Strategies – A Generic Knowledge Representation Mechanism for Software Development Activities

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

Abstract

This paper introduces a knowledge representation called *strategy* designed to support the application of formal methods in software engineering. Strategies represent development knowledge used to perform different software engineering activities. The development of an artifact is modeled as a problem solving process. An important goal is to guarantee semantic properties of the developed product. Strategies support stepwise automation of development tasks. Since the definition of strategies is generic, they can be employed in different phases of the software lifecycle. 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

All efforts to automate software engineering activities and to reuse previously gained experience must be based on a representation of the knowledge used by software engineers. This representation must be easily implementable on machines and be complemented by some process model that describes how to make use of the represented knowledge.

In this paper, we present such a knowledge representation mechanism, called *strategy*. Strategies are specifically designed to support the application of formal methods in software engineering. Formal methods make it possible to guarantee *semantic* properties of the developed product (this may be a specification, a design, a program, test cases, or the like). This is in contrast to CASE tools that usually do not take semantic issues into account. The concept of strategy is independent of a particular formalism.

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 is usually described in text books. In contrast, the ability to decide which strategy may successfully be applied in a particular situation requires human intuition and a deep understanding of the problem at hand. While these heuristics are hardly mechanizable, strategies can be implemented.

The basic idea underlying strategies is to conceive software engineering activities as problem solving processes. For some development problem, an *acceptable* solution has to be constructed. The notion of acceptability captures the semantic requirements the developed product has to fulfill. The notion of strategy is *generic* in the definition of problems, solutions, and acceptability. This means that strategies can be used to formalize a variety of software development activities, two of which are presented in this paper.

In problem solving with strategies, problems are solved by reduction to a number of subproblems that are in turn solved by application of strategies. This process terminates when the generated subproblems are so simple that they can be solved directly.

The use of strategies to support software engineering activities has the following advantages:

- Development methods formalized by strategies can be combined freely and be enhanced, changed and adapted to special project contexts in a routine way.
- *Strategicals* provide ways to define more powerful strategies by combination of existing ones.
- The parts of a strategy that are responsible to guarantee acceptability of the developed solution are well isolated. Only these parts have to be verified to obtain trustworthy support systems.

As already mentioned, merely representing development knowledge does not suffice. The knowledge representation mechanism must therefore be complemented by concepts for the machine supported application of this knowledge. For strategies, this is achieved as follows:

- We give a modular representation of strategies that easily maps to encapsulation mechanisms of modern programming languages.
- An abstract problem solving algorithm describes how development activities with strategies can be carried out by machine (where appropriate user interaction will be necessary).
- The parts of the algorithm where user interaction can be replaced by automatic procedures are clearly identified, making stepwise automation possible.
- A generic system architecture provides detailed concepts for the implementation of support systems for strategy-based problem solving.

We formally define strategies, strategicals, strategy modules and the abstract problem solving algorithm in the language Z [Spi92b]. This provides precise definitions of these notions and supports reasoning about strategies.

We describe instances of the strategy framework supporting the development activities of specification acquisition and program synthesis. The prototype system IOSS (Integrated Open Synthesis System) is an implementation of the instance for program synthesis. Its architecture is an instance of the proposed system architecture.

Different support systems implementing different instantiations of the strategy framework have a strong potential for successful combination. Such a combination can provide integrated tool support for different software development activities.

The rest of the paper is organized as follows: After giving examples of the kind of knowledge that can be expressed by strategies in Section 2, we present a formal definition of strategies in the specification language Z in Section 3. Section 4 introduces strategicals that can be used to define more powerful strategies from simpler ones. Steps toward an implementation of strategies are taken in Section 5. The system architecture described in Section 6 further elaborates the implementation concepts. An instantiation of the framework supporting program synthesis is presented in Section 7, together with a description of the implemented system IOSS. Section 8 presents an instantiation for specification acquisition. We are then able to compare the two instantiations in Section 9, and to compare strategy-based problem solving with tactical theorem proving and other related work (Section 10). Finally, we summarize in Section 11.

2 Knowledge Formalizable by Strategies

Strategies describe established ways of procedure that can be used to tackle a given problem. They give hints how to proceed, but they cannot guarantee that the problem is solved successfully in every case. Their aim is not to trivialize problems (by solving them fully automatically) but to give guidance and keep track of what remains to be done to fully solve a problem.

We illustrate the kind of knowledge that can be represented by strategies by way of examples from specification acquisition and program development.

2.1 Developing Z Specifications

One of the factors that contribute to the relatively good acceptance of Z in industry is the existence of a methodology [PST91] that gives guidance for its use. This methodology recommends to proceed in the following way when developing Z specifications:

1. develop the global definitions
2. develop the global state and the initial state
3. develop the system operations:
   (a) develop the operations for the normal case
   (b) develop the operations for error cases
   (c) define total operations, combining the operations for the normal and the error cases

When we view the development of a specification as a problem solving process, then the above steps constitute subproblems that all have to be solved to obtain the final solution. The following questions arise:

- Are the subproblems independent of each other?
  In our example, they are certainly not, because we can only define the system operations when we know on which state they operate.
- How are the solutions of the subproblems combined to form the solution to the original problem?
  In our example, they are simply concatenated. They next example will show that this is not always the case.
- What conditions must the solutions to the subproblems fulfill, so that the final solution is acceptable?
  In our example, we require that the operations refer to the state defined earlier and that they do not use any global definitions that are not contained in the global definitions part of the specification.

In formalizing such ways of procedure as strategies, all these questions will be given precise answers.

2.2 Developing While Loops

Gries' approach to the development of correct programs [Gri81] mostly deals with the development of loops. For **while** loops, the approach can be summarized as follows. Given a precondition P and a postcondition R,

1. develop a loop invariant I by weakening the postcondition R appropriately
2. develop a loop condition C such that $\neg\ C \wedge I \Rightarrow R$
3. develop the initialization *init* of the loop such that it establishes the invariant, starting from the precondition P
4. develop a bound function *bf* on a set with a well-founded ordering such that *bf* is not minimal as long as C holds
5. develop the loop body *body* such that the bound function is decreased while the invariant is maintained

This procedure is also formalizable as a strategy. However, it is not as clear as in the previous example what the subproblems are. Since, in program synthesis, problems have to do with the development of programs from specifications, we have only two subproblems here: the development of the initialization and the development of the loop body.

The conditions I and C and the function bf are developed "on the side". We call this information *external information* because it is just an input for the problem solving process that is needed to set up the specifications for the programs *init* and *body*. How this input is obtained is not part of the definition of a strategy to develop **while** loops. One possibility to obtain the external information is to ask the user (of an implemented support system for strategy-based problem solving). Another possibility is to try to compute it automatically. A (semi-) automatic procedure could implement the heuristics that Gries gives for the development of loop invariants from postconditions. It is also possible to ask the user in a first version of a system and then gradually replace user interaction by automatic procedures. Strategies provide the necessary separation of concerns.

The developed parts are assembled to yield the final solution *init*; **while not** C **do** *body* **od**. The conditions that were given in the problem description, namely that the initialization establishes the loop invariant, that the loop body decreases the bound function while maintaining the invariant, and that the loop invariant and the negation of the loop condition entail the postcondition, guarantee that the developed loop is totally correct with respect to the given pre- and postconditions.

The two examples presented here give an impression of what kind of software development knowledge can be represented by strategies. Strategies can give no guarantee for success. The subproblems generated by a strategy need not be independent. The solutions to subproblems usually must fulfill certain conditions so as to guarantee that the initial problem is solved. For different application areas, problems and solutions may look different. External information is used to represent inputs that are needed to define problems or solutions. The process of obtaining external information can be subject to gradual automation, thus automating the whole problem solving process.

3 Formal Definition of Strategies

Strategies describe possible steps during a development. 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 define the notion of relation that will be used to define strategies. Second, we introduce *constituting relations*, the building blocks of strategies. Finally, strategies are defined as sets of constituting 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 [Spi92b]. A summary of the Z notation used in this paper is given in Appendix B.

3.1 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]. 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* to denote attribute values.

Having introduced *Attribute* and *Value* as basic types, we can define tuples as finite partial functions from attributes to values, where $\mathbb{P}$ is the powerset operator:

$$\begin{array}{|l} tuple : \mathbb{P}(Attribute \mathrel{+\!\!\!\!\!\!\;+\!\!\!\!\!\rightarrow} Value) \end{array}$$

Relations are sets of tuples that all have the same domain. This domain is called the *scheme* of the relation. Note that in Z function applications are written without parentheses.

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

Domain restriction and domain subtraction as they are used for the usual notion of relation are also needed for relations of database theory.

$$(\_\lhd_r\_) == (\lambda\, attrs : \mathbb{F}\, Attribute;\ r : relation \bullet (\{t : r \bullet attrs \lhd t\}))$$
$$(\_\ntriangleleft_r\_) == (\lambda\, attrs : \mathbb{F}\, Attribute;\ r : relation \bullet \{t : r \bullet attrs \ntriangleleft t\})$$

Here, $\lhd$ restricts the domain of a relation to its left argument, and $\ntriangleleft$ subtracts its left argument from the domain of the relation, see Appendix B. The operator $\mathbb{F}$ denotes final sets.

A *join* is a total function combining 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.

$$\begin{array}{|l} \_ \bowtie \_ : relation \times relation \longrightarrow relation \\ \hline \forall\, r_1, r_2, r : relation \bullet \\ \qquad r_1 \bowtie r_2 \\ \qquad = \{t : tuple \mid \operatorname{dom} t = scheme\ r_1 \cup scheme\ r_2 \wedge \\ \qquad\qquad\qquad scheme\ r_1 \lhd t \in r_1 \wedge scheme\ r_2 \lhd t \in r_2\} \end{array}$$

The join operation is associative and commutative. Hence, the join can also be defined for finite sets of relations. The definition of this operation, denoted $\bowtie$, is straightforward and not presented here.

3.2 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*. Acceptability is a relation between problems and solutions.

$$\begin{array}{|l} \_acceptable\_for\_ : Solution \longleftrightarrow Problem \end{array}$$

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* between problem and solution attributes.

$$\begin{array}{|l} P\_init : ProblemAttribute \\ S\_final : SolutionAttribute \\ cor : ProblemAttribute \rightarrowtail\!\!\!\!\!\twoheadrightarrow SolutionAttribute \\ \hline cor\ P\_init = S\_final \end{array}$$

3.3 Constituting Relations

Each strategy will be defined by as set of *constituting relations*. These relations represent the dependencies between the subproblems generated by a strategy. Their schemes consist of arbitrary attributes for problems and solutions. The schemes are divided into *input attributes* and *output attributes*. 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.

$$
\begin{array}{|l}
const\_rel : \mathbb{P}\ relation \\
\hline
\forall\, cr : const\_rel \bullet \forall\, t : cr;\ a : scheme\ cr \bullet \\
\qquad scheme\ cr \subseteq (ProblemAttribute \cup SolutionAttribute) \land \\
\qquad (a \in ProblemAttribute \Rightarrow t\ a \in Problem) \land \\
\qquad (a \in SolutionAttribute \Rightarrow t\ a \in Solution)
\end{array}
$$

$$
\begin{array}{|l}
IA, OA : const\_rel \longrightarrow \mathbb{F}\ Attribute \\
\hline
\forall\, cr : const\_rel \bullet \langle IA\ cr, OA\ cr \rangle\ \mathsf{partition}\ scheme\ cr
\end{array}
$$

It is now possible to define dependency relations on constituting relations. A constituting relation directly depends on another if one of its input attributes is an output attribute of the other relation. The depending constituting relation is considered to be "larger". The transitive closure of the direct dependency relation with respect to some set of constituting relations yields the dependency relation.

$$
\begin{array}{|l}
\_\sqsubset_d\_ : const\_rel \leftrightarrow const\_rel \\
\sqsubset : \mathbb{P}\ const\_rel \longrightarrow (const\_rel \leftrightarrow const\_rel) \\
\hline
\forall\, cr_1, cr_2 : const\_rel;\ crs : \mathbb{P}\ const\_rel \bullet \\
\qquad ((cr_1 \sqsubset_d cr_2 \Leftrightarrow OA\ cr_1 \cap IA\ cr_2 \neq \varnothing) \land \\
\qquad (\sqsubset (crs) = \{cr_1, cr_2 : crs \mid (\exists\, chain : \mathrm{seq}\ crs \bullet head\ chain = cr_1 \land last\ chain = cr_2 \land \\
\qquad\qquad\quad (\forall\, i : 1 \mathinner{..} \#chain - 1 \bullet chain\ i \sqsubset_d chain(i+1)))\}))
\end{array}
$$

Instead of writing $(cr, cr') \in (\sqsubset crs)$, we write $cr \sqsubset_{crs} cr'$.

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 except the value of *P_init* must be determined in the problem solving process, i.e. each attribute except *P_init* must occur as an output attribute of some constituting relation.

4. Each attribute value should be determined only once, i.e. the sets of output attributes of all constituting relations must be disjoint.

5. Each solution to a subproblem is used further, i.e. it occurs as an input attribute of some constituting relation. (For the subproblems, it is not necessary to state such a requirement because they are used to generate the solutions to the subproblems.)

6. 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. This means that each subproblem must be set up before it is solved.

7. The dependency relation on the constituting relations must not be cyclic.

Finite sets of constituting relations fulfilling these requirements are called *admissible*. In the following formal definition of admissibility, each line of the predicate part of the axiomatic box formalizes one of the previous requirements. The function *partsols* yields all solution attributes of a relation scheme except *S_final*, and $scheme_s\ crs = scheme(\bowtie crs)$. The inverse of a relation (or function) r is denoted $r^{\sim}$

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

From this definition it follows that (i) each attribute a except *P_init* occurs as an output attribute of exactly one constituting relation, and (ii) each input attribute of a constituting relation except *P_init* must be an output attribute of a smaller relation. This means, there is an order in which all attribute values can be determined.

Lemma 1

$$
\begin{array}{l}
\forall\ crs : \mathbb{F}\ const\_rel \mid admissible\ crs \bullet \\
\quad (\forall\ a : scheme_s\ crs \setminus \{P\_init\} \bullet \exists_1\ cr_a : crs \bullet a \in OA\ cr_a) \\
\quad \land \\
\quad (\forall\ cr : crs \bullet IA\ cr \subseteq (\bigcup\{cr' : crs \mid cr' \sqsubset_{crs} cr \bullet OA(cr')\}) \cup \{P\_init\})
\end{array}
$$

Proof

The first part of the lemma follows from requirements 3 and 4 of the definition of admissibility. The second part follows from requirements 5 and 7.

■

3.4 Strategies

It is now possible to define strategies as admissible sets of constituting relations that fulfill certain conditions. An admissible set *strat* is a strategy if

1. $scheme_s$ *strat* contains the attributes *P_init* and *S_final*.
2. For each problem attribute of $scheme_s$ *strat*, the corresponding solution attribute is a member of the scheme, and vice versa.
3. If a member of the relation $\bowtie$ *strat* contains acceptable solutions for all problems except *P_init* then it also contains 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 solved correctly. For strategies solving the problem directly, this condition means that they must produce only acceptable solutions.

Again, each of the requirements corresponds to one conjunct in the formal definition. The function *subprs* yields all problem attributes occurring the schemes of a set of relations, except *P_init*.

