc.) or real objects (e.g., a solution or a behaviour protocol). Places can be marked with tags (*In* for entering, *Out* for exiting place, *FG* for global fusion set). An FG tagged place is common to several nets (e.g., the Knowledge Base). Transitions can be tagged with HI (for hierarchical invocation transition). This means that the process is continued in the called subnet. The dotted boxes show which places are corresponding in the calling net and in the called net. Shaded transitions and places are taken into account by the IM (see below).

Problem Solving is started in the page "*Problem Processing*" (Figure 1). The problem solver (PS) strives for one goal to choose out of the set of goals: "*deliberate.*" A goal may be viewed as a set of facts about the environment which the problem solver wants to become true (Newell, 1982). A goal can be expressed as a *predicative description* which is to be achieved by a problem solution. For example, the goal to create a program which tests if a natural number is even, "even(n)", can be expressed by the description:

"funct even = (nat n) bool: exists ((nat k) 2 * k = n)".

The goal is processed in the page "*Goal Processing*" (Figure 2). If the PS comes up with a solution, the used knowledge is optimized: *deductive knowledge optimization*. When the PS encounters a similar problem, the solution time will be shorter. The net is left when there are no tokens in "*Goals,*" "*Goal,*" and "*Solutions.*"

In the page "Goal Processsing" (Figure 2), the PS checks whether his set of problem solving operators is sufficient for a solution: "*operational?*"/"*non-operational?*"

An operational goal is processed according to the page "*Operational Goal Processing*" (Figure 3). A plan is *synthesized* by applying problem solving operators, or it is created by *analogical* reasoning. The plan is a partially ordered sequence or hierarchy of domain-specific problem-solving operators (or of domain-unspecific heuristics; see below). In either case, the plan is *executed*. Execution might necessitate further plan refinement, so arrows lead also back from "execute" to "Plan." Execution leads to a problem solving *protocol* which is used in combination with the knowledge base to *evaluate* the outcome. The *result of the evaluation* generates an impasse or a success and is transferred back to the page "*Goal Processing.*"

The *reaction* of the PS to *success* is: leave "*Goal Processing*" with a *solution*. The reaction to an impasse is the creation of subgoals to use weak heuristics for problem solving. Now there is a recursive call to "Problem Processing." "Goal Processing" and "Operational Goal Processing" are called again. This time, within Operational Goal Processing a plan to use heuristics is synthesized and executed. (Simple examples for these weak heuristics are to use a dictionary, to find an expert to consult, and so on.) A memory trace of the situation which led to the impasse is kept. If the use of heuristics is successful, then the result is *twofold*:

- The heuristically based solution to the impasse is related to the memory trace of the impasse situation. Thus, within "*Goal Processing,*" new *domain-specific* problem-solving operators are inductively *acquired*.

- Within "*Problem Processing,*" the *domain-unspecific* heuristic knowledge used is deductively *optimized*. So next time the PS encounters an impasse, he or she will be more skilled and efficient in using a dictionary, finding someone to consult, and so forth.

Finally, a non-operational goal is processed according to the page "*Non-operational Goal Processing*" (Figure 4). The problem is decomposed and the subsolutions are composed to a final solution.

It is possible and necessary to refine the theory's transitions and places, but for our purpose this theory is sufficient. Important are the following theoretically and empirically validated statements:

- New knowledge is acquired only at impasses after successful application of weak heuristics.
- Information is helpful only at impasses and if synchronized with the knowledge state of the PS.

What *design principles* does the ISP-DL Theory imply for the ABSYNT help system and for the Internal Model (IM) which is intended to represent the PS's actual domain knowledge?

Concerning the help system

(1) It should not interrupt the learner but *offer* information upon request.

(2) There should be attractive and *easily useable means of evaluation* for all problem solving phases.

(3) *Different problem-solving phases*—synthesizing (planning), executing (implementing), and evaluating—should be supported.

(4) Information should be *user-centered*, that is, closely tailored to the knowledge state of the learner. The learner should be able to use pre-knowledge as much as possible.

Concerning the Internal Model

(1) It should distinguish between *newly acquired* knowledge and *improved* knowledge, where knowledge can only be improved after successful application.

(2) Its content should reflect *performance data* such as speedups from earlier to later tasks, asking for or actively looking for help, and corrections or redesign of solution proposals.

(3) It should represent both *planning* knowledge and *implementation / coding* knowledge.

The ABSYNT Problem Solving Monitor

ABSYNT is a visual programming language based on ideas stated in an introductory computer science textbook (Bauer & Goos, 1982). It is a tree representation of pure LISP without the list data structure and is aimed at supporting the acquisition of basic functional programming skills, including abstraction and recursive systems. The motivation and analysis of ABSYNT with respect to properties of visual languages is described in Möbus and Thole (1989). The ABSYNT Problem Solving Monitor provides an *iconic program-*

ming environment (Chang, 1990). Its main components are a visual editor, trace, and a *help component*: a *hypotheses testing environment*.

In the editor (Figure 5), ABSYNT programs can be constructed. There is a head window and a body window. The left part of Figure 5 shows the tool bar of the editor: The bucket is for deleting nodes and links. The hand is for moving, the pen for naming, and the line for connecting nodes. Next, there is a constant, parameter, and "higher" self-defined operator node (to be named by the learner, using the pen tool). Constant and parameter nodes are the *leaves* of ABSYNT trees. Then several primitive operator nodes follow ("if", "+", "-", "*", . . .). Editing is done by selecting nodes with the mouse and placing them in the windows and by linking, moving, naming, or deleting them. Nodes and links can be created *independently*: If a link is created before the to-be-linked nodes are edited, then shadows are automatically created at the link ends. They serve as place holders for nodes to be edited later. Shadows may also be created by clicking into a free region of a window. In Figure 5, a program is actually under development by a student. There are subtrees not yet linked and nodes not yet named or completely unspecified (shaded areas). The upper part of Figure 5 shows the Start window for calling programs. This is also where the visual trace starts if selected by the student. In the visual trace, each computational step is made visible by representing computation goals and results within the nodes (Möbus & Schröder, 1990).

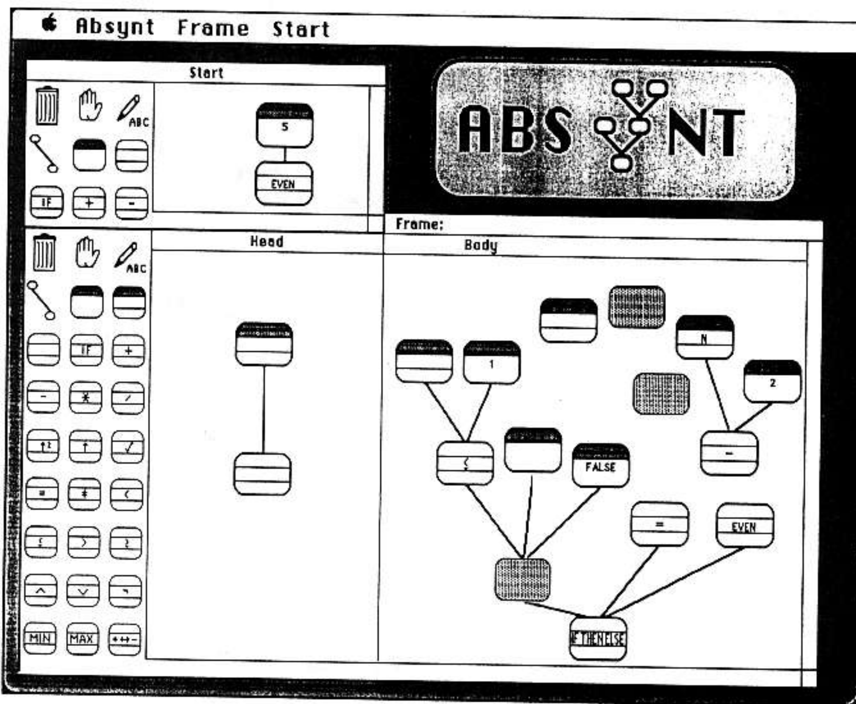


Figure 5. A snapshot of the visual editor of ABSYNT

In the *hypotheses testing environment* (Figure 6), the PS may state hypotheses (bold parts of the program in the upper worksheet in Figure 6) about the correctness of programs or parts thereof for given programming tasks. The hypothesis is: "It is possible to embed the boldly marked fragment of the program in a correct solution to the current task!" The PS then selects the current task from a menu, and the system analyzes the hypothesis. If the

hypothesis can be confirmed, the PS is shown a copy of the hypothesis. If this information is not sufficient to resolve the impasse, the PS may ask for more information (completion proposals). If the hypothesis cannot be confirmed, the PS receives the message that the hypothesis cannot be completed to a solution known by the system.

