Cognition and Computer Programming

edited by

Karl F. Wender
Franz Schmalhofer
Heinz-Dieter Böcker

Cognition and Computer Programming

edited by

Karl F. Wender
Department of Psychology
University of Trier
Germany

Franz Schmalhofer
German Center for Artificial Intelligence
University of Kaiserslautern
Germany

Heinz-Dieter Böcker
Integrated Publication and Information Systems Institute (IPSI)
Gesellschaft für Mathematik und Datenverarbeitung (GMD)
Darmstadt
Germany

Ablex Publishing Corporation
Norwood, New Jersey

Copyright (c) 1995 by Ablex Publishing Corporation

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, microfilming, recording or otherwise, without permission of the publisher.

Printed in the United States of America

Library of Congress Cataloging-in-Publication Data
Cognition and computer programming / edited by Karl F. Wender, Franz Schmalhofer, Heinz-Dieter Böcker.
p. cm. — (Ablex series in computational science)
Includes bibliographical references and index.
ISBN 1-56750-094-3 (cl). — ISBN 1-56750-095-1 (ppk)
1. Electronic digital computers—Programming. 2. Cognition.
I. Wender, Karl Friedrich. II. Schmalhofer, F. (Franz), 1952-
III. Böcker, Heinz-Dieter, 1949- IV. Series.
QA76.6.C6218 1994
005.1'01'9—dc20 93-44140
CIP

Ablex Publishing Corporation
355 Chestnut Street
Norwood, New Jersey 07648

CHAPTER 3

Online Modeling the Novice–Expert Shift in Programming Skills on a Rule-Schema-Case Partial Order*

Claus Möbus, Olaf Schröder, and Heinz-Jürgen Thole
University of Oldenburg
Department of Computational Sciences
Oldenburg, Germany

INTRODUCTION

The development of models of learners' *knowledge acquisition* processes is an important topic of cognitive science basic and applied research. Modeling knowledge acquisition processes has been recognized as a necessary extension to *status models*, that is, of bugs in skills (Brown & Burton, 1982; Brown & VanLehn, 1980; Sleeman, 1984), because models of this kind raise the question of the origins of the hypothesized knowledge structures. Research on knowledge acquisition processes (Anderson, 1983, 1986, 1989; Rosenbloom, Laird, Newell, & McCarl, 1991; Rosenbloom & Newell, 1986, 1987) also touches on applied research questions such as: Which order is the best for a set of tasks to be worked on? Why is information useless to one person and helpful to another? How is help and instructional material to be designed?

Answering these questions requires hypotheses about the learner's knowledge states and knowledge acquisition processes. This is especially true within help and tutoring systems (Frasson & Gauthier, 1990; Kearsley, 1988; Sleeman & Brown, 1982; Wenger, 1987), in which online diagnosis of the learner's knowledge (*learner model*) is necessary in order for the system to react

*The research for this chapter was sponsored by Deutsche Forschungsgemeinschaft under Grant No. Mo 292/3-3.

in an adequate way. For example, if several reactions are possible, the learner model should select the most appropriate one. The learner model has to be both efficient and valid. But to achieve both goals is a difficult problem (Self, 1990, 1991) because there is only a limited source of information—the learner's stream of actions.

We develop an adaptive help system that supports learners working on planning tasks. The help system has knowledge about a big solution space in order to be capable to recognize not only standard but also "unusual" solutions. In order to meet the requirements mentioned, we developed a *theoretical framework of problem solving and learning* that serves as a base for interpreting the student's actions and verbalizations and for constructing the learner model. There are two versions of this model:

- An *Internal Model* (IM) diagnoses the actual domain knowledge of the learner at different stages in the knowledge acquisition process (*state model*). It is based on the computer-assessable data provided by the interaction of the student with the system. The IM is designed to be an integrated part of the help system ("internal" to it) in order to provide online diagnosis, user-centered feedback, and help.
- An *External Model* (EM) is designed to simulate the knowledge acquisition processes of learners (*process model*) on a level of detail not available to the IM (i.e., including verbalizations). The EM is not part of the help system ("external" to it), but supports the design of the IM.

The application domain of our help system is ABSYNT—a functional visual programming language (Janke & Kohnert, 1989; Möbus & Schröder, 1989; Möbus & Thole, 1989) which is a tree representation of pure LISP. The ABSYNT problem-solving monitor (Möbus, 1990, 1991; Möbus & Thole, 1990) supports learners acquiring basic functional programming concepts. It provides help for the learner while working on programming tasks. Currently, we apply the concepts originally developed for ABSYNT to the design of a help system for modeling discrete systems with Petri nets (Möbus, Pitschke, & Schröder, 1992). In this chapter, we focus on the internal model (IM) of the ABSYNT problem-solving monitor. In the next section, our theoretical position on problem solving and learning is described. Then a short description of ABSYNT is provided. After that, the IM is described in some detail, including empirical hypotheses and a case study. Finally, some extensions, prospects, and conclusions are discussed.

THEORETICAL POSITION

For modeling knowledge changes and knowledge acquisition processes, a theoretical position of *problem solving* and *learning* is necessary that is able to

describe the shift of the learner from novice to expert (Elio & Scharf, 1990). In contrast to, for example, Elio and Scharf, our model is tightly constrained by empirical data. We think that it is useful in general to describe problem-solving processes for a given task by the following phases (similar to Gollwitzer, 1990; Gollwitzer, Heckhausen, & Steller, 1990):

1. *Deliberation phase*: For our concerns, this phase just consists of the decision of the problem solver to strive for the goal of the given task. A goal may be viewed as a set of facts about the environment which the problem solver wants to become true (Newell, 1982). More precisely, 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 "even" problem can be implemented by a function with a name such as "even," one parameter "n" which has the type "natural number," the output type of the function which is a boolean truth value, and the body of the function which has to meet the declarative specification: "There exists a natural number k such that 2 * k = n." This goal is achieved if a program is created which satisfies this description.
2. *Synthesizing phase*: The problem solver is concerned with how to achieve the goal. This requires planning knowledge for the elaboration of goals and implementation knowledge for the realization of these goals in the domain. The problem solver has to decide how to differentiate the goal into one or several subgoals and how to instantiate the subgoals. The synthesizing phase results in a set or sequence of sets of intended actions. As an alternative to synthesizing, a plan might be created by analogical reasoning.
3. *Execution phase*: The planned actions are executed.
4. *Evaluation phase*: The result is evaluated. That is, it is checked whether the solution obtained satisfies the task goal.

Novices and experts differ in various ways (e.g., Chi, Feltovich, & Glaser, 1981; Simon & Simon, 1978) in these phases. Novices work sequentially and engage much in planning. Many control decisions are necessary, and many subgoals have to be set. In contrast, experts need only a few control decisions and subgoal settings. They just plug in their "canned solution schemes" (Soloway, 1986). These schemes (Anderson, 1990; Bartlett, 1932) may provide solutions for whole tasks or for subtasks. The components of the schemes are recalled in *parallel*. Thus, while executing one schema, there is no prespecified order of action steps. So we hypothesize that the order of action steps is *indeterminate*. Control decisions like setting subgoals are only necessary

between, but not within, the solution schemes. Our theory assumes that only the *set* but not the *sequence* of programming actions contained in the schema can be predicted.

How does the knowledge acquisition process proceed? We found that knowledge acquisition while working on problems alternates between *impasse-driven* and *success-driven learning* (IDL-SDL) (Möbus & Thole, 1990; Schröder, 1990). According to IDL (Laird, Rosenbloom, & Newell, 1987; VanLehn, 1988, 1990, 1991b), the learner traps into impasses if he or she encounters unfamiliar situations. The learner gets stuck because the knowledge needed for the actual situation is not acquired. In response to an impasse, the learner employs weak heuristics, for example, asking for help. If problem solving with help is successful, the learner acquires new knowledge which enables him or her to overcome the impasse. According to SDL (Anderson, 1986, 1989; Lewis, 1987; Rosenbloom & Newell, 1987; Vere, 1977; Wolff, 1987), already acquired knowledge is optimized if used in familiar situations. This means that the solution schemes characteristic for experts are created from chains of more simple pieces of knowledge. As a result, less control decisions and subgoals are necessary, and performance will get faster in future situations. Thus, IDL-SDL theory makes a distinction between:

- *acquired* but not yet improved domain knowledge
- *improved* domain knowledge
- domain-unspecific *weak heuristics* for the acquisition of domain knowledge. Problem solving with these heuristics in response to an impasse can again be described by the four phases above: The problem solver considers the possibilities to get help and chooses the most promising one, for example, to ask. He plans how to effectively make use of the help, for example, what and whom to ask. Again the plan is executed, and the result is evaluated. Thus, the impasse should lead to a new problem-solving phase where the problem solver considers, plans, and uses help to generate a subgoal, to decide between two subgoals, or to localize a bug.

Figure 3.1 summarizes our theoretical framework as a higher order Petri net (Reisig, 1985). The IDL-SDL net of Figure 3.1 shows that goals are reached by problem solving, and the knowledge used is optimized. The subnet "problem solving" contains four phases: deliberation, planning, execution, and evaluation. Evaluation might reveal that the problem solution is faulty. This is an impasse, and a subgoal is set to resolve it. So the subnet "problem solving" is called recursively, and IDL (acquiring new knowledge) occurs in response to a subgoal solution. For example, new domain knowledge is acquired as a result of asking for help. SDL might occur as the result of a

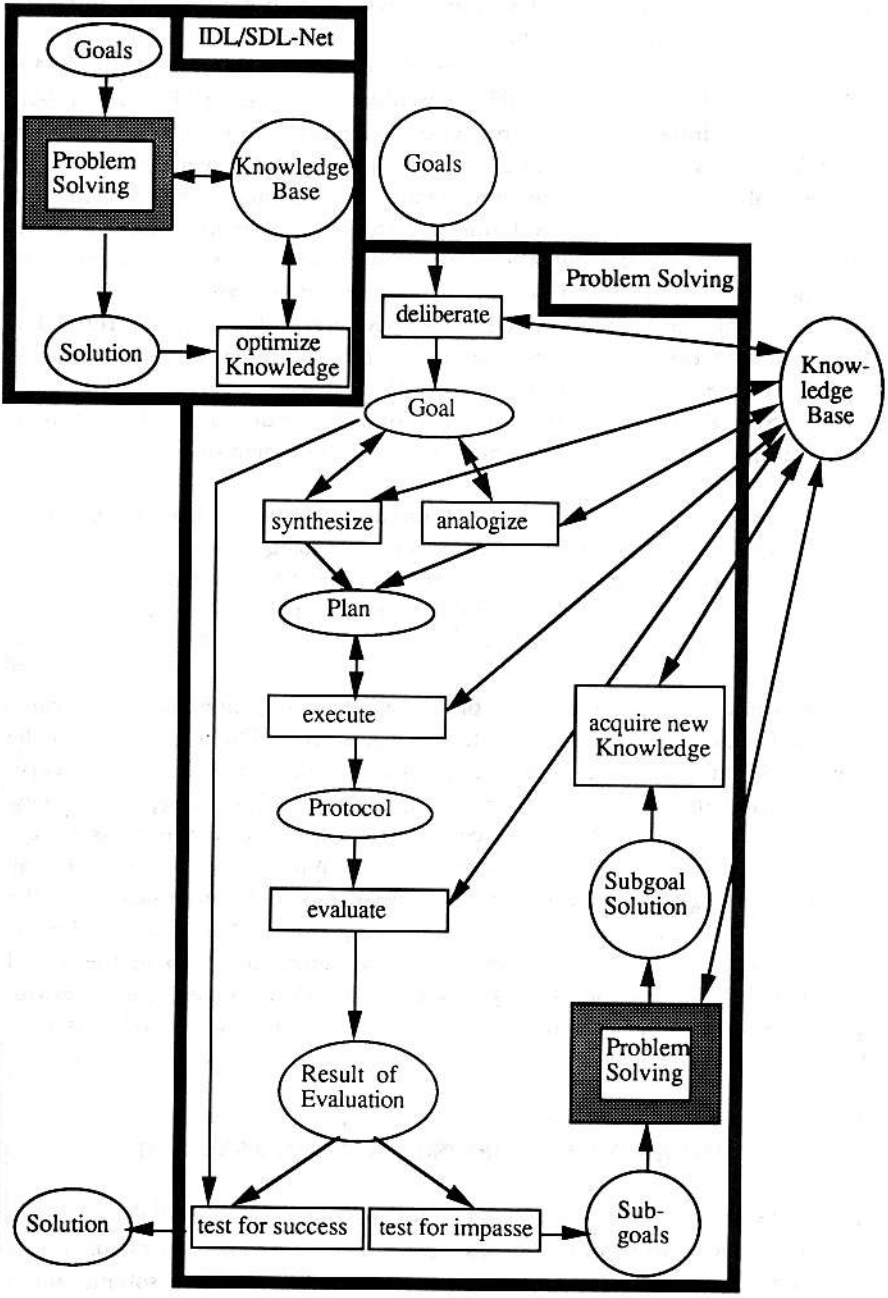


Figure 3.1. Sketch of the theoretical position as a higher order Petri Net

successful evaluation phase. Existing knowledge is optimized (by composition) so it can be used more efficiently.