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

Figure 1 illustrates the definition of strategies, where arrows denote the propagation of attribute values.

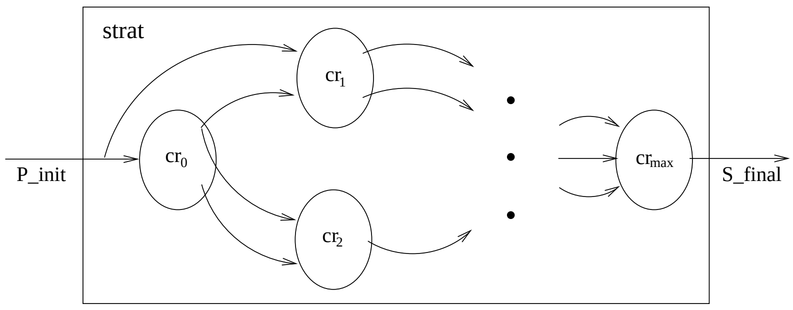


Figure 1: Definition of strategies

Note that the values of the output attributes of a constituting relation need not be independent. Strategies will usually be defined such that a subproblem and its corresponding solution are output attributes of the same constituting relation, where the solution must fulfill certain requirements with respect to the problem.

From the definition of strategies, it follows that there is at least one member of a strategy that has *P_init* as its only input attribute. This means that the problem solving process can be started at all. Furthermore, there is exactly one maximal member in a strategy that has *S_final* as its only output attribute and that depends on all other members of the strategy.

Lemma 2

$$
\begin{array}{l}
\forall\, \textit{strat} : \textit{strategy} \bullet \\
\quad (\exists\, cr_0 : \textit{strat} \bullet \textit{IA}\ cr_0 = \{P\_init\}) \\
\quad \wedge \\
\quad (\textbf{let}\ cr_{max} == (\mu\, r : \textit{strat} \mid S\_final \in \textit{OA}\ r) \bullet \\
\qquad (\{S\_final\} = \textit{OA}\ cr_{max} \wedge (\forall\, cr : (\textit{strat} \setminus \{cr_{max}\}) \bullet cr \sqsubset_{strat} cr_{max})))
\end{array}
$$

Proof

The first part of the lemma follows from the fact that $\sqsubset_{strat}$ does not contain cycles and Lemma 1. The second part follows from requirements 2, 5 and 7 of the admissibility definition for *strat*.

■

Renaming of attributes (except *P_init* and *S_final*) does not change the semantic content of a strategy. Hence, we can define an equivalence relation *_equiv_* on strategies, which will be used in Section 4.1.

The definitions presented in this section constitute the theoretical foundation of our approach. The next sections show how strategies can be combined and how they can be represented to make them implementable.

4 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.

We define three strategicals 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. To make the REPEAT strategical more widely applicable, we also define a LIFT strategical that transforms a strategy to develop one item into a strategy to develop several items of the same kind.

4.1 The THEN Strategical

The idea of this strategical is to replace one subproblem p generated by strategy $strat_1$ by the subproblems generated by strategy $strat_2$. The effect is the same as when reducing a problem first with $strat_1$ and then reducing p by $strat_2$. The difference is that p and its corresponding solution $cor\,p$ are not generated explicitly. This is illustrated in Figure 2.

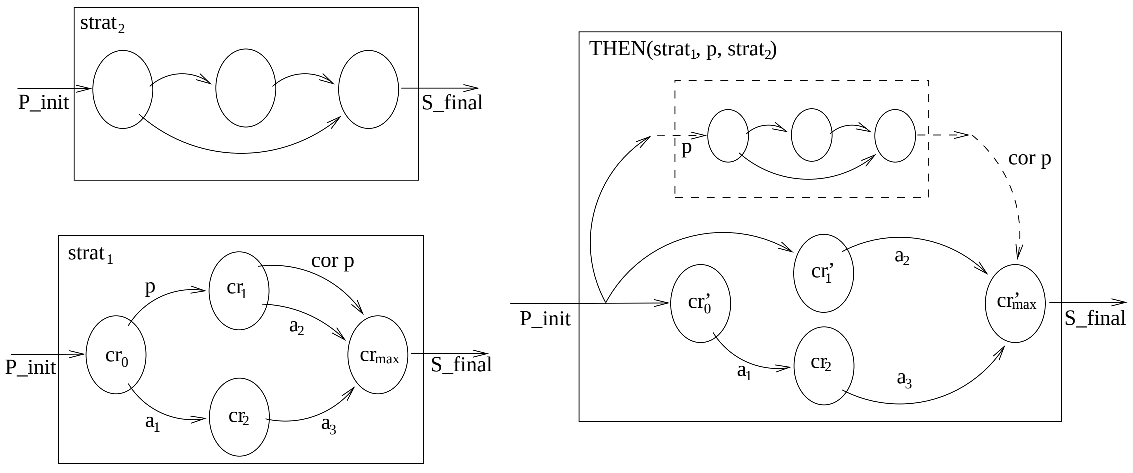


Figure 2: THEN strategical

In the strategy defined by THEN($strat_1, p, strat_2$), p plays the role of *P_init*, and $cor\,p$ plays the role of *S_final* in $strat_2$. The attribute values for p and $cor\,p$ are no longer explicitely set up. Hence, all those attribute values that are needed to define the value of p must be supplied to all constituting relations that rely on the value of p. Similarly, all attribute values that are needed to determine the final solution of $strat_2$ must be supplied to all constituting relations that have $cor\,p$ as an input attribute. Furthermore, we must guarantee that all attribute values are determined relative to the *same* values for p and $cor\, p$, i.e. there must be unique values for p and $cor\, p$ such that the $\textsc{Then}(strat_1, p, strat_2)$ equals $strat_1 \cup strat_2$, except that the attributes p and $cor\, p$ are removed from their schemes. This is achieved by joining the members of the two strategies with all members they depend on. The effect of this definition is that the constituting relations that make up $\textsc{Then}(strat_1, p, strat_2)$ have more input attributes than the ones in $strat_1$ and $strat_2$. Independent subproblems, however, remain independent.

The following function defines the transformation of the constituting relations that is necessary to replace p and $cor\, p$. A constituting relation cr is joined with all constituting relations it depends on, and the attributes p and $cor\, p$ are removed from its scheme.

$$\begin{array}{|l}
transform_{Then} : const\_rel \times (\mathbb{P}\, const\_rel) \times ProblemAttribute \mathrel{+\!\!\!\!\!\rightarrow} const\_rel \\
\hline
\forall\, cr, cr_t : const\_rel;\ crs : \mathbb{P}\, const\_rel;\ p : ProblemAttribute \bullet \\
\quad (cr_t = transform_{Then}(cr, crs, p) \Leftrightarrow \\
\quad\quad cr \in crs \land \\
\quad\quad p \in scheme_s\, crs \land \\
\quad\quad (\mathbf{let}\ lo == \bowtie \{r : crs \mid r \sqsubset_{crs} cr\} \bullet \\
\quad\quad\quad IA\, cr_t = (IA\, cr \cup scheme\, lo) \setminus \{p, cor\, p\} \land \\
\quad\quad\quad OA\, cr_t = OA\, cr \setminus \{p, cor\, p\} \land \\
\quad\quad\quad cr_t = scheme\, cr_t \lhd_r (lo \bowtie cr)))
\end{array}$$

To define $\textsc{Then}(strat_1, p, strat_2)$, we must first guarantee that the names of the subproblems generated by $strat_1$ and $strat_2$ are different by choosing a strategy $strat_2'$ that is equivalent to $strat_2$ and fulfills this requirement. Then, in $strat_2'$, P_init is replaced by p and S_final is replaced by $cor\, p$, using the function $replace : Attribute \times Attribute \times relation \longrightarrow relation$. Each of the resulting constituting relations is then transformed using the function $transform_{Then}$.

$$\begin{array}{|l}
\textsc{Then} : strategy \times ProblemAttribute \times strategy \mathrel{+\!\!\!\!\!\rightarrow} strategy \\
\hline
\forall\, strat_1, strat_2 : strategy;\ p : ProblemAttribute \bullet \\
\quad ((strat_1, p, strat_2) \in \operatorname{dom} \textsc{Then} \Rightarrow p \in subprs\ strat_1) \\
\quad \land \\
\quad (\exists\, strat_2' : strategy \mid strat_2'\ equiv\ strat_2 \land subprs\ strat_1 \cap subprs\ strat_2' = \varnothing \bullet \\
\quad\quad (\mathbf{let}\ strat_{2,r}' == \{cr : strat_2' \bullet replace(S\_final, cor\, p, replace(P\_init, p, cr))\} \bullet \\
\quad\quad \textsc{Then}(strat_1, p, strat_2) \\
\quad\quad\quad = \{cr : strat_1 \cup strat_{2,r}' \bullet transform_{Then}(cr, strat_1 \cup strat_{2,r}', p)\}))
\end{array}$$

Whenever $\textsc{Then}(strat_1, p, strat_2)$ is defined, it yields a strategy:

Lemma 3

$$\begin{array}{l}
\forall\, strat_1, strat_2 : strategy;\ p : ProblemAttribute \mid (strat_1, p, strat_2) \in \operatorname{dom} \textsc{Then} \bullet \\
\quad \textsc{Then}(strat_1, p, strat_2) \in strategy
\end{array}$$

The next lemma states that $\textsc{Then}(strat_1, p, strat_2)$ conforms to our intuition: its join contains exactly those tuples that can also be obtained by joining $strat_1$ and $strat_{2,r}'$ (where $strat_{2,r}'$ is defined as before), and then dropping the values of p and $cor\, p$.

Lemma 4

$$\bowtie (\textsc{Then}(strat_1, p, strat_2)) = \{p, cor\, p\} \mathrel{\ntriangleleft}_r (\bowtie (strat_1 \cup strat_{2,r}'))$$

where $strat_{2,r}'$ *is defined as in the definition of* $\textsc{Then}$

Finally, Lemma 5 states that $\textsc{Then}$ does not introduce any new dependencies, i.e. if two constituting relations of $\textsc{Then}(strat_1, p, strat_2)$ are dependent, also their untransformed versions are.

Lemma 5

$$cr'_t \sqsubset_{\text{THEN}(strat_1,p,strat_2)} cr_t \;\Rightarrow\; cr' \sqsubset_{strat_1 \cup strat'_{2,r}} cr$$

where $strat'_{2,r}$ *is defined as in the definition of* THEN *and*
$cr_t = transform_{Then}(cr, strat_1 \cup strat'_{2,r}, p) \wedge cr'_t = transform_{Then}(cr', strat_1 \cup strat'_{2,r}, p)$

The proofs of these lemmas are given in Appendix A.1. An example of a strategy defined with THEN is given in Section 7.2.3.

4.2 The REPEAT Strategical

The first argument of this strategical is the strategy *strat* to be repeated. Repetition here means that a subproblem p generated by *strat* should again be reduced by a finite iteration of *strat* or another strategy *terminate* that does not generate new subproblems. The attribute p and the strategy *terminate* are the other arguments of REPEAT. A strategy defined with REPEAT does not itself perform an iteration but only one step of an iteration. How often *strat* is iterated is decided elsewhere, e.g. by the user of an implemented system. The strategy REPEAT(*strat*, p, *terminate*) is distinguished from *strat* only by restriction of one constituting relation, as shown in Figure 3.

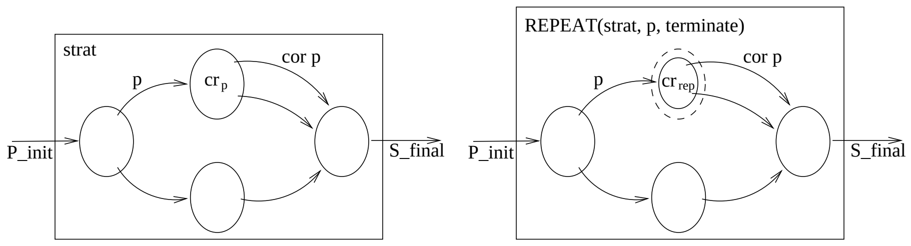


Figure 3: REPEAT strategical

The new constituting relation cr_{rep} is a subset of cr_p. Problem p is either solved by *terminate* or by a finite iteration of *strat*. The finite iteration is characterized by the fact that there is a finite sequence of tuples of $\bowtie$ *strat*, such that the subproblem p of tuple i is the initial problem *P_init* for tuple $i+1$; for the solutions, the analogous condition holds. The last tuple must contain a pair that is a member of $\bowtie$ *terminate*.

```
REPEAT : $strategy \times ProblemAttribute \times strategy \rightarrowtail strategy$
──────────
$\forall\, strat, terminate : strategy;\ p : ProblemAttribute \bullet$
    $((strat, p, terminate) \in \operatorname{dom} \text{REPEAT}$
        $\Rightarrow p \in subprs\ strat \wedge scheme_s\ terminate = \{P\_init, S\_final\})$
    $\wedge$
    $(\mathbf{let}\ cr_p == (\mu\, cr : strat \mid cor\ p \in OA\ cr) \bullet$
    $(\mathbf{let}\ cr_{rep} == \{t : cr_p \mid \{P\_init \mapsto t\ p, S\_final \mapsto t(cor\ p)\} \in \bowtie terminate$
                    $\vee$
                    $(\exists\, n : \mathbb{N}_1;\ ts : \operatorname{seq}(\bowtie strat) \mid \#ts = n \bullet$
                        $(ts\ 1)\ P\_init = t\ p \wedge (ts\ 1)\ S\_final = t(cor\ p) \wedge$
                        $(\forall\, i : 2 \mathinner{..} n \bullet (ts\ i)\ P\_init = (ts(i-1))\ p \wedge$
                                        $(ts\ i)\ S\_final = (ts(i-1))\ (cor\ p)) \wedge$
                        $\{P\_init \mapsto (ts\ n)\ p, S\_final \mapsto (ts\ n)(cor\ p)\} \in \bowtie terminate)\} \bullet$
    $IA\ cr_{rep} = IA\ cr_p \wedge$
    $\text{REPEAT}(strat, p, terminate) = ((strat \setminus \{cr_p\}) \cup \{cr_{rep}\})))$
```

Whenever REPEAT(*strat*, p, *terminate*) is defined, it yields a strategy:

Lemma 6

$$\forall\, strat, terminate : strategy;\ p : ProblemAttribute \mid (strat, p, terminate) \in \operatorname{dom} \textsc{Repeat} \bullet$$
$$\textsc{Repeat}(strat, p, terminate) \in strategy$$

Proof

Since $\textsc{Repeat}(strat, p, terminate)$ is distinguished from *strat* only by additional requirements on membership in one of its constituting relations, cr_p, it follows immediately that $\textsc{Repeat}(strat, p, terminate)$ is a strategy.

■

4.3 The Lift Strategical

It is possible that a strategy must be changed to make the Repeat strategical applicable. In specification acquisition, for instance, we may have the problem to define a list of Z operations. We might want to solve this problem by repeated application of a strategy *define_schema* that defines one schema. As it is, this strategy cannot serve as an argument to Repeat, because it defines only one schema and not a list of schemas. It cannot be applied to the subproblems it generates, namely to define the declaration part and to define the predicate part of the schema. We first have to "lift" it so that it can generate a list of schemas instead of a single schema.

