Computer-Aided Formal Methods: A Generic Concept

Maritta Heisel

Technische Universität Berlin
FB Informatik – FG Softwaretechnik
Franklinstr. 28-29, Sekr. FR 5-6, D-10587 Berlin, Germany
heisel@cs.tu-berlin.de

Abstract. We present a formalism-independent approach to the design of support systems for the application of formal methods in software engineering. Its basis is a knowledge representation mechanism called strategy. Strategies represent development knowledge used to perform different software engineering activities. The development of an artefact is modeled as a problem solving process. The definition of strategies is generic in the definition of problems, solutions and acceptability of a solution with respect to a problem. The notion of strategy is complemented by a generic system architecture that serves as a template for the implementation of support tools for strategy-based problem solving. Two different instantiations of the strategy framework and an implemented program synthesis system are presented.

1 Introduction

The idea behind Computer-Aided Software Engineering is that the process of developing software can (and should) be supported by software tools. Tools would lead their users through a well-defined process and relieve them from tedious bookkeeping tasks. CASE tools support traditional, informal approaches to software engineering.

For the application of formal methods, tool support is even more important. Their use involves considering much more detail than necessary with traditional methods. Hence, applying formal methods without tool support is a tedious and error-prone task. The peculiarities of formal methods lead to specific requirements for tools that support their application in software engineering practice:

Requirements for Formal Methods Specific Tool Support

Usability by non-experts. Only if tools are built that allow non-experts to work with formal methods, they will have a chance to enter industrial practice. Today, most existing tools are parsers, type checkers and documentation tools for specifications, or theorem provers for the underlying logics. Only few provide support for the *methodological* aspects of formal methods. But for non-experts, methodological support is crucial.

Guarantee semantic properties. A tool must support the development process in a way that eases rigorous mathematical reasoning and establishes confidence that the product indeed fulfills the required properties. There

must be a clear identification of the steps in the development process that are responsible for establishing semantic properties.

Balance User Guidance and Flexibility. Since the application of formal methods is a non-trivial task, it is important not to leave the user alone with a mere formalism but to develop explicit techniques to guide its use. On the other hand, a tool should not unnecessarily restrict its users. Therefore, it must support the *combination* of different techniques and be *customizable* by informed users who develop specialized techniques for their project contexts.

Provide Overview of Development. Exactness and rigor entail a higher level of detail that must be handled. It is crucial for developers to have tool support that provides an *overview* of the development process and the relations between subtasks.

This paper presents a *formalism independent approach* to the design of tools that support the peculiarities of formal methods. The formal basis of this approach is a knowledge representation mechanism called *strategy*. Strategies make development knowledge implementable. In the strategy framework, a development activity is conceived as constructing a solution for a given problem. The solution to be constructed must be *acceptable* for the problem. Acceptability captures the semantic requirements concerning the product of the development process. The notion of strategy is generic in the definition of problems, solutions and acceptability. Hence, different development activities can be expressed as strategies. Using so-called *strategicals*, more powerful strategies can be defined by combination of existing ones.

To make strategies implementable, they are represented as *strategy modules*. This leads to a uniform interface between strategies which enables combination of methods and enhances the adaptability of a support tool. Moreover, those parts of a strategy that are responsible for the acceptability of solutions are isolated. Only they have to be verified in order to gain confidence in a non-verified support system. Finally, the parts of a strategy amenable to automation are clearly identified. Thus, it becomes possible to gradually replace user interaction by automatic procedures.

Strategies, strategicals, and strategy modules are formally defined in the language Z [Spi92]. This does not only provide precise definitions of these notions but also makes reasoning about strategies possible.

The strategy framework is completed by a *generic system architecture*. This architecture shows how to implement support tools for strategy-based development. It is designed to meet the requirements expressed above. Different support systems implementing different instantiations of the strategy framework have a strong potential for successful combination. Such a combination would provide integrated tool support for different software development activities. A program synthesis system called IOSS (Integrated Open Synthesis System) is an instance of this architecture.

In the following, we present the formal foundation of the strategy framework in Sect. 2. The system architecture is described in Sect. 3. An instantiation of the framework to support program synthesis is presented in Sect. 4, together with

a description of the implemented system IOSS. Sect. 5 presents an instantiation for specification acquisition. We are then able to compare the two instantiations in Sect. 6 and to compare strategy-based problem solving with tactical theorem proving and other related work (Sect. 7). Finally, we summarize in Sect. 8.

2 Strategies

Strategies describe possible steps during a development. Examples are how to decompose a system design to guarantee a particular property, how to conduct a data refinement, or how to implement a particular class of algorithms. This kind of knowledge can be found in text books on software engineering.

A strategy works by problem reduction. For a given problem, it determines a number of subproblems. From their solutions the strategy produces a solution to the initial problem. Finally, it tests if that solution is acceptable according to some notion of acceptability. The solutions to subproblems are naturally obtained by strategy applications as well. In general, the subproblems of a strategy are not independent of each other and of the solutions to other subproblems. This restricts the order in which the various subproblems can be set up and solved.

We first give a formal definition of strategies. Second, we sketch functions to define new strategies from existing ones. Finally, we describe how strategies can be represented in a way suitable for implementation.

2.1 Formal Definition of Strategies

Formally, strategies are defined as relations, relating a problem to the subproblems needed so solve it, and the final solution to the solutions of the subproblems. The formal definition is expressed in the specification language Z.

Definition of Database Relations. Since in the context of strategies it is convenient to refer to the subproblems and their solutions by *names*, our definition of strategies is based on the the notion of relation as used in the theory of relational databases [Kan90], instead of the usual mathematical notion of relation. In this setting, relations are sets of tuples. A tuple is a mapping from a set of *attributes* to domains of these attributes. In this way, each component of a tuple can be referred to by its attribute name. In order not to confuse these domains with the domain of a relation as it is frequently used in Z, we introduce the type *Value* as the domain for all attributes.

With the basic types *Attribute* and *Value*, we can define tuples as finite partial functions from attributes to values: $tuple : \mathbb{P}(Attribute \pfun Value)$. Relations are sets of tuples that all have the same domain. This domain is called the *scheme* of the relation.

$$\begin{array}{|l} relation : \mathbb{P}(\mathbb{P}\ tuple) \\ \hline \forall r : relation \bullet \forall t_1, t_2 : r \bullet \mathrm{dom}\ t_1 = \mathrm{dom}\ t_2 \end{array}$$

A *join* combines two relations. The scheme of the joined relation is the union of the scheme of the given relations. On common elements of the schemes, the values of the attributes must coincide.

$$\_ \bowtie \_ : relation \times relation \longrightarrow relation$$

$$\begin{aligned}
&\forall\, r_1, r_2, r : relation \bullet \\
&\quad r_1 \bowtie r_2 \\
&\quad = \{t : tuple \mid \mathrm{dom}\, t = scheme\, r_1 \cup scheme\, r_2 \wedge \\
&\qquad\qquad scheme\, r_1 \lhd t \in r_1 \wedge scheme\, r_2 \lhd t \in r_2\}
\end{aligned}$$

The join operation is associative and commutative. Hence, the join can also be defined for finite sets of relations. This operation is denoted $\bowtie$.