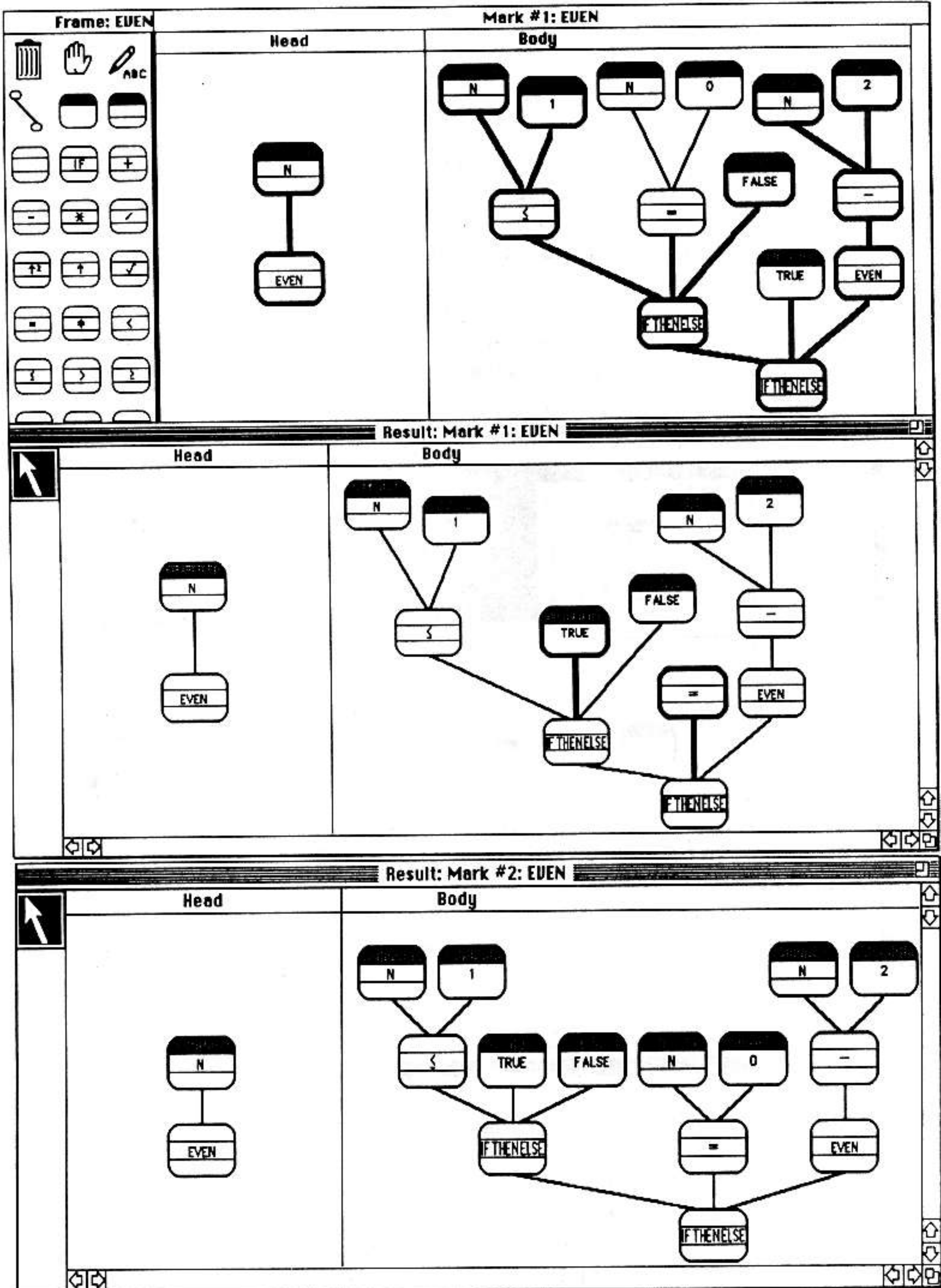


Figure 6. Snapshot of the ABSYNT Hypotheses Testing Environment

The upper part of Figure 6 shows a solution proposal to the "even" problem just constructed by a student: "Construct a program that determines whether a number is even!" This solution does not terminate for odd arguments. Despite that, the *hypothesis* (bold program fragment in the upper part of Figure 6) is embeddable in a correct solution. So the hypothesis is returned as feedback to the student (thin program fragment in the middle part of Figure 6). The student then may ask for a completion proposal generated by the system. In the example the system completes the hypothesis successively with the constant "true" and with the "="-operator (bold program fragments in the middle part of Figure 6). Internally, the system generates a complete solution visible in the lower part of Figure 6. So the student's solution in the upper part of Figure 6 may be corrected by an interchange of program parts.

One reason for the hypotheses testing approach is that in programming, a bug usually *cannot be absolutely localized*, and there is a variety of ways to debug a wrong solution. Hypotheses testing leaves the decision of which parts of a buggy solution proposal to keep to the PS and thereby provides a rich data source about the PS's knowledge state. Single subject sessions with the ABSYNT Problem Solving Monitor revealed that hypotheses testing was heavily used. It was almost the only means of debugging wrong solution proposals, despite the fact that the subjects had also the visual trace available. This is partly due to the fact that in contrast to the trace, hypotheses testing does not require a complete ABSYNT program solution.

The answers to the learner's hypotheses are generated by rules defining a *goals-means-relation (GMR)*. These rules may be viewed as "pure" expert domain knowledge not influenced by learning. Thus we will call this set of rules EXPERT in the remainder of the paper. Currently, EXPERT contains about 650 rules and analyzes and synthesizes several millions of solutions for 40 tasks (Möbus, 1990, 1991; Möbus & Thole, 1990). One of them is the "even" task just introduced; more tasks will be presented later. We think that such a large solution space is necessary because we observed that especially novices often construct unusual solutions due to local repairs. (This is exemplified by the clumsy-looking student proposal in the upper part of Figure 6.)

With respect to the *design principles* mentioned at the end of the last section, the ABSYNT Problem Solving Monitor does not interrupt but *offers* help and has attractive means of *evaluation* (hypotheses testing, visual trace). Incorporation of a *planning* level is in progress (see also the discussion section). Concerning *user-centered help*, the completions shown in the middle part of Figure 6 (bold program fragments) and the complete solution in the lower part of Figure 6 were generated by EXPERT rules. EXPERT analyzes and synthesizes solution proposals but is not *adaptive* to the learner's knowledge. Usually EXPERT is able to generate a large set of *possible* completions. Thus the main function of the *IM* (internal student model) is to *select* a completion from this set which is maximally *consistent* with the learner's current knowledge state, and thus to provide user-centered help.

The IM contains simple GMR rules and composites of them. It is continuously updated according to theoretical and empirical constraints. Therefore, GMR rules, rule composition, and empirical constraints will be described before presenting the IM.

GMR Rules

This section describes the goals-means-relation GMR. The set of GMR rules may be split in two ways: *rule type* (simple, composed) vs. *database* of the rules (EXPERT, POSS, IM).

- There are three kinds of *simple rules*: *goal elaboration rules*, *rules implementing one ABSYNT node* (operator, parameter, or constant), and *rules implementing program heads*.

- *Composite rules* are created by merging at least two successive rules parsing a solution. Composites may be produced from simple rules and composites. A composite is called a *schema* if it contains at least one pair of variables which can be bound to a goal tree and a corresponding ABSYNT program subtree, respectively. If a composite is fully instantiated (i.e., its variables can only be bound to node names or node values), then it is called a *case*.

Concerning the *data base* of the GMR rules, EXPERT contains the expert domain knowledge (only simple rules). The sets IM and POSS will be described below.

Figure 7 shows examples for simple rules depicted in their visual representations. Each rule has a *rule head* (left hand side, pointed to by the arrow) and a *rule body* (right hand side, where the arrow is pointing from). The rule head contains a *goals-means-pair* where the goal is contained in the ellipse and the means (implementation of the goal) is contained in the rectangle. The rule body contains one goals-means-pair or a conjunction of pairs, or a primitive predicate (is_parm, is_const).

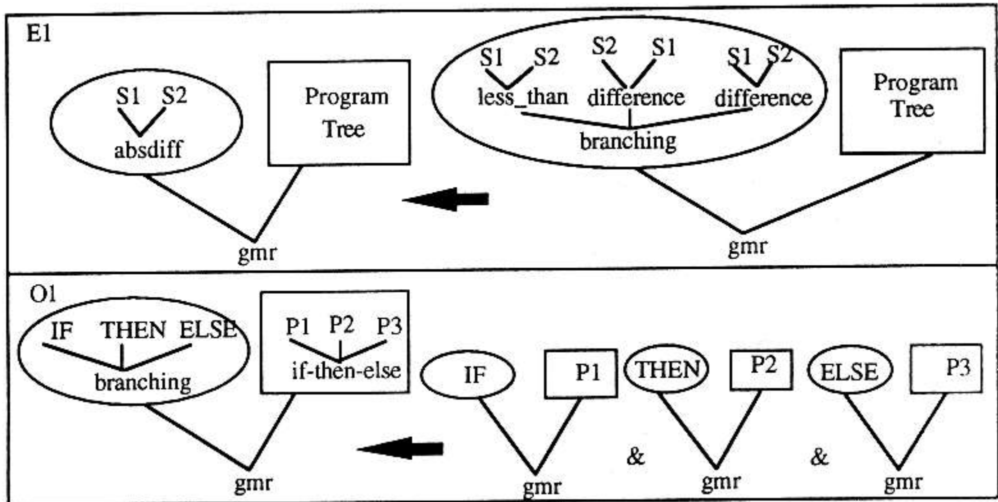


Figure 7. A goal elaboration rule (E1) and a rule (O1) implementing the ABSYNT node "if-then-else"

The first rule of Figure 7, E1, is a goal elaboration rule. It can be read:

If	(*rule head*):
	your main goal is "absdiff" with two subgoals S1 and S2,
then	leave space for a program tree yet to be implemented, and (*rule body*):
If	in the next planning step you create the new goal "branching" with the three subgoals
	"less_than (S1, S2)," "difference (S2, S1)," and "difference (S1, S2),"
then	the program tree solving this new goal will also be the solution for the main goal"

O1 in Figure 7 is a simple rule implementing the ABSYNT "if-then-else" operator node:

If	(*rule head*):
	your main goal is "branching" with three subgoals (IF, THEN, ELSE),
then	*implement* an "if-then-else"-node (or "if-"-node) with three links leaving from its input, and leave space above these links for three program trees P1, P2, P3 yet to be implemented; and (*rule body*):
if	in the next planning step you pursue the goal IF,
then	its solution P1 will also be at P1 in the solution of the main goal, and
if	in the next planning step you pursue the goal THEN,
then	its solution P2 will also be at P2 in the solution of the main goal, and
if	in the next planning step you pursue the goal ELSE,
then	its solution P3 will also be at P3 in the solution of the main goal.