The "lifted" strategy will generate one problem in addition to the problems generated by its argument strategy. This new problem can be used for the repetition. Besides a strategy, Lift needs the following arguments:

1. a function *p_down* that converts a "bigger" problem, e.g. to develop a list of schemas, into a "smaller" one, e.g. to develop one schema,
2. an injective function *p_combine* that makes a new problem out of the original problem and a partial solution, and
3. a function *s_combine* to combine the solutions of a "bigger" and a "smaller" problem.

These functions must be defined in such a way that the correctness of the lifted strategy can be guaranteed:

$$\forall\, pr : Problem;\ sol, sol' : Solution \mid$$
$$sol'\ acceptable\_for\ p\_down\ pr \wedge sol\ acceptable\_for\ p\_combine(pr, sol') \bullet$$
$$s\_combine(sol', sol)\ acceptable\_for\ pr$$

This strategical is illustrated in Figure 4.

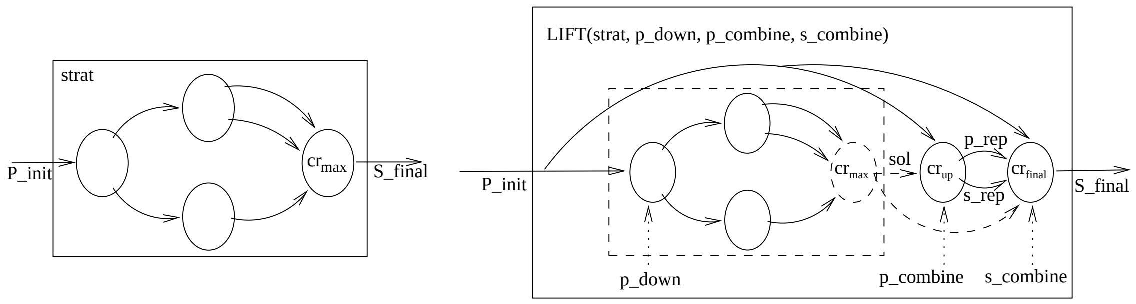


Figure 4: Lift strategical

Lift generates two new attributes, *p_rep* and *s_rep* that achieve the lifting. While the argument strategy *strat* can solve a "smaller" problem, Lift(*strat*, *p_down*, *p_combine*, *s_combine*) is used to solve a "bigger" problem. Hence, the problem *P_init* given to Lift(*strat*, *p_down*, *p_combine*, *s_combine*) must be transformed into a "smaller" problem by the function *p_down*. For *p_down*(*P_init*), all the attribute values of *strat* are determined as required by *strat* except the value for *S_final*. The solution *sol* that would have been bound to *S_final* by strategy *strat* is now propagated into the new problem *p_rep* using the function *p_combine*. Its solution *s_rep* is combined with the solution *sol* of the "smaller" problem using the function *s_combine*.

Similar to Then we need to transform the constituting relations of *strat*. Since Lift(*strat*, *p_down*, *p_combine*, *s_combine*) solves a "bigger" problem, this problem must be transformed by *p_down* into a "smaller" one to make *strat* applicable. This concerns only those constituting relations whose schema contains *P_init*.

$$
\begin{array}{|l}
\mathit{transform}_{\mathit{Lift}} : \mathit{const\_rel} \times (\mathit{Problem} \pfun \mathit{Problem}) \pfun \mathit{const\_rel} \\
\hline
\forall\, \mathit{cr} : \mathit{const\_rel}; \mathit{p\_down} : \mathit{Problem} \pfun \mathit{Problem} \mid \mathit{S\_final} \notin \mathit{scheme}\ \mathit{cr} \bullet \\
\quad \mathit{IA}(\mathit{transform}_{\mathit{Lift}}(\mathit{cr}, \mathit{p\_down})) = \mathit{IA}\ \mathit{cr} \land \\
\quad \mathit{OA}(\mathit{transform}_{\mathit{Lift}}(\mathit{cr}, \mathit{p\_down})) = \mathit{OA}\ \mathit{cr} \land \\
\quad \mathit{transform}_{\mathit{Lift}}(\mathit{cr}, \mathit{p\_down}) = \\
\qquad \mathbf{if}\ \mathit{P\_init} \notin \mathit{scheme}\ \mathit{cr}\ \mathbf{then}\ \mathit{cr} \\
\qquad \mathbf{else}\ \{t : \mathit{tuple} \mid t \oplus \{\mathit{P\_init} \mapsto \mathit{p\_down}(t\ \mathit{P\_init})\} \in \mathit{cr}\}
\end{array}
$$

The constituting relations of Lift(*strat*, *p_down*, *p_combine*, *s_combine*) are the transformed constituting relations of *strat* and two new ones. The first of these, *cr_up*, defines the attributes *p_up* and *s_up* for the "lifted" problem and its solution, using the function *p_combine*. The second one, cr_{final}, assembles the final solution of the lifted strategy by combining the solution of the transformed strategy *strat* with the "bigger" solution *s_up*.

$$
\begin{array}{|l}
\textsc{Lift} : \mathit{strategy} \\
\qquad \times (\mathit{Problem} \pfun \mathit{Problem}) \\
\qquad \times (\mathit{Problem} \times \mathit{Solution} \pinj \mathit{Problem}) \\
\qquad \times (\mathit{Solution} \times \mathit{Solution} \pfun \mathit{Solution}) \\
\qquad \pfun \mathit{strategy} \\
\hline
\forall\, \mathit{strat} : \mathit{strategy}; \mathit{p\_down} : \mathit{Problem} \pfun \mathit{Problem}; \\
\quad \mathit{p\_combine} : \mathit{Problem} \times \mathit{Solution} \pinj \mathit{Problem}; \\
\quad \mathit{s\_combine} : \mathit{Solution} \times \mathit{Solution} \pfun \mathit{Solution} \mid \\
\qquad (\forall\, \mathit{pr} : \mathit{Problem}; \mathit{sol}, \mathit{sol}' : \mathit{Solution} \mid \\
\qquad\quad \mathit{sol}'\ \mathit{acceptable\_for}\ \mathit{p\_down}\ \mathit{pr} \land \mathit{sol}\ \mathit{acceptable\_for}\ \mathit{p\_combine}(\mathit{pr}, \mathit{sol}') \bullet \\
\qquad\quad \mathit{s\_combine}(\mathit{sol}', \mathit{sol})\ \mathit{acceptable\_for}\ \mathit{pr}) \bullet \\
\quad \exists\, \mathit{p\_up} : \mathit{ProblemAttribute}; \mathit{s\_up} : \mathit{SolutionAttribute} \mid \\
\qquad \mathit{cor}\ \mathit{p\_up} = \mathit{s\_up} \land \{\mathit{p\_up}, \mathit{s\_up}\} \cap \mathit{subprs}\ \mathit{strat} = \varnothing \bullet \\
\qquad (\mathbf{let}\ \mathit{cr}_{\mathit{max}} == (\mu\, \mathit{cr} : \mathit{strat} \mid \mathit{OA}\ \mathit{cr} = \{\mathit{S\_final}\}) \bullet \\
\qquad (\mathbf{let}\ \mathit{crs}_{\mathit{new}} == \{\mathit{cr} : \mathit{strat} \setminus \{\mathit{cr}_{\mathit{max}}\} \bullet \mathit{transform}_{\mathit{Lift}}(\mathit{cr}, \mathit{p\_down})\}; \\
\qquad\quad \mathit{ias} == \mathit{IA}\ \mathit{cr}_{\mathit{max}} \cup \{\mathit{P\_init}\}; \mathit{oas} == \{\mathit{p\_up}, \mathit{s\_up}\} \bullet \\
\qquad (\mathbf{let}\ \mathit{cr}_{\mathit{up}} == \{t : \mathit{tuple} \mid \operatorname{dom} t = \mathit{ias} \cup \mathit{oas} \land \\
\qquad\qquad\qquad (\exists\, \mathit{sol} : \mathit{Solution} \mid (\mathit{IA}\ \mathit{cr}_{\mathit{max}} \lhd t) \cup \{\mathit{S\_final} \mapsto \mathit{sol}\} \in \mathit{cr}_{\mathit{max}} \bullet \\
\qquad\qquad\qquad\quad t\ \mathit{p\_up} = \mathit{p\_combine}(t\ \mathit{P\_init}, \mathit{sol}) \land \\
\qquad\qquad\qquad\quad t\ \mathit{s\_up}\ \mathit{acceptable\_for}\ t\ \mathit{p\_up})\}; \\
\qquad\quad \mathit{cr}_{\mathit{final}} == \{t : \mathit{tuple} \mid \operatorname{dom} t = \{\mathit{P\_init}, \mathit{p\_up}, \mathit{s\_up}, \mathit{S\_final}\} \land \\
\qquad\qquad\qquad (\mathbf{let}\ \mathit{sol} == (\mu\, \mathit{sol} : \mathit{Solution} \mid t\ \mathit{p\_up} = \mathit{p\_combine}(t\ \mathit{P\_init}, \mathit{sol})) \bullet \\
\qquad\qquad\qquad\quad t\ \mathit{S\_final} = \mathit{s\_combine}(\mathit{sol}, t\ \mathit{s\_up}))\} \bullet \\
\qquad \mathit{IA}\ \mathit{cr}_{\mathit{up}} = \mathit{ias} \land \mathit{OA}\ \mathit{cr}_{\mathit{up}} = \mathit{oas} \land \\
\qquad \mathit{IA}\ \mathit{cr}_{\mathit{final}} = \{\mathit{P\_init}, \mathit{p\_up}, \mathit{s\_up}\} \land \mathit{OA}\ \mathit{cr}_{\mathit{final}} = \{\mathit{S\_final}\} \land \\
\qquad \textsc{Lift}(\mathit{strat}, \mathit{p\_down}, \mathit{p\_combine}, \mathit{s\_combine}) = \mathit{crs}_{\mathit{new}} \cup \{\mathit{cr}_{\mathit{up}}, \mathit{cr}_{\mathit{final}}\})))
\end{array}
$$

The injectivity of the function *s_combine* guarantees that the tuples of cr_{up} and cr_{final} are defined with respect to the same solution *sol*.

Whenever $\textsc{Lift}(strat, p\_down, p\_combine, s\_combine)$ is defined, it yields a strategy:

Lemma 7

$$
\begin{array}{l}
\forall\, strat : strategy;\ p\_down : Problem \nrightarrow Problem; \\
\quad p\_combine : Problem \times Solution \rightarrowtail\!\!\!\nrightarrow Problem; \\
\quad s\_combine : Solution \times Solution \nrightarrow Solution \mid \\
\qquad (strat, p\_down, p\_combine, s\_combine) \in \operatorname{dom} \textsc{Lift} \bullet \\
\qquad \textsc{Lift}(strat, p\_down, p\_combine, s\_combine) \in strategy
\end{array}
$$

The proof of this lemma is given in Appendix A.2. An example of a strategy defined with Lift and Repeat is given in Section 8.2.3.

We have now defined strategies formally and given means to define more powerful strategies by combining simpler ones. So far, strategies are purely declarative. Our goal, however, is to make strategies *applicable* for problem solving. To this end, we must take a more procedural perspective of strategies.

5 Problem Solving With Strategies

Strategies and strategicals as they are defined so far are the conceptual basis for strategy-based problem solving. To make strategies applicable mechanically, we proceed in two steps: first, we represent strategies as modules that are implementable, using the encapsulation constructs offered by modern programming languages. Second, we present an abstract algorithm that defines how strategy-based problem solving proceeds. This algorithm is expressed as a set of algebraically defined Z functions. These functions are meant to denote recursive algorithms that could easily be implemented in a functional programming language. It can be shown that, if the algorithm yields a solution to a given problem, then this solution is acceptable.

5.1 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. In an implemented support system for strategy-based problem solving, 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 set *subp* of subproblems it produces,
- the dependency relation *_depends_* on them and their solutions,
- for each subproblem, a procedure *setup* that defines it, using the information in the initial problem and the subproblems and solutions it depends on,
- for each solution to a subproblem, a predicate *local_accept* that checks if the solution conforms to the requirements stated in the constituting relation of which it is an output attribute,
- a procedure *assemble* describing how to assemble the final solution,
- a test *accept* of acceptability for the assembled solution.

Optionally, an *explain* component may be added that explains *why* a solution is acceptable for a problem.

In the following, we define a number of functions that have a strategy as their argument and yield one piece of information as described before, i.e. each of these functions defines one component of a strategy module for its argument strategy.

The function *subprs* introduced in Section 3.4 yields the subproblems generated by a strategy. The dependency relation must be defined on pairs of problems instead of pairs of constituting relations:

$$
\begin{array}{|l}
\mathit{Depends} : \mathit{strategy} \longrightarrow (\mathit{ProblemAttribute} \longleftrightarrow \mathit{ProblemAttribute}) \\
\hline
\forall\, \mathit{strat} : \mathit{strategy};\, p_1, p_2 : \mathit{ProblemAttribute} \mid \{p_1, p_2\} \subseteq \mathit{scheme}_s\ \mathit{strat} \bullet \\
\quad (\mathbf{let}\ \mathit{cr}_1 == (\mu\, r : \mathit{strat} \mid p_1 \in \mathit{OA}\ r); \\
\qquad \mathit{cr}_2 == (\mu\, r : \mathit{strat} \mid p_2 \in \mathit{OA}\ r) \bullet \\
\qquad\quad (p_1, p_2) \in \mathit{Depends}(\mathit{strat}) \Leftrightarrow \mathit{cr}_1 \sqsubset_{\mathit{strat}} \mathit{cr}_2)
\end{array}
$$

It is possible for a combination of values for the input attributes of a constituting relation to be related to several combinations of values for the output attributes. In these cases, the basic type *ExtInfo* is used to select one of the possible values. This external information can be derived from user input or be computed automatically. By means of external information, relations are transformed into functions.

$$[\mathit{ExtInfo}]$$

A function that sets up a problem depends on the strategy *strat* and the subproblem p to be defined. $(\mathit{Setup\ strat})\, p$ is a function that takes a tuple and some external information as its arguments and yields a problem. It is defined with respect to the constituting relation cr_p for which p is an output attribute. Each tuple t for which the function $(\mathit{Setup\ strat})\, p$ is defined contains at least the values of the input attributes of cr_p. If the values of the input attributes are consistent with cr_p, then the value yielded by the setup function must also be consistent with cr_p. The external information is used to choose among different possible values that satisfy these conditions.