Using Database Relations to Define Strategies. We now use the definition of relations given above to define strategies. The first step is to instantiate the *Attribute* and *Value* sets for problem solving. Second, we define constituting relations and the notions of admissibility and determinability of attribute values. Finally, strategies can be defined.

Problems, Solutions, Acceptability. Problems and solutions are generic parameters for the notion of strategy. The sets *Problem* and *Solution* are defined as subsets of *Value* with an empty intersection. Acceptability is a relation between problems and solutions: $\_acceptable\_for\_ : Solution \leftrightarrow Problem$. The sets *ProblemAttribute* and *SolutionAttribute* are subsets of *Attribute* with an empty intersection. Both have countably many elements.

We use the distinguished attributes *P_init* and *S_final* to refer to the initial problem and its final solution. Moreover, we assume a bijective correspondence $cor : ProblemAttribute \rightarrowtail\!\!\!\!\!\rightarrow SolutionAttribute$ between problem and solution attributes.

Constituting Relations. Each strategy is defined by a set of so-called *constituting relations*. These relations represent the dependencies between the subproblems generated by a strategy. Their schemes consist of arbitrary attributes for problems and solutions. They are divided into *input attributes IA* and *output attributes OA*. The constituting relations restrict the values of the output attributes, given the values of the input attributes. Thus, they determine an order on the subproblems that must be respected in the problem solving process.

$$const\_rel : \mathbb{P}\, relation$$

$$\begin{aligned}
&\forall\, cr : const_rel \bullet \forall\, t : cr;\ a : scheme\, cr \bullet \\
&\quad scheme\, cr \subseteq (ProblemAttribute \cup SolutionAttribute) \wedge \\
&\quad (a \in ProblemAttribute \Rightarrow t\ a \in Problem) \wedge \\
&\quad (a \in SolutionAttribute \Rightarrow t\ a \in Solution)
\end{aligned}$$

It is now possible to define dependency of constituting relations. A constituting relation directly depends on another ($cr_1 \sqsubset_d cr_2$) if one of its input attributes is an output attribute of the other relation: $OA\, cr_1 \cap IA\, cr_2 \neq \emptyset$. The depending constituting relation is considered to be "larger". The transitive closure of the direct dependency relation yields the dependency relation $\sqsubset$.

A set of constituting relations that defines a strategy must conform to our intuition of problem solving, i.e.

1. The original problem to be solved must be known, i.e. *P_init* must always be an input attribute.

2. The solution to the original problem is the last item to be determined, i.e. *S_final* must always be an output attribute.
3. Each attribute value should be determined only once, i.e. the sets of output attributes of all constituting relations must be disjoint.
4. Each solution to a subproblem is used further, i.e. it occurs as an input attribute of some constituting relation.
5. A solution must directly depend on the corresponding problem, i.e. if a solution attribute is an output attribute of a constituting relation, then the corresponding problem attribute must occur in the scheme of this constituting relation.

Sets of constituting relations fulfilling these requirements are called *admissible*. The function *partsols* yields all solution attributes of a relation scheme except *S_final*. Each line of the predicate corresponds to one of the above requirements.

$$
\begin{array}{|l}
admissible\_ : \mathbb{P}(\mathbb{F}\ const\_rel) \\
\hline
\forall\, crs : \mathbb{F}\ const\_rel \bullet \\
\quad admissible\ crs \\
\quad \Leftrightarrow (\forall\, cr, cr' : crs \mid cr \neq cr' \bullet \\
\qquad\qquad (P\_init \in scheme\ cr \Rightarrow P\_init \in IA\ cr) \\
\qquad\qquad \land\ (S\_final \in scheme\ cr \Rightarrow S\_final \in OA\ cr) \\
\qquad\qquad \land\ OA\ cr \cap OA\ cr' = \varnothing \\
\qquad\qquad \land\ (\forall\, a : partsols\ cr \bullet \exists\, cr'' : crs \bullet a \in IA\ cr'') \\
\qquad\qquad \land\ (\forall\, a : partsols\ cr \mid a \in OA\ cr \bullet cor^{\sim} a \in scheme\ cr))
\end{array}
$$

The predicate $determinable\_ : \mathbb{P}(\mathbb{F}\ const\_rel)$ requires that there is an order in which all the attribute values can be determined. It says that the dependency relation must not be cyclic, and that each input attribute of a constituting relation except *P_init* must be an output attribute of a smaller relation.

Strategies. It is now possible to define strategies as relations that fulfill certain conditions:

1. The scheme of the relation must contain the attributes *P_init* and *S_final*.
2. For each problem attribute of the scheme, the corresponding solution attribute must be in the scheme, and vice versa.
3. There must be a uniquely defined set of constituting relations that is admissible and determinable, such that the relation is the join of these constituting relations.
4. If a member of the relation contains acceptable solutions for all problems except *P_init* then it must also contain an acceptable solution for *P_init*. This means, if all subproblems are solved correctly, then the original problem must be solved correctly, too.

The last condition guarantees that a problem that is solved exclusively by application of strategies is always correctly solved. For strategies solving the problem directly, this condition means that the solution to be generated must be acceptable. The function *subprs* yields all problem attributes of a relation scheme except *P_init*.

$$
\begin{array}{|l}
strategy : \mathbb{P}\ relation \\
\hline
\forall\, strat : strategy \bullet \\
\quad \{P\_init, S\_final\} \subseteq scheme\ strat \\
\quad \land\ (\forall\, a : ProblemAttribute \bullet a \in scheme\ strat \Leftrightarrow cor\ a \in scheme\ strat) \\
\quad \land\ (\exists_1\, crs : \mathbb{F}\ const\_rel \bullet \\
\qquad admissible\ crs \land determinable\ crs \land strat =\!\bowtie\ crs) \\
\quad \land\ (\forall\, res : strat \bullet \\
\qquad (\forall\, a : subprs\ strat \bullet (res\ (cor\ a))\ acceptable\_for\ (res\ a)) \\
\qquad \Rightarrow (res\ S\_final)\ acceptable\_for\ (res\ P\_init))
\end{array}
$$

This definition lays the theoretical foundation of our approach. Fig. 1 illustrates the definition of strategies, where arrows denote the propagation of attribute values.

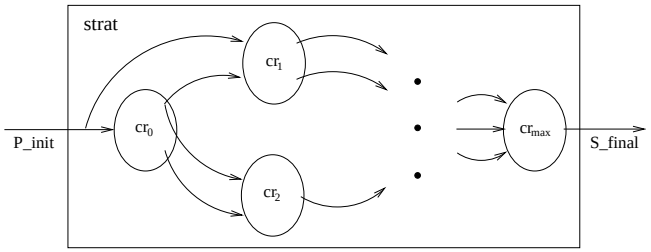

Fig. 1. Definition of strategies

2.2 Strategicals

Strategicals are functions that take strategies as their arguments and yield strategies as their result. They are useful to define higher-level strategies by combination of lower-level ones or to restrict the set of applicable strategies, thus contributing to a larger degree of automation of the development process.