Composition of Rules

In our theory, composites represent improved sped-up knowledge. Simple rules and composites constitute a partial order from simple rules ("micro rules") to solution schemata to specific cases representing solution examples for tasks. In this section we will define rule composition.

If we view the rules as Horn clauses (Kowalski, 1979), then the composite RIJ of two rules RI and RJ can be described by the inference rule:

$$\frac{\text{RI: } (F \leftarrow P \;\&\; C) \qquad \text{RJ: } (P' \leftarrow A)}{\text{RIJ: } (F \leftarrow A \;\&\; C)\sigma}$$

The two clauses above the line resolve to the resolvent below the line. A, C are conjunctions of atomic formulas. P, P', and F are atomic formulas. σ is the most general unifier of P and P'. RIJ is the result of unfolding RI and RJ—a sound operation (Hogger, 1990).

For example, we can compose the *schema* C7 (Figure 8) out of the set of simple rules {O1, O5, L1, L2}, where:

```
O1: gmr(branching(IF,THEN,ELSE),if-pop(P1,P2,P3)):-
        gmr(IF,P1),gmr(THEN,P2),gmr(ELSE,P3).
O5: gmr(equal(S1,S2), eq-pop(P1,P2)):- gmr(S1,P1),gmr(S2,P2).
L1: gmr(parm(P), P-pl):- is_parm(P).
L2: gmr(const(C), C-cl):- is_const(C).
C7: gmr(branching(equal(parm(Y),const(C)),parm(X),ELSE),
                                    if-pop(eq-pop(Y-pl,C-cl),X-pl,P)):-
        is_parm(Y),is_const(C),is_parm(X),gmr(ELSE,P).
```

where:

if-pop	=	primitive ABSYNT operator "if-then-else" (or "if")
eq-pop	=	primitive ABSYNT operator "="
P-pl, X-pl, Y-pl	=	unnamed ABSYNT parameter leaves
C-cl	=	empty ABSYNT constant leaf

We also can describe the composition of node implementing rules RI and RJ with a shorthand notation:

$$RIJ \quad = \quad RI_k \quad \bullet \quad RJ$$

The index k denotes the place k in the goal tree of the head of RI. A place k is the k-th variable leaf numbered from left to right (e.g., $O1_3$ = ELSE). The semantics of "•" can be described in three steps. First, the variable in place k in the goal term in the head of RI is substituted by the goal term in the head of RJ. Second, the call term P in the body of RI which contains the to-be-substituted variable unifies with the head of RJ and is replaced by the body of RJ. Third, the unifier σ is applied to the term resulting from the second step, leading to the composed rule RIJ. Thus, the variables affected by the unification in step two are replaced by their bindings.

For example, $O1_2 \bullet L1$ = gmr(branching(IF, parm(P), ELSE), if-pop(P1,P-pl,P3)):- gmr(IF,P1), is_parm(P), gmr(ELSE, P3). C7 can be composed from the rule set {O1, O5, L1, L2} in 16 different ways. Two possibilities are:

$$C7 = (O1_2 \bullet L1)_1 \bullet ((O5_2 \bullet L2)_1 \bullet L1)$$
$$C7 = (((O1_1 \bullet O5)_3 \bullet L1)_2 \bullet L2)_1 \bullet L1$$

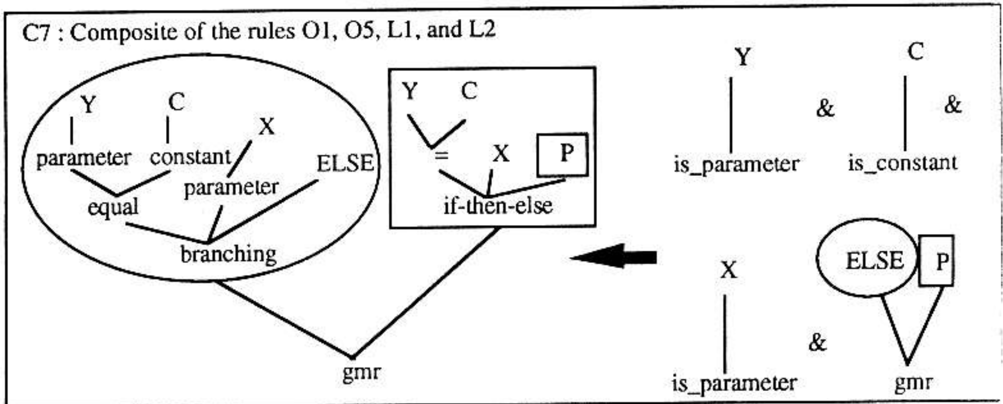


Figure 8. The composite C7

Empirical Constraints of Simple Rules, Chains, Schemata, and Cases

Rules, rule chains, and schemata give rise to different *empirical predictions*. The purpose of this section is to introduce hypotheses about the application of novice and expert knowledge, viewed as simple GMR rules and composites. Some of these hypotheses will be used in the Internal Model.

Any approach designed to represent changing knowledge states must mirror the shift from novice to expert. In general, novices work *sequentially*, set more subgoals, and need more control decisions, while experts work in *parallel*, set fewer subgoals, and need fewer control decisions (Chase & Simon, 1973; Elio & Scharf, 1990; Gugerty & Olson, 1986; Simon & Simon, 1978). Here, this difference is reflected in the partial order from simple rules to schemata to specific cases.

In order to demonstrate this difference, it is necessary to specify hypotheses about the problem-solving behavior. According to the ISP-DL Theory, a plan is synthesized from a goal, and execution of operators leads to a protocol of actions and verbalizations (Figure 3). Thus, with respect to the theory, we make a distinction between the problem-solving phases of *planning* and *execution*: A *plan synthesizer* or *"planner" synthesizes* plans, and an *operator executor* or *"coder" executes* operators to implement the plans. The coder has domain-specific knowledge (GMR rules) for implementing ABSYNT trees but no planning knowledge. The coder also has very limited execution knowledge: pattern matching without unification (except for parameter and higher operator names and constant values). More complex processes are left to the planner whose job is to guide the coder, based on domain-specific planning knowledge and on weak heuristics (to be specified by the External Model, as stated earlier).

For illustration of a hypothetical interaction sequence between planner and coder, we assume that the goal "branching (equal (parm(y), const(0)), parm(x), ELSE)" is to be implemented and that the coder has knowledge about the set of simple GMR rules {O1, O5, L1, L2}. Figure 9 shows how the interaction might proceed: At time t_0, the planner delivers the goal. The coder has no rule for it, so he rejects the goal. So the planner chops the goal into subgoals. Next, he may present the subgoal "parm(y)" to the coder. The coder now has a rule, L1, instantiates it to L1', and edits an ABSYNT parameter node with the name "y". Next, the planner delivers the subgoal "parm(x)". The planner uses L1 again, leading to the instantiation L1", and programs a parameter x. Then the planner comes up with "const(0)". The coder uses L2, applying L2' and programming a constant node 0. Next, the subgoal "equal(S1,S2)" is given. The planner instantiates O5 to O5' and creates a "=" node with two open links: their upper ends are shadows (place holders for nodes). After time t_j, the planner tells the coder that "equal(S1, S2)" has "parm(y)" as its first subgoal. So the coder connects the first input link of the "=" node to the parameter y. Next, the planner tells the coder that "equal(S1,S2)" has "const(0)" as its second subgoal, so the coder connects the second input link of the "=" node to the constant 0. Thus, the coder has to rearrange the position of the nodes and/or the orientation of the links. This is symbolized by the hand in Figure 9. Next, the planner comes up with the "branching(IF,THEN,ELSE)" subgoal. The coder implements it, instantiating O1 to O1'. After time t_m, the planner tells the coder that "branching (IF,THEN,ELSE)" has "parm(x)" as its second subgoal and "equal(S1, S2)" as its first subgoal. So the coder connects the second and first input link of the "if-then-else" node to the parameter x and to the "=" node, respectively. Again, the position of links and/or nodes on the screen may have to be rearranged. Now the goal is solved.

Thus, the planner does not know about the coder's knowledge, and vice versa. There is no fixed order of application of GMR rules. The order solely depends on how the goals are delivered to the coder by the planner. In the example, the coder created the sequence of rule instantiations (L1', L1", L2', O5', O1') depending on the goals delivered by the planner.

In contrast to this sequence, if the same goal "branching (equal (parm(y), const(0)), parm(x), ELSE)" is given and the coder knows the schema C7, then the interaction shown in Figure 10 will be produced. Again, at time t_0 the planner delivers the goal. This time the coder instantiates C7 to C7' and implements the ABSYNT tree contained in C7' without requiring subgoals and linking instructions from the planner.