$$
\begin{array}{|l}
\mathit{Setup} : \mathit{strategy} \longrightarrow (\mathit{ProblemAttribute} \nrightarrow (\mathit{tuple} \times \mathit{ExtInfo} \nrightarrow \mathit{Problem})) \\
\hline
\forall\, \mathit{strat} : \mathit{strategy};\, p : \mathit{ProblemAttribute} \bullet \\
\quad \mathrm{dom}(\mathit{Setup}(\mathit{strat})) = \mathit{subprs\ strat} \land \\
\quad (p \in \mathit{subprs\ strat} \Rightarrow \\
\qquad (\exists_1\, \mathit{cr}_p : \mathit{strat} \mid p \in \mathit{OA}\ \mathit{cr}_p \bullet \\
\qquad\quad \forall\, t : \mathit{tuple};\, i : \mathit{ExtInfo} \mid (t, i) \in \mathrm{dom}(\mathit{Setup}(\mathit{strat})(p)) \bullet \\
\qquad\qquad \mathrm{dom}\, t \supseteq \mathit{IA}\ \mathit{cr}_p \land \\
\qquad\qquad ((\mathit{IA}\ \mathit{cr}_p \lhd t) \in (\mathit{IA}\ \mathit{cr}_p) \lhd_r \mathit{cr}_p \Rightarrow \\
\qquad\qquad\quad (\mathit{IA}\ \mathit{cr}_p \lhd t) \cup \{p \mapsto (\mathit{Setup}(\mathit{strat})(p))(t, i)\} \in (\mathit{IA}\ \mathit{cr}_p \cup \{p\}) \lhd_r \mathit{cr}_p)))
\end{array}
$$

For the intermediate solutions, we may have local acceptability conditions that are stated in the constituting relation cr_s of which the solution is an output attribute. Each tuple which is in the domain of $(\mathit{Local\_Accept\ strat})\, s$ contains at least the values of the input attributes that are needed to define the value of s and its corresponding problem $cor^{\sim} s$. If the values of the input attributes and the problem attribute $cor^{\sim} s$ are consistent with cr_s, then the value of s must also be consistent with cr_s.

$$
\begin{array}{|l}
\mathit{Local\_Accept} : \mathit{strategy} \longrightarrow (\mathit{SolutionAttribute} \nrightarrow (\mathit{tuple} \longleftrightarrow \mathit{Solution})) \\
\hline
\forall\, \mathit{strat} : \mathit{strategy};\, s : \mathit{SolutionAttribute} \bullet \\
\quad \mathrm{dom}(\mathit{Local\_Accept}(\mathit{strat})) = \mathit{partsols\ strat} \land \\
\quad (s \in \mathit{partsols}(\bowtie \mathit{strat}) \Rightarrow \\
\qquad (\exists_1\, \mathit{cr}_s : \mathit{strat} \mid s \in \mathit{OA}\ \mathit{cr}_s \bullet \\
\qquad\quad \forall\, t : \mathit{tuple};\, \mathit{sol} : \mathit{Solution} \mid (t, \mathit{sol}) \in \mathit{Local\_Accept}(\mathit{strat})(s) \bullet \\
\qquad\quad (\mathbf{let}\ \mathit{inp} == \mathit{IA}\ \mathit{cr}_s \cup \{\mathit{cor}^{\sim} s\} \bullet \\
\qquad\qquad \mathrm{dom}\, t \supseteq \mathit{inp} \land \\
\qquad\qquad ((\mathit{inp} \lhd t) \in \mathit{inp} \lhd_r \mathit{cr}_s \Rightarrow (\mathit{inp} \lhd t) \cup \{s \mapsto \mathit{sol}\} \in (\mathit{inp} \cup \{s\}) \lhd_r \mathit{cr}_s))))
\end{array}
$$

The conditions for the *Assemble* function can be expressed in a similar way.

$$\begin{array}{l}
\mathit{Assemble} : \mathit{strategy} \longrightarrow (\mathit{tuple} \times \mathit{ExtInfo} \rightarrow\!\!\!\!\!+ \ \mathit{Solution}) \\
\hline
\forall\, \mathit{strat} : \mathit{strategy} \bullet \\
\quad \exists_1\, cr_{max} : \mathit{strat} \mid \mathit{S\_final} \in \mathit{OA}\ cr_{max} \bullet \\
\quad\quad \forall\, t : \mathit{tuple};\ i : \mathit{ExtInfo} \mid (t, i) \in \operatorname{dom}(\mathit{Assemble}\ \mathit{strat}) \bullet \\
\quad\quad\quad \operatorname{dom} t = (\mathit{scheme}\ cr_{max}) \setminus \{\mathit{S\_final}\} \land \\
\quad\quad\quad t \in \{\mathit{S\_final}\} \lhd_r cr_{max} \Rightarrow \\
\quad\quad\quad\quad t \cup \{\mathit{S\_final} \mapsto \mathit{Assemble}(\mathit{strat})(t, i)\} \in cr_{max}
\end{array}$$

A tuple may only be a member of the set *Accept strat* if it is a member of $\bowtie$ *strat*. Thus, *Accept strat* will usually represent a sufficient condition for membership in a strategy that can be checked mechanically.

$$\begin{array}{l}
\mathit{Accept} : \mathit{strategy} \longrightarrow (\mathbb{P}\ \mathit{tuple}) \\
\hline
\forall\, \mathit{strat} : \mathit{strategy} \bullet \mathit{Accept}(\mathit{strat}) \subseteq (\bowtie \mathit{strat})
\end{array}$$

A Z specification does not specify what happens if a function is applied to an argument that does not lie in the domain of the function. An algorithm implementing the function could either not terminate or report failure. For our problem solving algorithm, we want that a failure is reported if a problem cannot be set up, a solution cannot be assembled properly, or a partial solution is found to be not locally acceptable. This is achieved by defining free types into which problems, solutions, and tuples are embedded and that contain error values to indicate that some of the previous functions are undefined. Thus, partial functions are totalized by yielding members the free types instead of problems, solutions, or tuples.

$$\begin{array}{l}
\mathit{total\_P} ::= \mathit{fail\_P} \mid \mathit{ok\_P}\langle\!\langle \mathit{Problem} \rangle\!\rangle \\
\mathit{total\_S} ::= \mathit{fail\_S} \mid \mathit{ok\_S}\langle\!\langle \mathit{Solution} \rangle\!\rangle \\
\mathit{total\_t} ::= \mathit{fail\_t} \mid \mathit{ok\_t}\langle\!\langle \mathit{tuple} \rangle\!\rangle
\end{array}$$

Strategy modules are algorithmic descriptions of strategies. They are obtained by application of the functions *subprs*, *Depends*, *Setup*, *Local_Accept*, *Assemble* and *Accept* to a strategy *strat*, and totalizing the results of *Setup* and *Assemble*. An error value is returned if and only if the corresponding function is undefined. Strategy modules are defined as a schema type that resembles record types in programming languages. The components of schema types are selected with the dot notation, e.g. for *sm* : *StrategyModule*, we write *sm.subp* to denote the subproblems generated by the strategy.

$$\begin{array}{l}
\textit{StrategyModule} \\
\hline
\mathit{subp} : \mathbb{P}\ \mathit{ProblemAttribute} \\
\mathit{\_depends\_on\_} : \mathit{ProblemAttribute} \longleftrightarrow \mathit{ProblemAttribute} \\
\mathit{setup} : \mathit{ProblemAttribute} \rightarrow\!\!\!\!\!+ \ (\mathit{tuple} \times \mathit{ExtInfo} \longrightarrow \mathit{total\_P}) \\
\mathit{local\_accept} : \mathit{SolutionAttribute} \rightarrow\!\!\!\!\!+ \ (\mathit{tuple} \longleftrightarrow \mathit{Solution}) \\
\mathit{assemble} : \mathit{tuple} \times \mathit{ExtInfo} \longrightarrow \mathit{total\_S} \\
\mathit{accept} : \mathbb{P}\ \mathit{tuple} \\
\hline
\exists\, \mathit{strat} : \mathit{strategy} \bullet \\
\quad (\mathit{subp} = \mathit{subprs}\ \mathit{strat} \land \\
\quad (\mathit{\_depends\_on\_}) = \mathit{Depends}\ \mathit{strat} \land \\
\quad \mathit{local\_accept} = \mathit{Local\_Accept}\ \mathit{strat} \land \\
\quad \mathit{accept} = \mathit{Accept}\ \mathit{strat} \land \\
\quad (\forall\, p : \mathit{ProblemAttribute};\ t : \mathit{tuple};\ i : \mathit{ExtInfo} \bullet \\
\quad\quad (((t, i) \in \operatorname{dom}((\mathit{Setup}\ \mathit{strat})(p)) \Leftrightarrow \mathit{setup}(p)(t, i) \in \operatorname{ran} \mathit{ok\_P}) \land \\
\quad\quad ((t, i) \in \operatorname{dom}((\mathit{Setup}\ \mathit{strat})(p)) \Rightarrow \mathit{setup}(p)(t, i) = \mathit{ok\_P}((\mathit{Setup}\ \mathit{strat})(p)(t, i))) \land \\
\quad\quad ((t, i) \in \operatorname{dom}(\mathit{Assemble}\ \mathit{strat}) \Leftrightarrow \mathit{assemble}(t, i) \in \operatorname{ran} \mathit{ok\_S}) \land \\
\quad\quad ((t, i) \in \operatorname{dom}(\mathit{Assemble}\ \mathit{strat}) \Rightarrow \mathit{assemble}(t, i) = \mathit{ok\_S}((\mathit{Assemble}\ \mathit{strat})(t, i)))))
\end{array}$$

A function $mod\_rep : strategy \longrightarrow StrategyModule$ transforms a strategy into a strategy module.

5.2 An Abstract Problem Solving Algorithm

In this section, we present an abstract algorithm that describes strategy-based development. This algorithm is expressed as a set of functions in Z.

Problem solving with strategies usually needs user interaction. The basic type *UserInput* comprises all possible user input. User interaction is modeled by giving a sequence of user inputs to the various functions. If such a sequence is not long enough, the functions are undefined. This corresponds to the situation where an interactive systems expects user input that is not supplied.

A *heuristic function* is a function that converts user input, which is needed to determine the value of some attribute of a strategy, into external information. The heuristic function may also depend on the values of other attributes, which are supplied to it as a tuple. Heuristic functions are those parts of a strategy implementation that can be implemented with a varying degree of automation, from interactive to fully automatic. It is also possible to automate them gradually by replacing interactive parts with semi or fully automatic ones. We cover the case that a heuristic function is independent of user input by using a dummy value in the sequence of user inputs.

$$heuristic\_function : StrategyModule \times Attribute \longrightarrow (tuple \times UserInput \rightarrow\!\!\!\!\!\!+\; ExtInfo)$$

The set *available_strategies* denotes the set of all available strategy modules. The function $choice : Problem \times (\mathbb{P}\, StrategyModule) \times UserInput \rightarrow\!\!\!\!\!\!+\; StrategyModule$ is used to select a strategy to solve a given problem from the available strategies.

We now can give the top-level problem solving algorithm. Its arguments are a problem and a list of user inputs. Since *solve* will be applied recursively, its result is not only a solution but also a user input list. A strategy to be applied to the problem is selected, and the function *apply* is called that applies the strategy to the problem. If the application of this strategy is successful, the value of *S_final* of the computed tuple and the input list yielded by *apply* form the result of the *solve* function. Otherwise, another trial is made with the user input list yielded by *apply*.

```
$solve : Problem \times \mathrm{seq}\ UserInput \rightarrow\!\!\!\!\!+ (Solution \times (\mathrm{seq}\ UserInput))$
────────────
$\forall\, pr : Problem;\ input\_list : \mathrm{seq}\ UserInput \bullet$
$solve(pr, input\_list) =$
    $(\mathbf{let}\ sm == choice(pr, available\_strategies, head\ input\_list) \bullet$
    $(\mathbf{let}\ t == apply(pr, sm, tail\ input\_list) \bullet$
    $\mathbf{if}\ first\ t = fail\_t\ \mathbf{then}\ solve(pr, second\ t)$
    $\mathbf{else}\ ((ok\_t^{\sim}(first\ t))\ S\_final, second\ t)))$
```

The function *apply* first calls another function *solve_subprs* to solve the subproblems generated by the strategy. It then sets up the final solution and checks it for acceptability. Each time a failure can occur, this is checked by the function and propagated into the result if necessary.

```
$apply : Problem \times StrategyModule \times \mathrm{seq}\ UserInput \rightarrow\!\!\!\!\!+ (total\_t \times \mathrm{seq}\ UserInput)$
────────────
$\forall\, p : Problem;\ sm : StrategyModule;\ input\_list : \mathrm{seq}\ UserInput \bullet$
$apply(p, sm, input\_list) =$
    $(\mathbf{let}\ s == solve\_subprs(\{P\_init \mapsto p\}, sm.subp, sm, input\_list) \bullet$
    $\mathbf{if}\ first\ s = fail\_t\ \mathbf{then}\ (fail\_t, second\ s)$
    $\mathbf{else}\ (\mathbf{let}\ tup == ok\_t^{\sim}(first\ s);$
            $input\_list' == second\ s \bullet$
        $(\mathbf{let}\ ext\_info == heuristic\_function(sm, S\_final)(tup, head\ input\_list') \bullet$
        $(\mathbf{let}\ final\_solution == sm.assemble(tup, ext\_info) \bullet$
        $\mathbf{if}\ final\_solution = fail\_S\ \mathbf{then}\ (fail\_t, tail\ input\_list')$
        $\mathbf{else}\ (\mathbf{let}\ s' == tup \cup \{S\_final \mapsto ok\_S^{\sim} final\_solution\} \bullet$
            $\mathbf{if}\ s' \notin sm.accept\ \mathbf{then}\ (fail\_t, tail\ input\_list')$
            $\mathbf{else}\ (ok\_t(s'), tail\ input\_list'))))))$
```

The function *solve_subprs* calls *solve* recursively for all subproblems contained in its second argument. Its first argument is the tuple generated so far. The function *choose_minimal* selects a minimal problem attribute from the set of open problems. The appropriate setup function defines the corresponding problem. Its solution, generated by *solve*, is then checked for local acceptability.

```
$solve\_subprs : tuple \times (\mathbb{P}\, ProblemAttribute) \times StrategyModule \times \mathrm{seq}\, UserInput$
        $\rightarrowtail\!\!\!\!+ (total\_t \times \mathrm{seq}\, UserInput)$
────────────
$\forall\, t : tuple;\ pas : \mathbb{P}\, ProblemAttribute;\ sm : StrategyModule;\ input\_list : \mathrm{seq}\, UserInput \bullet$
$solve\_subprs(t, pas, sm, input\_list) =$
    $\mathbf{if}\ pas = \varnothing\ \mathbf{then}\ (ok\_t(t), input\_list)$
    $\mathbf{else}\ (\mathbf{let}\ p == choose\_minimal(sm.(\_depends\_on\_), pas, head\ input\_list) \bullet$
        $(\mathbf{let}\ ext\_info == heuristic\_function(sm, p)(t, head(tail\ input\_list)) \bullet$
        $(\mathbf{let}\ pv == ((sm.setup)(p))(t, ext\_info) \bullet$
        $\mathbf{if}\ pv = fail\_P\ \mathbf{then}\ (fail\_t, tail(tail\ input\_list))$
        $\mathbf{else}\ (\mathbf{let}\ new\_pr == ok\_P^{\sim}\, pv \bullet$
            $(\mathbf{let}\ s == solve(new\_pr, tail(tail\ input\_list)) \bullet$
            $(\mathbf{let}\ sol == first\ s;\ input\_list' == second\ s \bullet$
            $\mathbf{if}\ (t \cup \{p \mapsto new\_pr\}, sol) \notin sm.local\_accept(cor\ p)$
            $\mathbf{then}\ (fail\_t, input\_list')$
            $\mathbf{else}\ solve\_subprs((t \cup \{p \mapsto new\_pr, cor\ p \mapsto sol\}), pas \setminus \{p\}, sm, input\_list'))))))$
```

The following lemmas show that the functions *solve*, *apply* and *solve_subprs* model strategy-based problem solving in an appropriate way: Whenever *solve* yields a solution to a problem, this solution is acceptable.