Three strategicals are defined that are useful in different contexts. The THEN strategical composes two strategies. Applications of this strategical can be found in program synthesis. The REPEAT strategical allows for a stepwise repetition of a strategy. The wish for such a strategical arises in the context of specification acquisition where often several items of the same kind have to be developed. In order to make the REPEAT strategical more widely applicable, a LIFT strategical transforms a strategy to develop one item into a strategy to develop several items of the same kind. For reasons of space, we can only present THEN and REPEAT.

The THEN Strategical. This strategical has the following signature:

$$
\begin{array}{|l}
\textsc{Then} : strategy \times ProblemAttribute \times strategy \pfun strategy
\end{array}
$$

The effect of applying $\textsc{Then}(strat_1, p, strat_2)$ is the same as when reducing a problem first with $strat_1$ and then reducing the generated subproblem p by $strat_2$. The difference is that p and its corresponding solution $cor\ p$ are not generated explicitly. This is illustrated in Fig. 2. An example of a strategy defined with THEN is given in Sect. 4.2.

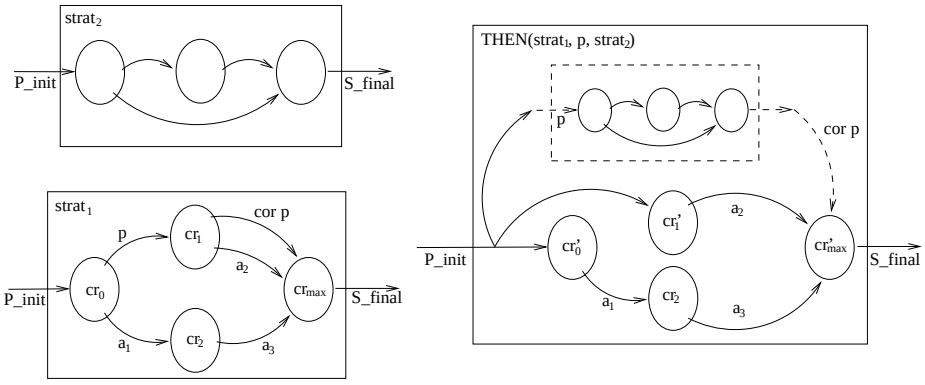

Fig. 2. THEN Strategical

The REPEAT **Strategical.** The first argument of this strategical is a strategy *strat* that is to be repeated. Repetition here means that a subproblem p generated by *strat* should again be reduced by a finite iteration of *strat*. The iteration is terminated by another strategy *terminate* that does not generate new subproblems.

$$\text{REPEAT} : \mathit{strategy} \times \mathit{ProblemAttribute} \times \mathit{strategy} \nrightarrow \mathit{strategy}$$

Hence, REPEAT$(\mathit{strat}, p, \mathit{terminate})$ is distinguished from *strat* only in some additional requirements concerning the reduction of p, as indicated in Fig. 3. An example of a strategy defined with REPEAT is given in Sect. 5.2.

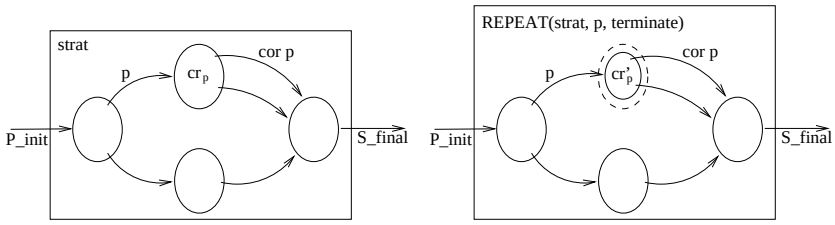

Fig. 3. REPEAT Strategical

2.3 Modular Representation of Strategies

To make strategies implementable, we must find a suitable representation for them that is closer to the constructs provided by programming languages than relations of database theory. Implementations of strategies should be independent of each other with a uniform interface between them. Thus, the implementation of a strategy is a module with a clearly defined interface to other strategies and the rest of the system. A strategy module consists of the following items:

- the names of subproblems it produces,
- the dependency relation on them,
- for each subproblem, a procedure how to set it up using the information in the initial problem and the subproblems and solutions it depends on, and possibly some externally provided information,
- a procedure describing how to assemble the final solution,
- a test of acceptability for the assembled solution, and
- optionally a procedure providing an explanation *why* a particular solution is acceptable.

The last item is not strictly necessary for a strategy to work. Still, one might be interested in a more detailed documentation of why a particular solution "works" for a given problem. This yields the following definition of a strategy module:

$$
\begin{array}{|l}
\hline
\mathit{StrategyModule} \\
\mathit{subprs} : \mathbb{P}\ \mathit{ProblemAttribute} \\
\mathit{depends} : \mathit{ProblemAttribute} \leftrightarrow \mathit{ProblemAttribute} \\
\mathit{setup} : \mathit{ProblemAttribute} \pfun (\mathit{tuple} \times \mathit{ExtInfo} \pfun \mathit{Problem}) \\
\mathit{assemble} : \mathit{tuple} \times \mathit{ExtInfo} \pfun \mathit{Solution} \\
\mathit{accept}\_ : \mathbb{P}\ \mathit{tuple} \\
\mathit{explain} : \mathit{tuple} \pfun \mathit{Explanation} \\
\hline
\end{array}
$$

The predicate part of this schema cannot be given here for reasons of space. Note that the dependency relation is now defined for single problems instead of constituting relations. It is possible for a combination of values for the input attributes of a constituting relation to be related to several values of the output attributes. In these cases, the basic type *ExtInfo* is used to select one of the possible values. This external information can be user input or be computed automatically. The basic type *Explanation* serves its obvious purpose.

3 System Architecture

The definition of strategies is parameterized by the notions of problem, solution, and acceptability. Therefore, it is possible to design a *generic* system architecture to support strategy-based development processes. Fig. 4 gives a general view of the architecture which is described in more detail in [HSZ95]. Two global data structures represent the state of development: the *development tree* and the *control tree*. The development tree represents the entire development that has taken place so far. Nodes contain problems, information about the strategies applied to them, and solutions to the problems as far as they have been found. Links between siblings represent dependencies on other problems or solutions.

The data in the control tree is concerned only with the future development. Its nodes represent open tasks. They point to nodes in the development tree that do not yet contain solutions. The degrees of freedom to choose the next problem to work on are also represented in the control tree. The third major component of the architecture is the strategy base. It represents knowledge for strategy-based problem solving by strategy modules.