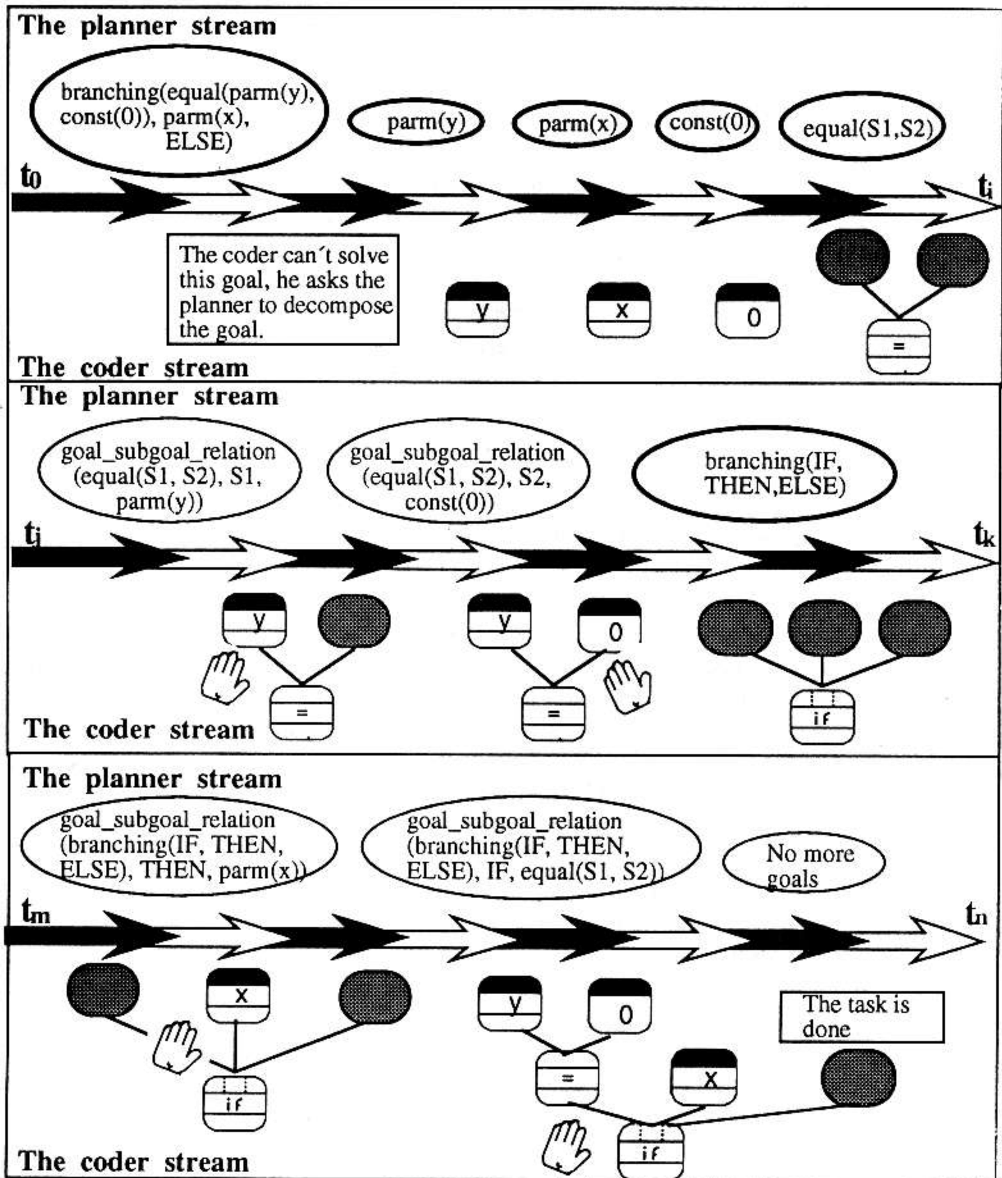


Figure 9. Squence of interactions between planner and coder while solving the goal "branching (equal (parm(y), const(0)), parm(x), ELSE)" with the set {O1, O5, L1, L2} of simple rules

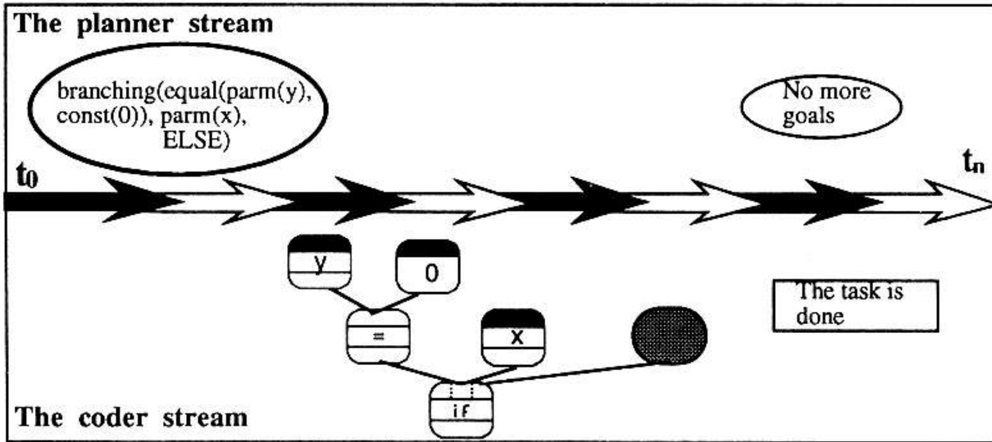


Figure 10. Sequence of interactions between planner and coder while solving the goal "branching (equal (parm(y), const(0)), parm(x), ELSE)" with the schema C7

If we compare the first interaction (Figure 9) where the coder knows {O1, O5, L1, L2} with the second one (Figure 10) where the coder knows C7, we observe:

- In the first sequence, the coder implements five program fragments corresponding to the subgoals delivered by the planner. In the second sequence, the coder implements just one program tree corresponding to the goal.
- In the first sequence, the planner gives explicit information about linking program fragments, and the coder rearranges program fragments accordingly, if necessary. In the second sequence, there is no such information.

In order to enable *empirical predictions*, we associate the following empirical claims with these observations:

Implementation of ABSYNT program fragments: If the coder applies a certain GMR rule, then exactly the ABSYNT program fragment contained in it is implemented in an uninterrupted sequence of programming actions (like positioning a node, drawing a link, etc.). We do not postulate order constraints *within* this sequence, but we expect the sequence not to be interrupted by programming actions stemming from *different* rule instantiations.

Verbalization of goals: Following the theoretically motivated distinction of a planner and a coder, selecting goals and subgoals for implementation by the coder is an act of planning involving control decisions. So is seems reasonable that at these decision points, the selected goals may be verbalized (Ericsson & Simon, 1984). The verbalizations explained by the selection of a certain GMR rule may be intermixed with the rule's programming actions but not with verbalizations and actions stemming from different rule instantiations.

Correction of positions: If the just implemented program fragment solves a dangling call or calls for another fragment already implemented, then it is to be connected with this

existing fragment. Now, corrective programming actions are likely: lengthening links, changing their orientation, and moving nodes.

If we compare the application of a single composite to the application of a set of simple rules (like C7 vs. {O1, O5, L1, L2}), then the following empirical consequences are assumed to result:

Implementation of ABSYNT program fragments (implementation hypothesis): For the set of simple rules, the order of rule applications is indeterminate, but the programming actions described by each rule should be continuous. *Actions of different rule instantiations should not interleave.* In contrast, when applying the composite, there are no order constraints on the programming actions at all since just one rule is applied.

Verbalization of goals (verbalization hypothesis): In the example, if the coder's knowledge contains C7, the planner has to make one control decision. If the coder knows only {O1, O5, L1, L2}, the planner has to make at least five control decisions (depending on how the goal is decomposed). Thus, we expect that applying composites is accompanied by *fewer goal verbalizations* than applying corresponding sets of simple rules.

Correction of positions (rearrangement hypothesis): In case of the composite, there are no open GMR calls to be implemented, and there are no to-be-linked program fragments left by earlier rule applications. Thus, we expect that applying composites leads to *fewer position corrections* of ABSYNT nodes and links than applying the corresponding sets of simple rules.

Performance time (time hypothesis): Planning, selecting, and verbalizing goals, and correcting positions of nodes and links are internal or external actions that are expected to need time (Rosenbloom & Newell, 1987). Thus, we expect that applying composites is *faster* than applying the corresponding sets of simple rules.

These predictions have not yet been investigated empirically, except for the implementation hypothesis (see below). But both the implementation hypothesis and the time hypothesis are used in the construction of the Internal Model to be described now.

The Internal Model (IM)

The IM is a set of domain specific knowledge (simple GMR rules and composites) utilized and continuously updated. As stated earlier, the IM covers the subset of the ISP-DL Theory shaded in Figures 1 to 4. So before describing it in detail, we will sketch it in terms of the ISP-DL Theory.

Concerning Figure 1: The PS is faced with a programming task (*goal*) and constructs a solution proposal (*solution*). The solution is parsed, using the *knowledge base* (rules in the IM and—as far as needed—in EXPERT). Subsequently, the rules just used for parsing are *optimized* by composition.

