The Cognitive Psychology of Knowledge
G. Strube and K.F. Wender (Editors)

CHAPTER 10

LEARNING PROGRAM ABSTRACTIONS: MODEL AND EMPIRICAL VALIDATION[1]

Franz Schmalhofer, Ralph Bergmann, Stefan Boschert, and Jörg Thoben

German Research Center for Artificial Intelligence
Kaiserslautern, Germany

ABSTRACT

This paper proposes a construction-integration process that yields a coarse-grained representation from a concrete computer program in terms of a learner's situational knowledge (i.e a procedure schema). A procedure schema is a cognitive structure that is important for the comprehension, design, implementation, and maintenance of computer programs. A formal analysis yielded state and sequence abstraction mappings as a sufficient condition for the formation of such a schema. A simulation furthermore specified the required prior domain knowledge. Four think-aloud studies revealed the knowledge, which different learners actually used and showed how construction and integration episodes were interleaved. While learners, who elaborated programs in general terms, were found to form abstractions with deductive justifications, other learners induced and tested hypotheses from sample program inputs.

Computer programs, such as an operating system written in the programming language C, an artificial intelligence system written in LISP, or even a Pascal program written by a computer science student, are rather complex patterns. Such programs must first be designed according to some specifications before they can be implemented on a computer. Later, they need to be maintained

[1] The research reported herein was financially supported by Grant No. Schm 648/1 of the Deutsche Forschungsgemeinschaft to Franz Schmalhofer.

and updated according to new requirements. Programmers, therefore, need to be able to analyze programs which were written by other programmers.

Various descriptions are known in the field of computer science which facilitate the design, implementation, and maintenance of computer programs (Yourdon, 1989). Such descriptions, which are often expressed in some abstract language, are used for the construction and comprehension of computer programs. One description can be obtained in terms of program processes that transform the input data into the outputs of the program (data flow). Another representation, the control flow, is structured in terms of the sequence in which program actions will occur. Whereas in the data flow, the links between program actions represent the passage of data, in the control flow the links represent the passage of execution control. A program can also be abstractly described in terms of what the program is supposed to accomplish (i.e., production of a certain output), thus a decomposition according to the major program functions is achieved (Adelson, 1984). Pennington (1987) furthermore suggested an abstraction where the different actions of a program are individually represented by their preconditions and their consequences.

Cognitive psychology research has investigated how humans design and implement new computer programs (Jeffries, Turner, Polson, & Atwood, 1981) as well as how they comprehend and debug existing ones. Detienne and Soloway (1990) have identified different strategies that subjects may apply in comprehension. In plan-based understanding it is assumed that the knowledge of the programmer already includes an abstract plan which is suited for comprehending the specific computer program. Through activation and instantiation processes (Schank & Abelson, 1977) some cues of the program evoke the respective plan or schema and there is a subsequent match between the evoked plan and the information extracted from the program code. Since these program readers can match their internal representations of the plans more or less directly with the program, they do not need to explicitly (re)construct the causal relationships among the data and processing steps of the program. Without an abstract plan, a programmer may construct a coherent representation of how the different processing steps and respective changes in the data occur over time during the execution of the program. Such program traces can be formed from the concrete input data as well as from symbol input specifications. These are called actual and symbolic traces, respectively. A formal description of these plan- and program-based comprehension processes has been presented by Bergmann, Boschert, and

Schmalhofer (1992). Similar to Adelson (1984), they proposed plan-based comprehension processes for advanced learners, and program-based comprehension processes for beginners.

In accordance with the two described comprehension strategies, Pennington (1987) has distinguished the knowledge representations which result from comprehending computer programs with the described processes. Plan knowledge represents, at an abstract level, the programmer's understanding that patterns of program instructions "go together" to accomplish certain goals (Soloway, Ehrlich, & Black, 1983). A second knowledge structure represents the sequence of processing steps as they occur over time during the execution of the program. In an attempt to embed her empirical findings and theorizing in a well-established cognitive framework, Pennington speculated that a programmer's plan knowledge is similar to situation models, which have been established in the text comprehension literature (van Dijk & Kintsch, 1983). A situation model reflects the structures and possible transitions in a domain at a relatively general level.

In our previous work, where we developed the KIWi cognitive model (Schmalhofer, Boschert, & Kühn, 1990), we have investigated the formation of situation models for simple programming constructs. In the KIWi-model a situation model may be acquired from a text or from a sequence of examples. Depending on the specific learning material peripheral representations are additionally constructed. When learning from a text, the text base (a propositional structure) is supposedly constructed before the situation model may be formed. The text base is a veridical representation of the meaning of a text and its structure (Kintsch & van Dijk, 1978; Miller & Kintsch,1980). When learning from examples, a template (Anderson, Farrell, & Sauers, 1984) which represents the structural relations among the surface features of examples is first constructed (Schmalhofer & Glavanov, 1986; Schmalhofer & Boschert, 1991).

Similar to Pennington's findings about two different cognitive representations of computer programs (abstract plan knowledge and representation of sequence of processing steps), the KIWi-model distinguishes between situational representations and the more peripheral representations which are formed on route to constructing a situation model. Since the KIWi-model described the learning of individual programming constructs rather than acquiring knowledge about or from complete programs, the situation model consisted of abstract representations of individual programming constructs and the more peripheral representation described the surface structure of the code

for the individual steps that are available in the programming language. Pennington's experimental findings fit very well with the central assumptions of the original KIWi-model. In order to apply this theoretical work for investigating the acquisition of knowledge from complete programs, the KIWi-model must, however, be extended. With the KIWi-model we can thus elaborate Pennington's assumption that the programmer's plan knowledge is represented in the form of a situation model.

KIWi-model

The extended KIWi-model has the same basic structure as the initial model, and again describes the knowledge acquisition processes from different materials (text and instructions on the one hand, and examples and computer programs on the other). The main difference concerns the assumption that domain and task knowledge should be distinguished as two separate but interrelated components of the conceptual models (Breuker & Wielinga, 1989).

Previous research has already addressed a number of issues concerning domain and task knowledge. It is generally believed that domain knowledge, which is often referred to as the user's mental model of a system, is important for successfully interacting with the system, e.g., explaining and predicting its behavior (Norman, 1983; Schmalhofer & Kühn, 1991). Kieras and Bovair (1984) have shown, in psychological experiments, that a mental model, or domain knowledge, can also play an important role in learning to operate a device. Domain knowledge has also been found to be differentially useful for various goal-oriented procedures (Halasz & Moran, 1983).

The initial KIWi-model addressed only the acquisition of domain knowledge (e.g., the learning of programming constructs of the LISP-system). Therefore, task knowledge did not need to be modelled as part of the conceptual models. Programs describe procedures which are to be executed by a computer. In order to model the acquisition of knowledge from these programs, task knowledge must be represented as part of the conceptual models. We will first describe the basic assumptions of the extended model and then discuss the acquisition of domain and task knowledge.