There is some difference of our theoretical position to the SOAR architecture (Laird et al., 1987; Rosenbloom et al., 1991). In SOAR knowledge optimization ("chunking") can take place only after an impasse. In SOAR all knowledge changes stem from impasses. But it seems questionable whether all knowledge acquisition events can reasonably be described as resulting from impasses (VanLehn, 1991b). In our theory, knowledge is optimized ("composed") not after an impasse but after successful problem solving steps not preceded by an impasse, whereas impasses lead to the acquisition of *new* knowledge. Using SOAR terminology, in our IDL-SDL theory knowledge is optimized *within* the same problem space, whereas in SOAR knowledge is optimized *across* problem spaces.

IDL-SDL theory makes recommendations for the design of a help system. According to the theory, the learner will appreciate help only if:

- there is an actual *impasse* (without impasse, there is no need for help),
- it is synchronized to the learner's actual *knowledge* state, and
- it is synchronized to the actual *problem-solving level*. Help should not address implementation details if there is a planning problem, and vice versa.

According to these guidelines, the system does not interrupt the learner, but the decision to receive help is left to the learner. The learner may make use of help but is not constrained in his or her freedom to learn by discovery. On the one hand, the learner may engage in *free problem solving*, express new solution ideas, hypothesize their correctness, and receive proposals by the system on how to complete or correct the solutions. Thus, the learner may evaluate (parts of) a solution and require completion proposals from the system that are intended to encourage *self-explanation* (Chi, Bassok, Lewis, Reiman & Glaser, 1989) and *replanning*. Synchronizing help to the actual knowledge state is the job of the Internal Model (IM). Finally, planning with goal nodes is incorporated into the help system, but it is not yet addressed by the IM.

THE ABSYNT PROBLEM-SOLVING MONITOR

The ABSYNT problem-solving monitor provides an iconic environment (Glinert, 1990) and is aimed at supporting novices' acquisition of functional programming concepts up to recursive systems. As a problem-solving monitor, ABSYNT delivers help and proposals for solutions to given tasks, but it does not have a curricular component. The main components of ABSYNT

are: a visual program editor, a visual trace, and a help component. The user is free to switch among these components.

The *visual program editor* allows construction and syntactical checking of programs. The editor is not shown here, but many of its features are also visible in the hypotheses environment (Figures 3.2, 3.3, and 3.4): An ABSYNT program consists of a head tree and a body tree (see the upper half of Figure 3.2). Also there is a start tree (not shown) from which programs may be called. The nodes of the trees are constants, parameters, primitive and self-defined operator nodes. The links between the nodes are the "pipelines" for control and data flow. Programs are edited by taking nodes with the mouse from a menu bar (not shown) and connecting them.

The design of the visual program editor resulted from runnable specifications of the ABSYNT interpreter (Möbus & Thole, 1989) and of the "How-to use-it" knowledge (Janke & Kohnert, 1989). An analysis of ABSYNT in terms of properties of visual languages and cognitive design principles is provided in Möbus and Thole (1989).

The *visual trace* resulted from the runnable specifications of the interpreter. It makes each computational step of the ABSYNT interpreter visible. There is also a prediction environment in which the learner can predict the

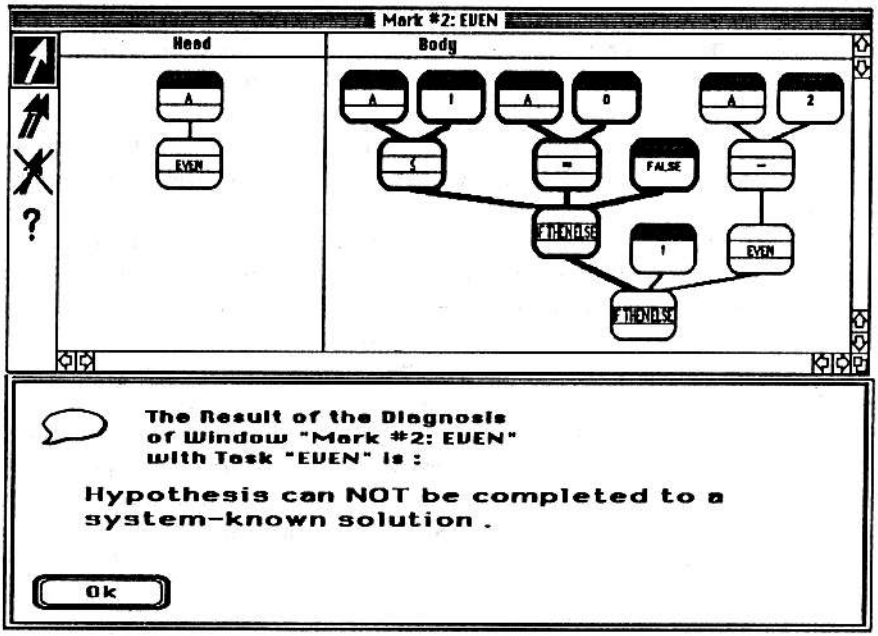


Figure 3.2. A snapshot of the ABSYNT interface showing an incorrect program with a user hypothesis (bold) which leads to the system's feedback: cannot be completed to a solution known by the system"

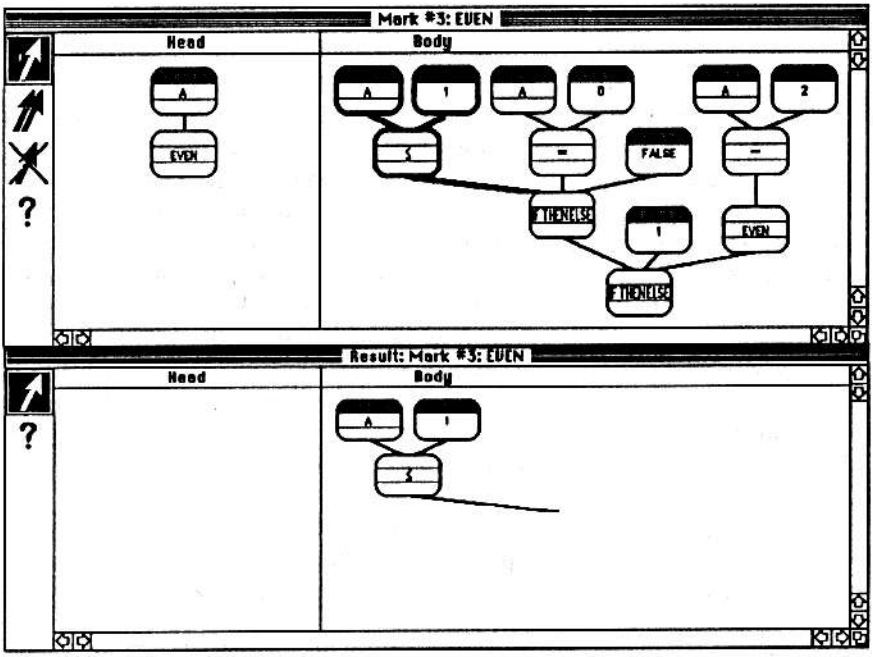


Figure 3.3. The ABSYNT interface showing another hypothesis. The system returns the hypothesis (lower half of the screen) to indicate its correctness

computation steps of the interpreter with a mouse and keyboard, with the semantic rules given as a help (Möbus & Schröder, 1990), which are visual representations of the runnable specifications.

The *help component* consists of two parts: a hypotheses environment and a set of *visual planning rules*. In the hypotheses environment, the learner may state hypotheses about the correctness of (parts of) solution proposals. Because in our system the control of feedback selection is left to the problem solver, we expect that the feedback content offered to the student will be more effective than help which is not under user control (McKendree, 1990).

Figures 3.2, 3.3, and 3.4 depict snapshots of the hypotheses environment. The learner programmed a solution proposal to the problem "even" (program testing whether a natural number is even). Then the learner stated a hypothesis by marking a part of the solution (bold parts in Figure 3.2): "The boldly marked fragment is part of a correct solution!" But the system's answer is negative (lower window of Figure 3.2). Now the learner narrows the hypothesis (Figure 3.3). The hypothesis has become smaller and more general. Correspondingly, the system's feedback space has become larger. This time the answer is positive: The selected program fragment is embeddable in a correct solution to the "even" task. This is indicated to the learner by

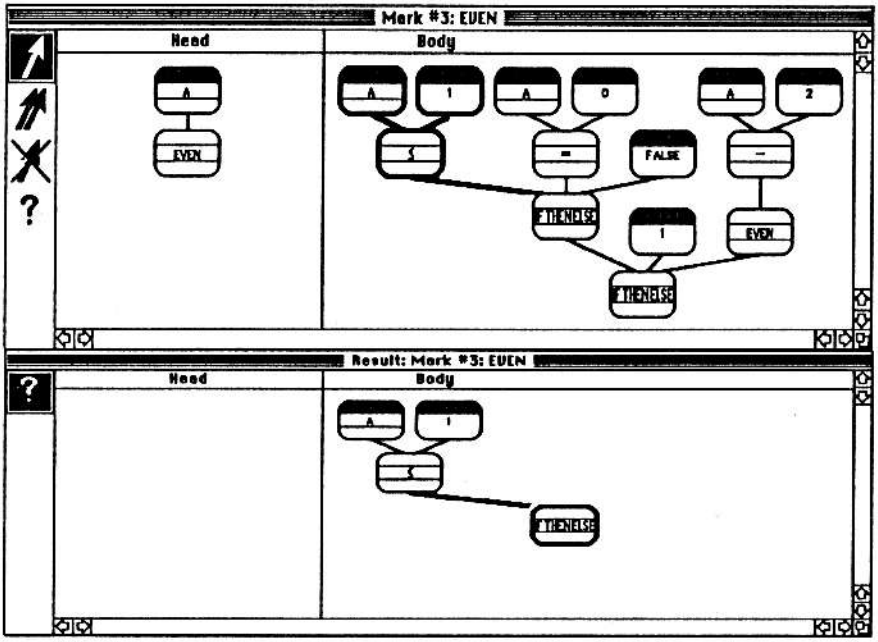


Figure 3.4. On demand (bold line, lower part of the figure) the system shows the next node of an internally generated complete solution

returning to the hypothesis in the lower window. In Figure 3.4 (lower window), the learner asks for a completion proposal of the open connection line (bold), and the system delivers an "if-then-else" node. Internally the system has generated a complete solution to the task in response to the hypothesis (see also Figure 3.5), but only one node is shown to the learner in order to give only the minimal necessary information to resolve the impasse. The feedback given by the system can be viewed as help on the language level, intended to support free, unguided problem solving (Möbus & Thole, 1990).

One reason for the *hypotheses testing approach* is that in programming a bug usually cannot be localized absolutely. There are a variety of ways to debug a wrong solution. Hypotheses testing leaves this decision to the learner and thereby provides a rich data source about the problem-solving process.

The answers to the learner's hypotheses are generated by rules defining a goals-means-relation (GMR). The GMR can be looked at as a rule-based inference system, a grammar, or an AND-OR graph with parametrized nodes. These rules may be viewed as "pure" expert knowledge not influenced by learning. (They correspond to what Corbett, Anderson, & Patterson (1988) call the "ideal student model.") Thus, we call this set of rules EXPERT in the remainder of the chapter. Currently, EXPERT contains about 1200

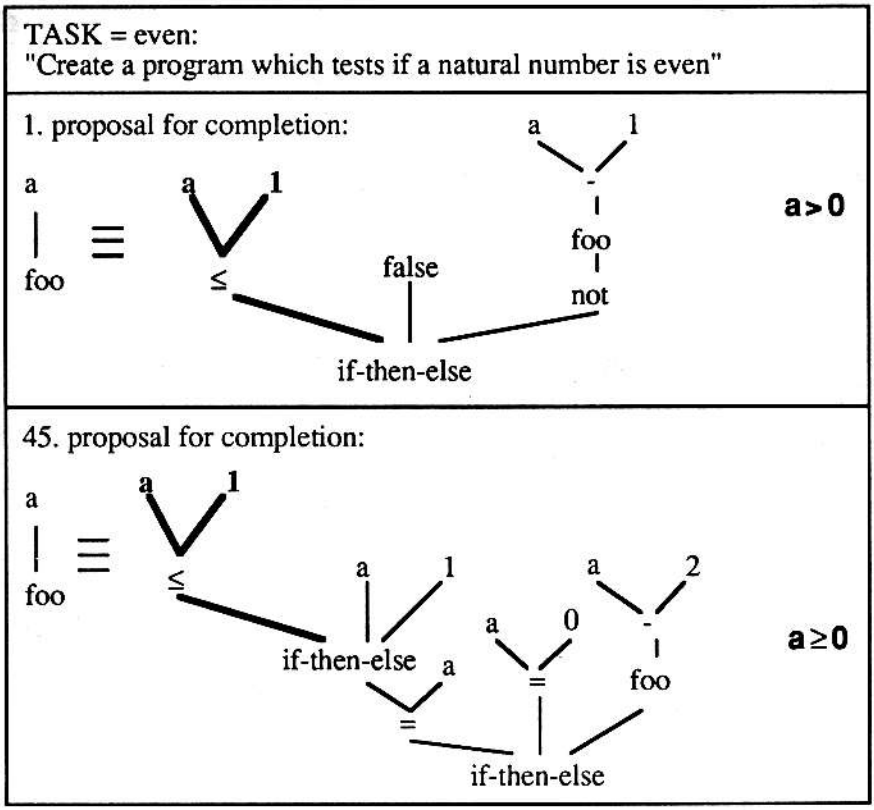


Figure 3.5. The first and the 45th completion generated by the system in response to a learner's hypothesis

rules and analyzes and synthesizes several millions of solutions for 42 tasks (Möbus, 1990, 1991; Möbus & Thole, 1990).