Since these new composites may be based on EXPERT rules, they are not directly inserted into the IM: According to ISP-DL Theory, a rule can only be improved after it is successfully

applied. This implies for the IM that it cannot at the same time be augmented by a new simple rule (from EXPERT) and by composites built from the same simple rule. For this reason, in addition to the IM there is a set *POSS* of possible candidates for future composites of the IM. Composites of the rules used for parsing a solution proposal are generated and kept in POSS as candidates. Only those surviving a later test are moved into the IM. These rules represent the result of "*deductive knowledge optimization,*" that is, *improved* knowledge.

Concerning Figure 2: If parsing the solution is possible solely with rules in the IM, then the IM is considered as sufficient to construct the solution, and "Goal Processing" is terminated ("*reaction to success*"). But if parsing the solution requires additional EXPERT rules, then the IM may be augmented by these (simple) rules, which represent the result of "*inductive knowledge acquisition*", that is, knowledge *newly acquired* in response to impasses.

Concerning Figure 3: The parse tree represents the student's hypothetical solution *plan*, which *execution* led to a *protocol*: the sequence of programming actions, verbalizations, and corrections exhibited by the student. We call that part of the protocol consisting only of the student's programming actions (creating nodes and links, naming nodes) the student's *action sequence*. The action sequence is used to evaluate the parse rules:

Since knowledge improvement should result in sped-up performance (*time hypothesis*), a composite is moved from POSS to IM only if the PS shows a *speedup from an earlier to a later action sequence* where both sequences can be produced by the composite.

The IM contains only GMR rules (simple rules and composites) which proved to be *plausible* with respect to an action sequence at least once. This is defined now. With respect to some action sequence, GMR rules form four subsets:

(1) Rules not containing any program fragments ("goal elaboration rules") are *nondecisive* with respect to the action sequence. (But verbalizations can be related to the goal elaboration rules; see Möbus & Thole, 1990).

(2) Rules whose head contains a program fragment which is part of the final result produced by the action sequence, and which was programmed in a *noninterrupted*, temporally continuous subsequence (see the *implementation hypothesis*). These rules are *plausible* with respect to the action sequence.

(3) Rules also containing a program fragment which is part of the final result of the action sequence, but this fragment corresponds only to the result of a *non*continuous action subsequence *interrupted* by other action steps. These rules are *implausible* with respect to the action sequence.

(4) Rules whose head contains a program fragment which is not part of the final result produced by the action sequence. These rules are *irrelevant* to the action sequence.

A *credit* rewards the usefulness of the rules in the IM. It is the product of the length of the action sequence explained by the rule and the number of its successful applications (frequency of being plausible). Thus, the credit depends on the empirical evidence gathered for a rule.

During the knowledge acquisition process, the IM is utilized and continuously updated according to a processing cycle shown in Figure 11:

- *Start* (Top of Figure 11): The first programming task is presented. Initially, both sets IM and POSS are empty.
- Now the learner solves the first task presented. Thus, an *action sequence* is produced, leading to a *solution* to the task. The action sequence is saved in a log file.
- *First Test:* IM and POSS are empty, so nothing happens.
- *First Parse:* The learner's ABSYNT program solution to the actual task is parsed with the EXPERT rules, leading to a set of parse rules.
- *First Generate*: The EXPERT rules just used for parsing are compared to the action sequence. The *plausible* parse EXPERT rules are put into the IM and get credit. Then the composites of all parse rules are created and compared to the action sequence. The plausible composites are kept in POSS. They are candidates of improved knowledge useful for future tasks. For each plausible composite, the time needed by the PS to perform the corresponding action sequence is attached. Now the Generate phase is finished, resulting in an updated POSS and IM.
- Now the next task is presented to the PS. The PS creates an ABSYNT action sequence and solution to it.

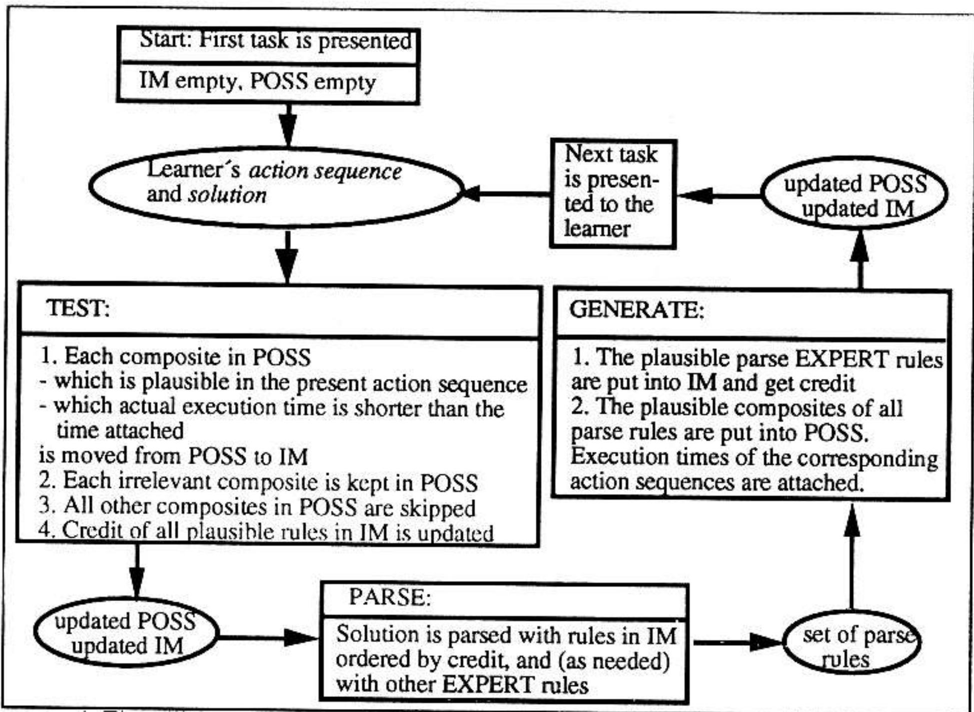


Figure 11. The utilizing and updating cycle of the IM during the knowledge acquisition process

- *Second Test*: Each composite in POSS is checked if
  (a) it is plausible with respect to the action sequence, and
  (b) the time needed by the PS to perform the respective continuous action sequence is shorter than the time attached to the composite. This means that the PS performs the action set *faster* than the previous corresponding action set which led to the creation of the composite.
  The composites meeting these requirements are put into the IM. Composites irrelevant to the action sequence of the solution just created are left in POSS. They might prove as useful composites on future tasks. All other composites violate the two requirements. They are skipped; that is, they are composites implausible to the actual sequence, or composites which predict a more speedy action sequence than observed. This means that the PS performs the action set *slower* than the previous corresponding action set which led to the creation of the composite. This slow-down is inconsistent with our model assumption that the PS prefers composites to simple rules; thus, the composite is not transfered to the IM but skipped.
  Finally, the credits of all rules in the IM which are plausible with respect to the present action sequence are updated. Thus, the second test leads to an updated POSS and IM.

- *Second Parse*: Now the solution of the second task is parsed with the rules of the IM ordered by their credits. As far as needed, EXPERT rules are also used for parsing.

- *Second Generate*: The plausibility of EXPERT rules which have just been used for parsing is checked. The plausible EXPERT parse rules are again put into the IM and get credit. Furthermore, the composites of all actual parse rules are created. The plausible composites are put into POSS; they will be tested on the next test phase. Again, the time needed for the corresponding action sequence is stored with each composite.

With respect to the design principles from ISP-DL Theory for the IM mentioned earlier, the IM contains *simple rules* (stemming from EXPERT) representing newly acquired but not yet improved knowledge, and *composites* representing various degrees of expertise. Empirical *performance* data such as speedup and the concept of plausibility are used for updating the IM. Furthermore, the IM predicts order constraints on action sequences, verbalization, rearrangement, and time. As in the ABSYNT Problem Solving Monitor, *planning* knowledge is not yet accounted for in the IM, but work is in progress. Finally, *help* generation will be discussed below.

An Illustration of the IM

Figure 12 shows a continuous fragment of the action sequence of a PS on a programming task. Again we will focus on the rules O1, O5, L1, L2, and C7 (see Figures 7 and 8). When S2 performs the sequence of Figure 12, O1, L1 and L2 are already in the IM from earlier tasks. O5 is not yet in the IM but only in the set of EXPERT rules. C7 has not yet been created.

After the subject has solved the task, the *Test Phase* (Figure 11) starts. Since the only composite we look at here (C7) has not been created, we only consider the fourth subphase: credit updating. O1 is *implausible* with respect to Figure 12 because the actions corresponding to the rule head of O1 are not continuous but *interrupted*. They are performed at

11:15:52, 11:15:58, 11:16:46, and 11:16:55 (Figure 12). Thus, the action sequence corresponding to the rule head of O1 is interrupted at 11:16:42 and 11:16:50.

L1 and L2 are also implausible. Actions corresponding to L1 are performed the first time at 11:15:08 and 11:15:29. Thus, this sequence is interrupted at 11:15:16 and 11:15:22. L1-like actions are shown a second time by the PS at 11:16:42 and 11:16:50. These are interrupted, too. Actions corresponding to L2 are performed at 11:15:16 and 11:15:34, with interruptions at 11:15:22 and 11:15:29. So since O1, L1, and L2 are implausible, their credits are not changed.

Now the subject's solution is *parsed* with rules in the IM and, as needed, with additional EXPERT rules (Figure 11). O1, O5, L1, and L2 are among the parse rules in this case, as no other rules have a higher credit and are able to parse the solution.