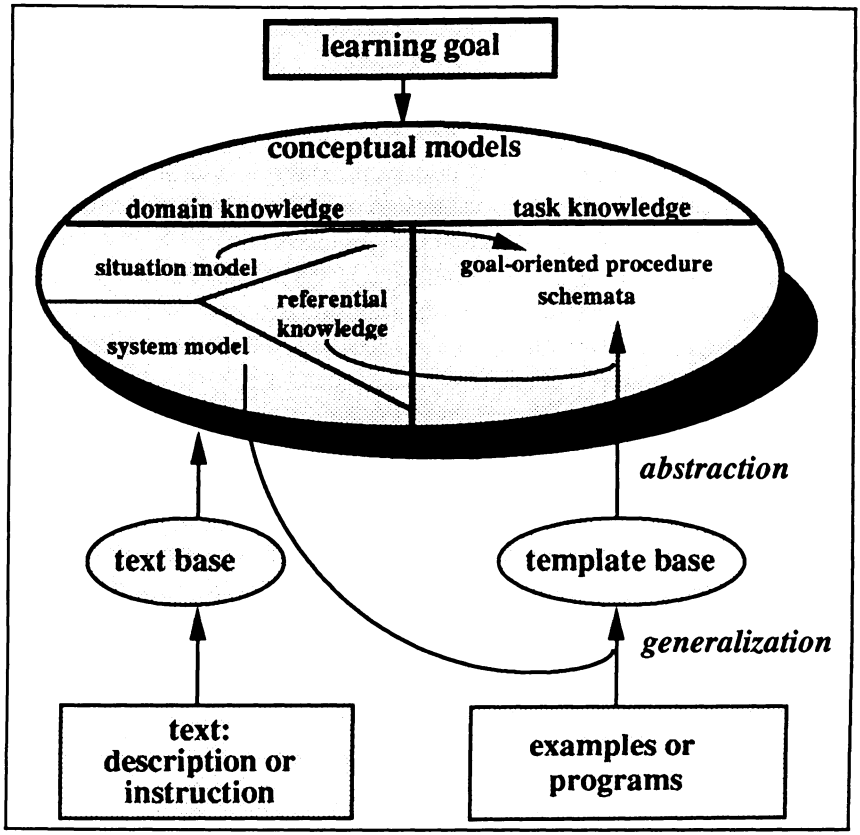

Figure 1: KIWi-model.

Basic Assumptions

The cognitive model for the acquisition of domain and task knowledge from different types of learning materials is presented in Figure 1. It shows the different knowledge structures and the transfer processes between them. The model can be described by the following basic assumptions:

Multiple Knowledge Representations: Three different knowledge representations for computer programs are assumed: the template base and the text base, which are material-related representations, and conceptual models, which are independent of any specific learning materials. The template base consists of patterns (templates) which represent the structural

relations among the surface features of examples or programs (Anderson, Farrell, & Sauers, 1984). The text base, a propositional structure, represents the structure and the meaning of a descriptive or instructive text (Kintsch, 1974). The conceptual models consist of domain and task knowledge.

Construction and Integration: Information processing is assumed to be organized in cycles. In each cycle a construction process, which results in an initial incoherent structure (comprising template base, textbase, and conceptual models), is followed by an integration process, which focuses the initial structure to a coherent whole (Kintsch, 1988). The structure is modeled as an associative network. The construction process is modeled as a production system, where sloppy but robust inference rules construct the different representations and relate them to each other. Robust rules are used because they are powerful enough to generate the appropriate elements and flexible enough to operate in many different contexts. The construction process involves: 1) forming peripheral representations (text base or template base) closely related to the learning material, 2) elaborating these representations by associations from the long-term memory, 3) generating more strategic inferences (McKoon & Ratcliff, 1992) which build parts of the conceptual models, and 4) assigning connection strengths to all pairs of representational elements that have been created in the construction process. As sloppy rules are used, the construction process results in an initial, elaborated, incoherent, and possibly contradictory structure. An integration process is therefore used to shape the initial structure into a coherent form via relaxation procedures in the connectionist manner (Rumelhart & McClelland, 1986). In this process activation is spread around the constructed elements until the system stabilizes. Depending upon the strength of their connections the elements will be activated differentially. As contradictory elements will be negatively connected and relevant items tend to be strongly interrelated (derived from the same phrase in the learning material or related semantically or experientially in the long-term memory), contradictions will be resolved and the relevant elements will be highly activated. The highly activated elements are then selected and carried over in the short-term buffer for the next processing cycle.

Learning from Text and Examples: Conceptual models (domain or task knowledge) can be acquired from a text or from a sequence of examples or programs. Thus, depending on the specific learning material, peripheral representations are first constructed: the text base, when learning from text, or the template base, when learning from examples or programs.

Informational Equivalence of Learning Materials: Further processing of the text base or the template base may yield conceptual models. When a conceptual model that was formed from a text can also be induced from a sequence of examples or a program (with the same prior knowledge), the text and the examples are said to be informationally equivalent.

Learning Goal and Prior Knowledge: Which knowledge structure a learner constructs in a learning situation depends upon his specific learning goal. The learning goal is determined by those tasks which a learner expects to be confronted with in the future. In other words, the general goal of learning is to construct knowledge so that various tasks can be solved relatively easy. Depending on the learning goal, knowledge which is only implicitly contained in the different knowledge representations must be made explicit. Learning itself is a knowledge-based process. The learner's prior knowledge together with his learning goal thus determines the knowledge acquired from the learning materials.

Duality of Domain and Task Knowledge in the Conceptual Models: Domain and task knowledge are two different interrelated types of knowledge in the coceptual models. Domain knowledge comprises: 1) a system model about a programming environment (e.g. LISP, Pascal), 2) a situation model about an application domain (e.g. mathematics, scheduling), and 3) the references between the system model and the situation model. System knowledge can be about different systems, like a LISP-system, which was studied in our previous work, or some technical device like a pocket-calculator. System knowledge about LISP may consist of knowledge about data representations in LISP, the evaluation of terms in LISP, and interactions with the LISP-system. The situation model may be about a mathematical characterization of programming constructs in terms of functions (Schmalhofer & Kühn, 1991) or about the arrangement of rooms in terms of a mental map (Morrow, Bower, & Greenspan, 1989). Task knowledge describes classes of problems which can be solved by applying a more or less specific procedure (Schmalhofer & Thoben, 1992). Such tasks can be defined by an initial state and a final state. The corresponding procedure consist of a sequence of operations which transform the initial state into the final state. Task knowledge consists of procedure schemata which specify the application conditions (input-output relation, argument specification) and the respective sequence of operations.

Acquisition of Domain Knowledge

Domain knowledge can be acquired from a descriptive text or from a sequence of examples. While the subjects are studying a text, the text base is formeded as a peripheral knowledge structure. The general knowledge from this text base is added to the domain knowledge, which builds part of the conceptual models. When reading a text, all elements that can be constructed from the text and the prior knowledge are added to the existing structure, which is then integrated (Kintsch, Welsch, Schmalhofer, & Zimny, 1990). The specific learning goal focuses on the relevant domain knowledge and thereby limits the amount of possible strategic inferences. A function schema provides the basis for the domain knowledge (i.e. system model). Elements of the textbase will be constructed on-line while processing a sentence. Immediate processing is also used when already available elements of the domain knowledge are encountered in the text. Newly formed elements usually cannot be assigned on-line to the function schema until the whole sentence is processed. During the integration, a new knowledge unit, which is assumed to be true, is set in relation to the already existing knowledge and is connected with it. By relating the new knowledge unit to the already constructed model, the information from the learning materials can be found to be redundant, contradictory, an operationalization of general knowledge, or truly new knowledge.

When a person learns from examples, a template is formed as the peripheral knowledge structure. By explanation-based generalization (Mitchell, Keller, & Kedar-Cabelli, 1986), a template is constructed from the specific examples, under the guidance of the already constructed domain knowledge. If no domain knowledge is available to explain the example, only a similarity-based generalization can be achieved. Furthermore, the domain knowledge may be extended by learning from these examples. Again, the formation of the different representations is achieved by a construction phase followed by an integration phase. The importance of learning from examples has been recently demonstrated by van Lehn, Jones, and Chi (1992) and Reimann and Schult (1992). An implementation of the acquisition of system knowledge has been presented by Schmalhofer and Kühn (1991).

The Acquisition of Task Knowledge

Task knowledge may be acquired from an instructive text (Kieras & Bovair, 1986) or from programs. The acquisition of task knowledge from computer programs is similar to the acquisition of a system model from examples. The

general learning goal is to form a procedure schema that, if refined in a concrete system environment like LISP, yield the given program. The acquisition of task knowledge from programs is performed as a construction-integration process.