Because *partial* solutions can be analyzed, synthesized, and completed, hypotheses testing is possible, as illustrated earlier.

For *adaptive* help generation, the EXPERT GMR rules have to be augmented by an internal student model (see the next section). The necessity of a learner model is illustrated in Figure 3.5. It shows the first and the 45th complete solution, which can be generated by the system in response to the learner's hypothesis (bold). So there are many strikingly different completion possibilities, and it is necessary to select one. The function of the internal model is to select the completion possibility which is maximally consistent with the learner's current knowledge state. This should reduce the learner's surprise to a minimum: "Least surprise principle of help."

The GMR rules can be translated into *visual planning rules* (Möbus, 1991).

The visual planning rules are based on the visual representations of rule examples shown in the appendices (they are explained in the next section). The planning rules are the second part of the help component. In contrast to the hypotheses feedback on the *language* level, the planning rules focus also on the *goal* level. The goal names contained in the planning rules can be viewed as labels for predicative goal descriptions such as the description of the "even" predicate stated earlier. The planning rules serve to encourage help-guided planning which starts from the presentation of the task. An explorative feasibility study with the planning rules showed that they can be used by programming novices for constructing solution proposals for programming tasks, even if initially the subjects have no idea how to solve the task.

The planning rules do not give reasons for the goals and implementations contained in them. (This is another problem we work on.) So one might be inclined to think that our subjects felt like the person in the Chinese room in Searle's (1980, 1984) thought experiment. But after obtaining solutions with the help of the planning rules, the subjects tried to explain the solutions to themselves (Chi et al., 1989; VanLehn, 1991a) and reported feelings of insight in a number of cases and thus "escaped the Chinese room" (Boden, 1989).

THE INTERNAL MODEL

The Internal Model (IM) is described in three subsections. In the first subsection, the knowledge contained in the IM is described, leading to a set of empirical constraints. In the second subsection, the evolution of the IM and its motivation by the empirical constraints is described, and an example is given. In the third subsection, an empirical case study is provided.

The Rule-Schema-Case Partial Order and its Empirical Consequences

This subsection describes the domain-specific planning knowledge of ABSYNT, as contained in GMR rules. It contains visual representations of GMR rules and composites of varying specifity. Furthermore, the empirical consequences which are associated with several types of rules are pointed out. But first, the terms to be used are listed below.

All rules containing correct planning knowledge are called *GMR rules*. There are two ways to split this set GMR (see Figure 3.6):

1. *Simple rules and composites:*
   - *Simple rules*. There are three kinds of simple rules: rules describing the differentiation of programming goals (goal elaboration rules),

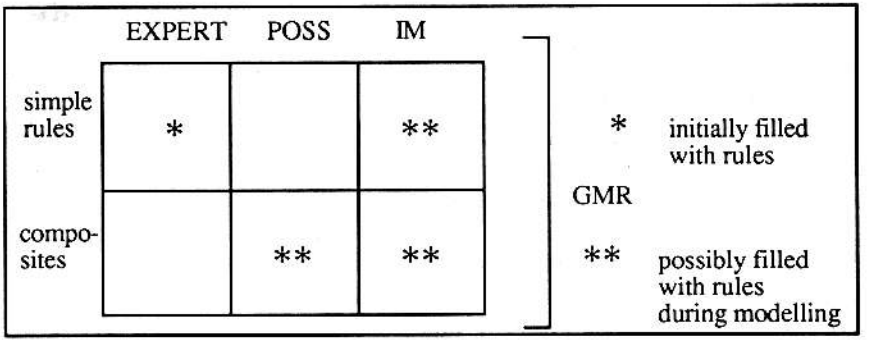


Figure 3.6. The partitions of the Goals Means Relation (GMR)

rules implementing one ABSYNT node, and rules implementing ABSYNT program heads. Generally the variables in the simple rules can be bound to *subtrees*, except for variables for parameter names, constant values, and names of higher (self-defined) operators.

- *Composites*, are rules that are created by merging at least two successive rules of a parse tree of a solution. Composites may be produced from simple rules and composites. Composites are not contained in the set of EXPERT rules. A composite that contains at least one variable which can be bound to a subtree is called a *schema*. If all variables in a composite can only be bound to node names or values, then the composite is a *case*. Finally, there are cases for whole tasks or *solution examples*.

2. *EXPERT, POSS, and IM:* The other way to split the set GMR is into the database for the rules. There are three subsets—EXPERT, POSS, and IM. EXPERT contains the expert domain knowledge described above. The sets IM and POSS are created at run-time and are described later. As the figure indicates, there are no simple rules in POSS and no composites in EXPERT.

GMR Rules. This subsection provides examples for *simple rules* that are depicted in their visual representations in Appendix A. 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 in which the goal is contained in the ellipse and the means (implementation) is contained in the rectangle. The rule body contains one or a conjunction of several goals-means-pairs, or a primitive predicate.

Rules E1 to E4 in Appendix A are *goal elaboration rules*. Goal elaboration rules are planning rules; they do not contain ABSYNT program fragments. For example, Rule E1 can be read:

(*rule head*):

- If your main goal is "absdiff" with two subgoals S1 and S2, then leave space for a program tree yet to be implemented.

(*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 which solves this new goal will also be the solution for the main goal."

Rules O1 to O4 and L1 and L2 of Appendix A are GMR rules implementing one ABSYNT node. Rules O1 to O4 implement one *operator* node. Rules L1 and L2 implement one *leaf* node (parameters and constants). O1 can be read:

(*rule head*):

- If your main goal is "branching" with three subgoals—IF, THEN, ELSE—then program an "if-then-else" node with three connections leaving from this node, and
  leave space above these connections for three program trees P1, P2, P3 yet to be implemented.

(*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.

The rules L1 and L2 do not contain goals-means-pairs in their rule bodies. Instead, the rule body of L1 requires that the program tree *X* must be a parameter. Similarly, C in L2 must be a constant. L1 can be read:

- If your main goal is to implement a parameter *X*,
  then *X* has to be a parameter name that has to be put into a parameter node,
  which has to be implemented as a part of the ABSYNT program.

Finally, Rule H1 in Appendix A shows a rule implementing the head of an ABSYNT program solving the task goal "absdiff," which means: "Create a program for a 2-arity function with a head and a yet unknown body. The body is a solution to the goal absdiff(parameter(X), parameter(Y))." This goal

means that the program has to compute the absolute difference of two numbers.

Figure 3.7 shows the solution of a subject—Subject 2—working on ABSYNT problems to the "absdiff" problem. The rules in Appendix A parse and generate this solution.

The composition of rules: Schemes and cases. In our theory, composites represent improved speed-up knowledge. Together with the simple rules they constitute a partial order from general planning rules to solution schemes to specific cases representing complete solution examples. In this section we define composition and provide and describe examples.

If we view the rules as Horn clauses (Kowalski, 1979; Robinson, 1992), then the composite of two rules can be described by the inference rule

$$\frac{(F \leftarrow P \,\&\, C) \qquad\qquad (P' \leftarrow A)}{(F \leftarrow A \,\&\, C)\sigma}$$

The two clauses above the line resolve to the resolvent below the line. A, C are conjunctions of atoms. P,P', and F are atoms. σ is the most general unifier of P and P'.

The Composites C1 to C6 in Appendix B can be created from the rules in Appendix A. As an example, it is explained how the Composite C1 results from O4 and two instances of L1.

O4: gmr(difference(S1, S2), -(P1, P2)) ←
gmr(S1, P1) & gmr(S2, P2).
L1: gmr(parameter(X),X) ← is_parameter(X).

O4 and L1 are expressed as Horn clauses. Applying the inference rule with σ = [S1 / parameter(X), P1 / X] leads to the intermediate rule instantiations O4σ and L1σ :

O4σ: gmr(difference(parameter(X), S2), -(X, P2)) ←
gmr(parameter(X), X) & gmr(S2, P2).
L1σ: gmr(parameter(X), X) ← is_parameter(X).

which resolves to:

C0: gmr(difference(parameter(X), S2), -(X, P2)) ←
is_parameter(X) & gmr(S2, P2).

This corresponds to an intermediate composite C0 (a schema) not depicted in Appendix B. Now we compose C0 with L1. In order to avoid name clashes,

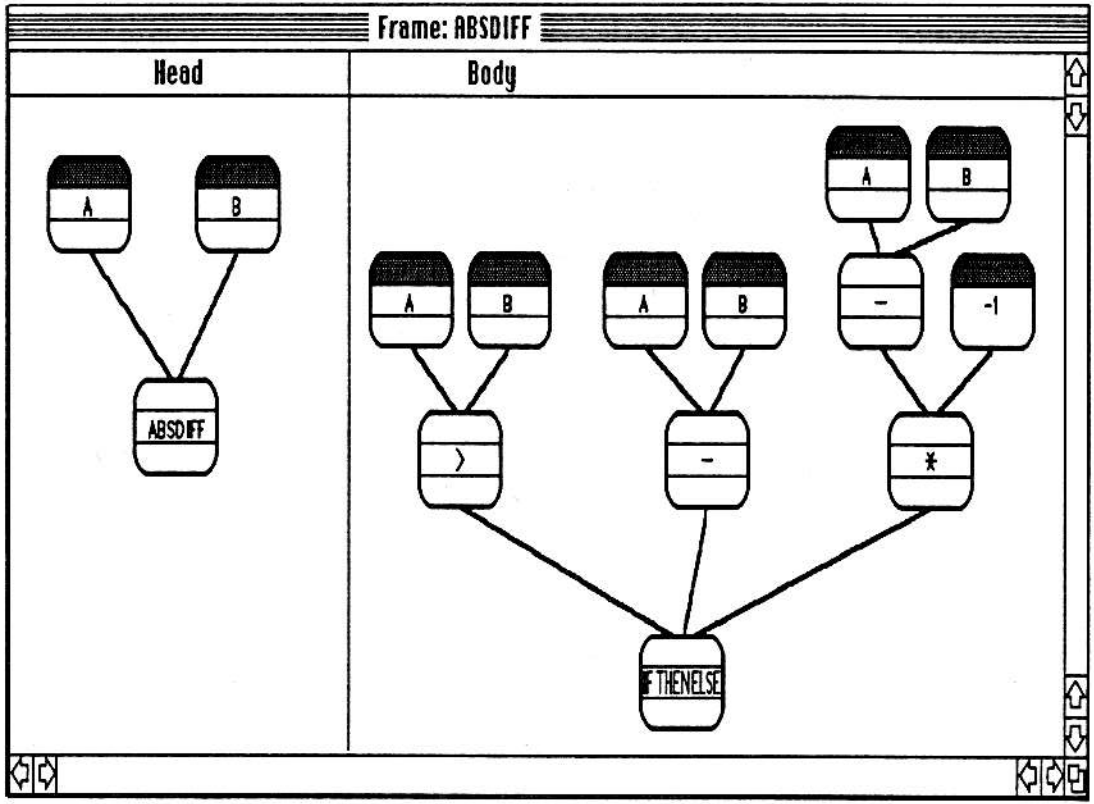


Figure 3.7. Solution created by Subject 2 to the task "program computing the absolute difference of two numbers"

we rename the variable X in L1 by Y. Applying the inference rule with σ = [S2 / parameter(Y), P2 / Y] leads to C0σ and L1σ:

C0σ: gmr(difference(parameter(X), parameter(Y)), -(X, Y)) ←
is_parameter(X) & gmr(parameter(Y), Y).
L1σ: gmr(parameter(Y), Y) ← is_parameter(Y).

which resolves to:

C1: gmr(difference(parameter(X), parameter(Y)), -(X, Y)) ←
is_parameter(X) & is_parameter(Y).

This corresponds to the Composite C1 in Appendix B. Thus, the Composite C1 states that the goal "difference of two parameter nodes" can be implemented by a tree which consists of the subtraction operator connected to two parameter nodes. C2 of Appendix B states that the same goal can be implemented by a tree in which the two parameter nodes are subtracted in reversed order, with the result multiplied by -1. Thus, C1 and C2 are *cases* for the difference of two parameters. C3, C4, C5, and C6 are composites of weakly increasing specifity. C3, C4, and C5 are *schemes* for the "absdiff" task which contain large parts of a solution in their rule heads. Finally, C6 contains a complete solution to the goal "absdiff" in its rule head. Thus, the Composite C6 contains a specific solution example (*case*) which corresponds to the solution of Subject 2 shown in Figure 3.7. Thus, it is shown that we can derive schemes and cases from GMR rules by rule composition. At the present moment it is a research question as to how to relate schemes to well-known but not precisely defined topics in problem solving, for example, "strategies," "tactics," or "styles."

The empirical constraints of rules, schemes, and cases. Simple GMR rules and composites give rise to different empirical predictions. We make use of some of them in the IM (see later). In this case we want to show which predictions are possible.

This requires hypotheses about the application of knowledge represented as GMR rules by a problem solver. According to our theory, plans are synthesized and executed. So we propose an interaction of two processes:

- A *synthesizing* process selects and instantiates GMR goal-elaboration rules. It delivers goals, subgoals, and goal-subgoal-relations for the current task: "planner."
- An *executing* process implements program fragments according to these goals, using GMR node-implementation rules. It also connects

program fragments according to the goal-subgoal-relations delivered by the synthesizing process: "coder."