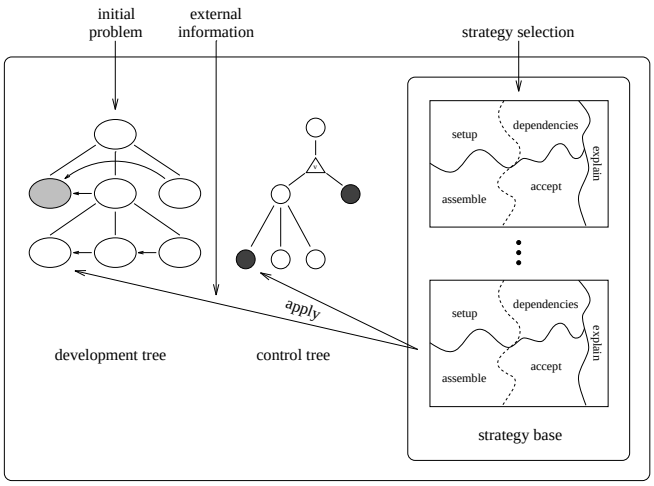

Fig. 4. General view of the system architecture

A development roughly proceeds as follows: the initial problem is the input to the system. It becomes the root node of the development tree. The root of the control tree is set up to point to this problem. Then a loop of strategy applications is entered until a solution for the initial problem has been constructed.

To apply a strategy, first the problem to be reduced is selected from the leaves of the control tree. Second, a strategy is selected from the strategy base. Applying the strategy to the problem means to extend the development tree with nodes for the new subproblems, install the functions of the strategy in these nodes, and set up dependency links between them. The control tree also is extended according to the dependencies between the produced subproblems.

If a strategy immediately produces a solution and does not generate any subproblems, or if solutions to all subproblems of a node in the development tree have been found, the functions to assemble and accept a solution are called, and, if successful, the solution is recorded in the respective node of the development tree. When a solution is produced the control tree shrinks because it only contains references to unsolved problems. The process terminates when the control tree vanishes. The result of the process is a development tree where all nodes contain acceptable solutions.

4 Instantiation for Program Synthesis

We present the instantiation of the framework as it is used for the implementation of IOSS, a system that supports the development of provably correct imperative programs. First, the generic parameters are instantiated. Then some example strategies are given. Finally, the implemented prototype system IOSS is described.

4.1 Problems, Solutions, Acceptability and Explanations

For the definition of the generic parameters, we also use a Z-like notation, without formalizing the syntax and semantics of formulas and programs, however.

Problems are specifications of programs, expressed as preconditions and postconditions that are formulas of first-order predicate logic. To aid focusing on the relevant parts of the task, the postcondition is divided into two parts, *invariant* and *goal*. In addition to these we have to specify which variables may be changed by the program (result variables), which ones may only be read (input variables), and which variables must not occur in the program (state variables). The state variables are used to store the value of variables before execution of the program for reference of this value in its postcondition. The function *free* yields the free variables of a formula. The predicate *valid* refers to the semantics of a formula and expresses its logical validity.

Solutions are programs in an imperative Pascal-like language. Furthermore, solutions contain additional pre- and postconditions. If the additional precondition is not equivalent to *true*, the developed program can only be guaranteed to work if both the originally specified and the additional precondition hold. The additional postcondition gives information about the behavior of the program, i.e. it says *how* the goal is achieved by the program. To exclude trivial solutions, the additional precondition is required not to be *false*.

ProgrammingProblem

$pre, goal, inv : First\_Order\_Formula$
$res, inp, state : \mathbb{P}\ Variable$

$\mathsf{disjoint}\ \langle res, inp, state \rangle$
$free(pre \wedge goal \wedge inv) \subseteq res \cup inp \cup state$
$valid(pre \Rightarrow inv)$

ProgSolution

$prog : Program$
$apr, apo : First\_Order\_Formula$

$satisfiable(apr)$

A solution is *acceptable* if and only if the program is totally correct with respect to both the original and the additional the pre- and postconditions, does not contain state variables (function *vars*), and does not change input variables (function *asg*). Checking for acceptability of a solution amounts to proving verification conditions on the constructed program.

$$\_correct\_for\_ : ProgSolution \leftrightarrow ProgrammingProblem$$

$$\forall\, p : ProgrammingProblem;\ s : ProgSolution \bullet$$
$$s\ correct\_for\ p$$
$$\Leftrightarrow (valid(p.pre \wedge s.apr \Rightarrow \langle s.prog \rangle (p.goal \wedge p.inv \wedge s.apo))$$
$$\wedge\ vars(s.prog) \cap p.state = \varnothing \wedge asg(s.prog) \cap p.inp = \varnothing)$$

The formula $pre \Rightarrow \langle prog \rangle post$ is a formula of dynamic logic [Gol82], a logic for proving properties of imperative programs. It denotes the total correctness of program *prog* with respect to precondition *pre* and postcondition *post*.

Explanations for solutions are provided as formal proofs in dynamic logic. In IOSS, proofs are represented as tree structures that can be inspected at any time during development.

4.2 Strategies for Program Synthesis

We present two strategies. The first one is used to develop compound statements; the second is a combined strategy using the THEN strategical that serves to develop loops together with their initialization.

The notation we use is semi-formal and resembles Z. The type *Value* denotes the disjoint union of the types *ProgrammingProblem* and *ProgSolution*.

The *protection* Strategy. This strategy is based on the idea that a conjunctive goal can be achieved by a compound statement. The part of the goal achieved by the first statement must be an invariant for the second one. It produces two subproblems and is defined as follows:

$protection = \bowtie \{prot\_first, prot\_second, prot\_sol\}$

where *prot_first* is defined by

$$
\begin{array}{l}
IA\ prot\_first = \{P\_init\} \\
OA\ prot\_first = \{P\_first, S\_first\} \\
prot\_first = \{\, t : scheme\ prot\_first \longrightarrow Value \mid \\
\quad \exists\, g_1, g_2 : First\_Order\_Formula \bullet \\
\qquad (valid(t(P\_init).goal \Leftrightarrow g_1 \wedge g_2) \wedge \\
\qquad t(P\_first) = \langle\, pre \Rrightarrow t(P\_init).pre, \\
\qquad\qquad\qquad goal \Rrightarrow g_1, \\
\qquad\qquad\qquad inv \Rrightarrow true, \\
\qquad\qquad\qquad res \Rrightarrow t(P\_init).res \cap free(g_1), \\
\qquad\qquad\qquad inp \Rrightarrow t(P\_init).inp \cup (t(P\_init).res \setminus free(g_1)), \\
\qquad\qquad\qquad state \Rrightarrow t(P\_init).state\,\rangle) \wedge \\
\quad t(S\_first)\ correct\_for\ t(P\_first)\}
\end{array}
$$

The precondition for the first statement is the same as for the original problem. The invariant may be invalidated in achieving goal g_1, hence the *inv* component of the value of *P_first* is *true*. Only the variables occurring free in g_1 may be changed; the other result variables of *P_init* become input variables for *P_first*. The state variables remain unchanged.