The Construction Episode. For the given sequence of concrete operators (i.e. the program) robust production rules construct: 1) state descriptions (initial, intermediate and final) induced by the application of the concrete operators (i.e the concrete execution trace of the program in terms of the system model), 2) abstract descriptions of the induced concrete states, and 3) abstract operators which describe various possible descriptions among the abstract states (i.e. descriptions in terms of situational knowledge). These production rules may construct symbolic descriptions of concrete and abstract states as well as descriptions for some sample input data. What is being constructed thus depends upon the learner's prior domain knowledge (see Figure 1). The prior domain knowledge, which is represented in the long-term memory, comprise: 1) a system model about a specific programming environment (e.g. data structures and programming constructs), 2) a situation model about an application domain, and 3) the references between the system model and the situation model.

The Integration Episode. The representation resulting from this construction process may be redundant, incoherent and even partially contradictory. For example different abstract operators may have been constructed to describe alternative transitions between various states. The integration process transforms this initial representation into a coherent form and an abstract procedure schema is obtained. Previous research used a connectionist approach for performing this integration process (Kintsch, 1988; Kintsch et. al., 1990). In particular, a spreading activation mechanism was used. In order to obtain syntactically sound procedure schema, which indeed describes a valid transition from the initial to the final state, our model uses a symbolic integration process. This integration process involves two steps: The first step establishes a consistent and redundancy free path of abstract operations by a dependency analysis (Schmalhofer, Bergmann, Kühn, & Schmidt., 1991). In the second step an additional integration is achieved by generalizing the path of abstract operations into the final procedure schema. This schema is added to the task knowledge and may later on be used to develop new programs, possibly in a different programming language (Vorberg & Goebel, 1991; Goebel & Vorberg, 1991). The whole construction-integration process thus consists of five phases, a construction episode with three phases followed by

an integration episode with two phases with the two episodes being organized in cycles.

A Formal Characterization of the Abstraction of Task Knowledge

The acquisition of task knowledge as a construction-integration process and the KIWi-model have now been presented in general terms. In this model task knowledge is formed by a transition from the template base to a conceptual model. Certain prior knowledge is needed for successfully performing this process. In this section we will first formally characterize the relation between the template base as a more peripheral knowledge structure and the procedure schema as a conceptual model. We will furthermore formally describe the prior knowledge which is required for successfully forming a procedure schema from a computer program. Finally, a method for constructing a procedure schema will be described in terms of a formal logic for describing actions (Lifschitz, 1987).

The Relation between Template Base and Procedure Schema

It was assumed that procedure schemata are abstractions of concrete program execution traces, which are stored as part of the template base. Michalski and Kodratoff (1990) have recently pointed out that abstraction has to be distinguished from generalization. While generalization transforms a description along a set superset dimension, abstraction transforms a description along a level of detail dimension. While generalization often uses the same representation language, abstraction usually involves a change in the representation space (e.g. from the system level to the situational level) to transform the representation language into a simpler language than the original (Michalski & Kodratoff, 1990). The central idea of abstraction, therefore, may be stated as to achieve a reduction of the level of detail of a description. The acquisition of knowledge from programming events has often be described as generalization (Weber, Bögelsack, & Wender, in press) rather than abstraction.

A program is composed of operations which are executed in a specific order and changes the state of the system. Therefore program abstraction has two independent dimensions: On the first dimension a change in the level of detail for the representation of single states is described. On the second dimension a change in the level of detail is declared by reducing the number of the states

contained in a program. As a consequence, a change of the representation of the state description and a change of the operations which describe the state transitions is required.

Sufficient Conditions for the Formation of a Procedure Schema

Since the goal is to construct domain-specific program abstractions, we assume that the system model about the programming environment and the situation model about the application domain can be formalized as two STRIPS worlds (e.g., Fikes, Hart, & Nilsson, 1972; Lifschitz, 1987; Knoblock, 1989). A STRIPS world W is a Triple (R, T, Op) over a first-order language L, where R is a set of essential sentences (Lifschitz, 1987) which describe aspects of a state of the world.

As usual, a state of a world W (i.e., individual states in the programming system or the application domain) is described by a subset of the essential sentences from R. T is a static theory which allows the deduction of additional properties of a state in the world. The theory T is implicitly assumed to be valid in all states of the world. Concrete programming constructs or abstract operations are formalized as a set of STRIPS-operators Op. Op is specified by descriptions $<P_\alpha, D_\alpha, A_\alpha>_\alpha \in Op$, where P_α is the precondition formula, D_α is the delete list, and A_α is the add list (Fikes et al., 1972). Concrete programs or abstract procedures will be formalized as plans. A plan p in a world W is a sequence $(o_1,...,o_n)$ of operators from Op. In a world W an initial state $s_0 \in \mathbf{S}$ and a plan $p=(o_1,...,o_n)$ induce a sequence of states $s_1 \in \mathbf{S},...,s_n \in \mathbf{S}$ where $s_{i-1} \cup T \vdash P_{o_i}$ and $s_i = (s_{i-1} \setminus D_{o_i}) \cup A_{o_i}$.

The abstraction of a program requires a reduction in the level of detail for the description of the individual states and a reduction in the number of states. Both of these changes in the level of detail imply a change of the representation for states as well as for operators. Therefore, we assume that two world descriptions, $W_c = (R_c, T_c, Op_c)$ (the concrete world) and $W_a = (R_a, T_a, Op_a)$ (the abstract world), are given. The problem of program abstraction can now be described as transforming a plan p_c from the concrete world W_c (i.e., a program) into a plan p_a in the abstract world W_a (i.e., a sequence of operations), with several conditions being satisfied. This transformation can formally be decomposed into two mappings, a state abstraction mapping **a**, and a sequence abstraction mapping **b** as follows:

A state abstraction mapping **a**: $\mathbf{S}_c \rightarrow \mathbf{S}_a$ is a mapping from $\mathbf{S}_c$, the set of all states in the concrete world, to $\mathbf{S}_a$, the set of all states in the abstract world,

that satisfies the following conditions:

(a) If $sc \cup T_C$ is consistent, then $\mathbf{a}(sc) \cup T_a$ is consistent for all $sc \in \mathbf{S}_C$.

(b) If $sc \cup sc' \cup T_C$ is consistent, then $\mathbf{a}(sc \cup sc') \supseteq \mathbf{a}(sc) \cup \mathbf{a}(sc')$ for all $sc, sc' \in \mathbf{S}_C$.

The state abstraction mapping transforms a concrete state description s into an abstract state description and thereby changes the representation of a state from concrete to abstract.

A sequence abstraction mapping $\mathbf{b}$: $N \rightarrow N$ relates an abstract state sequence $(sa_0, ...,sa_n)$ to a concrete state sequence $(sc_0,...,sc_m)$ by mapping the indices i of the abstract states sa_i onto the indices j of the concrete states sc_j, so that the following properties hold:

(a) $\mathbf{b}(0) = 0$ and $\mathbf{b}(n) = m$: The initial state and the goal state of the abstract sequence must correspond to the initial and goal state of the respective concrete state sequence.

(b) $\mathbf{b}(u) < \mathbf{b}(v)$ iff $u < v$: The order of the states defined through the concrete state sequence must be maintained for the abstract state sequence.

A plan p_a (i.e., an abstract procedure) is an abstraction of a plan p_c (i.e., a program) if there exists a state abstraction mapping $\mathbf{a}$: $\mathbf{S}_C \rightarrow \mathbf{S}_a$ and a sequence abstraction mapping $\mathbf{b}$: $N \rightarrow N$, so that:

If p_c and an initial state sc_0 induce the state sequence$(sc_0,...,sc_m)$ and $sa_0 = \mathbf{a}(sc_0)$ and $(sa_0,...,sa_n)$ is the state sequence which is induced by sa_0 and the abstract plan p_a, then $\mathbf{a}(sc_{\mathbf{b}(i)}) = sa_i$ holds for all $i \in \{1,...,n\}$.

This definition of abstraction is represented by Figure 2. The concrete space shows the sequence of m operators together with the induced state sequence. Selected states induced by the concrete plan (i.e., sc_0, sc_2, and sc_m) are mapped by the state abstraction mapping $\mathbf{a}$ into states of the abstract space. The sequence abstraction mapping $\mathbf{b}$ maps the indices of the abstract states to the corresponding concrete states. It becomes clear that the program abstraction can be defined by a pair of abstraction mappings $(\mathbf{a},\mathbf{b})$.