This synthesize-execute-interaction enables the application of GMR rules in any order. The actual order is determined by the synthesizing process. We state the following *processing* hypotheses:

- *Implementation* of ABSYNT fragments. If a certain GMR node-implementation rule is executed, then the program fragment contained in it (the "means" of the goals-means-pair) is implemented in an uninterrupted sequence of programming actions (positioning a node, naming a constant, parameter, or higher operator node, and drawing a link).
- *Verbalization* of goals. Selecting goals and subgoals is part of the synthesizing process and involves control decisions. So the goals and subgoals of a rule may be verbalized when this rule is applied.
- *Correction* of positions. If two program fragments have to be connected according to a goal-subgoal-relation, corrective programming actions are possible—lengthening links, changing their orientation, and moving nodes.

These processing hypotheses lead to hypotheses about *performance differences* between novices and experts. Because novices and experts are assumed to differ with respect to the partial order from simple rules to schemes to cases, we compare the application of a composite to a corresponding set of simple rules and state the following *performance difference* hypotheses:

- *No-interleaving hypothesis.* Because program fragments of GMR rules are assumed to be implemented within uninterrupted sequences, actions of different rule instantiations should not interleave. So for the set of simple rules we can make across-rule-predictions of which actions should or should not occur together. For the composite, there are no such predictions because it consists of just one rule. For example, if the "absdiff" task is solved using the Rule C6 in Appendix B, then there are no constraints on the action steps.
- *Verbalization hypothesis.* Applying the composite requires less goals to be selected than applying the set of simple rules (see again C6). So there should be *less* verbalizations in the first case.
- *Rearrangement hypothesis.* When applying the set of simple rules, the corresponding program fragments have to be connected according to goal-subgoal-relations. Thus, corrective programming actions are likely. In contrast, applying the composite requires no information about goal-subgoal-relations (see again C6: Everything is contained

within one rule). So the composite should require *less* position corrections.

- *Time hypothesis.* Selecting, elaborating, and verbalizing goals, and rearranging program fragments cost time. So the application of the composite should be *faster* than the application of the corresponding set of simple rules.

These relationships are illustrated in Figure 3.8, using the composites of Appendix B as example ("•" means composition, so, for example, C4 can also be expressed as C3•C1). The rules are organized in a partial order which reflects degree of sequence constraints of programming actions, degree of verbalizations, rearrangements, and performance time.

Applying the sets of rules {C1, C2, C3}, {C2, C3•C1}, {C1, C3•C2}, and {C3•C1•C2} all lead to the same solution (Figure 3.7). But application of {C1, C2, C3} should be accompanied by more verbalizations, rearrangements, and longer performance time than the other rule sets of Figure 3.8. Furthermore, {C1, C2, C3} imposes more constraints on the sequence of programming actions. {C1, C2, C3} predicts the following sets of events (programming actions, verbalizations, and rearrangements):

- *Events corresponding to C1.* The goal to program the difference of two parameters *X*, *Y* is possibly verbalized. The subtree in the rule head of C1 is implemented. (The programming actions are: create a "-" node, two links, two parameter nodes and name the parameters.)
- *Events corresponding to C2.* The goal to program the difference of two parameters *X*, *Y* is possibly verbalized. The subtree in the rule head of C2 is implemented.

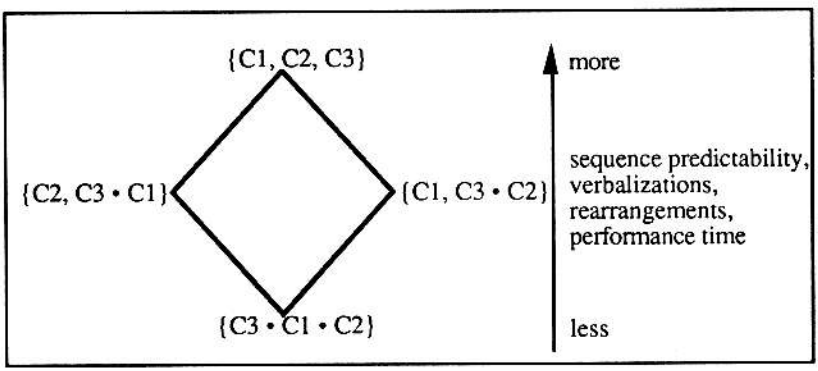


Figure 3.8. Partial order of sets of composites (see Appendix B), where C4 = C3•C1, C5 = C3•C2, C6 = C3•C1•C2

- *Events corresponding to C3.* The goal to program the task "absdiff" is possibly verbalized. The subtree in the rule head of C3 is implemented.
- *Events corresponding to goal-subgoal-information (C1-C3).* The program tree of C1 and the first open link of C3 are possibly rearranged (if not already connected), so they are connected.
- *Events corresponding to goal-subgoal-information (C2-C3).* The program tree of C2 and the second open link of C3 are possibly rearranged (if not already connected), so they are connected.

Nothing is predicted about the order of events within or across sets. But events from different sets or from different rules within one rule set are expected not to interleave.

{C3•C1•C2} = {C6} imposes no sequence constraints on programming actions and predicts only one verbalization and no rearrangements because its rule head contains the whole solution. There is only one set of events denoted by C6:

- *Events corresponding to C6.* The goal to program the task "absdiff" is possibly verbalized, and the program tree in the rule head of C6 is implemented.

The IM, to be described next, makes use of the no-interleaving hypothesis and the time hypothesis. We also describe a study of the no-interleaving hypothesis. The other hypotheses have not yet been tested empirically. This is work for the future.

Evolution of the IM in the Problem-Solving Process

The IM was developed according to the following principles, which are motivated by the constraints of IDL-SDL theory and by the empirical constraints stated earlier.

- IDL-SDL theory distinguishes between *acquired* and *improved* knowledge. The IM contains simple rules representing acquired but not yet improved knowledge and composites representing various degrees of expertise.
- The IM contains only rules whose program trees were executed in an uninterrupted sequence (*no-interleaving hypothesis*).
- Knowledge improvement should result in speeding up performance, as stated earlier (*time hypothesis*). Thus, a composite becomes part of the IM only if the student shows a speed-up from an earlier to a later

action sequence where both sequences can be produced by the composite or the corresponding set of simple rules.

- A credit value of the rules in the IM rewards the usefulness of a rule: The usefulness of a rule is the product of the length of the action sequence (number of programming actions) explained by the rule and the number of its successful applications (i.e., its actions are performed in an uninterrupted sequence).
- IDL-SDL theory does not permit improvement of knowledge not yet acquired and applied. So a rule newly acquired by IDL (a simple rule) cannot be improved before being applied at least once. This means that the IM should not be extended at the same time by a new simple rule and by composites built from this rule. Therefore, possible composites have to wait for incorporation in the IM. They are kept in a set POSS of possible composite candidates for the IM.

In some rare circumstances the learner's data pattern can be simultaneously explained by a simple rule not yet in the IM and a composite in POSS which is built from it. For practical reasons we currently permit that both rules enter the IM in such situations, although this does not strictly correspond to the theory. These situations would not occur if each simple rule would explain only one single programming action. This means to decompose the current "simple rules" into even more elementary rules—another aspect of future work.

Given a certain sequence of programming actions, we describe four subsets of rules (simple rules and composites):

1. Rules which do not contain any program fragments ("goal elaboration rules"). Currently the IM does not contain these rules, so they are *nondecisive* with respect to the action sequence. (We work on incorporating planning into the IM. Furthermore, fragments of verbalizations can be related to the goal elaboration rules; Möbus & Thole, 1990.)
2. Rules whose heads contain a program fragment that is part of the final result produced by the action sequence, and that was programmed in a *noninterrupted*, temporally continuous subsequence. Thus, these rules fit into the no-interleaving hypothesis. They are *plausible* with respect to the action sequence.
3. Rules whose heads also contain a program fragment that is part of the final result produced by the action sequence, but this fragment corresponds only to the result of a noncontinuous subsequence of actions *interrupted* by other action steps. These rules are *implausible* with respect to the action sequence. Examples for plausible and implausible rules are given later in this section.

4. Rules whose heads contain a program fragment that is *not part* of the final result produced by the action sequence. These rules are *irrelevant* to the action sequence.

Figure 3.9 shows the development of the IM during the knowledge acquisition process. Figure 3.10 shows the changes between the different sets of rules during this process. We will now go through Figures 3.9 and 3.10.

Start Top of Figure 3.9). The first programming task is presented. Both sets IM and POSS are empty.

Now the learner solves the first task. An action sequence is produced leading to a solution.

First Test Phase. Since IM and POSS are empty, nothing happens.

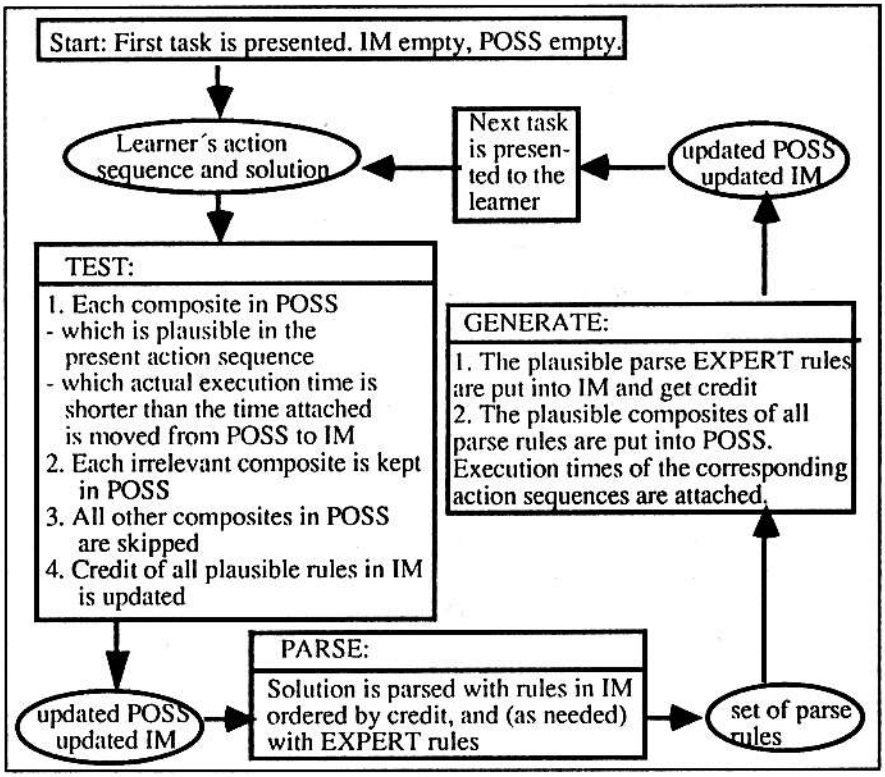


Figure 3.9. Development of the IM during the knowledge acquisition process

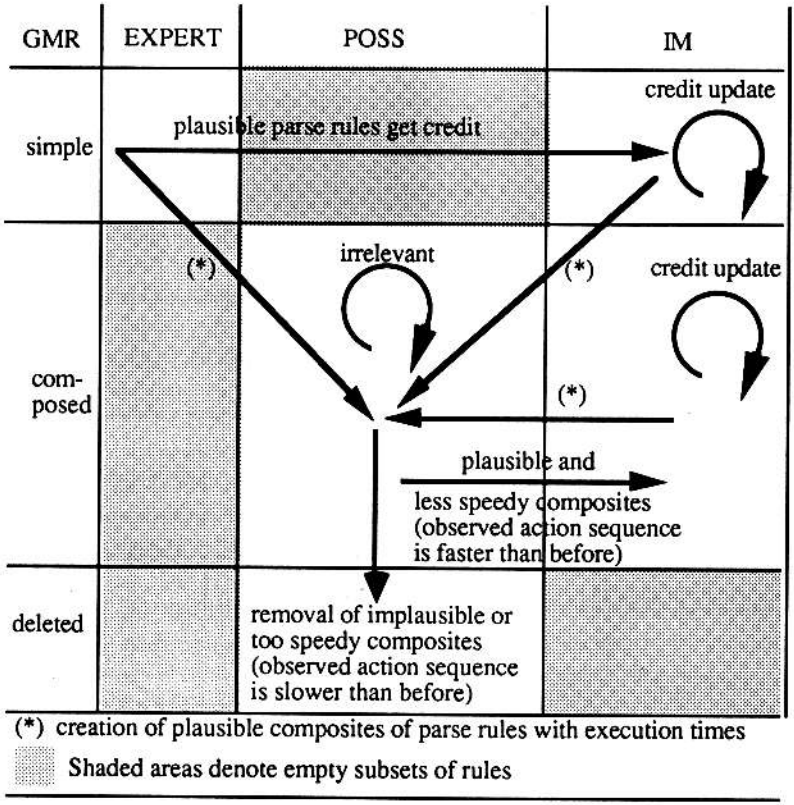


Figure 3.10. Changes between the rule sets during development of the IM

First Parse Phase. The learner's solution to the first programming task is parsed with the EXPERT rules, leading to a set of parse rules. The purpose of parsing is to check the correctness of the solution, to obtain candidates for the IM, and to be able to generate candidate composites for POSS.