Lemma 8

$$\forall\, pr : Problem;\ sol : Solution;\ i_1, i_2 : \mathrm{seq}\, UserInput \mid (sol, i_2) = solve(pr, i_1) \bullet \\ sol\ acceptable\_for\ pr$$

If *apply* yields a tuple (as opposed to an error value), this tuple belongs to the join of some strategy and contains acceptable solutions for all subproblems.

Lemma 9

$$\begin{aligned}
&\forall\, pr : Problem;\ sm : StrategyModule;\ i_1, i_2 : \mathrm{seq}\, UserInput;\ tt : total\_t \mid \\
&\qquad (tt, i_2) = apply(pr, sm, i_1) \wedge tt \in \mathrm{ran}\ ok\_t \bullet \\
&\quad \exists\, strat : strategy \mid sm = mod\_rep\ strat \bullet \\
&\quad (\mathbf{let}\ t == ok\_t^{\sim}\ tt \bullet \\
&\qquad t \in (\bowtie strat) \wedge t\ P\_init = pr\ \wedge \\
&\qquad (\forall\, p : subprs\ strat \bullet t(cor\ p)\ acceptable\_for (t\ p)))
\end{aligned}$$

Lemma 10 states that, if *solve_subprs* is called with an argument list that satisfies the conditions stated there, also the arguments of the recursive call fulfill these conditions, i.e. *solve_subprs* preserves certain *invariants*: there is a strategy such that the domain of the tuple generated so far consists of *P_init* and those subproblems of the strategy that are not contained in the second argument of the function, and their corresponding solutions; the attribute values of the tuple are constistent with all constituting relations of the strategy, whose scheme is a subset of the domain of the tuple; all generated solutions are acceptable for their corresponding problems.

Lemma 10 *For $solve\_subprs(t, pas, sm, i_1)$, we have the following invariants:*

$$\begin{aligned}
&\forall\, t : tuple;\ pas : \mathbb{P}\, ProblemAttribute;\ sm : StrategyModule \bullet \\
&\quad INV(t, pas, sm) \\
&\quad \Leftrightarrow \\
&\quad (\exists\, strat : strategy \mid sm = mod\_rep\ strat \bullet \\
&\qquad \mathrm{dom}\ t = \{P\_init\} \cup \bigcup\{p : (subprs\ strat \setminus pas) \bullet \{p, cor\ p\}\}\ \wedge \\
&\qquad (\forall\, cr : strat \mid scheme\ cr \subseteq \mathrm{dom}\ t \bullet scheme\ cr \lhd t \in cr)\ \wedge \\
&\qquad (\forall\, p : ProblemAttribute \mid p \in (\mathrm{dom}\ t \setminus \{P\_init\}) \bullet t(cor\ p)\ acceptable\_for\ t\ p))
\end{aligned}$$

Proof

Lemma 8 follows from Lemma 9, using the fact that *solve* is defined such that the first component of its result is a tuple t that belongs to the strategy implemented by the chosen strategy module *sm*.

□

The first part of Lemma 9 follows from the fact that all valid results yielded by *apply* (i.e. the first component of the result is in ran *ok_t*) satisfy the *accept* predicate of the strategy module *sm*. The definition of *StrategyModule* entails that for each strategy module there is a corresponding strategy, and that the *accept* predicate is sufficient for a tuple to be a member of the join of the strategy.

The second part follows from Lemma 10 and the fact that the invariants hold for the arguments supplied to *solve_subprs* in *apply*.

□

As already shown, there exists a strategy such that *sm* is its modular representation. Since *solve_subprs* defines the attribute p as well as *cor p*, the first invariant stated in Lemma 10 holds.

The second invariant holds because the values of all problem attributes are defined using the function *sm.setup*, which relies on the global function *Setup*. The function *Setup* guarantees consistency with the corresponding constituting relation. The new values for solution attributes are checked for consistency with the corresponding constituting relation by the predicate *sm.local_accept* that relies on the global function *Local_Accept*.

The third invariant follows by an inductive argument on the maximal depth of the recursion, using Lemma 8 as inductive hypothesis. The base case are strategies that solve the problem directly. For these strategies, *solve_subprs* terminates immediately, and the third invariant is vacuously true.

■

From Lemma 10, we can deduce that *solve_subprs* computes all attribute values except *S_final* such that they are consistent with the constituting relations of the applied strategy:

$$pas = \varnothing \Rightarrow t \in \bowtie (strat \setminus \{cr_{max}\})$$

where $cr_{max} == (\mu\, r : strat \mid S\_final \in OA\ r)$.

In this section, we have transformed purely declarative strategies into a more procedural representation. Problem solving functions show how this representation is used to perform strategy-based development. These functions, however, have been defined to demonstrate that developed solutions are acceptable. They are very abstract and do not take adequate user support into account, as is necessary for implemented support systems.

6 System Architecture

We now define a system architecture that describes how to implement support systems for strategy-based problem solving. In contrast to the functions of the previous section, this system architecture takes the user into account and allows for much more flexibility in the problem solving process than the abstract algorithm of Section 5.2.

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. 5 gives a general view of the architecture which is described in more detail in [HSZ95b].

This architecture is a sophisticated implementation of the functions given in the last section. Introducing data structures that represent the state of the development makes the development process can be more flexible than with a naive implementation of these functions, where all intermediate results would be buried on the run-time stack. Using the system architecture, it is not necessary to first solve a subproblem completely before starting to work on another one.

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.

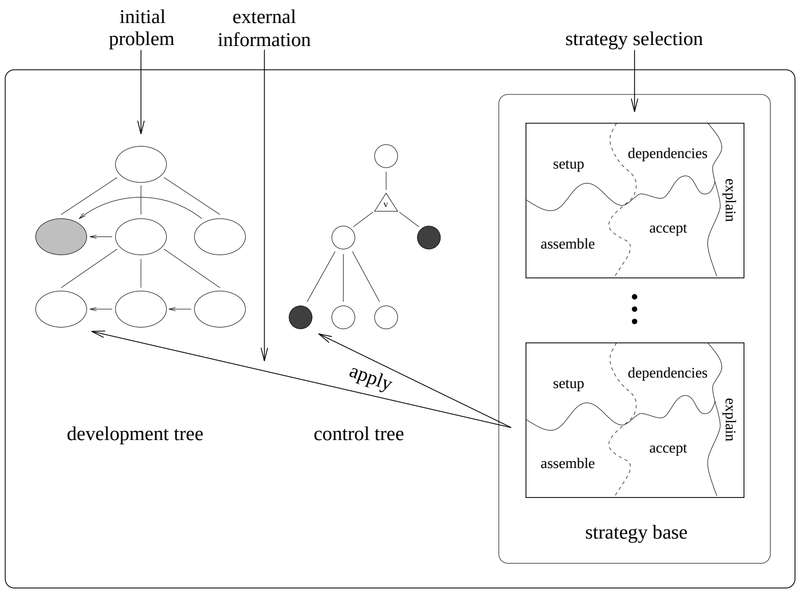


Figure 5: General view of the system architecture

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.

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 module in these nodes, and set up dependency links between them. The control tree is also 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 not only the developed solution; it is a development tree where all nodes contain acceptable solutions. This data structure provides a valuable documentation of the development process that can be kept for later reference.

This architecture guarantees the greatest possible flexibility in strategy-based problem solving. The user can always obtain an overview of the state of development and the context in which a certain problem has to be solved. The modular implementation of the strategy base makes it possible to incorporate new strategies in a routine manner. The architecture is independent of the kind of development activity that is to be supported and hence can be re-used for different instantiations of the strategy framework. Two such instantiations are presented in the following. They support different phases of the software lifecycle and use different formalisms.

7 Instantiation for Program Synthesis

We first 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. We instantiate the generic parameters and then give some example strategies. Finally, we describe the implemented prototype system IOSS.

7.1 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. For syntactic combination of formulas, we use the subscript "s", e.g. $\wedge_s$ and $\Rightarrow_s$. To refer to the semantics of formulas, we use predicates like *valid* and *satisfiable*.

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.

$$\begin{array}{|l}
\textit{ProgProblem} \\
\hline
\textit{pre}, \textit{goal}, \textit{inv} : \textit{First\_Order\_Formula} \\
\textit{res}, \textit{inp}, \textit{state} : \mathbb{P}\ \textit{Variable} \\
\hline
\mathsf{disjoint}\ \langle \textit{res}, \textit{inp}, \textit{state} \rangle \\
\textit{free}(\textit{pre} \wedge_s \textit{goal} \wedge_s \textit{inv}) \subseteq \textit{res} \cup \textit{inp} \cup \textit{state} \\
\textit{valid}(\textit{pre} \Rightarrow_s \textit{inv}) \\
\hline
\end{array}$$

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*.

$$\begin{array}{|l}
\textit{ProgSolution} \\
\hline
\textit{prog} : \textit{Program} \\
\textit{apr}, \textit{apo} : \textit{First\_Order\_Formula} \\
\hline
\textit{satisfiable}(\textit{apr}) \\
\hline
\end{array}$$

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.

$$\begin{array}{|l}
\_correct\_for\_ : ProgSolution \leftrightarrow ProgProblem \\
\hline
\forall\, pr : ProgProblem;\ sol : ProgSolution \bullet \\
\qquad sol\ correct\_for\ pr \\
\qquad \Leftrightarrow \\
\qquad (valid(pr.pre \wedge_s sol.apr \Rightarrow_s \langle sol.prog\rangle(pr.goal \wedge_s pr.inv \wedge_s sol.apo)) \wedge \\
\qquad vars(sol.prog) \cap pr.state = \varnothing \wedge \\
\qquad asg(sol.prog) \cap pr.inp = \varnothing)
\end{array}$$

The formula $pre \Rightarrow_s \langle prog\rangle post$ is a formula of dynamic logic [Gol82], a logic designed to prove 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.

7.2 Strategies for Program Synthesis

We present three strategies. With the first one, we can develop compound statements; the second serves to develop loops. Using the THEN strategical, these can be combined to yield a more powerful strategy to develop loops together with their initialization, thus implementing Gries' approach to the development of loops as introduced in Section 2.

The notation we use is semi-formal and resembles Z. The type *Value* denotes the disjoint union of the types *ProgProblem* and *ProgSolution*. Members of the schema types *ProgProblem* and *ProgSolution* are denoted as *bindings*. These are lists of pairs $component \Rrightarrow component\_value$.

7.2.1 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:

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

where *prot_first* is defined by

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

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 the existential quantifier in this definition. It 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. The constituting relation *prot_second* is defined by

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

The goal for *P_second* can be determined automatically. It consists of that part g_2 of the original goal that 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*, which is also a precondition for *P_second*. Another precondition is the additional postcondition guaranteed 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 occur in $t(S\_first).apo$). It is necessary to classify these variables because of the integrity condition $\mathit{free}(\mathit{pre} \wedge_s \mathit{goal} \wedge_s \mathit{inv}) \subseteq \mathit{res} \cup \mathit{inp} \cup \mathit{state}$ 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*. Also, the postcondition established by *S_first* must entail the additional precondition of *S_second*.

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.

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

7.2.2 The *loop* strategy

This strategy serves to construct a **while** loop. It is applicable only when the goal contains no quantifiers because the goal serves as the termination test of the loop. The strategy generates one subproblem.

$$\mathit{loop} = \{\mathit{loop\_body}, \mathit{loop\_sol}\}$$

where *loop_body* is defined by

$$\begin{array}{l}
IA\ loop\_body = \{P\_init\} \\
OA\ loop\_body = \{P\_loop, S\_loop\} \\
loop\_body = \{\ t : scheme\ loop\_body \longrightarrow Value \mid \\
\quad boolean\_expr(t(P\_init).goal) \land \\
\quad \exists\, bf, 0 : Term;\ t_0 : Variable;\ \prec : Term \leftrightarrow Term; \\
\quad loop\_inv : First\_Order\_Formula \mid \\
\quad\quad well\_founded\_ordering(0, \prec) \land \\
\quad\quad t_0 \notin (t(P\_init).res \cup t(P\_init).inp \cup t(P\_init).state) \land \\
\quad\quad valid(t(P\_init).pre \Rightarrow_s loop\_inv) \land \\
\quad\quad valid(t(P\_init).inv \land_s loop\_inv \land_s \neg_s(t(P\_init).goal) \Rightarrow_s 0 \prec bf) \bullet \\
\quad\quad t(P\_loop) = \langle\, pre \Rrightarrow t(P\_init).inv \land_s loop\_inv \land_s \neg_s(t(P\_init).goal) \land_s t_0 =_s bf, \\
\quad\quad\quad\quad\quad goal \Rrightarrow bf \prec t_0, \\
\quad\quad\quad\quad\quad inv \Rrightarrow t(P\_init).inv \land_s loop\_inv, \\
\quad\quad\quad\quad\quad res \Rrightarrow t(P\_init).res, \\
\quad\quad\quad\quad\quad inp \Rrightarrow t(P\_init).inp, \\
\quad\quad\quad\quad\quad state \Rrightarrow t(P\_init).state \cup \{t_0\}\,\rangle\ \land \\
\quad\quad t(S\_loop)\ acceptable\_for\ t(P\_loop)\}
\end{array}$$

To set up the problem *P_loop*, a bound function *bf* and a well-founded ordering $\prec$ on the carrier set of *bf* are needed, together with a constant *0* that is minimal with respect to $\prec$. The invariant of the loop to be developed consists of the invariant of the original problem and a formula *loop_inv* that usually contains invariant parts of the precondition $t(P\_init).pre$, e.g. ranges of variables. In IOSS, both *loop_inv* and *bf* must be given by the user, where for *loop_inv* the system makes proposals. The appropriate ordering, however, can often be inferred from the sort of the bound function, which is integer in many cases.

The goal is then to decrease the bound function, i.e. to make progress towards termination while maintaining the invariant. To express the decrease of the bound function, a new state variable t_0 is introduced. Otherwise, the variable classification remains unchanged.