For the construction of domain-specific abstraction mappings, a justification of the state abstraction mappings by the referential knowledge is necessary. Thereby the construction of abstraction mappings is restricted to possibly useful ones. The referential knowledge will be formalized as generic abstraction theory (Giordana, Roverso, & Saitta, 1991). Generic abstraction theories relate atomic formulae of an abstract language L' to terms of a

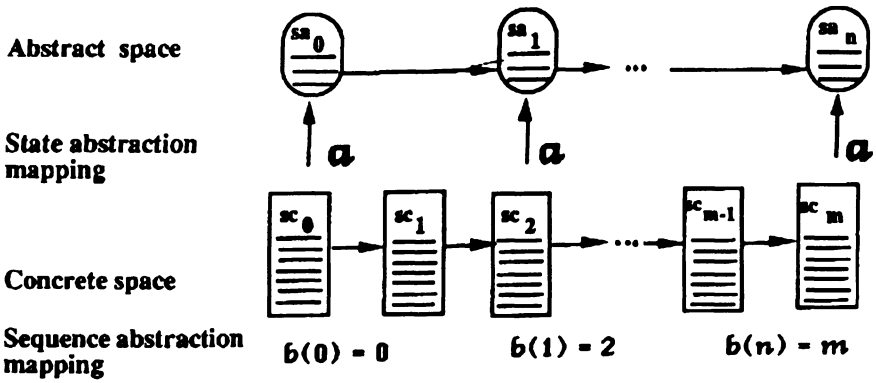

Figure 2: Definition of program abstraction.

corresponding concrete language L. The generic abstraction theory is restricted to a set of axioms of the form: $\psi \leftrightarrow D_1 \vee D_2 \vee ... \vee D_n$, where ψ is an atomic formulae of an abstract language L' and $D_1,...,D_n$ are conjunctions of predicates in L. Similarly, in this article, a generic state abstraction theory T_g is defined as a set of axioms of the form $\psi \leftrightarrow D_1 \vee D_2 \vee ... \vee D_n$, where ψ is an essential sentence of the abstract world and $D_1,...,D_n$ are conjunctions of sentences of the concrete world. For the state abstraction mapping **a**, it is required that: if $\psi \in \mathbf{a}(s_C)$ then $s_C \cup T_C \cup T_g \vdash \psi$. Since a minimal consistent state abstraction mapping (according to the $\supset$ relation) should be reached, the reverse implication, namely that every essential sentence for which $s_C \cup T_C \cup T_g \vdash \psi$, holds is an element of $\mathbf{a}(s_C)$, is not demanded.

A Formal Description of the Program Abstraction Method

As described in the previous section, the task of constructing an abstraction of a program can be seen as the problem of finding a deductively justified state abstraction mapping and a sequence abstraction mapping so that a related sequence of operators in the abstract world exists. This section describes PABS, a method which formalizes the construction of program abstractions. The five phases of the method implement the five processing phases in the acquisition of task knowledge.

In the first phase of PABS, execution of the concrete program p_C is simulated and the sequence of the induced states in the concrete space is computed. In

the second phase, for each of these states an abstract description is derived by utilizing the generic abstraction theory. In third phase, for each pair of the abstract states, it is checked whether there exists an abstract operation which is applicable, and which transforms abstract states into each other. A directed graph of candidate abstract operations is finally constructed in this phase. In phase four, a complete and consistent path from the initial abstract state to the final abstract state is searched. With this path a state abstraction mapping and a sequence abstraction mapping can be determined. Finally, explanation-based generalization is applied in phase five of PABS, to generalize the sequence of abstract operations into an abstract procedure schema with corresponding application conditions.

Phase I: Simulation of the Concrete Program. By simulating the execution of the concrete program p_C, the state sequence $(sc_1,...,sc_m)$ is computed, which is induced by the program p_C and a given initial state sc_0. During this simulation, the definition of the operators Op_C (i.e., programming constructs) and the static theory T_C are applied to derive all those essential sentences which hold in the respective states. The proofs that exist for the applicability of each operator can now be seen as an explanation for the effects caused by the operators. They are stored together with the respective relations which describe the states.

Phase II: Construction of State Abstractions. The second phase performs a prerequisite for the composition of the deductively justified state abstraction mapping. With the generic state abstraction theory T_g, an abstract state description sa'_i is derived for each state sc_i which was computed in the first step. Essential sentences R_a of the abstract world description W_a are checked as to whether they can be inferred from $sc_i \cup T_C \cup T_g$. If for $\psi \in R_a$, $sc_i \cup T_C \cup T_g \vdash \psi$ holds, then ψ is included in the state abstraction sa'_i. The proofs that exists for the verification of ψ in sa'_i are stored together with the respective essential sentence ψ. If the concrete and the abstract world descriptions consist of a finite set of essential sentences, and if the type of the theories T_C and T_g allows the determination of the truth of a sentence, the abstract state descriptions can be generated automatically. By allowing some user support in this phase, more concise abstract state descriptions may be obtained, thereby reducing the overall search in subsequent phases.

Phase III: Constructing Abstract Operations. The goal of the third phase is to identify candidate abstract operations for the abstract procedure. For each pair (sa'_u, sa'_v) with $u < v$, it is checked if there exists an abstract operator O_α described by $<P_\alpha, D_\alpha, A_\alpha>$ which is applicable in sa'_u and which

transforms sa'_u into sa'_v. If $sa'_u \cup T_a \vdash P_\alpha$, and if every sentence of A_α is contained in sa'_v, and none of the sentences of D_α is contained in sa'_v, then the operation O_α is noted to be a candidate for the abstract procedure. A directed graph is constructed, where the nodes of the graph are built by the abstract states sa'_i and where links between the states are introduced for those operations that are candidates for achieving the respective state transitions. Again, the proofs that exist for the validation of P_α in sa'_u are stored together with the corresponding operation.

Phase IV: Establishing a Consistent Path of Abstract Operations. From the constructed graph, a complete and consistent path $p_a = (o_1,...,o_n)$ from the initial abstract state sa'_0 to the final state sa'_m is searched. This graph defines a sequence abstraction mapping **b** through: $\mathbf{b}(i) = j$ iff the i-th abstract operation of the path connects state sa'_k with sa'_j. The consistency requirement for this path expresses that every essential sentence which guarantees the applicability of the operator o_{i+1} ($sa'_i \cup T_a \vdash P_\alpha$) is created by a preceding operator (through the add-list) and is protected until o_{i+1} is executed, or the essential sentence is already true in the initial state and is protected until o_{i+1} is executed. This condition assures that the abstract procedure represented by the path p_a is applicable, which means that the application conditions for all operations are satisfied in the states in which they are executed. A dependency analysis (Schmalhofer et al., 1991) can be performed to verify this condition.

Through the selected path $p_a = (o_1,...,o_n)$, a corresponding sequence of minimal abstract state descriptions $sa_0,...,sa_n$ with $sa'_{\mathbf{b}(i)} \supseteq sa_i$ is defined. These minimal abstract states contain exactly those sentences which are necessary to guarantee the applicability of the operations in the path. Thereby $sa_0,...,sa_n$ represents the state sequence which is induced by p_a. Note that a state abstraction mapping **a** is also implicitly defined through: $\mathbf{a}(sc) = \{\Psi \mid sc \cup T_C \cup T_g \vdash \Psi$ and $\exists\, i \in \{1,...,n\}$ such that $\Psi \in sa_i\}$. For realistic situations, the abstract world description will likely consist of a large number of essential sentences and abstract operations. Therefore it is expected that more than one abstraction can be found. In this situation a heuristic criterion (e.g., a good ratio n/m of the number of abstract to concrete states) can be chosen or the user has to decide which of the paths from the initial to the final state represent the intended abstraction.