First Generate Phase. The EXPERT rules just used for parsing are compared to the action sequence which produced the learner's solution and which is saved in a log file. The plausible EXPERT parse rules are put into the IM (see also Figure 3.10—arrow pointing from simple EXPERT rules to simple IM rules) and get credit (number of action steps described by each rule). These rules are hypothesized as *newly acquired* by the learner in response to impasses on solving the first task. In the next step of the generate phase, the composites of *all* rules just used for parsing the solution are built. Each composite is compared against the action sequence that produced the solution.

The *plausible composites* are kept in the set POSS because they are possible candidates of improved knowledge useful for future tasks (see also Figure 3.10—arrow pointing from simple EXPERT rules to composites in POSS). For each plausible composite, the time needed by the learner to perform the corresponding uninterrupted action sequence is attached.

Next, the learner solves the second task.

Second Test Phase. When the learner has finished the second task, each composite in POSS is checked to see if it is *plausible* with respect to the action sequence, and if the time needed by the learner to perform the respective continuous action sequence is *shorter* than the time attached to the composite. The composites meeting these two requirements are put into the IM. (In Figure 3.10, this is represented by the arrow pointing from composites in POSS to composites in IM.) Composites in POSS which are *irrelevant* to the action sequence of the solution just created are left in POSS. They might prove as useful composites on future tasks. (In Figure 3.10 its the arrow pointing from composites in POSS back to composites in POSS.) All *other* composites violate the two requirements. They are skipped (that is, composites implausible to the actual sequence, or composites that predict a more speedy action sequence than observed. This means that the learner has performed the action sequence more slowly than the sequence that led to the creation of the composite). In Figure 3.10, skipping these composites is represented by the arrow pointing to deleted rules. Finally, the credits of all rules in the IM which are plausible with respect to the present action sequence are updated. In Figure 3.10, this is represented by the two circular arrows within simple IM rules and within IM composites.

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

Second Generate Phase. Basically the same steps are performed as in the first generate phase. The EXPERT rules just used for parsing are compared to the action sequence of the task solution. The *plausible* EXPERT rules are again put into the IM and get credit (Figure 3.10—arrow from EXPERT to simple IM rules). As before, they are hypothesized as newly acquired knowledge in response to impasses on the task just performed. Furthermore, the composites of the parse rules are created. (In Figure 3.10, this is represented by the three arrows labeled with "(*)", since the parse rules may contain simple EXPERT rules, simple IM rules, and IM composites.) The *plausible composites* are put into POSS, where they will be tested on the next test phase. Again the time needed for the corresponding action sequence is stored with each composite.

When solving the next task, the next test phase will again split the

composites in POSS into rules for the IM, rules to keep in POSS, and rules to skip. The circle repeats with each new task solved.

An example. We illustrate a short episode of the IM with data from Subject 2. Figure 3.11 shows her solution to the problem: "addaddone"—"Create an ABSYNT program which adds two natural numbers. The addition operator can only be used for incrementing by 1."

For this example, we restrict our attention to the GMR Rules O1, O5, L1, L2, and C7 (see Appendices A and C). Subject 2 already worked on other tasks before solving the "addaddone" task. At this point, C7 has not been created. O1, L1, and L2 are already in the IM from earlier tasks. O5 is not yet in the IM, but in the set of EXPERT rules.

After the "addaddone" task is solved, the test phase starts (see Figure 3.9). Because the only composite we look at here (C7) has not been created yet, we can skip the first three subphases of the test phase and go to the fourth subphase: The credit of all plausible rules in the IM has to be updated. Figure 3.12 shows a fragment of the action sequence performed by Subject 2 on the "addaddone" task. O1, L1, and L2 are in the IM (see earlier). O1 is *implausible* 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 3.12). 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 at 11:15:08 and 11:15:29. Thus, this sequence is interrupted at 11:15:16 and 11:15:22. Actions corresponding to L2 are performed at 11:15:16 and 11:15:34. Thus, this sequence is *interrupted* at 11:15:22 and 11:15:29. So because O1, L1, and L2 are implausible, their credits are not changed.

Now the solution in Figure 3.11 is parsed with rules in the IM and, as needed, with additional EXPERT rules (Figure 3.9). O1, O5, L1, and L2 are among the parse rules, because no other rules have a higher credit and are able to parse the solution.

After the parse phase, the generate phase (Figure 3.9) 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 (Appendix C) 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 3.12)—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 the information "135 seconds" attached to it.

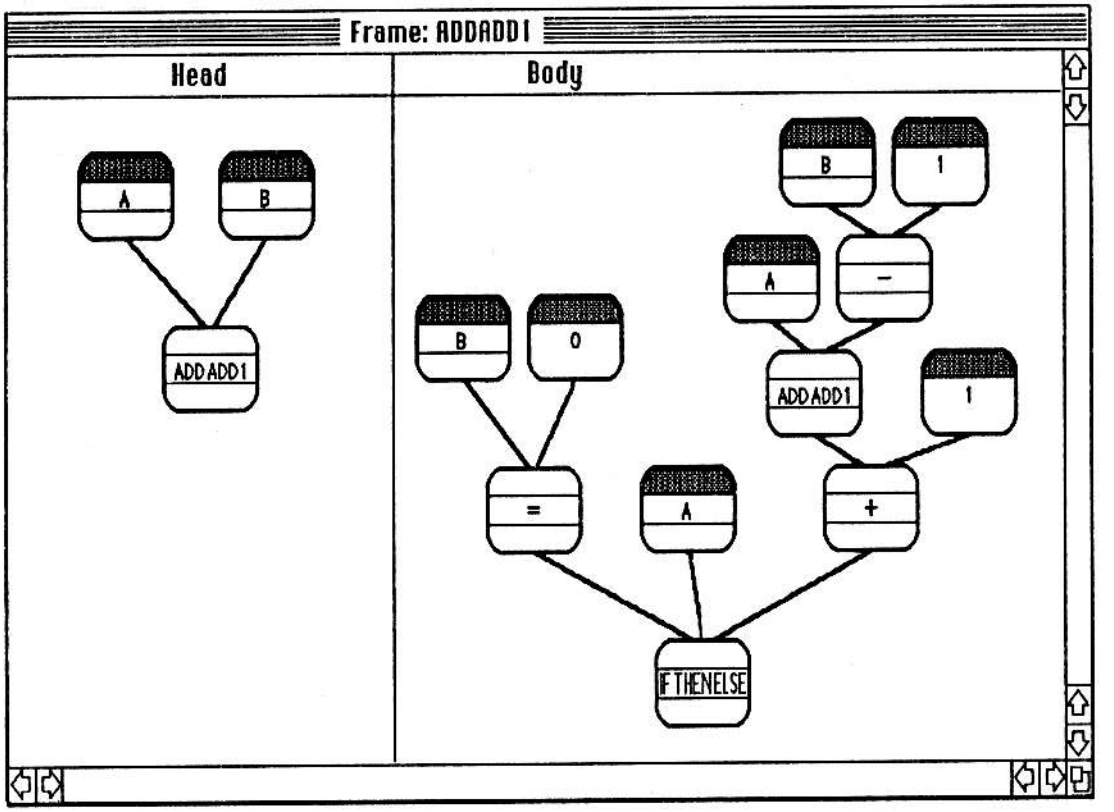


Figure 3.11. Solution of Subject 2 to the problem: "addaddone": expressing addition by "+1"

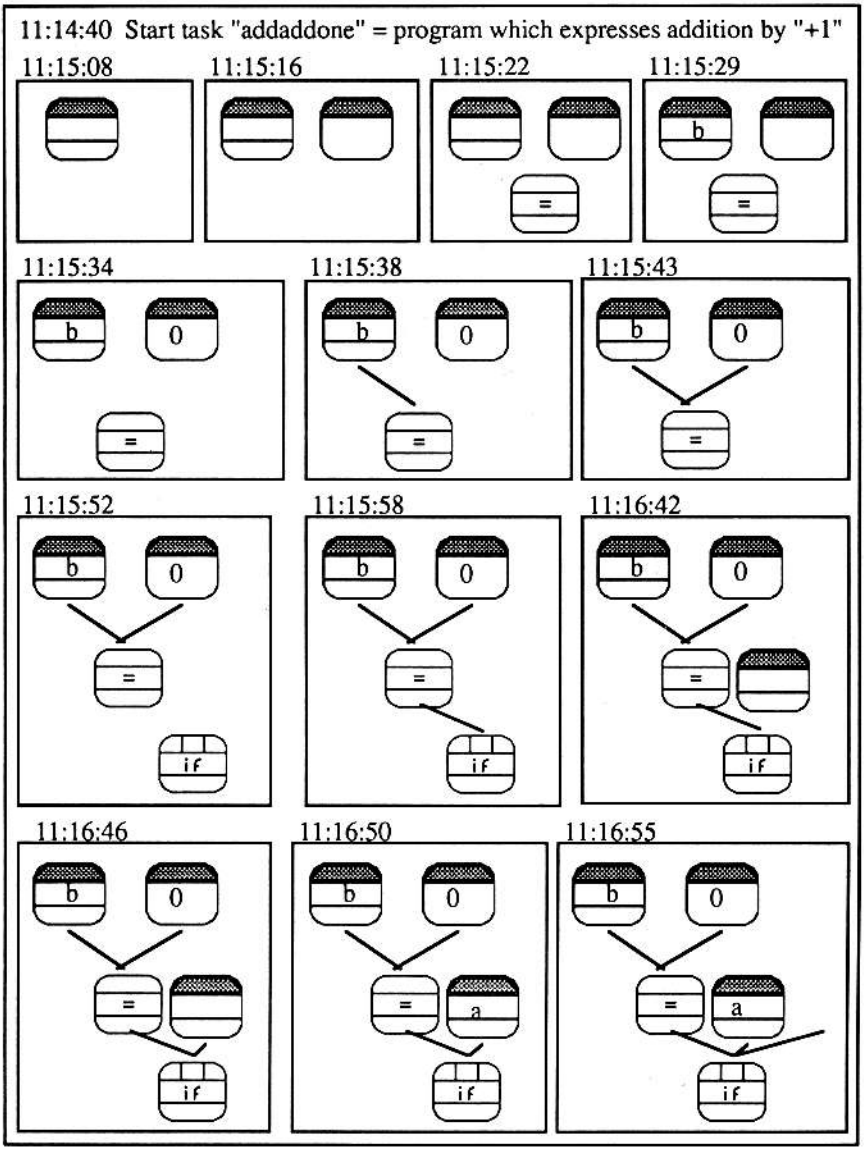


Figure 3.12. Part of the sequence of Subject 2s actions (October 23, 1989) leading to the solution in Figure 3.11

The following task was "diffdiffone"—"Create an ABSYNT program which subtracts two natural numbers. The subtraction operator can only be used for decrementing by one." Subject 2 created the solution shown in Figure 3.13, with an action subsequence shown in Figure 3.14.

The test phase following reveals that C7 is plausible again. The corresponding action sequence was performed in 92 sec (starting from the beginning of the task at 11:10:56, to 11:12:28). This is less than the 135 sec attached to C7. So C7 is moved into the IM and gets a credit of 13 because it describes 13 programming steps (see Figures 3.12 and 3.14). This credit will be incremented by 13 each time the composite is plausible again. O5 is also plausible (actions at 11:11:43, 11:11:48, and 11:11:53). So it enters the IM with a credit of 3.

An Empirical Case Study

In a case study the sequence predictions of the IM were investigated, as implied by the no-interleaving hypothesis. Before solving the programming tasks "addaddone" and "diffdiffone," Subject 2 solved seven tasks not involving abstraction or recursion. Her action sequences and solutions to these tasks were saved in a log file. The IM was run on the action sequences, leading to seven subsequent states of the IM.

Based on the state of the IM immediately before the i^{th} task, and the solution proposal of Subject 2 to the i^{th} task, uninterrupted sets of programming actions to the i^{th} solution were predicted. This *model trace* was compared to the actual *subject trace*: Subject 2's action sequence to the i^{th} task.

For each pair of adjacent programming actions in the subject trace which was expected to be adjacent according to the model trace (i.e., explained by the same IM rule instantiation), a "+" was noted. For each pair of *non*adjacent programming actions in the subject trace which was expected to be adjacent according to the model trace, a "−" was noted. Thus, "+" denote pairs fitting the predictions, and "−" denote contradictions.

Figure 3.15 illustrates this procedure for one of the seven tasks solved by Subject 2. Figure 3.15 shows:

a. a subset of the rules in the IM after solving the third task, "absdiff" (program computing the absolute difference of two numbers)
b. S2's solution to the fourth task, "quot" (program dividing two numbers such that the result is greater or equal than 1)
c. the predicted action sequence for this solution of "quot" (model trace)
d. the observed action sequence of S2 (subject trace) marked with "+" and "−" as described.