Note that there occurs an existential quantifier in this definition. This indicates that external information is necessary to set up the problem for *P_first*. In the implemented strategy of IOSS, the user is asked to indicate the goal for the first problem. *prot_second* is defined by

$$
\begin{array}{l}
IA\ prot\_second = \{P\_init, P\_first, S\_first\} \\
OA\ prot\_second = \{P\_second, S\_second\} \\
prot\_second = \{\, t : scheme\ prot\_second \longrightarrow Value \mid \\
\quad \exists\, g_2 : First\_Order\_Formula \bullet \\
\qquad (valid(t(P\_init).goal \Leftrightarrow t(P\_first).goal \wedge g_2) \wedge \\
\qquad t(P\_second) = \langle\, pre \Rrightarrow t(P\_first).goal \wedge t(S\_first).apo, \\
\qquad\qquad\qquad goal \Rrightarrow g_2 \wedge t(P\_init).inv, \\
\qquad\qquad\qquad inv \Rrightarrow t(P\_first).goal, \\
\qquad\qquad\qquad res \Rrightarrow t(P\_init).res, \\
\qquad\qquad\qquad inp \Rrightarrow t(P\_init).inp \cup (free(t(S\_first).apo) \\
\qquad\qquad\qquad\qquad \setminus (t(P\_init).res \cup t(P\_init).state)), \\
\qquad\qquad\qquad state \Rrightarrow t(P\_init).state\,\rangle) \wedge \\
\quad t(S\_second)\ correct\_for\ t(P\_second) \wedge
\end{array}
$$

$$valid(t(P\_first).goal \wedge t(S\_first).apo \Rightarrow t(S\_second).apr)\}$$

In this case, the goal for *P_second* can be determined automatically. It consists of that part g_2 of the original goal which was not achieved by solving the problem *P_first*, together with the invariant of *P_init*. The invariant for *P_second* is the goal of *P_first*; its precondition also is this goal, together with the additional postcondition established by *S_first*.

The result variables for *P_second* are the same as for the original problem. Its input variables are the input variables of *P_init* plus all variables newly introduced in solving *P_first* (these will occur in $t(S\_first).apo$). It is necessary to classify these variables because of the integrity condition $free(pre \wedge goal \wedge inv) \subseteq res \cup inp \cup state$ stated in the definition of programming problems.

The state variables again remain unchanged. The solution *S_second* is not only required to be acceptable for *P_second*. It must also be guaranteed that its additional precondition is entailed by the postcondition established by *S_first*.

The constituting relation *prot_sol* defines how the final solution is assembled from the solutions of the subproblems, where the final program is the sequential composition of the two programs developed in solving the subproblems.

$$\begin{array}{l} IA\ prot\_sol = \{S\_first, S\_second\} \\ OA\ prot\_sol = \{S\_final\} \\ prot\_sol = \{\, t : scheme\ prot\_sol \longrightarrow Value \mid \\ \quad t(S\_final) = \langle\, prog \Rrightarrow t(S\_first).prog;\ t(S\_second).prog, \\ \qquad\qquad apr \Rrightarrow t(S\_first).apr, \\ \qquad\qquad apo \Rrightarrow t(S\_second).apo \rangle\, \} \end{array}$$

A Combined Strategy. The *protection* strategy is frequently used for the development of *while* loops. This usually takes place in the following manner: first, a loop invariant is developed, e.g. using the heuristics given in [Gri81]. The goal of the original problem is *strengthened* using the *strengthening* strategy. This strategy replaces the goal of a programming problem by a stronger or equivalent one. The new goal consists of the loop invariant and the negation of the loop condition. Second, the *protection* strategy is applied. The first statement of the compound is the initialization of the loop that establishes the invariant. The second part of the compound consists of the loop itself which is developed with a strategy called *loop*. To define a new *while* strategy that encompasses these steps, the THEN strategical can be used:

$$while = \textsc{Then}(strenthening, P\_str, \textsc{Then}(protection, P\_second, loop))$$

where *P_str* is the only subgoal generated by the *strengthening* strategy.

This shows that strategicals are a means to improve user support by making larger development steps possible. More strategies for program synthesis can be found in [Hei94].

4.3 IOSS: An Implemented Program Synthesis System

IOSS is an instantiation of the architecture described in Sect. 3. It uses the instantiation given in Sect. 4.1. The basis for the implementation of IOSS is the *Karlsruhe Interactive Verifier* (KIV), a shell for the implementation of proof methods for imperative programs [HRS88]. It provides a functional *Proof Programming Language* (`PPL`) with higher-order features and a backtrack mechanism. Strategies are implemented as collections of `PPL` functions in separate modules. New strategies can be incorporated in a routine way. Currently a template file for new strategies supports this process; for the future, we envision tool support relieving the implementor of anything but the peculiarities of the newly implemented strategy. The graphical user interface of IOSS (see Fig. 5) is written in `tcl/tk` [Ous94] and integrates the graph visualization system `daVinci` [FW95] to display the development tree.

Fig. 5 shows the graphical user interface of IOSS. The main window displays

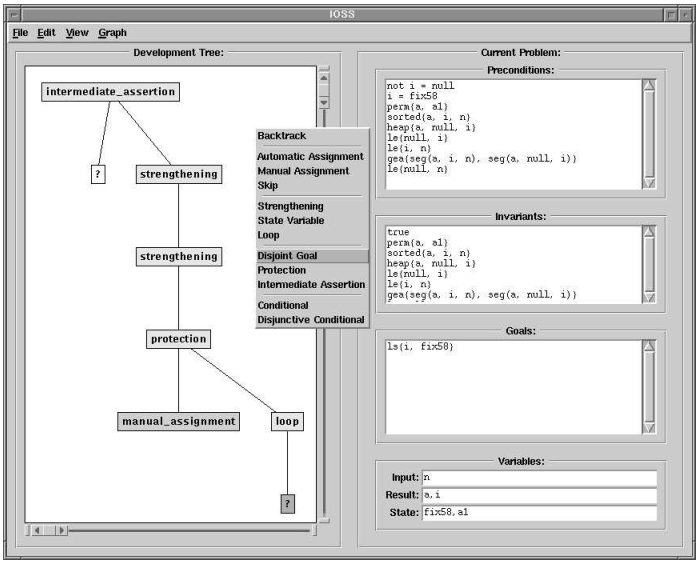

Fig. 5. The IOSS interface

the development task, represented by the development tree on the left-hand side of the window, and the current programming problem on the right-hand side of the window. The tree visualizes the process and the state of development. Each node is labeled with the name of the strategy applied to it. The state of the node is color coded, showing at a glance whether it is reducible, or solved, etc. The strategy menu is shown in the center of the window. Applications of strategies, inspection of nodes or the proof tree and graph manipulations like scaling are performed via mouse clicks or pull-down menus. For a more complete description of IOSS, the reader is referred to [HSZ95].

5 Instantiation for Specification Acquisition

The instantiation we present serves to develop specifications in Z. This fits well with the instance for IOSS since Z supports the explicit modeling of states. Z specifications will usually be implemented in an imperative language, like the one used in IOSS, and Z operation schemas can easily be transformed into programming problems of IOSS [Hei96].