Phase V: Constructing the Abstract Procedure. From p_a and the dependency network which justifies its consistency, a generalization of the abstracted operator sequence can be established. With the dependency network, which

functions as an explanation structure, explanation-based generalization can be applied to compute the least subgraph of the dependency network that connects all essential sentences of the final state with some sentences in the initial state. Within this subgraph, the remaining derivation trees which prove the applicability of the operations are generalized by standard goal regression as used by Mitchell, Keller, and Kedar-Cabelli (1986). Thereby constants are turned into variables. Operationality criteria are introduced for relations that

Concrete World:

Essential sentences: bitEq\2

Static Theory: Laws of boolean algebra to define simplify_bool\2

Operations:

Operation: and(x,y,z):
Precond: bitEq(x,t1) ∧ bitEq (y,t2) ∧ simplify_bool ((t1 and t2),t3)
Add: bitEq(z,t3)

Operation: or(x,y,z):
Precond: bitEq(x,t1) ∧ bitEq(y,t2) ∧ simplify_bool((t1 or t2),t3)
Add: bitEq(z,t3)

Operation: set(x,z):
Precond: bitEq(x,t1) ∧ simplify_bool(t1,t2)
Add: bitEq(z,t2)

Abstract World:

Essential sentences: natEq\2

Static Theory: Definition of simplify_nat\2, e.g., distribution law.

Operations:

Operation: add(x,y,z):
Precond: natEq(x,t1) ∧ natEq (y,t2) ∧ simplify_nat ((t1 + t2),t3)
Add: natEq(z,t3)

Operation: div2(x,z):
Precond: natEq(x,t1) ∧ simplify_nat((t1/2),t3)
Add: natEq(z,t3)

Operation: average(x,y,z):
Precond: natEq(x,t1) ∧ natEq(y,t2) ∧ simplify_nat(((t1 + t2)/2),t3
Add: natEq(z,t3)

Generic abstraction theory:

natEq(Y, (TN1+2*TN2+4*TN3)) ←
bitEq(X1,T1) ∧
bitEq(X2,T2) ∧
bitEq(X3,T3) ∧
X1\= X2 ∧ X1 \= X3 ∧ X2\= X3 ∧
map_to_N(T1,TN1) ∧
map_to_N(T2,TN2) ∧
map_to_N(T3,TN3).

natEq(Y, (TN1+2*TN2)) ←
bitEq(X1,T1) ∧
bitEq(X2,T2) ∧
X1\= X2 ∧
map_to_N(T1,TN1) ∧
map_to_N(T2,TN2).

Figure 3: Domain knowledge for program abstraction: system model, situation model, and referential knowledge.

describe the properties of generalized abstract operations. The final generalized explanation thus only contains operational relations, which describe the generalized operations, together with a generalized specification of the application conditions for the operator sequence. An algorithm for the determination of constraint relations with a locality property (Bergmann, 1990) is applied to construct this normal form for the abstracted procedure schemata.

The Abstraction of a Machine Level Program

To demonstrate the proposed method, the abstraction of an example program is presented in the following section. The goal is to find an abstraction for a machine language computer program which computes the average of two natural numbers. For that purpose, the required domain knowledge must be described first: the system model is formalized as the concrete world, the situation model as an abstract world, and the referential knowledge as a generic abstraction theory. Figure 3 shows a fraction of the formalized description.

The concrete world specifies the semantics of the operations of the machine language (and, or,...) in which the to be abstracted program is written. For example, and (x, y, z) specifies the assignment "z := x and y". The abstract world determines the high-level operations of the application domain arithmetic from which the abstraction is composed. These are some numeric operations on natural numbers (add, div2,...). The generic abstraction theory represents abstract data types (natural numbers) and their implementation (two or three-bit values).

The concrete program consists of 12 sequential steps as shown in Figure 4. The program computes the average of two natural numbers which are represented as two-bit values. The variables x1,...,x4 represent the input values, x5,...,x7 represent the output values, and x8, x9 are additional auxiliary variables.

Figure 5 shows the five phases of PABS which are applied for the abstraction of the example program: In the first phase, simulation of $P_{average}$ through symbolic execution (Bergmann, 1992) results in the concrete state sequence $sc_0,...,sc_{12}$. Each of the states is described by the actual value of the relevant boolean variables (bitEq), which depend on the symbolic program input $t_1,...,t_4$. In the goal state sc_{12}, the program output specification is derived. In

$P_{average}$:

1. xor(x1,x3,x5)
2. and(x1,x3,x8)
3. xor(x2,x4,x6)
4. xor(x6,x8,x6)
5. and(x2,x4,x7)
6. and(x2,x8,x9)
7. or(x7,x9,x7)
8. and(x4,x8,x9)
9. or(x7,x9,x7)
10. set(x6,x5)
11. set(x7,x6)
12. set(0,x7)

Figure 4: Example program $P_{average}$.

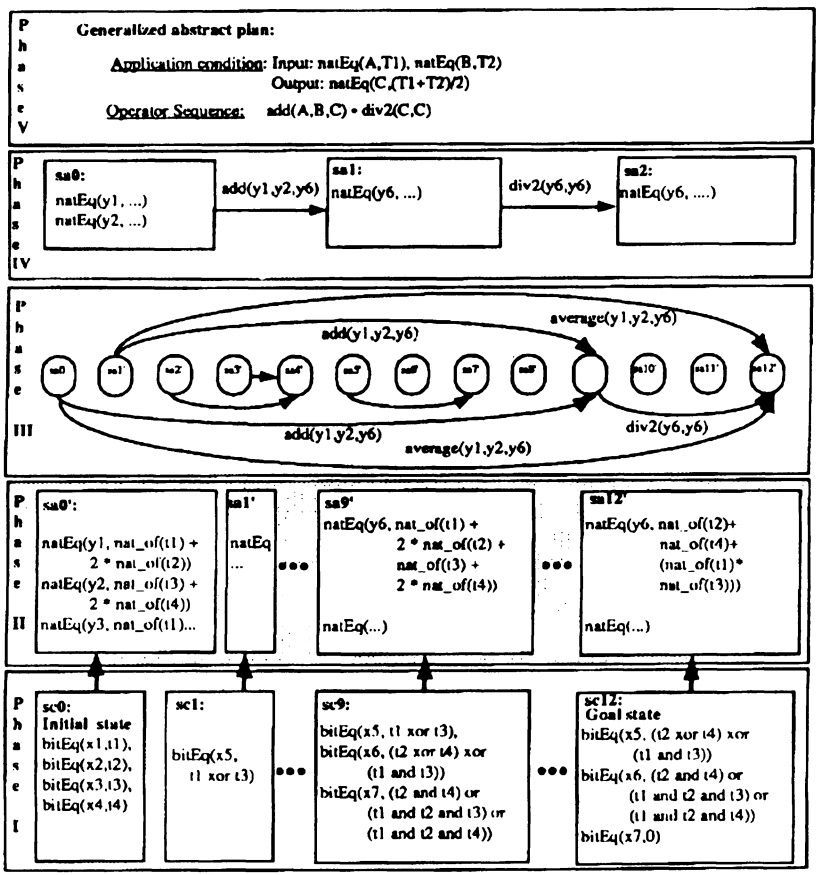

Figure 5: Five phases of PABS for the abstraction of the example program $P_{average}$

the second phase, each of the concrete states is abstracted by deriving essential sentences of the abstract state (the "natEq" predicates) by utilizing the generic abstraction theory. The generic abstraction theory describes the possible kinds of data abstraction by representing natural numbers as bit-vectors. The states $sa'_0,...,sa'_{12}$ computed in phase II contain the derived descriptions in terms of the abstract world. For example, the sentence "natEq(y1,nat_of(t1)+2* nat_of(t2))" in state sa'_0 expresses that there exists a natural number y1 whose value is computed from the value of the bits x1 and x2. The first clause of the generic abstraction theory from Figure 3 was applied to derive this data abstraction.