After the Parse Phase, the *Generate Phase* (Figure 11) starts. O5 is an EXPERT rule used for parsing. But O5 is implausible, since its corresponding actions were performed at 11:15:22, 11:15:38, and 11:15:43, with interruptions at 11:15:29 and 11:15:34. So O5 is not put into the IM. Then the composites of the parse rules are formed. C7 (Figure 8) is a composite formed from O1, O5, L1, and L2. This composite is plausible because it describes the uninterrupted sequence of programming actions from 11:15:08 to 11:16:55 (see Figure 12)—despite the fact that its components O1, O5, L1, and L2 are all implausible. Starting from the beginning of the task (at 11:14:40), the time for this action sequence is 135 seconds. Thus, the composite C7 is stored in POSS with "135 seconds" attached to it.

After solution of the next task, the now following Test phase reveals that C7 is plausible again. The corresponding action sequence (not depicted) was performed in 92 seconds, which is less than 135. So C7 is moved into the IM and gets a credit of 13 since it describes 13 programming steps (see Figure 12). This credit will be incremented by 13 each time the composite is plausible again.

An Empirical Analysis of the IM

As stated, the IM gives rise to several empirical predictions. We investigated the *implementation* hypothesis, stating that the programming actions described by a rule in the IM are performed in a continuous uninterrupted temporal sequence for a single subject creating non-recursive solutions to seven ABSYNT programming tasks. The IM was run offline based on the action sequences exhibited by the subject. For every solution, a *model trace* (order constraints on the action sequence) was predicted and compared to the *subject trace*, the subject's action sequence that led to the solution. Figure 13 shows

(a) a subset of the *rules* in the IM after the subject solved the third programming task of the task sequence. For each IM rule, the rule name and the program fragment in its head are depicted.

(b) the subject's *solution* to the next task, "quot" (to construct a program dividing the larger by the smaller of two numbers).

(c) the predicted constraints on the action sequence (*model trace*), given this solution. The program fragments predicted to be constructed in an uninterrupted sequence are boxed. Thus, for example, the if-then-else node rule predicts that programming the if-then-else node and three links leaving it are four programming actions performed in an uninterrupted sequence. So the corresponding fragment is boxed in Figure 13c.

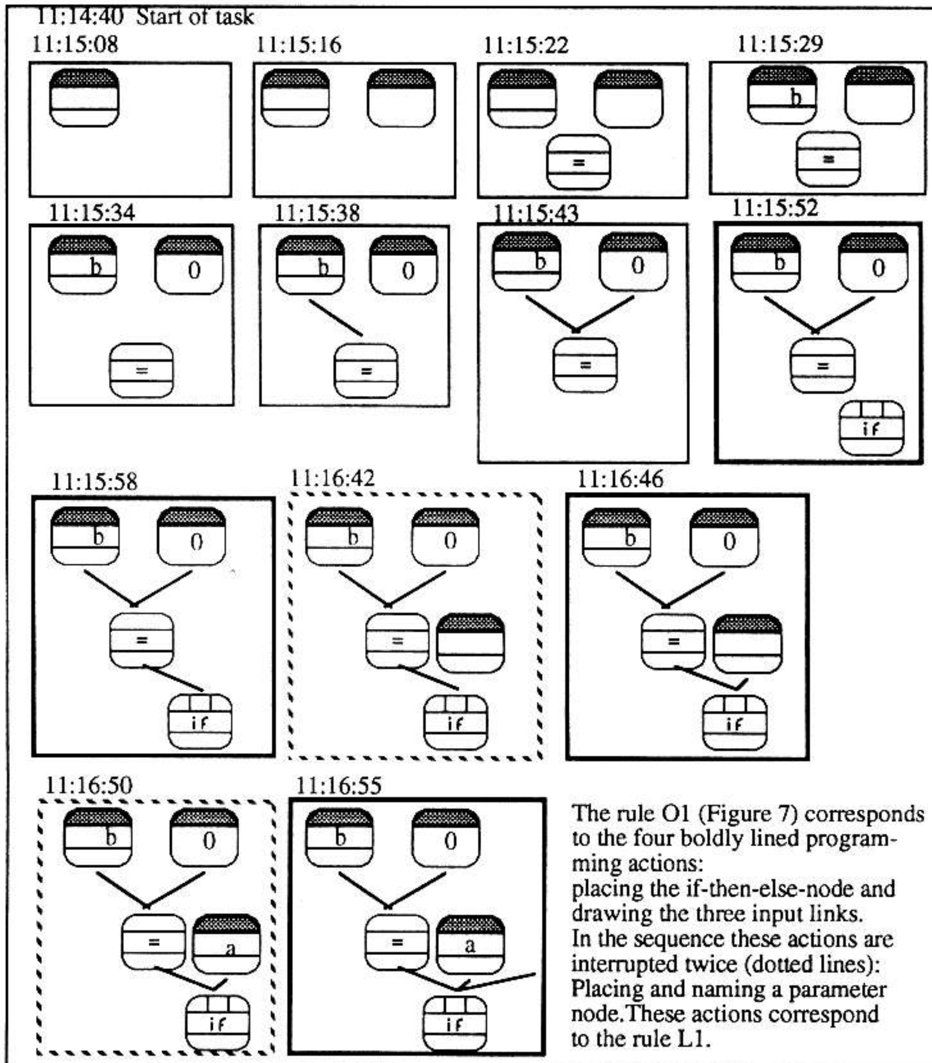


Figure 12. A continuous fragment of a sequence of programming actions of a subject

Similarly, placing a parameter node and naming this node are two actions predicted to occur subsequently. Instances of the same nodes are indexed.

(d) the observed actions sequence of the subject (*subject trace*). A "+" denotes a pair of programming actions explained by the same IM rule and perfomed subsequently. A "-" denotes a pair of programming actions explained by the same IM rule but interrupted by some other action(s). Thus, "+" denotes fits and "-" denotes contradictions to the predictions.

There were 80 "+" and 49 "-" for the seven action sequences. Since more "+" should lead to fewer runs than equally many "+" and "-", we applied the Runs test. There were 40 runs ($p < 0.001$).

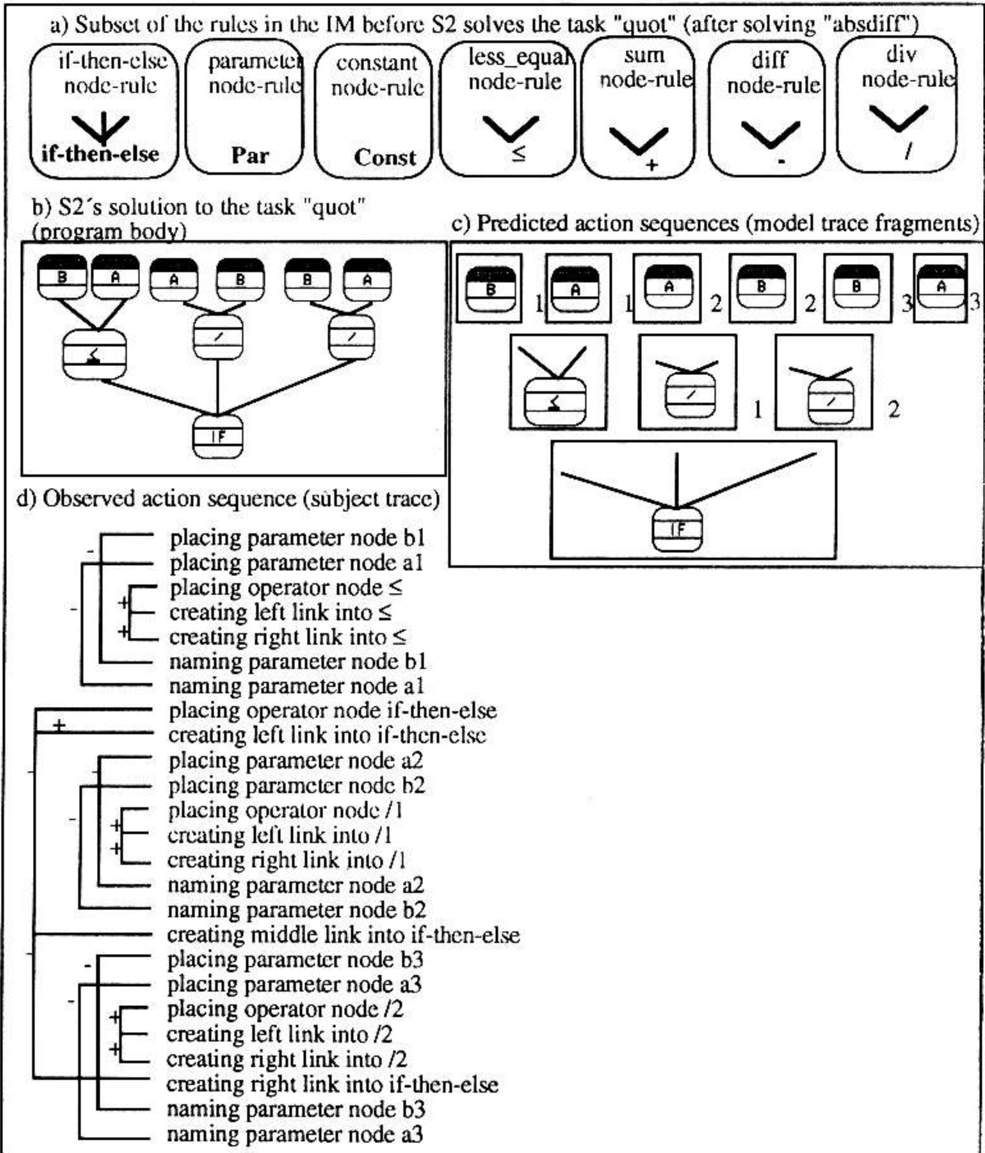


Figure 13. a) subset of the IM before solving "quot", b) S2's solution to "quot", c) predicted action sequences, d) observed action sequence corresponding (+) or contradicting (-) to the predictions

Discussion

We presented an approach to online diagnosis of students' knowledge which is intended to be based on a *theoretical framework* on problem solving and learning, to be empirically *valid*, and to support *adaptive help* generation. We will now discuss how far the IM meets these requirements and how we plan to improve it.