5.1 Problems, Solutions, and Acceptability

In contrast to program synthesis where problems and solutions are purely formal objects, specification acquisition transforms informal requirements into formal specifications. Hence, problems contain natural language descriptions of the purpose of the specification to be developed.

On the other hand, in order to develop a specification successively, one must know the parts of the specification that are already developed. Since problems should contain all information needed to solve them, problems must contain expressions of the chosen specification language, in our case Z.

Moreover, a problem contains a *schematic* Z expression that can be instantiated with an appropriate concrete Z expression. It specifies the syntactical class of the piece of specification to be developed and how it is embedded in its context. These considerations lead us to the basic types $[SynZ, Text, SchematicZ]$.

Semantically valid Z specifications are a subset of the syntactically correct ones: $SemZ : \mathbb{P}\, SynZ$. Z expressions can be associated with syntactical classes that are sets of Z expressions, e.g. *specification* or *schema*. $SyntacticalClass : \mathbb{P}(\mathbb{P}\, SynZ)$. The empty string ϵ is a syntactically correct Z expression.

Each schematic Z expression is associated with the syntactical class of Z expressions that it can be instantiated with. The function `NL` concatenates two Z expressions. In analogy to the Z reference manual, it can be interpreted to mean "new line". Since concatenating two arbitrary Z expressions does not always yield a syntactically correct Z expression this function is partial.

$$
\begin{array}{l}
syn\_class : SchematicZ \longrightarrow SyntacticalClass \\
instantiate : SchematicZ \times SynZ \nrightarrow SynZ \\
\_\mathtt{NL}\_ : SynZ \times SynZ \nrightarrow SynZ \\
\hline
\forall\, schem\_expr : SchematicZ \bullet \forall\, v : syn\_class\, schem\_expr \bullet \\
\quad (schem\_expr, v) \in \mathrm{dom}\, instantiate
\end{array}
$$

A specification problem consists of the parts mentioned above, where it is required that each Z expression belonging to the desired syntactical class can be combined with the Z part of the problem.

$$
\begin{array}{l}
SpecProblem \\
\hline
req : Text \\
context : SynZ \\
to\_develop : SchematicZ \\
\hline
\forall\, expr : SynZ \mid expr \in syn\_class\, to\_develop \bullet \\
\quad (context, instantiate(to\_develop, expr)) \in \mathrm{dom}(\_\mathtt{NL}\_)
\end{array}
$$

Solutions are Z expressions: $SpecSolution == SynZ$. A solution s is acceptable with respect to a problem p if and only if it belongs to the syntactical class of $p.to\_develop$ and the combination of $p.context$ with the instantiated schematic expression yields a semantically valid Z specification.

$$\_spec\_acceptable\_for\_ : SpecSolution \leftrightarrow SpecProblem$$

$$\begin{aligned}
&\forall\, s : SpecSolution; p : SpecProblem \bullet \\
&\quad s\ spec_acceptable_for\ p \\
&\quad \Leftrightarrow\ s \in syn_class(p.to_develop) \wedge \\
&\qquad p.context\ \mathtt{NL}\ instantiate(p.to_develop, s) \in SemZ
\end{aligned}$$

5.2 Strategies for Specification Acquisition

We present two strategies: the *state_based* strategy that captures the top-level methodology of the specification language Z; and a strategy to develop lists of schemas that is based on a strategy called *develop_schema* and is defined using the strategicals LIFT and REPEAT.

Again, we use a semi-formal Z-like notation without formalizing the syntax and semantics of Z and giving definitions for all the used functions and predicates.

The *State_Based* strategy. The Z methodology recommends to start with the global definitions, then to define the system state and the operations on the state. Finally, it may be necessary to make some more definitions in order to complete the specification. Since this is a top-level strategy, the given problem must admit to develop expressions of the syntactical class *specification*.

$$\begin{aligned}
state_based = \bowtie \{&global_defs, system_state, system_ops, other_defs, \\
&state_based_sol\}
\end{aligned}$$

where *global_defs* is defined by

$$\begin{aligned}
&IA\ global_defs = \{P_init\} \\
&OA\ global_defs = \{P_global, S_global\} \\
&global_defs = \{\, t : scheme\ global_defs \longrightarrow Value \mid \\
&\quad syn_class(t(P_init).to_develop) = specification \wedge \\
&\quad t(P_global) = \langle\, req \Rrightarrow t(P_init).req\,;\text{specify global definitions}, \\
&\qquad\qquad context \Rrightarrow t(P_init).context \\
&\qquad\qquad to_develop \Rrightarrow \mathsf{sp} : specification\rangle \wedge \\
&\quad t(S_global)\ spec_acceptable_for\ t(P_global)\}
\end{aligned}$$

Using the concatenation function ; for text, a natural-language description of the problem is added to the informal requirements. The schematic expression *to_develop* is given as a metavariable sp together with its syntactical class *specification*. The constituting relation *system_state* is defined by

$$\begin{aligned}
&IA\ system_state = \{P_init, S_global\} \\
&OA\ system_state = \{P_state, S_state\} \\
&system_state = \{\, t : scheme\ system_state \longrightarrow Value \mid \\
&\quad t(P_state) = \langle\, req \Rrightarrow t(P_init).req\,;\text{specify global system state}, \\
&\qquad\qquad context \Rrightarrow (t(P_init).context)\,\mathtt{NL}\,t(S_global) \\
&\qquad\qquad to_develop \Rrightarrow \mathsf{state_def} : schema_list\rangle \wedge \\
&\quad t(S_state)\ spec_acceptable_for\ t(P_state) \wedge \\
&\quad t(S_state) \neq \epsilon\}
\end{aligned}$$

To define *P_state*, the global definitions *S_global* are added to the context. The system state must be defined as a non-empty list of schemas. The constituting relation *system_ops* and *other_defs* are defined similarly. The operations are again required to be a non-empty list of schemas. On the other necessary definitions, no assumptions can be made.

The constituting relation *state_based_sol* assembles the final solution and states acceptability conditions that can only be checked when all partial solutions are known.

$IA\ state\_based\_sol = \{S\_global, S\_state, S\_ops, S\_other\}$
$OA\ state\_based\_sol = \{S\_final\}$
$state\_based\_sol = \{\ t : scheme\ state\_based\_sol \longrightarrow Value \mid$
$t(S\_final) = t(S\_global)\ \mathtt{NL}\ t(S\_state)\ \mathtt{NL}\ t(S\_ops)\ \mathtt{NL}\ t(S\_other) \wedge$
$t(S\_global)$ does not contain state or operation schemas $\wedge$
$t(S\_state)$ contains a state schema S that is not imported by any other schema in $t(S\_state)$ and an initial schema for $S \wedge$
$t(S\_ops)$ contains at least one operation schema $\wedge$
none of the operations defined in $t(S\_ops)$ has precondition *false*$\}$

A schema S is a *state schema* if it has neither inputs nor outputs and there are other schemas importing it. There must not be declarations of the kind $x : S$. Note that this can be checked only in context with the other parts of the specification. A schema is an operation schema if it imports a state schema with the $'$, Δ or Ξ convention.