In the third phase, the graph of the candidate abstract operations is expanded. For example, the link from sa'_0 to sa'_9 labeled "add(y1,y2,y6)" specifies that the add-operation is applicable in the abstract state sa'_0 and creates some of the effects (i.e., natEq(y6,...)) which are contained in the state sa'_9. Using this graph, a consistent path from sa'_0 to sa'_{12} is searched in phase IV. In the example the path $sa'_0 \rightarrow sa'_9 \rightarrow sa'_{12}$ is selected to describe the intended abstraction. Note that the path $sa'_0 \rightarrow sa'_{12}$ is also a consistent path which is assumed to be rejected. Through the selection of the path, a sequence abstraction mapping $\mathbf{b}: N \rightarrow N$ is defined by: $\mathbf{b}(0) = 0$, $\mathbf{b}(1) = 9$, and $\mathbf{b}(2) = 12$. The state abstraction mapping **a** maps the concrete states sc_0, sc_9, and sc_{12} to the derived abstract states sa_0, sa_1, and sa_2 which are subsets of the corresponding abstracted states sa'_0, sa'_9, and sa'_{12} computed in phase II. In the example **a** is defined as follows[2]:

a({bitEq(x1,T1,)bitEq(x2,T2)})={natEq(y1,nat_of(T1) + 2*nat_of(T2)}

a({bitEq(x3,T1,)bitEq(x4,T2)})={natEq(y2,nat_of(T1) + 2*nat_of(T2)}

a({bitEq(x5,T1,)bitEq(x6,T2), bitEq(x7,T3)})
={natEq(y6,nat_of(T1) + 2*nat_of(T2) + 4*nat_of(T3))}

Note that **a** exactly creates those preconditions and effects that are required for the abstract operations in the path. In the fifth phase, an explanation-based generalization computes the final abstract plan. This generalization is responsible, for example, for the replacement of the constant "nat_of(t1)+2*nat_of(t2)" by the variable A. The completely abstracted procedure expresses that whenever the average of the natural numbers T1 and T2 has to be computed, the sequence of adding T1 and T2 and dividing the result by two is an appropriate abstract procedure.

---

[2] T1, ..., T3 stand for boolean variables

Empirical validation

Four experiments were performed where successively more information was presented to the subjects. These experiments examined how well the formation of a procedure schema can be described as a construction-integration process, which knowledge constructions are actually verbalized in think-aloud protocols, and how the construction and integration phases are interleaved. In addition, the data can be used to test the specific formalization of the PABS-method.

Four computer science students participated in the first study. The information which was presented to the subjects consisted of a concrete program, some information about references between data structures in the concrete programming language structures in an abstract application domain, the specification of an abstract procedure, and one example of the abstract procedure. The subjects had two tasks: to form a symbolic program execution trace and to describe the different steps when designing the given concrete program. In the study the subjects solved several problems, whereby different programming languages (LISP, Pascal, machine language) and different application domains (e.g., set theory, arithmetic) were involved. The results show that the subjects had no problems in forming a symbolic execution trace and to explain the design of the program.

In the second study (three computer science students), only the concrete program was presented to the subjects and no other information about the application domain was available. The subjects were given two tasks: formation of the symbolic execution trace(as in the first study) and thereafter formation of an abstract procedure. The subjects had to solve two problems which were selected from the first study: a machine level program which computes the average of two natural numbers (see Figure 4) and a LISP program which computes the power set function. The subjects again had no problems in forming an execution trace. However, they had serious problems in finding a suitable application domain without any information about it. Therefore, the subjects mostly failed to construct the abstract procedure.

In the third study (two computer science students), the subjects received the concrete program and additional information about references between data structures in the concrete programming language and structures in the abstract

domain. With this additional information, the subjects were able to form the suitable abstract procedure in three of four cases.

In all three studies, the subjects could thus form a symbolic program execution trace when acquiring procedure knowledge (phase I of the PABS-method). In the second and the third study some empirical evidence for the formation of a procedure schemata by abstraction from the program execution trace (phase II - phase V) was found. Most subjects, however, applied supplementary strategies. For example, these subjects formed program traces from sample input data and often abstracted only the initial and the final states, but no intermediate states. Whereas these different strategies are consistent with the general model, they are not part of the specific PABS-method. Therefore, we decided to examine the abstraction of symbolic program execution traces more closely in a fourth study, which will be described next.

Method

Subjects. Four computer science students from the University of Kaiserslautern participated in this study. The students were paid for their participation.

Material. The instruction materials contained a concrete program in machine language (see Figure 4), information as to which patterns of bits make reference to which natural numbers, and the symbolic program execution trace, which had not been presented in the previous studies.

Procedure. The four subjects were first tested whether they had the necessary system knowledge about machine language (boolean algebra). The think-aloud was then practiced with a number multiplication task and an anagram task. During the study the subjects had the task to form the abstract procedure. The students solved the problem while giving think-aloud protocols, and no tutorial assistance was given. However, if a subject was lost in a problem, the experimenter would intervene and would stop the undertaking.

Results

One subject (Subject 3) formed the correct abstract procedure in about 30 minutes. The other subjects were lost in the problem and the experimenter stopped their activity after approximately one hour.

Because the raw protocols are extremely complex, we followed the suggestions made by Anderson, Farrell, & Sauers (1984) and constructed

schematic protocols. Such schematic protocols which characterize the essential features of the raw protocols and consist of a sequence of steps.

Table 1
Schematic Protocol for Subject 3

1. Comprehension of instructions (652 words, approximately n minutes)
2. Study of symbolic trace and generation of abstractions of individual states (359 words, approximately 4 minutes)
3. Review and summary (139 words)
4. Continue to study symbolic trace including local reviews (485 words)
5. Completion of abstract program (29 words)
6. Checking consistency between symbolic trace and abstract program (317 words)
7. Statement of final results (85 words)

Table 1 shows the schematic protocol for subject 3, who formed the correct abstract procedure. Consistent with the PABS-method, the subject worked through the symbolic trace, abstracted individual states of the program execution trace to states in the application domain arithmetic (e.g., protocol step 2 or 4), found possible arithmetic operations (e.g., step 2 or 4), and finally formed the correct abstract procedure (step 7). Contradictory to the method, subject 3 didn't study the complete program execution trace before abstracting individual states, but acted in a more cyclic mode: the subject worked through the symbolic trace and abstracted individual states until some abstract operation in the application domain could be found.

Figure 6 shows a graphical representation of the subject's time course of performance. A sequence of boxes and nodes in dark grey indicate the state sequence of the machine language programming system (boxes) together with the effecting programming steps (nodes). Boxes and nodes in light grey indicate the sequence of states and operations in the abstract application domain arithmetic. Thereby large boxes indicate symbolic states which can be derived from symbol input specifications, and small boxes below indicate states which can be derived from sample input data. Arrows indicate different processing episodes of subject 3 and the respective numbers refer to the steps in the schematic protocol. For example, the arrows labelled "2a" and "2c" in the second protocol episode refer to the study of symbolic trace (horizontal

arrows), and the generation of abstractions of individual states (vertical arrows), respectively.

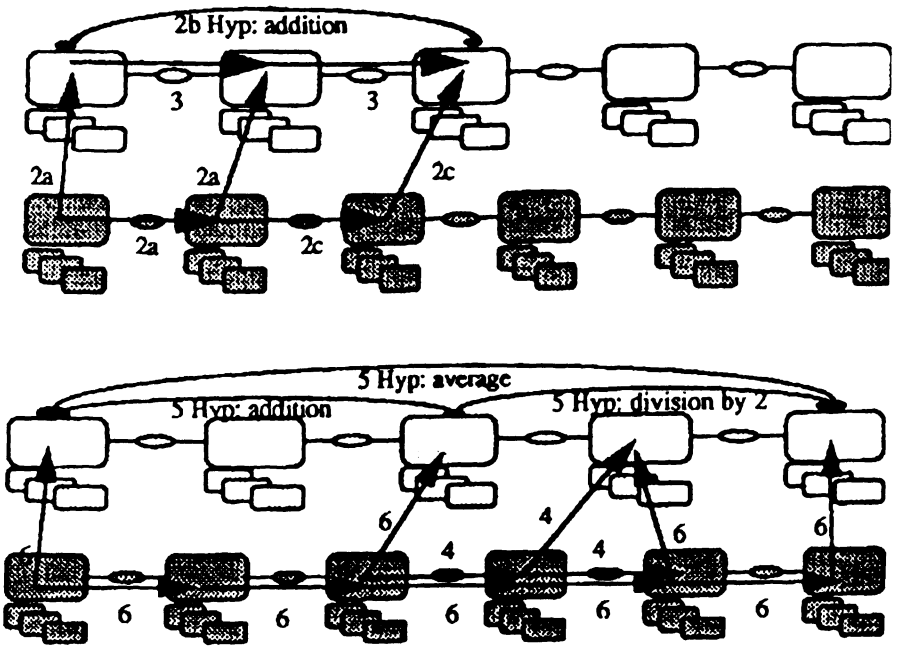

Figure 6: Subject 3: Time course of performance.