Foundation on a theoretical framework: In the section "The Internal Model", we showed how in our view the IM is related to the ISP-DL Theory. We tried to motivate the features of the IM by the theory. But still many aspects of the theory remain uncovered by the IM. Two of them are:

- *Generalization* of knowledge. Our observations from single-subject sessions with ABSYNT indicated use of previous solutions and positive transfer especially for recursive tasks. Thus composites in the IM should be generalized. *Generalization of composites* may be viewed as another way of knowledge optimization (e.g., Anderson, 1983; Wolff, 1987) in response to the successful utilization of knowledge (Figure 1). Additionally, generalized knowledge should also result from *analogizing* as an alternative to synthesizing a plan (Figure 3).

- Synthesizing a *plan*. Currently, the IM takes account only of the implementation level, but there is no representation of planning knowledge within the IM.

Concerning *generalization*, it is easy to construct generalized GMR rules (containing variables not only at the leaves of the goal trees and program trees) by using a concept hierarchy. Concerning a *planning level*, we want the learner to be able to construct plans with an extension of the ABSYNT language by new goal nodes so that *mixed* ABSYNT programs containing operator nodes and goal nodes will be possible. The learner will be able to test hypotheses and to receive error and completion feedback at this *planning* level even if he or she has no idea yet about the implementation. Thus, the learner may first *plan* a goal tree for the task at hand, test hypotheses about it, and debug it, if necessary. Afterwards, the learner may *implement* the goals by replacing them with operator nodes or subtrees.

From the user's point of view, the benefit of using goal nodes will be that hypotheses testing, will be possible at the *planning stage*, not just at the implementation stage. From a psychological point of view, the benefit is that *objective* data about the planning process can be obtained in addition to the verbalizations. Finally, from a help system design point of view, the benefit is that in addition to hypotheses testing it will be possible to offer *planning rules* as help to the learner. The planning rules will be visual representations of GMR goal elaboration rules.

Empirical validity. First of all, the model has to be empirically testable. We showed that many empirical predictions are possible based on the IM, and some more will be presented below. In addition, a first empirical investigation revealed that the IM describes a considerable portion of the subject trace. But the investigation also revealed how the IM might be improved. The following table shows how the "+" and "-" are distributed across different types of rules in the IM:

	Parameter node rule	*Constant node rule*	*Primitive operator node rules*	*Composites*
"+" cases	2	4	47	27
"-" cases	27	7	14	1

Thus, the parameter node rule, for example, is responsible for 2 "+" and for 27 "-": The subject usually does not place and name a parameter node in sequence. The same seems true for the constant node rule. Obviously, given that this result will be reproduced with other subjects, it should be possible to enhance the IM by splitting the parameter node rule (and the constant node rule as well) into two new rules; one for positioning and one for naming a parameter node. Then the current parameter node rule would have to be considered as a *composite* of these two new rules.

Adaptive help generation. The ultimate goal of the IM is to provide *adaptive* help or, more generally, to have an impact on the user-system-interaction in a way that takes account of the individual. In the ABSYNT Problem Solving Monitor, the need for the IM is obvious:

- There is a large solution space (the system is able to analyze and generate many solutions to given tasks) which is necessary because we want to be able to take care of novices' often unusual or unnecessarily complicated solutions (see Figure 6).

- Because of the large solution space, there is usually a large amount of completion proposals that can be generated by the system, so the problem is which one to select. The task of the IM is to enable *user-centered* selection.

But as indicated, the role of the IM will not be restricted to the completion of ABSYNT nodes. Extending completion to the planning level and offering visual planning rules as help will impose additional demands to the IM. Additionally, the IM does more than just help selection. The information provided to the student may be varied in several ways, and this gives rise to empirical predictions which in turn might support or weaken the IM. Figure 14 illustrates how information intended as help can be varied and what can be predicted. Basically, when the student is caught in an impasse and asks for a completion proposal, according to the IM there are two possible situations:

- The student lacks domain-specific implementation knowledge. Thus, with respect to the interaction of planning and coding described earlier, there is a *coding* problem.

- The student has knowledge of how to proceed but does not make use of it; there is a *planning* problem.

Therefore, it should be possible to *predict impasses* based on the IM. Another prediction is that information intended as help should address the right level. If the student is stuck because of a *planning* problem, he or she should get annoyed or even upset by information at the *coding* level (or implementation level), and vice versa. But more specific predictions within each level are possible too: Figure 14 depicts a hypothetical situation where the student just performed some programming actions, then gets stuck and asks for completion

proposals. According to the IM, there is a knowledge gap on the coding level, and after filling it, the student would be able to proceed (shaded part of the arrow in the upper right of Figure 14). Now there are several possibilities to react to the gap: The information provided might vary in *grain size* and *amount* (on the left of Figure 14).

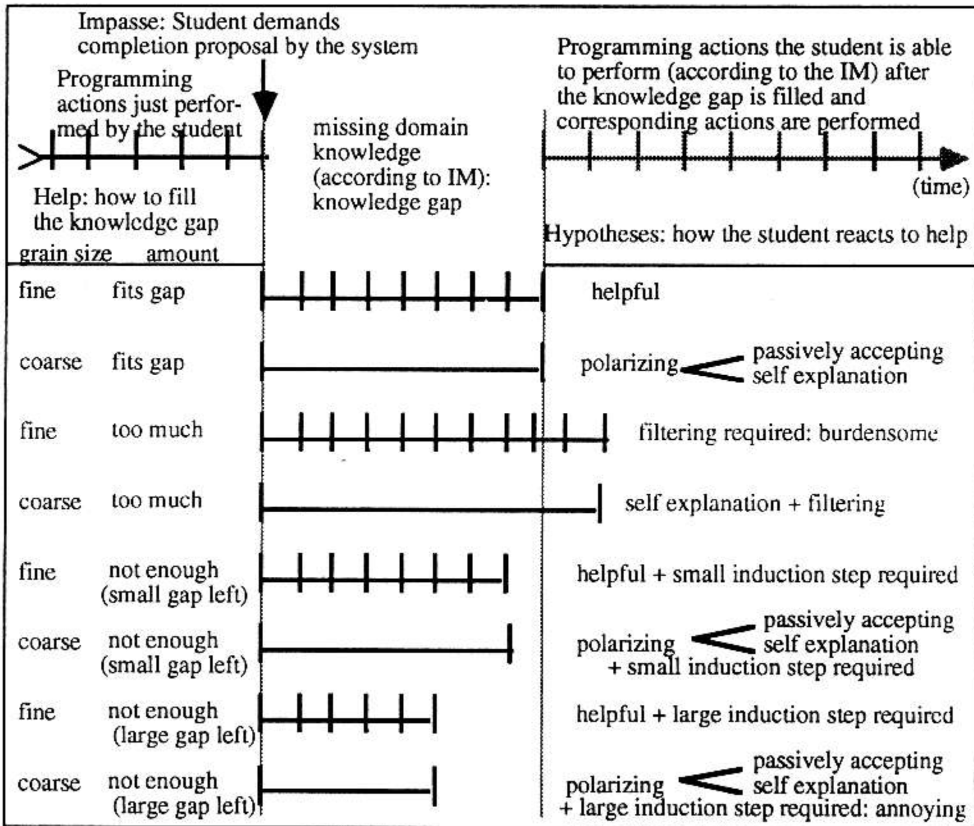


Figure 14. Types of information possibly provided in response to a knowledge gap diagnosed by the IM, and hypotheses concerning the student's reaction to this information

- *Grain size* concerns the rules underlying the completion proposal. If the grain size is *fine*, then the completion proposal may rest on a chain of simple rules which covers the gap. In this case, the completion proposal may consist of an ABSYNT subtree with an explanation of each programming step needed to construct this subtree, where the explanation is based on the goal structure of the chain of simple rules. If the grain size is *coarse*, then the completion proposal may rest on a single composite (to take the other extreme). The same subtree may be provided, but without an explanation.

- *Amount* concerns the relation between the completion and the gap. The completion proposal might *exactly fill* the gap, so subsequently the student can proceed by relying on her / his own knowledge. Alternatively, the completion proposal may contain *too much* information (more than necessary) or *not enough* information (the gap is not completely covered).