An Iterative Strategy. The second and third subproblems generated by the *state_based* strategy can be solved by repeated application of a strategy *define_schema* that serves to develop a single schema. For this purpose, we define a new strategy that generates lists of schemas instead of just one schema, using the strategical LIFT. According to the definitions of Sect. 2.2, we define

$$define\_schema\_list = \textsc{Repeat}(\textsc{Lift}(define\_schema), p\_rep, empty)$$

where *p_rep* is a problem attribute newly introduced by LIFT and *empty* is the terminating strategy that generates the empty specification ϵ.

6 Comparison of the Two Instantiations

The two instantiations of the strategy framework presented in Sects. 4 and 5 are distinguished in several important aspects. The differences show up in the following phenomena:

Instantiation of the generic parameters. Program synthesis performs the transition from a formal specification to a program. Both are formal objects, and hence the definition of acceptability can establish a formal relation between the two, namely correctness.

In specification acquisition, this is impossible because the description of the requirements is informal. Hence, the definition of acceptability can only refer to the formal specification in isolation and not to the requirements. In contrast to program synthesis, where all partial solutions are statements, the partial solutions in specification acquisition belong to different syntactical classes. This

makes it more difficult to define a general notion of acceptability. For individual strategies, stronger acceptability conditions than the general acceptability predicate can be stated, see e.g. the *state_based* strategy. These conditions reflect the purpose of the different parts of the specification in the context of a strategy, e.g. that the global definitions should not define the system state and that system operations should have a satisfiable precondition. Also other consistency and completeness criteria can be stated in the context of appropriate strategies.

Independent subproblems. In program synthesis, it is not uncommon that the subproblems generated by a strategy are independent of each other. For example, when developing a conditional, the two branches can be developed in any order or in parallel.

Specification acquisition, on the other hand, proceeds much more incrementally. Usually, later parts of the specification refer to the earlier parts. For example, in order to define the operations of a system, its state must be known. So far, none of the strategies defined for specification acquisition contains independent subproblems.

Working with incomplete solutions. The fact that subproblems in specification acquisition strongly depend on each other has yet another consequence. Experience has shown that it is unrealistic to assume that, if problem P_2 depends on the solution S_1 of problem P_1, it is possible to first solve P_1 completely and only then start working on P_2. This means that the process that implements problem solving with strategies must allow specifiers to work on a problem even if the solution it depends on is not yet completely known. Technically, this can be achieved by executing the *assemble* functions contained in the strategy modules (see Sect. 2.3), where for solutions not yet developed some dummy is used. As soon as a change in an earlier problem/solution occurs, the *assemble* functions must be executed once more. When a subproblem is finally solved, both the *assemble* and *accept* functions must be executed.

In program synthesis, such a feature would make the problem solving process more flexible and comfortable. However, it is not necessary to make strategy-based program synthesis feasible.

Use of repetition. Frequently, in specification acquisition, several items of the same kind must be developed to solve a problem. In this case, one might want to repeat the same strategy several times. This can be supported by the strategicals REPEAT and LIFT, as described in Sect. 5.2.

For program synthesis, a repetition of the same strategy is not as useful. To develop a program, it does not help to consider it as a concatenation of items of the syntactic class *statement*. This is due to the fact that programming problems provide much more detailed and semantical information than specification problems because they are formal. Their syntactical form may already suggest the strategy to apply. Consequently, strategy selection can rely more on the specific problem in program synthesis than in specification acquisition.

These considerations show that program synthesis and specification acquisition are fairly different activities. However, strategies are general enough to support them both.

7 Related Work

Our work relates to knowledge representation techniques and process modeling in classical software engineering, program synthesis and tactical theorem proving.

Representation of Design and Process Knowledge. In the Programmer's Apprentice project [RW88], programming knowledge is represented by *clichés*, i.e. prototypical examples of the artifacts in question. It may be difficult to set up a sufficiently complete cliché library that does not need to be extended for each new problem.

Wile [Wil83] and others [Ost87,SSW92] vote for a procedural representation of software development processes. This has the disadvantage to enforce a strict depth-first left-to-right processing.

Potts [Pot89] aims at capturing not only strategic but also heuristic aspects of design methods. He uses *Issue-based Information Systems* as a representation formalism. These tend to be specialized for a particular application domain.

Souquières and Lévy [SL93] have developed an approach to specification acquisition whose underlying concepts have much in common with the ones presented here. Specification acquisition is performed by solving *tasks*. The agenda of tasks is called a *workplan* and resembles our development tree. Tasks can be reduced by *development operators* similar to strategies. Development operators, however, do not guarantee semantic properties of the product.

In the German project KORSO [BJ95], the product of a development is described by a *development graph*. Its nodes are specification or program modules whose static composition and refinement relations are expressed by two kinds of vertices. There is no explicit distinction between "problem nodes" and "solution nodes". The KORSO development graph does not reflect single development steps, and dependencies between subproblems cannot be represented.

Program Synthesis. The strategy framework in general and IOSS in particular make it possible to integrate a variety of methods which can be expressed in its basic formalism. The synthesis systems CIP [CIP87], PROSPECTRA [HKB93] and LOPS [BH84], in contrast, are all designed to support specific methods. It is not intended to integrate these methods with other ones.

The approach underlying KIDS [Smi90] is to fill in algorithm schemas by constructive proof of properties of the schematic parts. This is achieved by highly specialized code (*design tactics*) for each schema. There is no general concept of design tactics or how to incorporate a new one into the system.

Tactical Theorem Proving. Tactical theorem proving has first been employed in Edinburgh LCF [Mil72]. *Tactics* are programs that implement "backward" application of logical rules. Tactical theorem proving is also used in the generic interactive theorem prover Isabelle [Pau94], in the verification system PVS [Dol95], and in KIV [HRS88], the theorem proving shell underlying IOSS.

The goal-directed, top-down approach to problem solving is common to tactics and strategies. Nevertheless, there are some important differences. First, a tactic is one monolithic piece of code. All subgoals are set up at its invocation. Dependencies between subgoals can only be expressed schematically by the use

of *metavariables*. Since tactics only perform goal reduction, there is no equivalent to the *assemble* and *accept* functions of strategies.

Another important difference concerns the roles of search, and tacticals or strategicals, respectively. In tactical theorem proving, proof search is promising because the theorem is known and need not be constructed. The purpose of strategy-based development, on the other hand, is to construct an artifact of the software development process in the first place. This makes search a hopeless enterprise. Consequently, the OR and FAIL tacticals that are used to program search are unnecessary in the context of strategy-based development. The REPEAT construct is realized differently in the two frameworks. While in search procedures, a "real" loop construct is necessary, the REPEAT strategical performs only one step of a loop; its purpose is to impose restrictions on the strategies to be applied. Only the THEN tactical or strategical is useful in both cases since it allows one to perform larger steps in a proof or a development. We conclude that the two activities — even though based on similar ideas — are quite different in their practical application.