The overall solution generated by the *loop* strategy is a loop with the negation of the original goal $t(P\_init).goal$ as the loop condition and the program that results in solving *P_loop* as the loop body. Accordingly, *loop_sol* is defined by

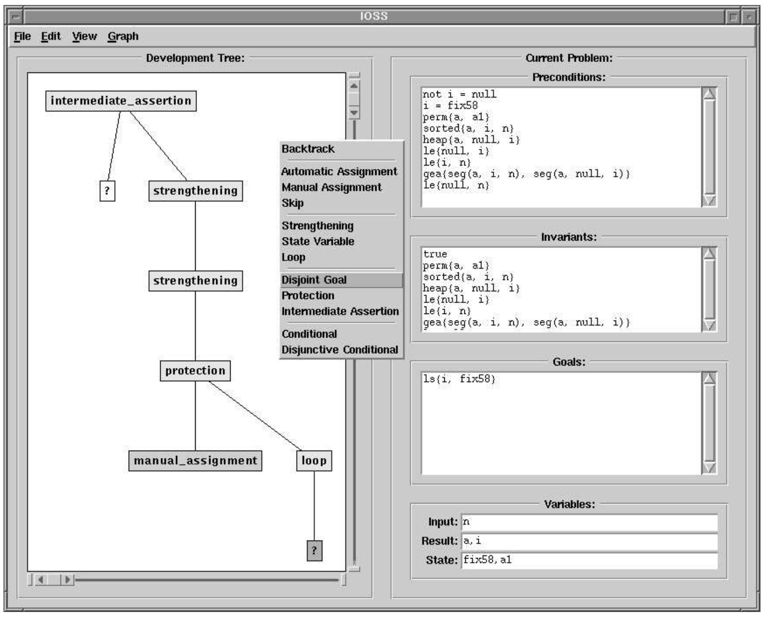
$$\begin{array}{l}
IA\ loop\_sol = \{P\_init, P\_loop, S\_loop\} \\
OA\ loop\_sol = \{S\_final\} \\
loop\_sol = \{\ t : scheme\ prot\_sol \longrightarrow Value \mid \\
\quad \exists\, loop\_inv : First\_Order\_Formula \mid \\
\quad\quad valid(t(P\_loop).inv \Leftrightarrow t(P\_init).inv \land loop\_inv) \bullet \\
\quad\quad t(S\_final) = \langle\, prog \Rrightarrow \textbf{while not } t(P\_init).goal\ \textbf{do } t(S\_loop).prog\ \textbf{od}, \\
\quad\quad\quad\quad\quad apr \Rrightarrow t(P\_init).pre \Rightarrow_s t(S\_loop).apr, \\
\quad\quad\quad\quad\quad apo \Rrightarrow loop\_inv\,\rangle\,\}
\end{array}$$

The formula *loop_inv* can be determined automatically: it is the same formula as used in *loop_body*.

7.2.3 A Combined Strategy

Usually, the development of a loop takes place as described in Section 2. Using strategies, the steps described there involve *strengthening* the goal of the original problem 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. For the development of the invariant, the heuristics given by Gries [Gri81] can be employed. 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 the *loop* strategy.

We use the THEN strategical to define a new *while* strategy that encompasses these steps:



Preconditions:

```
not i = null
i = fix58
perm(a, a1)
sorted(a, i, n)
heap(a, null, i)
le(null, i)
le(i, n)
gea(seg(a, i, n), seg(a, null, i))
le(null, n)
```

Invariants:

```
true
perm(a, a1)
sorted(a, i, n)
heap(a, null, i)
le(null, i)
le(i, n)
gea(seg(a, i, n), seg(a, null, i))
```

Goals:

```
ls(i, fix58)
```

Variables:

Input:	n
Result:	a, i
State:	fix58, a1

Figure 6: The IOSS interface

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

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

Strategies defined with THEN perform larger development steps than their component strategies. Thus, strategies can gradually approximate the complexity of development steps performed by human developers in practice. More strategies for program synthesis can be found in [Hei94].

7.3 IOSS: An Implemented Program Synthesis System

The program synthesis system IOSS is a research prototype that was built to validate the concept of strategy and the system architecture developed for the machine-supported application of strategies. Currently, it supports the application of the methods described in [Gri81] and [Der83]. Due to the uniform interface of strategy modules, the two approaches can be combined freely.

IOSS is an instantiation of the architecture described in Section 6 and uses the instantiation given in Section 7.1. 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. 6) is written in **`tcl/tk`** [Ous94] and integrates the graph visualization system **`daVinci`** [FW95] to display the development tree.

Fig. 6 shows the graphical user interface of IOSS. The main window displays 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, which also contains example developments, the reader is referred to [HSZ95a, HSZ95b].

8 Instantiation for Specification Acquisition

The instantiation of the previous section presupposes the existence of a formal specification for the program to be developed. However, developing the specification may be at least as difficult as transforming it into an executable program. Since formal specification languages often are not easy to handle, developers need support to use them appropriately. Strategies for specification acquisition not only propose an order in which the different parts of a specification can be developed. They also provide valuable validation mechanisms for the developed specifications.

The instantiation we present now 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 be easily transformed into programming problems of IOSS [Hei96]. The two instantiations are compared in Section 9.

8.1 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, 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. The schematic Z expression specifies the syntactic class of the piece of specification to be developed and how it is embedded in its context, see e.g. Section 8.2.2. These considerations lead us to the basic types

$[\mathit{SynZ}, \mathit{Text}, \mathit{SchematicZ}]$

Semantically valid Z specifications are a subset of the syntactically correct ones. To be able to state meaningful acceptability conditions that capture the role of a piece of specification in its context, Z expressions are associated with syntactic classes, e.g. *specification*, *schema*, *schema_list*. These syntactic classes are sets of Z expressions The empty string ϵ is a syntactically correct Z expression.

$$\begin{array}{|l}
\mathit{SemZ} : \mathbb{P}\,\mathit{SynZ} \\
\mathit{SyntacticalClass} : \mathbb{P}(\mathbb{P}\,\mathit{SynZ}) \\
\epsilon : \mathit{SynZ}
\end{array}$$

Each schematic Z expression is associated with the syntactic 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.

$$\begin{array}{|l}
\mathit{syn\_class} : \mathit{SchematicZ} \longrightarrow \mathit{SyntacticalClass} \\
\mathit{instantiate} : \mathit{SchematicZ} \times \mathit{SynZ} \rightarrow\!\!\!\!\!+\; \mathit{SynZ} \\
\_\mathtt{NL}\_ : \mathit{SynZ} \times \mathit{SynZ} \rightarrow\!\!\!\!\!+\; \mathit{SynZ} \\
\hline
\forall\, \mathit{schem\_expr} : \mathit{SchematicZ} \bullet \forall\, v : \mathit{syn\_class}\ \mathit{schem\_expr} \bullet \\
\qquad (\mathit{schem\_expr}, v) \in \operatorname{dom} \mathit{instantiate}
\end{array}$$

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

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

Solutions are Z expressions:

$$\mathit{SpecSolution} == \mathit{SynZ}$$

A solution *sol* is acceptable with respect to a problem *pr* if and only if it belongs to the syntactic class of $pr.to\_develop$ and the combination of $pr.context$ with the instantiated schematic expression yields a semantically valid Z specification.

$$
\begin{array}{|l}
\_spec\_acceptable\_for\_ : \mathit{SpecSolution} \leftrightarrow \mathit{SpecProblem} \\
\hline
\forall\, sol : \mathit{SpecSolution}; pr : \mathit{SpecProblem} \bullet \\
\qquad sol\ spec\_acceptable\_for\ pr \\
\qquad \Leftrightarrow \\
\qquad sol \in syn\_class(pr.to\_develop) \land \\
\qquad pr.context\ \mathtt{NL}\ instantiate(pr.to\_develop, sol) \in \mathit{SemZ}
\end{array}
$$

In practice, it is useful to define *SemZ* as those Z expressions that are accepted by available tools, such as the fuzz type checker [Spi92a].

8.2 Strategies for Specification Acquisition

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

Again, we use a semi-formal Z-like notation, neither formalizing the syntax and semantics of Z, nor giving definitions for all functions and predicates we use.

8.2.1 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 to complete the specification. Since this is a top-level strategy, the given problem must admit to develop expressions of the syntactic class *specification*. The type *Value* again denotes the disjoint union of *SpecProblem* and *SpecSolution*. Hence, we have

$$state\_based = \{global\_defs, system\_state, system\_ops, other\_defs, state\_based\_sol\}$$

where *global_defs* is defined by

$$
\begin{array}{l}
IA\ global\_defs = \{P\_init\} \\
OA\ global\_defs = \{P\_global, S\_global\} \\
global\_defs = \{\, t : scheme\ global\_defs \longrightarrow Value \mid \\
\qquad syn\_class(t(P\_init).to\_develop) = specification \land \\
\qquad t(P\_global) = \langle\, req \Rrightarrow t(P\_init).req\,; \text{“specify global definitions”}, \\
\qquad\qquad\qquad context \Rrightarrow t(P\_init).context \\
\qquad\qquad\qquad to\_develop \Rrightarrow \mathsf{sp} : specification \rangle \land \\
\qquad t(S\_global)\ spec\_acceptable\_for\ t(P\_global)\}
\end{array}
$$

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 denoted by sp : *specification*. This means that a Z expression belonging to the syntactic class *specification* must be developed, and the instantiation function is the identity. The constituting relation *system_state* is defined by

$$
\begin{array}{l}
\mathit{IA}\ \mathit{system\_state} = \{P\_\mathit{init}, S\_\mathit{global}\} \\
\mathit{OA}\ \mathit{system\_state} = \{P\_\mathit{state}, S\_\mathit{state}\} \\
\mathit{system\_state} = \{\ t : \mathit{scheme}\ \mathit{system\_state} \longrightarrow \mathit{Value}\ | \\
\quad t(P\_\mathit{state}) = \langle\ \mathit{req} \Rrightarrow t(P\_\mathit{init}).\mathit{req}\,;\ \text{“specify global system state”}, \\
\qquad\qquad \mathit{context} \Rrightarrow (t(P\_\mathit{init}).\mathit{context})\ \mathtt{NL}\ t(S\_\mathit{global}) \\
\qquad\qquad \mathit{to\_develop} \Rrightarrow \mathsf{state\_def} : \mathit{schema\_list}\rangle\ \wedge \\
\quad t(S\_\mathit{state})\ \mathit{spec\_acceptable\_for}\ t(P\_\mathit{state})\ \wedge \\
\quad t(S\_\mathit{state}) \neq \epsilon\}
\end{array}
$$

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* is defined by

$$
\begin{array}{l}
\mathit{IA}\ \mathit{system\_ops} = \{P\_\mathit{init}, P\_\mathit{state}, S\_\mathit{state}\} \\
\mathit{OA}\ \mathit{system\_ops} = \{P\_\mathit{ops}, S\_\mathit{ops}\} \\
\mathit{system\_ops} = \{\ t : \mathit{scheme}\ \mathit{system\_ops} \longrightarrow \mathit{Value}\ | \\
\quad t(P\_\mathit{ops}) = \langle\ \mathit{req} \Rrightarrow t(P\_\mathit{init}).\mathit{req}\,;\ \text{“specify system operations”}, \\
\qquad\qquad \mathit{context} \Rrightarrow (t(P\_\mathit{state}).\mathit{context})\ \mathtt{NL}\ t(S\_\mathit{state}) \\
\qquad\qquad \mathit{to\_develop} \Rrightarrow \mathsf{ops\_def} : \mathit{schema\_list}\rangle\ \wedge \\
\quad t(S\_\mathit{ops})\ \mathit{spec\_acceptable\_for}\ t(P\_\mathit{ops})\ \wedge \\
\quad t(S\_\mathit{ops}) \neq \epsilon\}
\end{array}
$$

Like the system state, the operations are defined by schemas. The empty list of operations is not permitted. The constituting relation *other_defs* is defined by

$$
\begin{array}{l}
\mathit{IA}\ \mathit{other\_def} = \{P\_\mathit{init}, P\_\mathit{ops}, S\_\mathit{ops}\} \\
\mathit{OA}\ \mathit{other\_def} = \{P\_\mathit{other}, S\_\mathit{other}\} \\
\mathit{other\_defs} = \{\ t : \mathit{scheme}\ \mathit{other\_defs} \longrightarrow \mathit{Value}\ | \\
\quad t(P\_\mathit{other}) = \langle\ \mathit{req} \Rrightarrow t(P\_\mathit{init}).\mathit{req}\,;\ \text{“other definitions”}, \\
\qquad\qquad \mathit{context} \Rrightarrow (t(P\_\mathit{ops}).\mathit{contex})\ \mathtt{NL}\ t(S\_\mathit{ops}) \\
\qquad\qquad \mathit{to\_develop} \Rrightarrow \mathsf{others} : \mathit{specification}\rangle\ \wedge \\
\quad t(S\_\mathit{other})\ \mathit{spec\_acceptable\_for}\ t(P\_\mathit{other})\}
\end{array}
$$

No assumptions can be made on the other necessary definitions.

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

$$
\begin{array}{l}
\mathit{IA}\ \mathit{state\_based\_sol} = \{S\_\mathit{global}, S\_\mathit{state}, S\_\mathit{ops}, S\_\mathit{other}\} \\
\mathit{OA}\ \mathit{state\_based\_sol} = \{S\_\mathit{final}\} \\
\mathit{state\_based\_sol} = \{\ t : \mathit{scheme}\ \mathit{state\_based\_sol} \longrightarrow \mathit{Value}\ | \\
\quad t(S\_\mathit{final}) = t(S\_\mathit{global})\ \mathtt{NL}\ t(S\_\mathit{state})\ \mathtt{NL}\ t(S\_\mathit{ops})\ \mathtt{NL}\ t(S\_\mathit{other})\ \wedge \\
\quad t(S\_\mathit{global})\ \text{does not contain state or operation schemas}\ \wedge \\
\quad t(S\_\mathit{state})\ \text{contains a state schema } S \text{ that is not imported by any} \\
\qquad\quad \text{other schema in } t(S\_\mathit{state}) \text{ and an initial schema for } S\ \wedge \\
\quad t(S\_\mathit{ops})\ \text{contains at least one operation schema}\ \wedge \\
\quad \text{none of the operations defined in } t(S\_\mathit{ops}) \text{ has precondition } \mathit{false}\}
\end{array}
$$

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 notation.

Applying the *state_based* strategy guarantees that the developed specification roughly conforms to the recommended Z methodology. Its acceptability conditions not only refer to the syntax of the developed specification, e.g. a list of schemas being non-empty, but also to its semantics, e.g. in distinguishing state and operation schemas. More detailed conditions can be stated in the strategies that are used to solve the subproblems generated by the *state_based* strategy.