On the left and middle part of Figure 14, the different combinations of grain size and amount of information are shown. They lead to different hypotheses (on the right of Figure 14). We will describe some of them:

- If the information is fine-grained and exactly fills the gap (first row in Figure 14), then we would expect that the student considers this information as *helpful*.
- If the information is coarse-grained and exactly fills the gap (second row), then the student misses explanations. S/He might either *passively accept* what is being offered, or engage in *self-explanation* (VanLehn, 1991a).
- If the information is fine-grained but exceeds the knowledge gap (third row), then the student has to"*filter*" the content relevant to the current situation. This might be experienced as *burdensome*.
- If the information leaves a small knowledge gap (fifth and sixth rows), then the student might try to induce one new simple rule and thereby cover the rest of the gap. (This situation seems similar to the induction of one subprocedure at a time by VanLehn's [1987] SIERRA program.)
- Finally, the last case to be considered here is that there is a large gap left, and the information offered is too coarse (last row). The student should experience such information as very inadequate to his current problem. She or he, then, should feel annoyed or even upset.

There remains much research, of course, to work out these hypotheses and put them to empirical test. But we think we have shown that the IM is an empirically fruitful approach to knowledge diagnosis and adaptive help generation which is testable and also touches upon further important research problems, like motivation and emotion.

References

Anderson, J.R. (1983). The architecture of cognition. Cambridge, MA: Harvard University Press.

Anderson, J.R. (1986). Knowledge compilation: The general learning mechanism. In R.S. Michalski, J.G. Carbonell, & T.M. Mitchell (Eds.), *Machine Learning*, vol. 2 (pp. 289-310). Los Altos: Kaufmann.

Anderson, J.R. (1989). A theory of the origins of human knowledge. *Artificial Intelligence*, *40*, 313-351.

Anderson, J.R., Boyle, C.F., Farrell, R., Reiser, B.J. (1987). Cognitive principles in the design of computer tutors. In P. Morris (Ed.), *Modelling cognition* (93-133) New York: Wiley.

Bauer, F.L., & Goos, G. (1982), *Informatik* (Vol. 1), Berlin: Springer (3rd ed.)

Brown, J.S., Burton, R.R. (1982). Diagnosing bugs in a simple procedural skill. In D. Sleeman & J.S. Brown (Eds.), *Intelligent Tutoring Systems* (pp. 157-183). New York: Academic Press.

Brown, J.S., & VanLehn, K. (1980). Repair theory: A generative theory of bugs in procedural skills. *Cognitive Science*, *4*, 379-426.

Chang, S.K. (Ed.). (1990). Principles of visual programming systems. Englewood Cliffs: Prentice Hall.

Chase, N.G., & Simon, H.A. (1973). Perception in chess. *Cognitive Psychology*, *4*, 55-81.

Elio, R., & Scharf, P.B. (1990). Modeling novice-to-expert shifts in problem solving strategy and knowledge organization. *Cognitive Science*, *14*, 579-639.

Ericsson, K.A., & Simon, H.A. (1984). Protocol analysis. Cambridge, MA: MIT Press.

Frasson, C., & Gauthier, G. (Eds.). (1990). Intelligent tutoring systems. Norwood, N.J.: Ablex.

Gollwitzer, P.M. (1990). Action phases and mind sets. In E.T. Higgins & R.M. Sorrentino (Eds.), *Handbook of motivation and cognition: Foundations of social behavior.* (Vol. 2, pp. 53-92).

Gregory, S. (1987). *Parallel logic programming in PARLOG: The language and its implementation.* Wokingham: Addison-Wesley.

Gugerty, L., & Olson, G.M. (1986). Comprehension differences in debugging by skilled and novice programmers. In E. Soloway & S. Iyengar (Eds.), *Empirical studies of programmers* (pp. 13-27). Norwood, NJ: Ablex.

Hogger, C.J. (1990). *Essentials of logic programming*. Oxford University Press.

Huber, P., Jensen, K., & Shapiro, R.M. (1990). Hierarchies in coloured Petri Nets. In G. Rozenberg (Ed.), *Advances in Petri Nets* 1990, LNCS, Heidelberg: Springer-Verlag.

Kearsley, G. (1988). *Online help systems.* Norwood, NJ: Ablex.

Kowalski, R. (1979). *Logic for problem solving.* Amsterdam: Elsevier Science Publishers.

Laird, J.E., Rosenbloom, P.S., & Newell, A. (1986). Universal subgoaling and chunking. *The automatic generation and learning of goal hierarchies.* Boston: Kluwer.

Laird, J.E., Rosenbloom, P.S., & Newell, A. (1987). SOAR: An architecture for general intelligence. *Artificial Intelligence, 33*, 1-64.

Lewis, C. (1987). Composition of productions. In D. Klahr, P. Langley, & R. Neches (Eds.), *Production system models of learning and development* (pp. 329-358). Cambridge: MIT Press.

Möbus, C. (1990). Toward the design of adaptive instructions and help for knowledge communication with the problem solving monitor ABSYNT. In V. Marik, O. Stepankova, & Z. Zdrahal (Eds.), *Artificial intelligence in higher education. Proceedings of the CEPES UNESCO International Symposium* (pp. 138-145). Prague, CSFR, October 23-25, 1989, Berlin-Heidelberg-New York: Springer-Verlag, LNAI 451.

Möbus, C. (1991). The relevance of computational models of knowledge acquisition for the design of helps in the problem solving monitor ABSYNT. In R. Lewis & H. Otsuki (Eds.), *Advanced research on computers in education* (pp. 137-144). Amsterdam: North-Holland.

Möbus, C., & Schröder, O. (1990). Representing semantic knowledge with 2-dimensional rules in the domain of functional programming. In P. Gorny & M. Tauber (Eds.), *Visualization in human computer interaction* (pp. 47-81). Berlin: Springer-Verlag.

Möbus, C., & Thole, H.-J. (1989). Tutors, instructions and helps. In T. Christaller (Ed.), *Künstliche Intelligenz KIFS 1987, Informatik-Fachberichte 202* (pp. 336-385). Heidelberg: Springer-Verlag.

Möbus, C., & Thole, H.J. (1990). Interactive support for planning visual programs in the problem solving monitor ABSYNT: Giving feedback to user hypotheses on the language level. In D.H. Norrie & H.W. Six (Ed.), *Computer assisted learning. Proceedings of the 3rd International Conference on Computer-Assisted Learning ICCAL 90,* Hagen, Germany, LNCS 438, Heidelberg: Springer-Verlag.

Neves, D.M., & Anderson, J.R. (1981). Knowledge compilation: Mechanisms for the automatization of cognitive skills. In J.R. Anderson (Ed.), *Cognitive skills and their acquisition* (pp. 57-84). Hillsdale, NJ: Lawrence Erlbaum.

Newell, A. (1982). The knowledge level. *Artificial Intelligence, 18*, 87-127

Rosenbloom, P.S., Laird, J.E., Newell, A., & McCarl, R. (1991). A preliminary analysis of the SOAR architecture as a basis for general intelligence. *Artificial Intelligence, 47*, 289-305.

Rosenbloom, P.S., & Newell, A. (1987). Learning by chunking: A production system model of practice. In D. Klahr, P. Langley, & R. Neches (Eds.), *Production system models of learning and development* (pp. 221-286). Cambridge, MA: MIT Press.

Schröder, O. (1990). A model of the acquisition of rule knowledge with visual helps: The operational knowledge for a functional, visual programming language. In D.H. Norrie & H.-W. Six (Eds.), *Computer-assisted learning: Proceedings of the 3rd Int. Conf. ICCAL 90* (pp. 142-157). Berlin: Springer-Verlag.

Self, J.A. (1990). Bypassing the intractable problem of student modeling. In C. Frasson & G. Gauthier (Eds.), *Intelligent tutoring systems* (pp. 107-123). Norwood, NJ: Ablex.

Self, J.A. (1991). Formal approaches to learner modelling. (Tech. Rep. No. AI-59). Lancaster, England: Lancaster University, Dept. of Computing.

Simon, H.A., & Simon, D.P. (1978). Individual differences in solving physics problems. In R.S. Siegler (Ed.), *Childrens' thinking: What develops?* (pp. 325-348). Hillsdale, NJ: Erlbaum.

Sleeman, D. (1984). An attempt to understand students' understanding of basic algebra. *Cognitive Science, 8*, 387-412.

Sleeman, D., & Brown, J.S. (1982). *Intelligent tutoring systems*. New York: Academic Press.

VanLehn, K. (1987). Learning one subprocedure per lesson. *Artificial Intelligence, 31*, 1-40.

VanLehn, K. (1988). Toward a theory of impasse-driven learning. In H. Mandl & A. Lesgold (Eds.), *Learning issues for intelligent tutoring systems* (pp. 19-41). New York: Springer-Verlag.

VanLehn, K. (1990). *Mind bugs: The origins of procedural misconceptions*. Cambridge: MIT Press.

VanLehn, K. (1991a). Two pseudo-students: Applications of machine learning to formative evaluation. In R. Lewis & S. Otsuki (Eds.), *Advanced research on computers in education* (pp. 17-25). Amsterdam: North-Holland.

VanLehn, K. (1991b), Rule acquisition events in the discovery of problem solving strategies. *Cognitive Science, 15*, 1-47.

Vere, S.A. (1977). Relational production systems. *Artificial Intelligence, 8*, 47-68.

Wenger, E. (1987). *Artificial intelligence and tutoring systems*. Los Altos, CA.

Wolff, J.G. (1987). Cognitive development as optimisation. In L. Bolc (Ed.), *Computational models of learning* (pp. 161-205). Berlin: Springer-Verlag.

Acknowledgement: We thank Jörg Folckers for reimplementing ABSYNT in LPA-PROLOG for Macintosh computer. Now we can switch off our LISP machine.