In Figure 3.15a, for each IM rule the program fragment in its head is shown in addition to the rule name. In Figure 3.15c, the program fragments

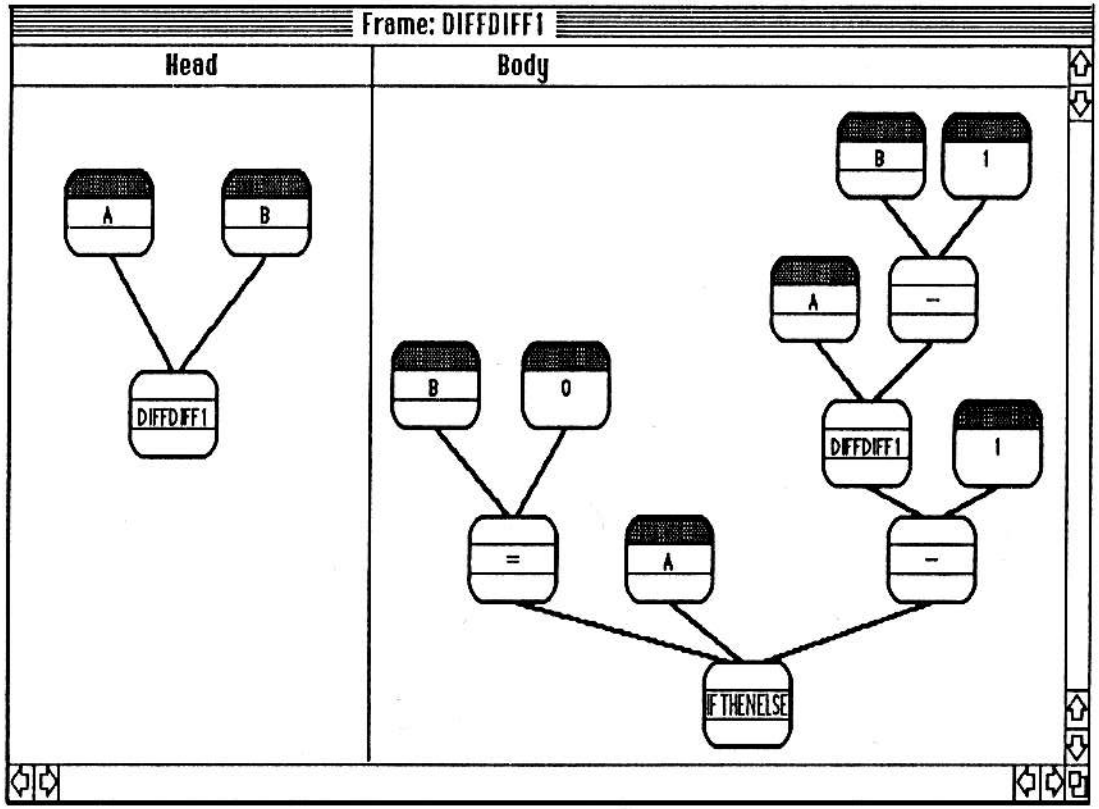


Figure 3.13. Solution of Subject 2 to the problem: "diffdiffone": Expressing subtraction by " – 1"

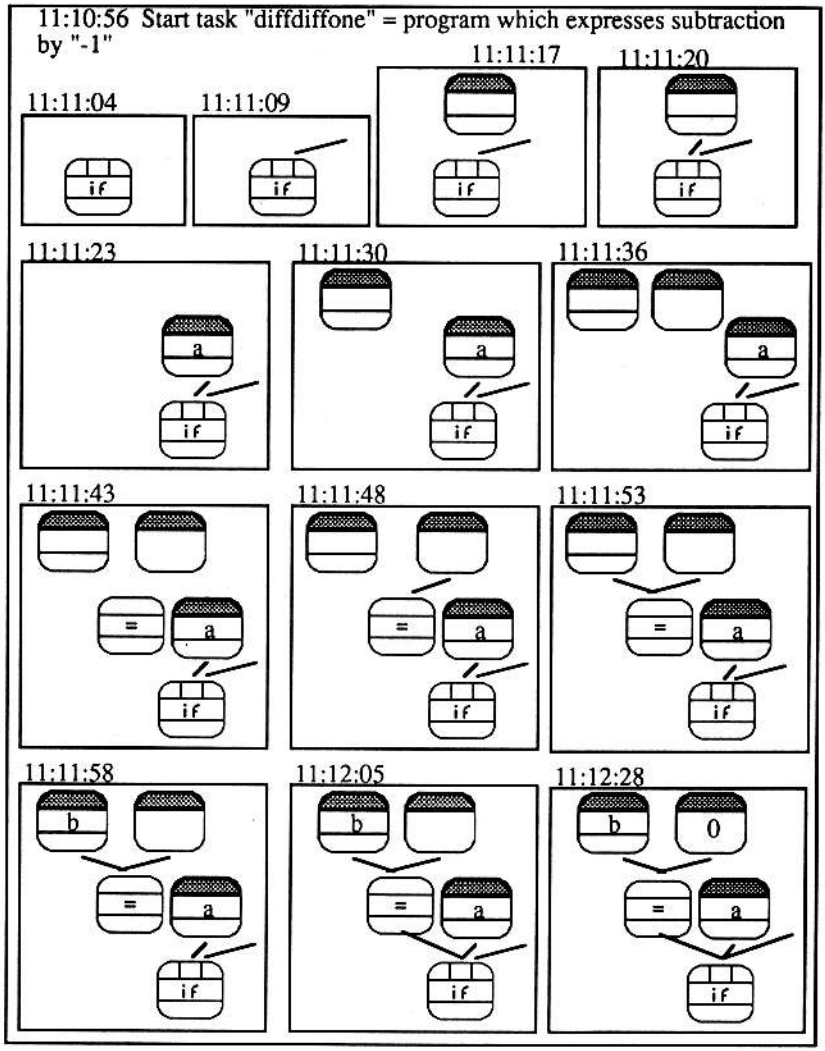


Figure 3.14. Part of the sequence of Subject 2s actions (October 26, 1989) performed on the task "diffdiffone"

predicted to be constructed in an uninterrupted sequence are boxed. Thus, for example, the if-then-else-node rule (Figure 3.15a) 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 3.15c. Similarly, placing a parameter node and

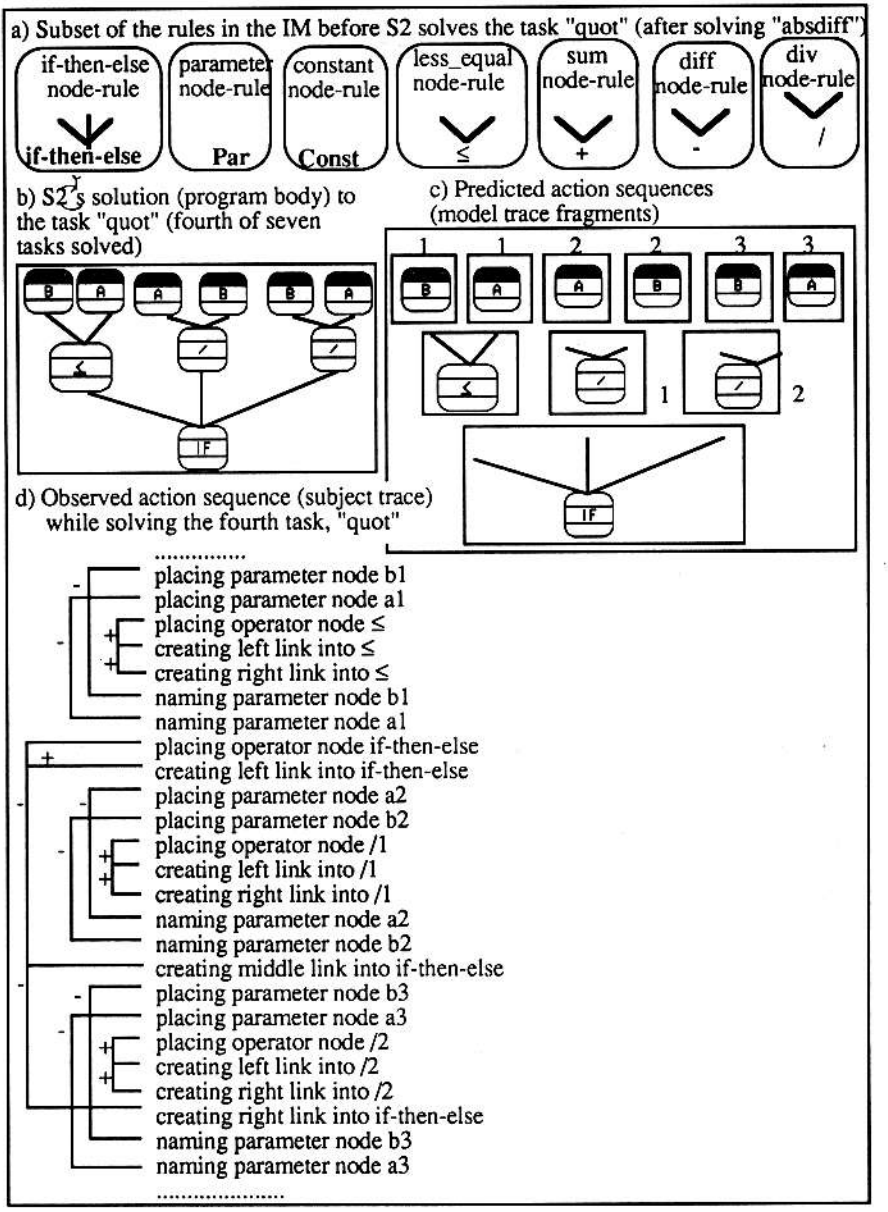


Figure 3.15. (a) subset of the IM before solving "quot", (b) S2s solution to "quot", (c) predicted action sequence fragments, (d) observed action sequence corresponding (+) or contradicting (−) to the predictions

Table 3.1. Fit of the IM Conditional to Various Rule Types (for all seven tasks).

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

naming this node are two actions predicted to occur subsequently. Different instances of nodes are indexed in Figure 3.15c and 3.15d.

This procedure led to 80 "+" and 49 "−" for the seven subject traces to the seven tasks. (As Figure 3.15 shows, the subject trace to the "quot" task led to 7 "+" and 8 "−", so the fit was bad in this task.) Because more "+" should lead to longer and thus fewer runs than about equally many "+" and "−", we applied the Runs-test. There were 40 runs ($p < 0.001$).

This result indicates that the IM adequately describes a considerable portion of the protocol of action sequences of Subject S2. Table 3.1 shows the distribution of "+" and "−" across different types of rules in the IM.

Thus, the parameter node rule, for example, is responsible for 2 of 80 "+", and for 27 of 49 "−": S2 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 improve the IM by splitting the parameter node rule (and the constant node rule) into two new rules: One for positioning and one for naming a node. Then the current parameter node rule would be a *composite* of these two new rules. As noted earlier, in general this means a decomposition of simple rules into more elementary rules, with each one explaining only one programming action.

FURTHER WORK

The IM is currently extended to the following directions:

- *Incorporating a planning level.* Currently the learner's hypothetical solution plan is the parse tree of the solution. We extended the ABSYNT language by goal nodes so that mixed programs containing ABSYNT operator nodes and goal nodes are possible. The learner is able to test hypotheses and to receive error and completion feedback at this planning level, even if the learner has no idea about the implementation in ABSYNT. We will extend the IM by goal nodes accordingly.

From the learner's point of view, the benefit of using goal nodes is that hypotheses testing is possible at the *planning stage*. From a psychological point of view, objective data about the planning process can be obtained, and the gap to the IDL-SDL theory is narrowed. Finally, from a help system design point of view, the benefit is that in addition to hypotheses testing it is possible to offer planning rules as help to the learner.

- *Designing help.* The ultimate goal of the IM is to deliver adaptive help. If the learner asks for a hypothesis completion, the IM rule with the *highest credit* which is able to complete the hypothesis will be used. Depending on the actual state of the IM, the completion may consist of a single node or of a subtree. When the IM is extended by a planning level, a completion proposal may also consist of a goal node (*plan completion*) or even a goal tree. Only if the IM does not contain any rule able to complete the current hypothesis, an EXPERT rule has to be used for completion.

Besides completion proposals, there are more possibilities to deliver help. For example, adaptive planning rules could be offered to the learner. Furthermore, it is possible to empirically test the effects of different kinds of help (i.e., "adaptive" vs. "nonadaptive"). For example, a completion proposal could be generated by a chain of simple rules or by a single composite. Secondly, the completion proposal might exactly cover the diagnosed knowledge gap or provide more or less information as needed to cover the gap. Several hypotheses about the effects of help variation along these dimensions are possible.

- *Malrules.* Malrules can automatically be created and integrated into the IM. If the problem solver's solution proposal cannot be recognized by the system, and an oracle (e.g. the teacher, another system, etc.) confirms that the solution is wrong, then it is trivial to generate a general malrule on the spot. It relates the task to the whole solution. If the problem solver tests hypotheses, then the system should be able to generate more specific malrules. The goals corresponding to the hypothesis and to the rest of the solution proposal can be identified using the ordinary EXPERT rules. Then a new malrule relating these goals can be created. This *mal goal elaboration rule* is able not only to reproduce the faulty student proposal, but also similar slips.
- *Generalizations.* Empirical studies with ABSYNT indicated use of previous solutions and positive transfer especially for recursive tasks. Thus, composites in the IM should be generalized at the nonleaf nodes of their goal trees and program trees. Generalization of composites may be viewed as another way of knowledge optimization

(Anderson, 1983, Wolff, 1987) in response to the successful utilization of knowledge (Figure 3.1).