8.2.2 The *define_schema* Strategy

This is a simple strategy, where a schema is defined in two parts: first the declaration part and then the predicate part. The strategy requires that solutions of syntactic class *schema* are permitted.

$$define\_schema = \{define\_decls, define\_pred, schema\_sol\}$$

where *define_decls* is defined by

$$\begin{array}{l}
IA\ define\_decls = \{P\_init\} \\
OA\ define\_decls = \{P\_decls, S\_decls\} \\
define\_decls = \{\ t : scheme\ define\_decls \longrightarrow Value \mid \\
\quad syn\_class(t(P\_init).to\_develop) \supseteq schema \\
\quad t(P\_decls) = \langle\ req \Rrightarrow t(P\_init).req\,;\ \text{“specify declaration part of schema”}, \\
\qquad\qquad\qquad context \Rrightarrow t(P\_init).context \\
\qquad\qquad\qquad to\_develop \Rrightarrow make\_schema(\mathsf{decls} : declaration\_list, true)\rangle \wedge \\
\quad t(S\_decls)\ spec\_acceptable\_for\ t(P\_decls)\}
\end{array}$$

Here, we have used the function *make_schema* instead of the graphical schema notation. The schematic expression $t(P\_decls).to\_develop$ says that a Z expression that belongs to the syntactic class *declaration_list* must be developed, and that the instantiation function is *make_schema*, with the developed expression and the predicate *true* as arguments. This trivial predicate must be used as long as the predicate part of the schema is not developed. The constituting relation *define_pred* is defined by:

$$\begin{array}{l}
IA\ define\_pred = \{P\_init, S\_decls\} \\
OA\ define\_pred = \{P\_pred, S\_pred\} \\
define\_pred = \{\ t : scheme\ define\_pred \longrightarrow Value \mid \\
\quad t(P\_pred) = \langle\ req \Rrightarrow t(P\_decls).req\,;\ \text{“specify predicate part of schema”}, \\
\qquad\qquad\qquad context \Rrightarrow t(P\_init).context \\
\qquad\qquad\qquad to\_develop \Rrightarrow make\_schema(t(S\_decls), \mathsf{pred} : predicate)\rangle \wedge \\
\quad t(S\_pred)\ spec\_acceptable\_for\ t(P\_pred)\}
\end{array}$$

Acceptability of the solution $t(S\_pred)$ requires that the developed predicate refers only to the declarations made in $t(S\_decls)$ and the global definitions of the context $t(P\_init).context$. The constituting relation *schema_sol* combines the two parts of the schema.

$$\begin{array}{l}
IA\ state\_based\_sol = \{S\_decls, S\_pred\} \\
OA\ state\_based\_sol = \{S\_final\} \\
state\_based\_sol = \{\ t : scheme\ state\_based\_sol \longrightarrow Value \mid \\
\quad t(S\_final) = make\_schema(t(S\_decls), t(S\_pred))\}
\end{array}$$

8.2.3 An Iterative Strategy

The second and third subproblems generated by the *state_based* strategy can be solved by repeated application of *define_schema*. For this purpose, a new strategy must be defined that generates lists of schemas instead of just one schema. According to the definitions of Section 4, we can define

$$\begin{array}{l}
define\_schema\_list = \\
\quad \textsc{Repeat}(\textsc{Lift}(define\_schema, p\_down, p\_combine, s\_combine), p\_rep, empty)
\end{array}$$

where *p_rep* is a problem attribute newly introduced by LIFT and *empty* is the terminating strategy that generates the empty specification ϵ. The other arguments of LIFT are defined as follows:

$$\begin{aligned}
&p\_down == (\lambda\, pr : SpecProblem \mid pr.to\_develop \supseteq schema\_list \bullet \\
&\quad \langle req \Rrightarrow pr.req,\, context \Rrightarrow pr.context,\, to\_develop \Rrightarrow \mathsf{sch} : schema\rangle)
\end{aligned}$$

$$\begin{aligned}
&p\_combine == (\lambda\, pr : SpecProblem;\ sol : SpecSolution \mid \\
&\qquad\qquad\qquad\qquad pr.to\_develop \supseteq schema\_list \land sol \in schema \bullet \\
&\quad \langle req \Rrightarrow pr.req,\, context \Rrightarrow pr.context\ \mathtt{NL}\ sol,\, to\_develop \Rrightarrow pr.to\_develop\rangle)
\end{aligned}$$

$$s\_combine == \_\mathtt{NL}\_$$

where *p_down* converts a problem to define a list of schemas into a problem to define a single schema; the function *p_combine* incorporates a developed schema into the *context* part of a problem; and the function *s_combine* concatenates two specifications, thus allowing to concatenate a schema with a list of schemas.

The function *p_combine* is injective, and if a list of schemas *sl* is acceptable for the combined problem, where the schema *sch* developed first is added to the context, then the whole schema list, consisting of concatenation of *sch* and *sl* is acceptable for the original problem. Hence, the requirements for the arguments of LIFT are fulfilled.

Defining the strategy *define_schema_list* with strategicals has the advantage that the existing strategy *define_schema* is re-used and that the user does not need to manually select the same strategy over and over again. The only possibilities left after application of *define_schema_list* are to develop one more schema or to terminate the iteration.

9 Comparison of the Two Instantiations

The two instantiations of the strategy framework presented in Sections 7 and 8 differ in several important aspects. The differences of program synthesis and specification acquisition are reflected in the respective instantiations. They show up in the following phenomena:

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

In specification acquisition, this is impossible because the requirements are described informally. Specification acquisition leads from informal to formal artifacts in the software engineering process. Hence, the general definition of acceptability can refer to the formal specification in isolation only, 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 syntactic classes. Thus the *general* notion of acceptability is static type correctness. For *individual* strategies, stronger acceptability conditions can be stated. 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 or that system operations should have a satisfiable precondition. Also consistency and completeness criteria can be stated in the context of particular strategies.

Independent subproblems. In program synthesis, the subproblems generated by a strategy are often 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, 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.

Incomplete solutions. The fact that subproblems in specification acquisition strongly depend on each other influences the way in which strategies are applied. 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. In the *state_based* strategy, the definition of the state and the operations will usually make use of the global definitions. But we cannot assume that a specifier foresees all necessary global definitions in advance. 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, we can achieve this by propagating incomplete solutions. When the specifier wants to work on a "later" subproblem, the *assemble* functions contained in the strategy modules (see Section 5.1) are executed, where dummy values are used for solutions that have not yet been developed. As soon as a change in an earlier problem/solution occurs, the *assemble* functions must be executed once more to propagate the results into later problem definitions. 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, like in *P_state* and *P_ops* of the *state_based* strategy. This can be supported by the strategicals REPEAT and LIFT, as described in Section 8.2. If several items of different syntactic classes have to be developed, like for the global definitions *P_global* of the *state_based* strategy, this can be achieved by a strategy called *iterate*.

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 semantic information than specification problems because they are formal. Their syntactic 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.

10 Related Work

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

Knowledge-Based Software Engineering (KBSE). This discipline seeks to support software engineering by artificial intelligence techniques. It comprises a variety of approaches to specification acquisition and program synthesis, see [LD89, LM91]. Our approach could also be subsumed under this field, because a knowledge representation mechanism is the heart of our approach.

A prominent example of KBSE, which is close to our aims, is the Programmer's Apprentice project [RW88]. There, programming knowledge is represented by *clichés*. These are prototypical examples of the artifacts in question, e.g. programs, requirements documents or designs. They can contain schematic parts. The programming task is performed by "inspection", i.e. choice of an appropriate cliché and its customization by combination with other clichés, instantiation of schematic parts, and structural changes. This is achieved by high-level editing commands. The assumption underlying the Apprentice approach is that a library of prototypical examples provides better user support than the representation of general-purpose knowledge. Our position is to prefer general-purpose knowledge because clichés to a large extent depend on the application domain. This makes it difficult to set up a sufficiently complete cliché library that does not need to be extended for each new problem.

Representation of Design and Process Knowledge. Wile [Wil83] presents the development language Paddle. Paddle is similar to conventional programming languages. Its control structures are called *goal structures*. Paddle programs are a means to express developments, i.e. procedures to transform a specification into a program. Since performing the thus specified process consists of executing the corresponding program, a disadvantage of this procedural representation of process knowledge is that it enforces a strict depth-first left-to-right processing of the goal structure. This restriction also applies to more recent approaches to represent software development processes by process programming languages [Ost87, SSW92].

Potts [Pot89] aims at capturing not only strategic but also heuristic aspects of design methods. He uses *Issue-based Information Systems* (IBIS) [CB88] as a representation formalism. IBIS representing heuristics tend to be specialized for a particular application domain. Our approach, in contrast, aims at representing general, domain independent problem solving knowledge.

Souquières and Lévy [Sou93, SL93] has 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. Therefore, incomplete reductions and a variable number of subtasks for the same operator can be admitted.

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" whose contents are not completely known and "solution nodes". In contrast to the development tree the KORSO development graph does not reflect single development steps. A branching in our development tree maps to a subgraph in their development graph where process information like 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. Their authors did not intend to integrate these methods with other ones, nor are these systems customizable. Moreover, the support of other activities than program synthesis was not a design goal for these systems.

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. Information about the development process is maintained implicitly. Working with KIDS, it is hard to keep track of "where" one is in a development. There is a logging and replay facility, but this provides no possibility to browse the state of development. Since design tactics are linearly programmed, there is no way to change the order of independent design steps or "interleave" tactics applications.

Tactical Theorem Proving. Tactical theorem proving has first been employed in Edinburgh LCF [Mil72]. The idea is to conduct interactive, goal-directed proofs by backward chaining from a goal to sufficient subgoals. *Tactics* are programs that implement "backward" application of logical rules. Tactical theorem proving is also used in modern theorem provers, e.g. 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 by the use of *metavariables*. These allow one to leave "holes" in a subgoal that are "filled" during proof of another subgoal by unification on metavariables. Dependencies not schematically expressible by metavariables cannot be realized by tactics. Since tactics only perform goal reduction, there is no equivalent to the *assemble* and *accept* functions of strategies. They are not necessary for the tactic approach because problems and solutions are identical except for instantiation of metavariables. In contrast, problems and solutions of strategies may be expressed in different languages, and the composition of solutions by *assemble* may not be expressible schematically.

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 proper loop construct is necessary, the REPEAT strategical performs only one step of a loop; its purpose is to impose restrictions on the strategies that may 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 idea — 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.

11 Conclusions

The concept of strategy is designed to support the application of formal methods in software engineering. The kind of knowledge that can be expressed as strategies are established ways of procedure as they are described in text books. The definition of strategies relies on relations, because different applications of the same strategy to a problem may lead to different subproblems and produce different solutions. Strategies do not fully automate a development task but provide guidance and validation. They leave a considerable degree of freedom in their application. The most important properties of the strategy framework are:

Methodological Support. When formal methods are used, it is important not to leave developers alone with a mere formalism. In contrast to other approaches, where tools deal with single documents and not with the process aspect of a development, the strategy framework aims at providing *methodological* support for software engineers. Making explicit not only dependencies but also independencies of problems in strategies allows for the greatest possible flexibility in the development process.

Genericity. The definition of strategies and the system architecture have the definitions of problems, solutions, and acceptability as generic parameters. This generic nature of strategies makes it possible to support quite different development activities, like specification acquisition and program synthesis.

Uniformity. The concept of strategy provides a uniform way of representing development knowledge. It is independent of the development activity that is performed and the formal method that is used. It gives rise to a uniform mathematical model of problem solving in the context of software engineering. Methods are uniformly represented as sets of strategies. Different methods can be combined freely as long as they rely on the same instantiation of the strategy framework.

Different instantiations of the strategy framework rely on the same principles. When conducting more and more development activities with strategies, software engineers can still use their previously acquired skills in strategy-based development; they need not learn entirely new ways of procedure. Moreover, when integrated tool support shall be provided, it is more promising to integrate different implemented instances of the strategy framework than totally unrelated systems.

Reuse. Strategies make development knowledge explicit. Knowledge represented as strategies can be communicated to others; it can be enhanced according to new experiences and insights; and it can be reused in different developments and by different persons.

Machine Support. Our approach provides concepts for machine-supported development processes. The uniform modular representation of strategies makes them implementable. The system architecture derived from the formal strategy framework gives guidelines for the implementation of support systems for strategy-based development. Representing the state of development by the data structure of development trees is essential for the practical applicability of the strategy approach. The practicality of the developed concepts is confirmed by the implemented system IOSS.

Documentation. The development tree does not only support the development process. Is also useful when the development is finished, because it provides a documentation of how the solution was developed and can be used as a starting point for later changes.

Semantic Properties. The concept of acceptability of a solution with respect to a problem captures the semantic properties that must be guaranteed by the developed products. Semantic constraints are on the one hand expressed in the general definition of acceptability that is part of every instantiation of the strategy framework. On the other hand, stronger acceptability conditions that take context information into account can be stated for individual strategies.

In an implementation, the functions *local_accept* and *accept* are the only components of a strategy module that are concerned with semantic properties. This enhances confidence in the development tool because only these functions have to be verified to ensure that the tool truly guarantees acceptability of the produced solutions.

Formality. By defining strategies formally, we were able to establish a theory of strategy-based problem solving. In several lemmas, we proved that strategies and strategicals conform to the intuition of problem solving, and we defined a problem solving algorithm that was proven to lead to acceptable solutions.

Stepwise Automation. Introducing the concept of heuristic function and using these functions in distinguished places in the development process, we have achieved a separation of concerns: the essence of the strategy, i.e. its semantic content, is carefully isolated from questions of replacing user interaction by semi or fully automatic procedures. Hence, gradually automating development processes amounts to local changes of heuristic functions.

Scalability. Using strategicals, more and more elaborate strategies can be defined. In this way, strategies can gradually approximate the size and kind of development steps as they are performed by software engineers. In connection with the stepwise automation facilities, this contributes to the scalability of the approach.

Customizibility. To incorporate a new method into a support system, the strategy base only has to be extended by the new strategies. This involves only local changes that do not affect existing components. The same holds for the automation of parts of the development.

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.

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 to profitably apply strategies.

Currently, we work on two more instantiations of the strategy framework. The first supports the specification of safety-critical systems. Here, a combination of Z and real-time CSP [Dav93] will be used. The second is intended to cover the design phase of the software development process by supporting the development of software architectures following certain architectural styles.

In the future, we want to gain even 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 contribute to the understanding of the requirements for automated support of almost all phases of the software development process.

For now, different instantiations of the strategy framework lead to different support systems. We will investigate how different instances of the system architecture can be combined; first ideas are reported in [Hei96]. 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 and his detailed comments on a draft of this paper.

References

[BH84] W. Bibel and K. M. Hörnig. LOPS – a system based on a strategical approach to program synthesis. In A. Biermann, G. Guiho, and Y. Kodratoff, editors, *Automatic Program Construction Techniques*, pages 69–89. MacMillan, New York, 1984.

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

[CB88] J. Conclin and M. Begeman. gIBIS: a hypertext tool for exploratory policy discussion. *ACM Transactions on Office Informations Systems*, 6:303–331, October 1988.

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

[Dav93] Jim Davies. *Specification and Proof in Real-Time CSP*. Cambridge University Press, 1993.

[Der83] Nachum Dershowitz. *The Evolution of Programs*. Birkhäuser, Boston, 1983.

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

[FW95] M. Fröhlich and M. Werner. Demonstration of the interactive graph-visualization system daVinci. 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, *Proceedings 9th International Conference on Automated Deduction*, LNCS 310, pages 131–140. Springer-Verlag, 1988.

[HSZ95a] Maritta Heisel, Thomas Santen, and Dominik Zimmermann. A generic system architecture for strategy-based software development. Technical Report 95-8, Technical University of Berlin, 1995.

[HSZ95b] Maritta Heisel, Thomas Santen, and Dominik Zimmermann. Tool support for formal software development: A generic architecture. In W. Schäfer and P. Botella, editors, *Proceedings 5-th European Software Engineering Conference*, 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.

[LD89] Michael Lowry and Raul Duran. Knowledge-based software engineering. In A. Barr, P.R. Cohen, and E.A. Feigenbaum, editors, *The Handbook of Artificial Intelligence*, chapter 20, pages 241–322. Addison-Wesley, Reading, MA, 1989.

[LM91] Michael R. Lowry and Robert D. McCartney, editors. *Automating Software Design.* AAAI Press, Menlo Park, 1991.

[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 International Conference 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 *International Conference on Software Engineering*, pages 217–226. IEEE Computer Society Press, 1989.

[PST91] Ben Potter, Jane Sinclair, and David Till. *An Introduction to Formal Specification and Z.* Prentice Hall, 1991.

[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 Transactions on Software Engineering*, 16(9):1024–1043, September 1990.