The three other subjects showed a similar performance. Therefore, the performance of one subject is presented in detail. Table 2 shows the schematic protocol for subject 2. In correspondence with the PABS-method, the subject studied the program execution trace and abstracted some individual states (e.g., protocol steps 5 and 8). Most protocol episodes were, however, supplementary to the proposed method: the subject built circuitries for adding or subtracting numbers and compared the circuitries with the concrete program (e.g., steps 4 and 6). Furthermore, the subject generated concrete input-output examples and tried to infer hypotheses about abstract operations from these examples (e.g., steps 9 and 10). Figure 7 shows a graphical representation of the subject's time course of performance.

Table 2
Schematic Protocol for Subject 2

---

1. Comprehension of instructions (538 words)
2. Generation of hypothesis about abstract operator(addition) (50 words)
3. Comprehension of instructions (135 words)
4. Build a circuitry for adding numbers (924 words)
5. Study symbolic trace (509 words)
6. Build own symbolic trace from program (483 words)
7. Reflection about previous actions (88 words)
8. Abstraction of individual states (84 words)
9. Generate concrete input-output examples (364 words)
10. Modification of first hypothesis based upon a single example (addition → subtraction) (27 words)
11. Generate concrete input-output examples for abstract operators (146 words)
12. Build negation circuitry (200 words)
13. Study symbolic trace (99 words)
14. Generate concrete input-output examples and infer abstract hypotheses (418 words)
15. Reflection about previous actions (91 words)
16. Modification of some abstraction rules (15 words)
17 Check input-output relations to test hypothesis (104 words)
18. Modification of some abstraction rules (25 words)
19. Check input-output relations to test hypothesis (74 words)
20. Reflection about previous actions (73 words)

Discussion

In this chapter we have proposed to describe the formation of a procedure schema as a construction-integration process with multiple knowledge representation on different levels of abstraction. Four think-aloud studies were performed to validate this model. The results showed that program abstraction can very well be described as such a construction-integration process. It was found that subjects used at least two types of construction rules: Most subjects preferred to construct specific examples from sample input data and abstract hypotheses, which were consistent with these examples. Some subjects were more sophisticated and constructed the program execution for all possible inputs by describing an input in general terms (i.e. a symbolic execution trace) rather than using an arbitrary example

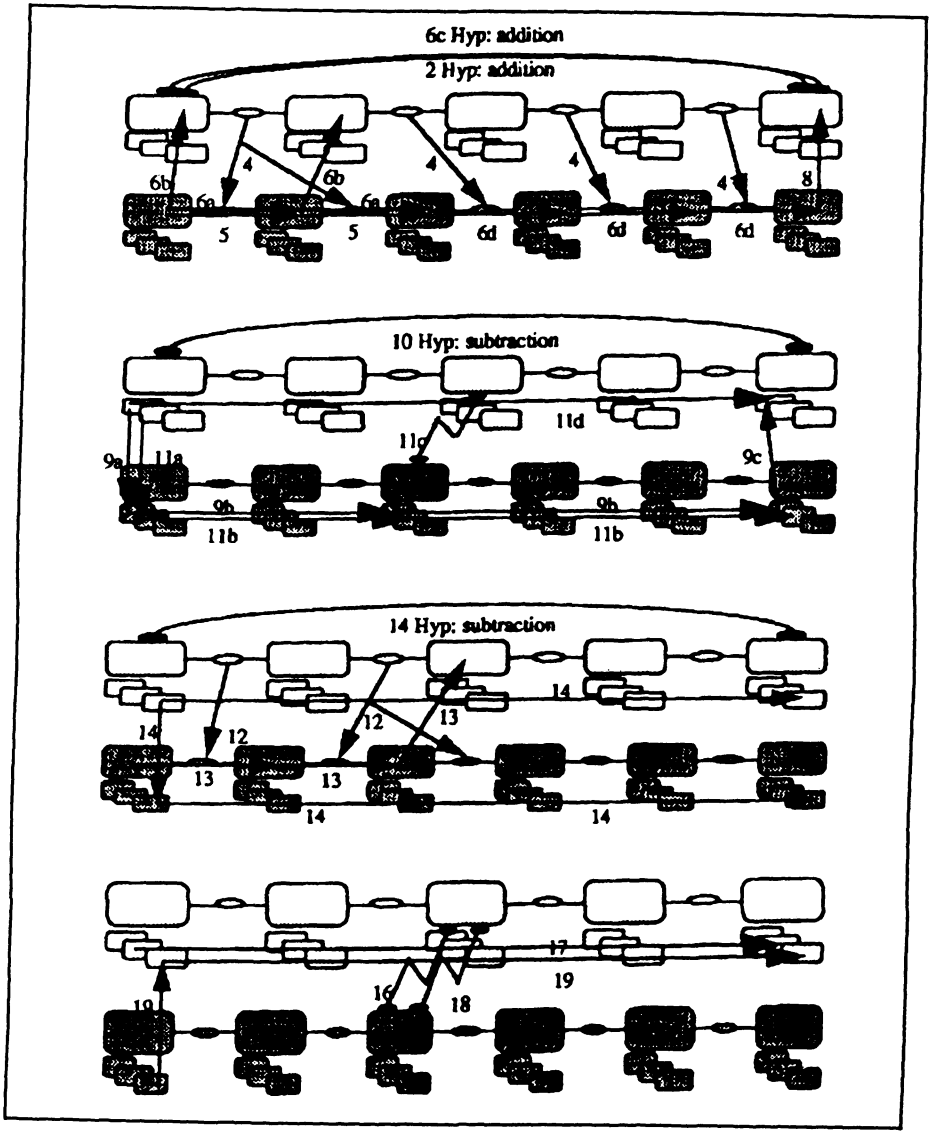

Figure 7: Subject 2: Time course of performance.

input. These subjects were also able to construct abstractions with deductive justifications. The think-aloud protocols also provide evidence for the integration process. For instance, after all the relevant knowledge has been constructed, some subjects systematically reviewed the coherent procedure schema and its relation to the specific program from which it was abstracted (see Table 1, steps 6 and 7).

In the current implementation of the PABS-method, we only used production rules that construct symbolic execution traces. The current implementation consequently describes only that subset of subjects. The modeling of the other subjects would require that production rules for the respective knowledge construction are being implemented. In order to obtain more predictive power with the proposed model for individual subjects, one would need to diagnose the individual's prior knowledge. Since computer programs can be executed on finite automata, one could very well use the diagnostic procedures of Buchner and Funke (1993), which rely on finite-state automata descriptions.

Independent of the specific knowledge construction rules being used, all subjects interleaved construction and integration episodes in a cyclic manner. The construction processes for the first segment of the program were often followed by an integration episode, construction processes for the following program segment and so on and so forth (see Figure 6). In the current research we have thus presented a model for the construction of task knowledge from computer programs within a construction-integration framework, which appears to be well suited for describing the learning of program abstractions.