- *Generate-and-test-method.* One feature of the IM which arose for practical reasons is its generate-and-test style. Composites are generated and retained only if they survive the test phase. The obvious alternative is to generate composites only as needed.
- *Supporting the IM by an EM* (External Model). The EM is designed to provide hypothetical reasons for the knowledge shifts in the IM. It is an extension of an earlier model of help-based knowledge acquisition based on IDL-SDL Theory which we developed for the related domain of the semantic knowledge for ABSYNT (Schröder, 1990). A trace of the model was compared step by step to the action and verbalization protocol of a single subject, coded into 13 categories. Sixty percent of the protocol was correctly described by the model.

The EM (Schröder & Möbus, 1992) is based on the analysis of the protocol of actions and verbalizations of another subject working on our sequence of ABSYNT programming tasks. The model proceeds as follows:

- *Task comprehension:* A programming task is presented to the model as a text graph of the task description. The model converts the text graph into propositions representing the learner's initial task representation.
- *Synthesizing phase:* The concepts mentioned in this representation (similar to Anderson, 1989) are activated. The concepts contain general preknowledge and possibly ABSYNT-specific knowledge. The general preknowledge consists of an input–output relation that defines the concept. The ABSYNT-specific knowledge describes how the concept is elaborated or implemented as an ABSYNT program fragment. The knowledge stored in the concepts is used for constructing a *situation model* (van Dijk & Kintsch, 1983) representing the actual solution plan for the current task.
- *Execution phase:* The situation model is translated into ABSYNT code.
- *Evaluation phase:* The code is checked. If no bugs are detected, then the knowledge stored in the concepts actually used is improved by composition (*SDL*). Thus, new concepts for composites are created.

Impasses may arise at several points in this process. For example, a certain activated concept might not yet contain ABSYNT-specific knowledge. In this case, the subject chooses a heuristic, for example, to create a subtask from the definition of the concept. If the subtask is solved, then the program fragment created is inserted into the concept slot representing ABSYNT-specific knowledge. Thus, new ABSYNT-specific knowledge is acquired (*IDL*).

EM and IM are closely related. The concepts of the EM correspond to the rules of the IM. The situation model or solution plan of the EM corresponds to the instantiated parse rules in the IM. EM and IM are intended to mutually constrain each other. For example, the EM explains acquisition and improvement of knowledge by certain impasses, heuristics, and successful solutions. The IM should hypothesize the same knowledge as the EM. On the other hand, the IM predicts impasses, verbalizations, sequence constraints on action steps, and speedups. The EM should be consistent with these predictions.

CONCLUSIONS

We introduced a *learner model (IM)* which is intended to describe the change of knowledge of the learner by moving from novice to expert as a gradual shift of acquiring new knowledge, schemata, and finally specific solutions. The model processes online the recognizable data of the action stream of the learner. It is based on theoretical constraints imposed by IDL-SDL, action phases, and case basedness. Some summarizing remarks are made concerning the relevance of the IM for help, its consistency with the theoretical constraints, and its testability.

- *Modeling knowledge on a rule-schema-case partial order.* In our system the often quoted dichotomy between rule-based vs. case-based knowledge representation (Slade, 1991) is resolved in favor of a dynamic rule-based system that models knowledge on a partial order of fine-grained rules, schemes, and cases (composites with variables only for node names and values). So there is only *one* principle needed to model the generation of schemes and cases. Case-based or analogical reasoning is both plausible for psychological reasons (e.g., Anderson, Farrell, & Sauers, 1984; Escott & McCalla, 1988; Weber, 1989; for the domain of functional programming) and profitable for efficiency of the diagnosis.
- *Relevance for hypotheses about the knowledge acquisition process.* The rules in the IM can be directly converted into domain specific information for the learner. Thus, the learner will receive:
  a. *adaptive hypothesis completion proposals.* These proposals will be generated by the IM. Thus, they will correspond to the actual knowledge state of the learner as hypothesized by the IM.
  b. *adaptive planning help.* The planning help supplied to the learner will be continuously updated in accordance with the IM.
- *Consistency with IDL-SDL theory.* According to IDL-SDL theory, *new*

knowledge is *acquired* in response to *impasses*, and *existing* knowledge is *improved* by successful *practice*.

EXPERT rules not in the IM are transferred to the IM if they are: (a) used for parsing, and (b) plausible. They represent hypothetical *newly acquired* knowledge.

Composites are transferred to the IM if: (a) they are plausible, and (b) the corresponding actions need less time than the corresponding chain of simple rules. The composites represent hypothetical *improved* knowledge.

- *Preference for rules explaining long action sequences.* Simple rules and composites explain increasingly large program fragments in a partial order up to specific solution examples (*cases*). In the credit scheme, composites are preferred because they explain larger program fragments than simple rules. The credit scheme also prefers rules often applied successfully. Thus, moderately specific composites should get a high priority. Furthermore, composites impose less constraints on the sequence of action steps than the corresponding chain of simple rules. Thus, composites are more likely to be plausible.
- *Empirical testability.* The IM represents the hypothetical knowledge states of the learner. Simple rules and composites provide an adequate description of the learner's knowledge, only if they correspond to the *sequence of action steps* performed by the learner. Hypotheses concerning sequence constraints, verbalizations, rearrangements, and time were stated already. There is also another possibility for the IM to fail: It may happen that learners' action sequences cannot be described by the IM at all because: (a) the corresponding simple rules are implausible, and (b) the corresponding composites drop out because there is no performance speedup.

Furthermore, IDL says that new knowledge is acquired in response to impasses. So if EXPERT rules are taken into the IM, then impasses should have occurred just before the corresponding action steps were performed. Thus, we expect that the learner makes use of help at exactly these points in the problem-solving process. The EM should of course make the same predictions.

Finally, we plan to let the learner state his expectations before receiving a completion proposal to a hypothesis. The learner then predicts how the system will complete his hypothesis. We would expect that the learner's prediction corresponds to the right side of the head of the rule used by the system for hypothesis completion. Furthermore, if this rule is in the IM, then the learner should be more confident in his prediction than if the rule is an EXPERT rule

not yet in the IM. In this latter case the learner lacks domain knowledge and thus is more inclined to guess.

- Last but not least we think that we made a contribution to a *theory of predictability of behavior.* According to our theory the novice-expert shift can be modeled by acquiring and composing GMR rules. So knowledge can be represented by rules, schemes, and cases. As we tried to show, this is accompanied by the impossibility to predict action-sequences which are denoted by one schema or one case. The empirical consequence of that prediction is: Fixed to the same granularity of the behavior protocol the sequence of actions of the expert is less predictable than the sequence of actions of the novice. Thus, we expect an upper limit of predictability of action sequences when becoming an expert.

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. II, pp. 289–310). Los Altos, CA: Morgan Kaufman.

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

Anderson, J. R. (1990). *Cognitive psychology and its implications* (3rd ed.). New York: Freeman.

Anderson, J. R., Farrell, R., & Sauers, R. (1984). Learning to program in LISP. *Cognitive Science, 8,* 87–129.

Bartlett, F. (1932). *Remembering.* Cambridge: Cambridge University Press.

Boden, M. A. (1989). Escaping from the Chinese room. In M. A. Boden (Ed), *Artificial intelligence in psychology* (pp. 82–100). Cambridge, MA: MIT Press.

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.

Chi, M. T. H., Bassok, M., Lewis, M., Reimann, P., & Glaser, R. (1989). Self-explanations: How students study and use examples in learning to solve problems. *Cognitive Science, 13,* 145–182.

Chi, M. T. H., Feltovich, P. J., & Glaser, R. (1981). Categorization and representation of physics problems by experts and novices. *Cognitive Science, 5,* 121–152.

Corbett, A. T., Anderson, J. R., & Patterson, E.J. (1988). Problem compilation and tutoring flexibility in the LISP tutor. *Proceedings Intelligent Tutoring Systems ITS'88* (pp. 423–429). Montreal.

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.

Escott, J. A., & McCalla, G. I. (1988). Problem solving by analogy: A source of errors in novice LISP programming. *ITS-88 Proceedings* (pp. 312–319). Montreal.

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

Glinert, E. P. (1990). Nontextual programming environments. In S. K. Chang (Ed.), *Principles of visual programming systems* (pp. 144–230). Englewood Cliffs, NJ: Prentice Hall.

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). New York: Guilford Press.

Gollwitzer, P. M., Heckhausen, H., & Steller, B. (1990). Deliberative and implemental mind sets: Cognitive tuning toward congruous thoughts and information. *Journal of Personality and Social Psychology, 59*(6), 1119–1127.

Janke, G., & Kohnert, K. (1989). Interface design of a visual programming language: Evaluating runnable specifications. In F. Klix, N. A. Streitz, Y. Waern, & N. Wandke (Eds.), *MACINTER II Man-Computer Interaction Research* (pp. 567–581). Amsterdam: North-Holland.

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

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

Laird, J., Newell, A., & Rosenbloom, P. (1986). *Universal subgoaling and chunking*. Boston: Kluwer.

Laird, J., Rosenbloom, P., & 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, MA: MIT Press.

McKendree, J. (1990). Effective feedback content for tutoring complex skills. *Human-Computer Interaction, 5*, 381–413.

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 Prague* (LNAI Vol. 451, pp. 138–145). Springer.

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 & O. Setsuko (Eds.), *Advanced research on computers in education* (ARCE '90 pp. 137–144). Amsterdam: North-Holland.

Möbus, C., Pitschke, K., & Schröder, O. (1992). Towards the theory-guided design of help systems for programming and modeling tasks. In C. Frasson, G. Gauthier, & G.I. McCalla (Eds.), *Intelligent tutoring systems. Proceedings of the 2nd International Conference on ITS '92* (LNCS 608, pp. 294–301). Berlin: Springer.

Möbus, C., & Schröder, O. (1989). Knowledge specification and instruction for a visual computer language. In F. Klix, N. Streitz, Y. Waern, & N. Wandke (Eds.), *MACINTER II man-computer interaction research* (pp. 535–565). 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. 7th Interdisciplinary Workshop in Informatics and Psychology* (LNCS 439, pp. 47–81). Berlin: Springer.

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

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 (Eds.), *Computer assisted learning. Proceedings of the 3rd International Conference on Computer-Assisted Learning ICCAL 90* (LNCS 438, pp. 36–49). Berlin: Springer.

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

Reisig, W. (1985). *Petri nets—An introduction. Springer EATCS monographs on theoretical computer science.* New York: Springer.

Robinson, J. A. (1992). Logic and logic programming. *Communications of the ACM, 35,* 40–65.

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., & Newell, A. (1986). The chunking of goal hierarchies: A generalized model of practice. In R. S. Michalski, J. G. Carbonell, & T. M. Mitchell (Eds.), *Machine learning* (Vol. II, pp. 247–288). Los Altos, CA: Morgan Kaufman.

Rosenbloom, P., & 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 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 International Conference on Computer-Assisted Learning ICCAL 90* (pp. LNCS 438, pp. 142–157). Berlin: Springer.

Schröder, O., & Möbus, C. (1992). Zur Modellierung des hilfegeleiteten Wissenserwerbs beim Problemlösen. In K. Reiss, M. Reiss, & H. Spandl (Eds.), *Maschinelles lernen—Modellierung von Lernen mit Maschinen* (pp. 23–62). Berlin: Springer.

Searle, J. (1980). Minds, brains and programs. *Behavioral and Brain Sciences, 3,* 417–457.

Searle, J. (1984). *Minds, brains, and science.* Cambridge, MA: Harvard University Press. (in German: Suhrkamp, 1986)

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 modeling* (Tech. Rep. AI-59). Lancaster, England: Department of Computing, Lancaster University.

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.

Slade, S. (1991). Case-based reasoning: A research paradigm. *AI Magazine, 12*(1), 42–55.

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.

Soloway, E. (1986). Learning to Program = Learning to Construct Mechanisms and Explanations. *Communications of the ACM, 29*(9), 850–858.

van Dijk, T. A., & Kintsch, W. (1983). *Strategies of discourse comprehension.* New York: Academic Press.

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.

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

VanLehn, K. (1991a). Two pseudo-students: Applications of machine learning to formative evaluation. In R. Lewis & O. Setsuko (Eds.), *Advanced research on computers in education (ARCE '90).* Amsterdam: North-Holland.

VanLehn, K. (1991b). Rule acquisition events in the discovery of problem solving strategies. *Cognitive Science, 13,* 1–47.