[Sou93] Jeanine Souquières. *Aide au Développement de Specifications.* Thèse d'Etat, Université de Nancy I, 1993.

[Spi92a] J. M. Spivey. The fuzz manual. Computing Science Consultancy, Oxford, 1992.

[Spi92b] 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.

A Proofs of Lemmas

A.1 Proofs of Lemmas 3, 4 and 5

We first prove Lemma 5, where we use the same abbreviations and declarations as introduced there. By the fact that $\sqsubset$ is the transitive closure of $\sqsubset_d$ with respect to some set of constituting relations, the lemma follows by an inductive argument from

$$cr'_t \sqsubset_d cr_t \Rightarrow cr' \sqsubset_{strat_1 \cup strat'_{2,r}} cr$$

The relation $cr_t \sqsubset_d cr'_t$ means $OA\ cr'_t \cap IA\ cr_t \neq \varnothing$. According to the definition of $transform_{Then}$, this is equivalent to

$$\begin{array}{l}(OA\ cr' \setminus \{p, cor\ p\}) \\ \quad \cap ((IA\ cr \cup scheme(\bowtie \{r : strat_1 \cup strat'_{2,r} \mid r \sqsubset_{strat_1 \cup strat'_{2,r}} cr\})) \setminus \{p, cor\ p\}) \\ \neq \varnothing\end{array}$$

This condition is equivalent to

$$\begin{array}{l}\exists\, a : Attribute \mid a \notin \{p, cor\ p\} \bullet \\ \quad a \in OA\ cr' \wedge a \in (IA\ cr \cup scheme(\bowtie \{r : strat_1 \cup strat'_{2,r} \mid r \sqsubset_{strat_1 \cup strat'_{2,r}} cr\}))\end{array}$$

If $a \in OA\ cr' \cap IA\ cr$, we immediately have $cr' \sqsubset_d cr$ and hence $cr' \sqsubset_{strat_1 \cup strat'_{2,r}} cr$. Otherwise,

$$\exists\, \overline{cr} : \{r : strat_1 \cup strat'_{2,r} \mid r \sqsubset_{strat_1 \cup strat'_{2,r}} cr\} \bullet a \in scheme\ \overline{cr}$$

Since the sets of all output attributes of $strat_1 \cup strat'_{2,r}$ except $p, cor\ p$ are disjoint and $a \notin \{p, cor\ p\}$, a cannot be an output attribute of $\overline{cr}$. Hence, $a \in IA\ \overline{cr}$. This yields $OA\ cr' \cap IA\ \overline{cr} \neq \varnothing$. It follows

$$cr' \sqsubset_d \overline{cr}, \text{ where } \overline{cr} \sqsubset_{strat_1 \cup strat'_{2,r}} cr$$

which finishes the proof of Lemma 5.

□

We now prove Lemma 3, where we use the same definitions and abbreviations as before. We have

$$\begin{array}{l}scheme_s(\text{THEN}(strat_1, p, strat_2)) \\ \quad = (scheme_s\ strat_1 \setminus \{p, cor\ p\}) \cup (scheme_s\ strat'_2 \setminus \{P\_init, S\_final\})\end{array}$$

where $(scheme_s\ strat_1 \setminus \{p, cor\ p\}) \cap (scheme_s\ strat'_2 \setminus \{P\_init, S\_final\}) = \varnothing$. It follows that the conditions 1 and 2 of the strategy definition are fulfilled.

The admissibility of $\text{THEN}(strat_1, p, strat_2)$ is fulfilled, as the following argumentation shows:

- Since $transform_{Then}$ does not change the property of P_init or S_final of being an input or output attribute of some $cr \in strat_1$ and these two attributes do no longer occur in $strat'_{2,r}$, the requirements 1 and 2 of the admissibility definition are fulfilled.

- The function $\mathit{transform}_{\mathit{Then}}$ removes the attributes p and $\mathit{cor}\,p$ from the input and output attributes of the constituting relation supplied as its first argument. Since these attributes are no longer in the scheme of $\textsc{Then}(\mathit{strat}_1, p, \mathit{strat}_2)$, this does not destroy the requirements 3 and 4. Hence, they also hold for the transformed constituting relations.
- Condition 5 holds because it holds for strat_1 as well as strat_2', and because the attribute $\mathit{cor}\,p$ that replaces $\mathit{S\_final}$ in strat_2' is an input attribute of a constituting relation of strat_1.
- The condition 6 is also not changed by $\mathit{transform}_{\mathit{Then}}$ because the attributes p and $\mathit{cor}\,p$ are always removed pairwise from the schemes of the constituting relations.
- We show condition 7 by contraposition. If there were cyclic dependencies in $\textsc{Then}(\mathit{strat}_1, p, \mathit{strat}_2)$, then by Lemma 5, there would also be a cyclic dependency in $\mathit{strat}_1 \cup \mathit{strat}_{2,r}'$. Since strat_1 and $\mathit{strat}_{2,r}'$ both cannot contain cycles (because strat_1 and strat_2' are strategies), a cycle in $\mathit{strat}_1 \cup \mathit{strat}_{2,r}'$ must contain members of both strat_1 and $\mathit{strat}_{2,r}'$. It follows that without loss of generality

  $$\exists\, \mathit{cr} : \mathit{strat}_1 \bullet \exists\, \mathit{chain} : \mathrm{seq}(\mathit{strat}_1 \cup \mathit{strat}_{2,r}') \bullet \mathit{head}\ \mathit{chain} = \mathit{cr} \wedge \mathit{last}\ \mathit{chain} = \mathit{cr} \wedge \\ (\forall\, j : 1..\#\mathit{chain} - 1 \bullet \mathit{chain}\ j \sqsubset_d \mathit{chain}(j+1))$$

  Since the cycle must contain members of $\mathit{strat}_{2,r}'$, there must be a minimal index i and a maximal index k of chain such that

  $$\mathit{chain}\ i \in \mathit{strat}_1 \wedge \mathit{chain}(i+1) \in \mathit{strat}_{2,r}' \wedge \\ \mathit{chain}\ k \in \mathit{strat}_{2,r}' \wedge \mathit{chain}(k+1) \in \mathit{strat}_1$$

  Since p and $\mathit{cor}\,p$ are the only common elements of strat_1 and $\mathit{strat}_{2,r}'$, only these can be involved in the direct dependencies $\mathit{chain}\ i \sqsubset_d \mathit{chain}(i+1)$ and $\mathit{chain}\ k \sqsubset_d \mathit{chain}(k+1)$.

  For index i, $\mathit{cor}\,p \in \mathit{OA}(\mathit{chain}\ i) \cap \mathit{IA}(\mathit{chain}(i+1))$ is impossible because in $\mathit{strat}_{2,r}'$ $\mathit{cor}\,p$ is always an output attribute (it takes the role of $\mathit{S\_final}$ in strat_2'). For index k, $p \in \mathit{OA}(\mathit{chain}\ k) \cap \mathit{IA}(\mathit{chain}(k+1))$ is impossible because in $\mathit{strat}_{2,r}'$ p is always an input attribute (it takes the role of $\mathit{P\_init}$ in $\mathit{strat}_{2'}$). It follows

  $$p \in \mathit{OA}(\mathit{chain}\ i) \cap \mathit{IA}(\mathit{chain}(i+1)) \wedge \mathit{cor}\,p \in \mathit{OA}(\mathit{chain}\ k) \cap \mathit{IA}(\mathit{chain}(k+1))$$

  Since $p \in \mathit{OA}(\mathit{chain}\ i)$, we get $\neg\ (\mathit{cr}_{\mathit{cor}\,p} \sqsubset_{\mathit{strat}_1} \mathit{chain}\ i)$, where $\mathit{cr}_{\mathit{cor}\,p}$ is the unique constituting relation of strat_1 that contains $\mathit{cor}\,p$ as an output attribute. Since i was chosen minimal, $\mathit{cr} \sqsubset_{\mathit{strat}_1} \mathit{chain}\ i$ holds. It follows $\neg\ (\mathit{cr}_{\mathit{cor}\,p} \sqsubset_{\mathit{strat}_1} \mathit{cr})$. Otherwise, by transitivity of $\sqsubset_{\mathit{strat}_1}$, we would get $\mathit{cr}_{\mathit{cor}\,p} \sqsubset_{\mathit{strat}_1} \mathit{cr} \sqsubset_{\mathit{strat}_1} \mathit{chain}\ i$.

  Since $\mathit{cor}\,p \in \mathit{IA}(\mathit{chain}(k+1))$, we have $\mathit{cr}_{\mathit{cor}\,p} \sqsubset_{\mathit{strat}_1} \mathit{chain}(k+1)$. Index k was chosen maximal, which yields $\mathit{chain}(k+1) \sqsubset_{\mathit{strat}_1} \mathit{cr}$. By transitivity of $\sqsubset_{\mathit{strat}_1}$, we get $\mathit{cr}_{\mathit{cor}\,p} \sqsubset_{\mathit{strat}_1} \mathit{cr}$. This is a contradiction because we had already concluded $\neg\ (\mathit{cr}_{\mathit{cor}\,p} \sqsubset_{\mathit{strat}_1} \mathit{cr})$. This finishes the proof that $\textsc{Then}(\mathit{strat}_1, p, \mathit{strat}_2)$ contains no cycles.

It remains to show the correctness of $\textsc{Then}(\mathit{strat}_1, p, \mathit{strat}_2)$, as required by condition 3 of the strategy definition. This condition follows immediately from Lemma 4, together with the facts that both strat_1 and strat_2' are strategies and that renaming of attributes does not destroy correctness.

□

To prove Lemma 4, we first show that

$$\forall\, \mathit{cr} : \mathit{strat}_1 \cup \mathit{strat}_{2,r}' \mid \mathit{cr} \neq \mathit{cr}_{\mathit{max}} \bullet \mathit{cr} \sqsubset_{\mathit{strat}_1 \cup \mathit{strat}_{2,r}'} \mathit{cr}_{\mathit{max}} \\ \text{where } \mathit{cr}_{\mathit{max}} == (\mu\, r : \mathit{strat}_1 \mid \mathit{S\_final} \in \mathit{OA}\ r)$$

Because of Lemma 2, this holds for $\mathit{cr} : \mathit{strat}_1 \setminus \{\mathit{cr}_{\mathit{max}}\}$. For $\mathit{cr} : \mathit{strat}_{2,r}' \setminus \{\mathit{cr}_{\mathit{max},2}\}$, we have $\mathit{cr} \sqsubset_{\mathit{strat}_{2,r}'} \mathit{cr}_{\mathit{max},2}$, where $\mathit{cr}_{\mathit{max},2} == (\mu\, r : \mathit{strat}_{2,r}' \mid \mathit{cor}\,p \in \mathit{OA}\ r)$. Since $\mathit{cor}\,p$ is an input attribute of some $cr : strat_1$, it follows $cr_{max,2} \sqsubset_{strat_1 \cup strat'_{2,r}} cr_{max}$, and by transitivity of $\sqsubset_{strat_1 \cup strat'_{2,r}}$, the previous proposition is shown.

This gives us

$$transform_{Then}(cr_{max}, strat_1 \cup strat'_{2,r}, p) = \{p, cor\, p\} \lhd_r \bowtie (strat_1 \cup strat'_{2,r})$$

It follows $\bowtie (\text{THEN}(strat_1, p, strat_2)) \subseteq \{p, cor\, p\} \lhd_r \bowtie (strat_1 \cup strat'_{2,r})$.

For the other direction, we consider some $t \in \{p, cor\, p\} \lhd_r \bowtie (strat_1 \cup strat'_{2,r})$ and show that it is also a member of $transform_{Then}(cr_{max}, strat_1 \cup strat'_{2,r}, p)$. This follows from

$$\forall\, cr_t : \text{THEN}(strat_1, p, strat_2) \bullet scheme\ cr_t \lhd t \in cr_t$$

This proposition is true because all $cr_t : \text{THEN}(strat_1, p, strat_2)$ are defined as $\{p, cor\, p\} \lhd_r \bowtie crs'$ for some $crs' \subseteq strat_1 \cup strat'_{2,r}$. By the definition of $\bowtie$, if a tuple is in the join of set of relations, its appropriate restriction is in every subset of that set.

This concludes the proof of Lemma 4 (and hence of Lemma 3).

■

A.2 Proof of Lemma 7

The conditions 1 and 2 of the strategy definition as well as the conditions 1 and 2 of the admissibility definition are easily verified. The other conditions for the admissibility of $\text{LIFT}(strat, p\_down, p\_combine, s\_combine)$ can be verified as follows:

- Condition 3 is fulfilled because it is fulfilled for $a : scheme_s\ strat \setminus \{S\_final\}$ in crs_{new}, for p_up and s_up in cr_{up} and for S_final in cr_{final}.
- Condition 4 holds because it holds for $strat$ and $scheme_s\ strat \cap \{p\_up, s\_up\} = \varnothing$
- Condition 5 holds because it holds for $strat$ and $s\_rep \in IA\ cr_{final}$.
- The condition 6 is fulfilled because it is fulfilled for $strat$ and and $\{p\_up, s\_up\} \subseteq scheme\ cr_{up}$.
- There are no cycles in $\text{LIFT}(strat, p\_down, p\_combine, s\_combine)$ because, first, there are no cycles in $strat$. Second, it holds

  $$\forall\, cr : crs_{new} \bullet cr \sqsubset_{\text{LIFT}(strat,\ldots)} cr_{up}$$

  because $cr \sqsubset_{strat} cr_{max}$ for $cr : strat \setminus \{cr_{max}\}$ and $IA\ cr_{max} \subseteq IA\ cr_{up}$. Third, we have $cr_{up} \sqsubset_d cr_{final}$.

  For $cr : crs_{new}$, $cr_{up} \sqsubset_{\text{LIFT}(strat,\ldots)} cr$ is impossible because $OA\ cr_{up} \cap scheme_s\ crs_{new} = \varnothing$. Finally, cr_{final} does not depend on any other constituting relation because S_final does not occur in the scheme of any constituting relation except cr_{final}.

It remains to show the correctness of $\text{THEN}(strat_1, p, strat_2)$, as required as condition 3 in the strategy definition. For some $t \in \bowtie (\text{LIFT}(strat, p\_down, p\_combine, s\_combine))$ that contains acceptable solutions for all subproblems, i.e. members of the set $subprs\ strat \cup \{p\_up\}$, we must show $t\ S\_final\ acceptable\_for\ t\ P\_init$. From the definition of $transform_{Lift}$ and the fact that $strat$ is a strategy, it follows that $\exists\, sol : Solution \bullet sol\ acceptable\_for\ (p\_down(t\ P\_init))$. Since the function $p\_combine$ is required to be injective, the solution that is used in cr_{up} to define $t\, p\_up$ and the one that is used in cr_{final} to define $t\, S\_final$ is the same. This gives us

$$sol\ acceptable\_for\ (p\_down(t\ P\_init)) \wedge t\ s\_up\ acceptable\_for\ p\_combine(t\ P\_init, sol)$$

which, according to the requirements on $p\_down, p\_combine$ and $s\_combine$ suffices to conclude $s\_combine(sol, t\ s\_up) = t\ S\_final\ acceptable\_for\ t\ P\_init$.

■