Acknowledgements

Karin Schweizer helped in designing and running one of the four studies. We would also like to thank Joachim Funke, Uli Glowalla, Walter Kintsch, and Gerd Weber for their comments on this paper.

References

Adelson, B. (1984). When novices surpass experts: The difficulty of a task may increase with expertise. *Journal of Experimental Psychology: Learning, Memory,and Cognition, 10,* 484-495.

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

Bergmann, R. (1990). Generierung von Skelettplänen als Problem der Wissensakquisition, Unpublished master's thesis, Universität Kaiserslautern.

Bergmann, R. (1992). Explanation-based learning for the automated reuse of programs. *Proceedings of the IEEE-Conference on Computer Systems and Software Engineering, COMPEURO92.*

Bergmann, R., Boschert, S., & Schmalhofer, F. (1992). Das Erlernen einer Programmiersprache: Wissenserwerb aus Texten, Beispielen und komplexen Programmen. In K. Reiss, M. Reiss, & H. Spandl (Eds.), *Maschinelles Lernen: Modellierung von Lernen mit Maschinen* (pp. 204-224). Berlin: Springer-Verlag.

Breuker, J. A., & Wielinga, W. J. (1989). Model driven knowledge acquisition. In P. Guida & G. Tasso (Eds.), Topics in the Design of Expert Systems (pp. 265-96). Amsterdam: North Holland.

Buchner, A., & Funke, J. (1993). Finite-state automata: Dynamic Task Environments in problem-solving research. The Quarterly Journal of Experimental Psychology, 46A(1), 83-118.

Detienne, F., & Soloway, E. (1990). An empirically-derived control structure for the process of program understanding. *International Journal of Man-Machine Studies, 33,* 323-342.

Fikes, R. E., Hart, P. E., & Nilsson, N. J. (1972). Learning and executing generalized robot plans. *Artificial Intelligence, 3,* 251-288.

Giordana, A., Roverso, D., & Saitta, L. (1991). Abstracting background knowledge for concept learning. In Y. Kodratoff (Ed.), *Lecture notes in artificial intelligence: Machine learning-EWSL-91* (pp. 1-13). Berlin: Springer.

Goebel, R., & Vorberg, D. (1991). Das Lösen rekursiver Programmierprobleme: Ein Simulationsmodell. *Kognitionswissenschaft, 2,* 27-36.

Halasz, F. G., & Moran, T. P. (1983). Mental models and problem solving in using a calculator. *Proceedings of CHI-83 Human Factors in Computing Systems.* New York: ACM.

Jeffries, R., Turner, A. A., Polson, P. G., & Atwood, M. E. (1981). The processes involved in designing software. In J. R. Anderson (Ed.), *Cognitive skills and their acquisition.* Hillsdale, NJ: Erlbaum.

Kieras, D. E., & Bovair, S. (1984). The role of a mental model in learning to operate a device. *Cognitive Science, 8,* 255-273.

Kintsch, W. (1974). *The representation of meaning in memory.* Hillsdale, NJ: Erlbaum.

Kintsch, W. (1988). The role of knowledge in discourse comprehension: A construction-integration model. *Psychological Review, 95,* 163-182.

Kintsch, W., & Van Dijk, T. A. (1978). Toward a model of text comprehension and production. *Psychological Review, 85,* 363-394.

Kintsch, W., Welsch, D., Schmalhofer, F., & Zimny, S. (1990). Sentence memory: A theoretical analysis. Journal of Memory and Language, 29, 133-159.

Knoblock, C. A. (1989). A theory of abstraction for hierarchical planning. *Proceedings of the Workshop on Change of Representation and Inductive Bias* (pp.81-104). Boston: Kluwer.

Lifschitz, V. (1987). On the semantics of STRIPS. *Reasoning about Actions and Plans: Proceedings of the 1986 Workshop* (pp. 1-9). Timberline, Oregon.

McKoon, G. & Ratcliff, R. (1992). Inference during reading. Psychological Review, 99, 440-466.

Michalski, R. S., & Kodratoff, Y. (1990). Research in machine learning: Recent progress, classification of methods, and future directions. In Y. Kodratoff & R. S.Michalski (Eds.),

Machine learning: An artificial intelligence approach (Vol. 3, pp. 3-30). San Mateo, CA: Morgan Kaufmann.

Miller, J. R., & Kintsch, W. (1980). Readability and recall of short prose passages: A theoretical analysis. *Journal of Experimental Psychology: Human Learning and Memory, 6,* 335-354.

Mitchell, T. M., Keller, R. M., & Kedar-Cabelli, S. T. (1986). Explanation-based generalization: a unifying view. *Machine Learning, 1,* 48-80.

Morrow, D., Bower, G., & Greenspan, S. (1989). Updating situation models during narrative comprehension. Journal of Memory and Language, 28, 292-312.

Norman, D.A. (1983). Some observations on mental models. In D. Gentner & A. L. Stevens (Eds.), *Mental models.* Hillsdale, NJ: Erlbaum.

Pennington, N. (1987). Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology, 19,* 295-314.

Reimann, P., & Schult, T. J. (1992). Transforming examples into cases. In F. Schmalhofer, G. Strube, & T. Wetter (Eds.), *Contemporary knowledge engineering and cognition* (pp. 139-146). Heidelberg: Springer.

Rumelhart, D. E., & McClelland, J. L. (1986). Parallel distributed processing. Explorations in the microstructure of cognition. Vol. 1: Foundations. Cambridge: MIT Press.

Schank, R. J., & Abelson, R. (1977). *Scripts, plans, goals,and understanding.* Hillsdale, NJ: Erlbaum.

Schmalhofer, F., Bergmann, R., Kühn, O., & Schmidt, G. (1991). Using integrated knowledge acquisition to prepare sophisticated expert plans for their re-use in novel situations. In T. Christaller (Ed.), *GWAI-91: 15.Fachtagung für Künstliche Intelligenz* (pp. 62-71). Berlin: Springer-Verlag.

Schmalhofer, F., & Boschert, S. (1991). *Learning from text and examples: How situation model, text representation,and template base depend on instruction materials and prior domain knowledge.* Unpublished manuscript, German Research Center for Artificial Intelligence, Kaiserslautern.

Schmalhofer, F., Boschert, S., & Kühn, O. (1990). Der Aufbau allgemeinen Situationswissens aus Text und Beispielen. *Zeitschrift für Pädagogische Psychologie, 4,* 177-186.

Schmalhofer, F., & Glavanov, D. (1986). Three components of understanding a programmer's manual: Verbatim, propositional, and situational representations. *Journal of Memory and Language, 25,* 279-294.

Schmalhofer, F., & Kühn, O. (1991).The psychological processes of constructing a mental model when learning by being told, from examples and by exploration. In M. J. Tauber & D. Ackermann (Eds.), *Mental models and human-computer interaction 2* (pp. 337-360). Amsterdam: North-Holland

Schmalhofer, F., & Thoben, J. (1992). The case-oriented construction of a knowledge base. AI Communications, 5(1), 3-18.

Soloway, E., Ehrlich, K., & Black, J. B. (1983). Beyond numbers: Don't ask "how many" ... ask "why". *Proceedings of the Conference on Human Factors in Computer Systems,* Boston, MA.

Van Dijk, T. A., & Kintsch, W. (1983). *Strategies of discourse comprehension.* San Diego, CA: Academic Press.

van Lehn, K., Jones, R. M., & Chi, M. T. H. (1992). A model of the self-explanation effect. *The Journal of the Learning Sciences, 2,* 1-59.

Vorberg, D., & Goebel, R. (1991). Das Lösen rekursiver Programmierprobleme: Rekursionsschemata. *Kognitionswissenschaft, 1,* 83-95.

Weber, G., Bögelsack, A., & Wender, K. F. (in press). Representation of programming episodes in the ELM model. In K. F. Wender, F. Schmalhofer & H. D. Boecker (Eds.), *Cognition and computer programming.* Norwood, NJ: Ablex.

Yourdon, E. (1989). *Managing the structured techniques.* Englewood Cliffs, NJ: Yourdon Press.