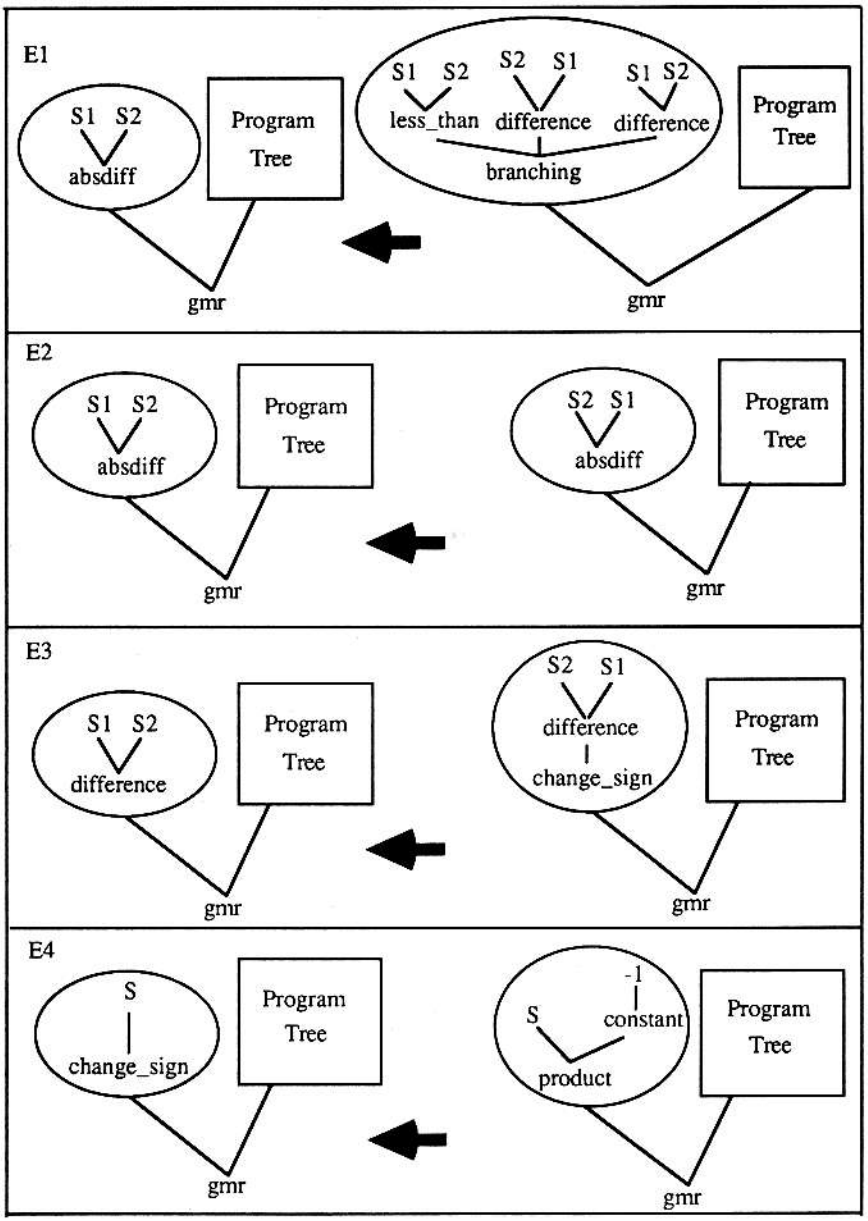
Vere, S. A. (1977). Relational production systems. *Artificial Intelligence, 8,* 47–68.

Weber, G. (1989). Automatische kognitive Diagnose in einem Programmier-Tutor. In D. Metzing (Ed.), *Künstliche Intelligenz GWAI 89* (pp. 331–336). Berlin: Springer.

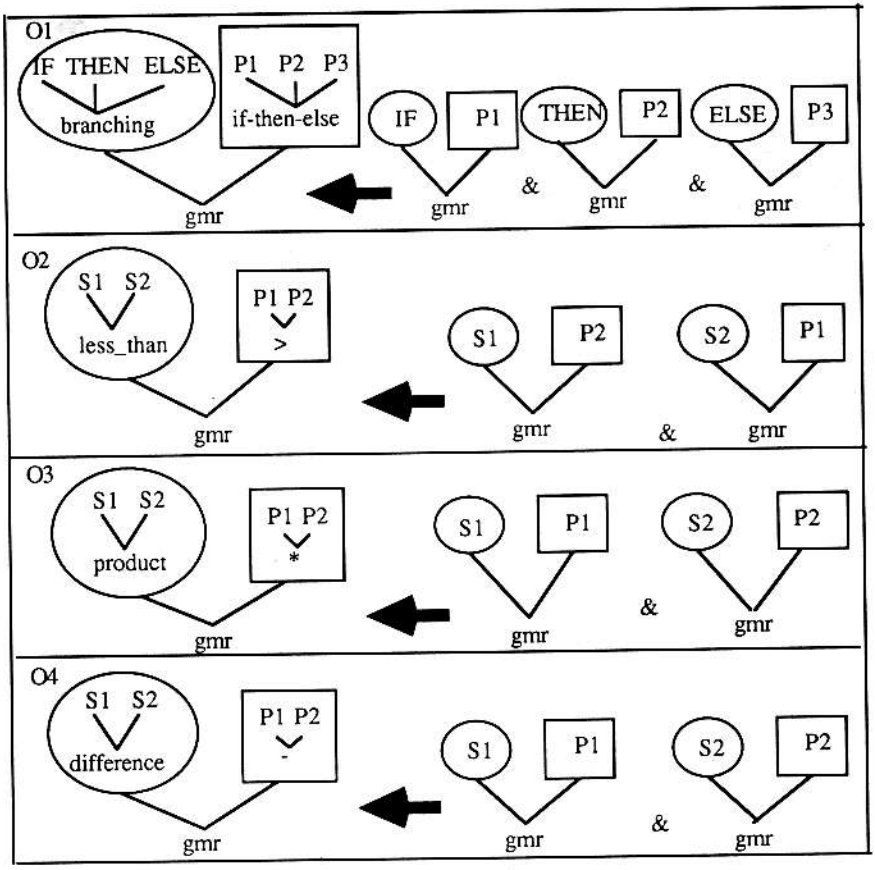
Wenger, E. (1987). *Artificial intelligence and tutoring systems.* Los Altos, CA: Morgan Kaufman.

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

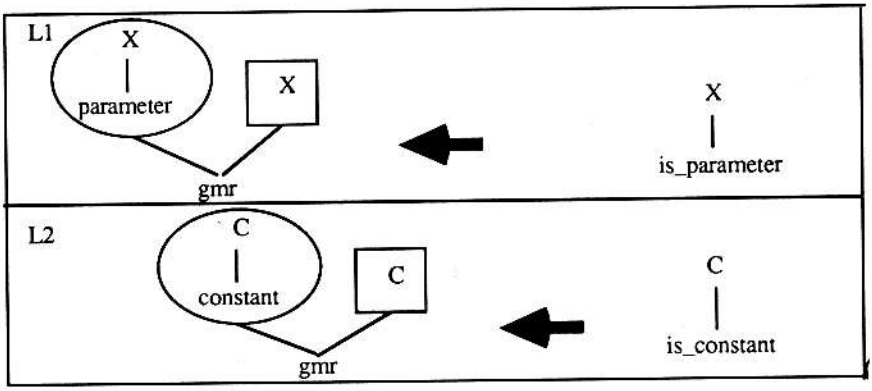
APPENDIX A: SIMPLE GMR RULES



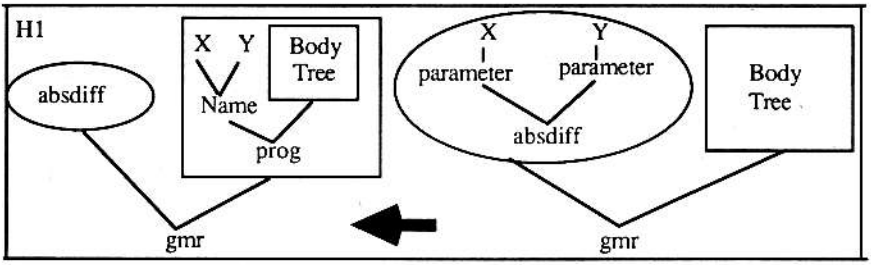
E1 to E4: Goal elaboration rules



O1 to O4: Rules implementing one operator node

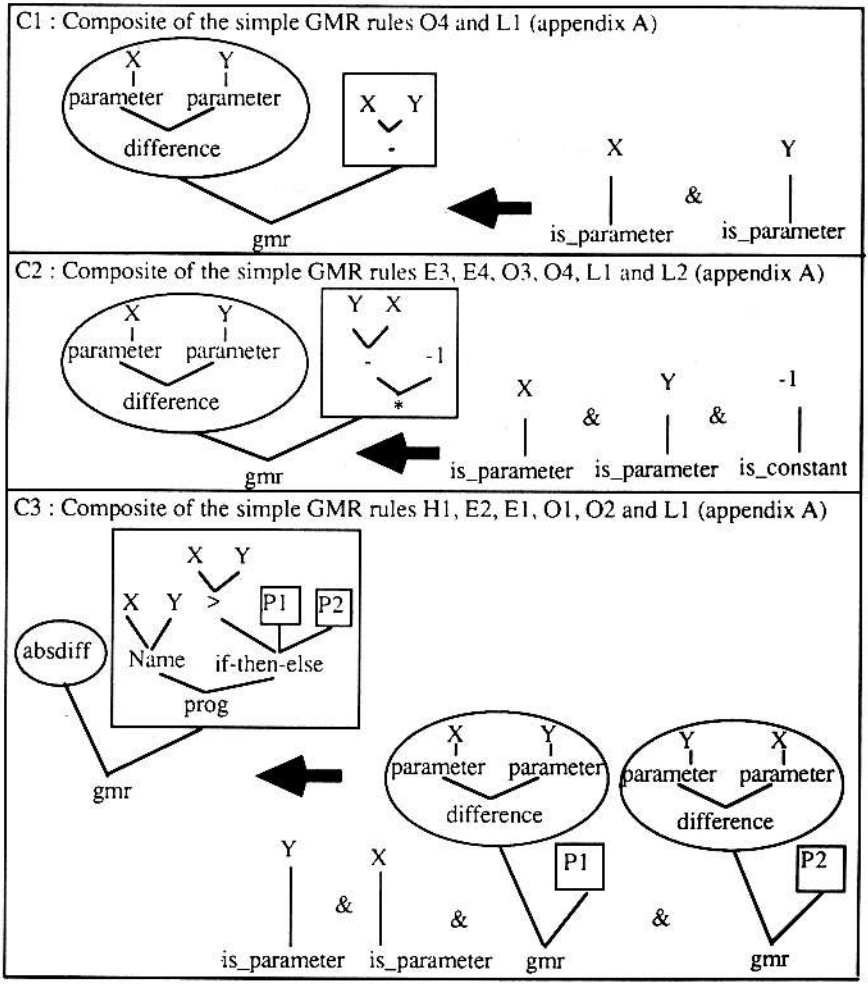


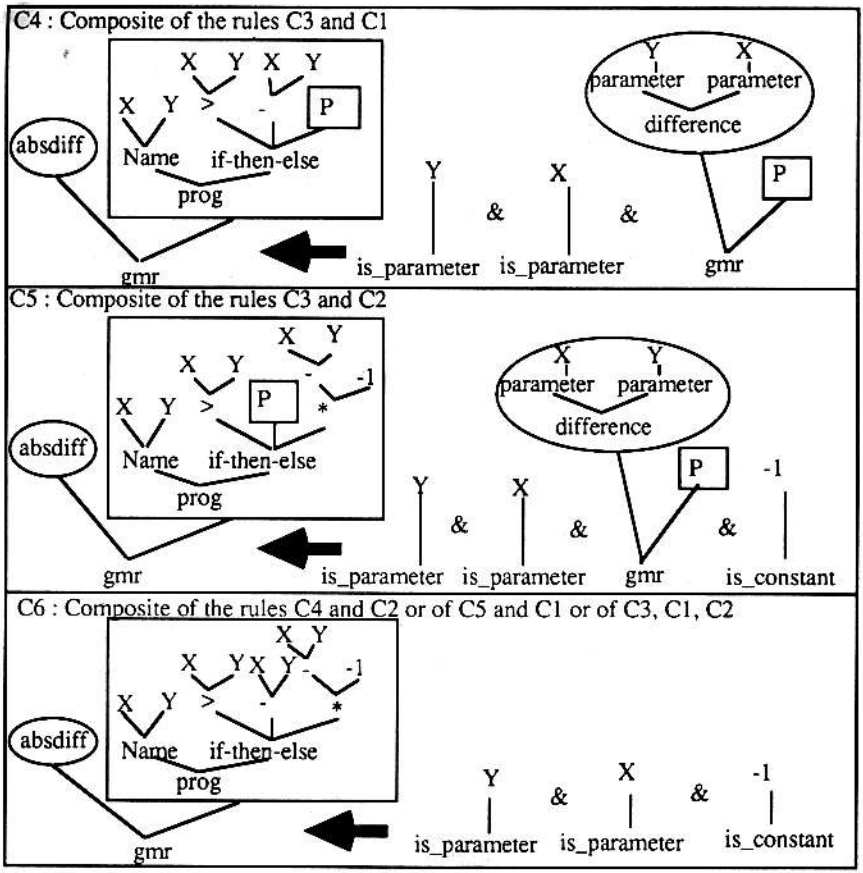
L1 and L2: Rules implementing a leaf node (parameter, constant)



H1: Rule implementing a program head

APPENDIX B: COMPOSITES




C4 : Composite of the rules C3 and C1
X Y X Y
X Y > - P
absdiff
Name if-then-else
prog
gmr
Y X
parameter parameter
difference
P
Y & X & P
is_parameter is_parameter gmr
C5 : Composite of the rules C3 and C2
X Y
X Y - -1
X Y > P *
absdiff
Name if-then-else
prog
gmr
X Y
parameter parameter
difference
P -1
Y & X & P & -1
is_parameter is_parameter gmr is_constant
C6 : Composite of the rules C4 and C2 or of C5 and C1 or of C3, C1, C2
X Y
X Y X Y - -1
X Y > - *
absdiff
Name if-then-else
prog
gmr
Y & X & -1
is_parameter is_parameter is_constant

APPENDIX C: MORE GMR RULES