Apart from these conceptual differences, there are differences in the kind of user support tactical theorem provers provide. Theorem proving systems like Isabelle or PVS usually do not maintain a data structure equivalent to the development tree. It is the users' responsibility to record their proof steps textually outside of the system.

8 Conclusions

We have demonstrated that strategies are a suitable concept for the representation of development knowledge. They aim at *methodological* support, in contrast to other tools that deal with single documents and not with the process aspect of a development. By making explicit not only dependencies but also independencies of problems, strategies allow for the greatest possible flexibility in the development process. Other tools enforce one fixed way of procedure on their users (see Sect. 7).

The generic nature of strategies makes it possible to support quite different development activities, like specification acquisition and program synthesis. Strategicals contribute to the scalability of the approach. The uniform representation as strategy modules makes strategies implementable and isolates those parts that are responsible for acceptability and the ones that can be subject to automation.

The generic system architecture that complements the formal strategy framework gives guidelines for the implementation of support systems for strategy-based development. The representation of the state of development by the data structure of development trees contributes essentially to the practical applicability of the strategy approach. This approach fulfills the requirements stated in Sect. 1 in the following way.

Usability by non-experts. A necessary prerequisite for the successful work with strategies is the familiarity with the involved formalisms. To use the instantiation

for specification acquisition, a good knowledge of the Z language is necessary. To develop programs with IOSS, the user should be familiar with Gries' method to develop correct programs [Gri81]. It is not required, however, to be a researcher in the area of formal methods in order to profitably apply strategies.

Guarantee Semantic Properties. The function *accept* is the only component of the interface of a strategy module that is concerned with semantic properties. This enhances confidence in the development tool because only the *accept* functions have to be verified to ensure that the tool truly guarantees acceptability of the produced solutions.

Balance User Guidance and Flexibility. Methods are uniformly represented as sets of strategies. Their common interface to the system kernel makes method combination possible. To incorporate a new method into the system, the strategy base only has to be extended by the new strategies. This involves only local changes that do not affect existing components.

More work is necessary if the notions of problem, solution or acceptability have to be changed. In this case, all strategies have to be revised, but the clear modularization still helps in identifying the code that has to be changed.

Provide Overview of Development. By maintaining the open subproblems and their dependencies in the development tree, we not only get an overview of the state of the development, but the entire development is mirrored in this data structure.

In the future, we want to gain more experience in expressing methods dealing with the different phases of the software life cycle as strategies, e.g. requirements engineering or maintenance. This will enhance understanding of the requirements of efficient and extensive tool support.

For now, different instantiations of the strategy framework lead to different isolated support systems. It will be investigated how different instances of the system architecture can be combined. This would provide integrated tool support for larger parts of the software lifecycle.

Acknowledgment. Thanks to Thomas Santen for his untiring willingness to discuss strategies.

References

[BH84] W. Bibel and K. M. Hörnig. LOPS – a system based on a strategical approach to program synthesis. pages 69–89, 1984.

[BJ95] M. Broy and S. Jaehnichen, editors. *KORSO: Methods, Languages, and Tools to Construct Correct Software*. LNCS 1009. Springer Verlag, 1995.

[CIP87] CIP System Group. *The Munich Project CIP. Volume II: The Program Transformation System CIP-S*. LNCS 292. Springer-Verlag, 1987.

[Dol95] Axel Dold. Representing, verifying and applying software development steps using the PVS system. In V.S. Alagar and Maurice Nivat, editors, *Proc. 4th AMAST*, LNCS 936. Springer-Verlag, 1995.

[FW95] M. Fröhlich and M. Werner. Demonstration of the interactive graph-visualization system. In *Proc. DIMACS Workshop on Graph Drawing*, LNCS. Springer-Verlag, 1995.

[Gol82] R. Goldblatt. *Axiomatising the Logic of Computer Programming.* LNCS 130. Springer-Verlag, 1982.

[Gri81] David Gries. *The Science of Programming.* Springer-Verlag, 1981.

[Hei94] Maritta Heisel. A formal notion of strategy for software development. Technical Report 94–28, Technical University of Berlin, 1994.

[Hei96] Maritta Heisel. An approach to develop provably safe software. *High Integrity Systems*, 1996. to appear.

[HKB93] B. Hoffmann and B. Krieg-Brückner, editors. *PROgram Development by SPECification and TRAnsformation, the PROSPECTRA Methodology, Language Family and System.* LNCS 680. Springer-Verlag, 1993.

[HRS88] Maritta Heisel, Wolfgang Reif, and Werner Stephan. Implementing verification strategies in the KIV system. In E. Lusk and R. Overbeek, editors, *Proc. 9th CADE*, LNCS 310, pages 131–140. Springer-Verlag, 1988.

[HSZ95] Maritta Heisel, Thomas Santen, and Dominik Zimmermann. Tool support for formal software development: A generic architecture. In W. Schäfer and P. Botella, editors, *Proc. 5-th European Software Engineering Conf.*, LNCS 989, pages 272–293. Springer-Verlag, 1995.

[Kan90] Paris C. Kanellakis. Elements of relational database theory. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 17, pages 1073–1156. Elsevier, 1990.

[Mil72] Robin Milner. Logic for computable functions: description of a machine implementation. *SIGPLAN Notices*, 7:1–6, 1972.

[Ost87] Leon Osterweil. Software processes are software too. In *9th Intl. Conf. on Software Engineering*, pages 2–13. IEEE Computer Society Press, 1987.

[Ous94] John K. Ousterhout. *Tcl and the Tk Toolkit.* Addison-Wesley, 1994.

[Pau94] L. C. Paulson. *Isabelle.* LNCS 828. Springer-Verlag, 1994.

[Pot89] Colin Potts. A generic model for representing design methods. In *Intl. Conf. on Software Engineering*, pages 217–226. IEEE Computer Society Press, 1989.

[RW88] Charles Rich and Richard C. Waters. The programmer's apprentice: A research overview. *IEEE Computer*, pages 10–25, November 1988.

[SL93] Jeanine Souquières and Nicole Lévy. Description of specification developments. In *Proc. of Requirements Engineering '93*, pages 216–223, 1993.

[Smi90] Douglas R. Smith. KIDS: A semi-automatic program development system. *IEEE Trans. on Software Engineering*, 16(9):1024–1043, Sept. 1990.

[Spi92] J. M. Spivey. *The Z Notation – A Reference Manual.* Prentice Hall, 2nd edition, 1992.

[SSW92] Terry Shepard, Steve Sibbald, and Colin Wortley. A visual software process language. *Communications of the ACM*, 35(4):37–44, April 1992.

[Wil83] David S. Wile. Program developments: Formal explanations of implementations. *Communications of the ACM*, 26(11):902–911, November 1983.