A Two-Layered Approach to Support Systematic Software Development

Maritta Heisel[1] and Stefan Jähnichen[2,3]

[1] Otto-von-Guericke-Universität Magdeburg, Fakultät für Informatik, Institut für Verteilte Systeme, D-39016 Magdeburg, Germany, email: heisel@cs.uni-magdeburg.de
[2] FG Softwaretechnik, Technische Universität Berlin, Sekr. FR 5-6, Franklinstr. 28/29, D-10587 Berlin, Germany, jaehn@cs.tu-berlin.de
[3] GMD FIRST, Rudower Chaussee 5, 12489 Berlin, Germany

Abstract. We present two concepts that help software engineers to perform different software development activities systematically. The concept of an *agenda* serves to represent technical process knowledge. An agenda consists of a list of steps to be performed when developing a software artifact. Each activity may have associated a schematic expression of the language in which the artifact is expressed and validation conditions that help detect errors. Agendas provide methodological support to their users, make development knowledge explicit and thus comprehensible, and they contribute to a standardization of software development activities and products.
The concept of a *strategy* is a formalization of agendas. Strategies model the development of a software artifact as a problem solving process. They form the basis for machine-supported development processes. They come with a generic system architecture that serves as a template for the implementation of support tools for strategy-based problem solving.

Keywords: Software engineering methodology, process modeling, formal methods

1 Introduction

Software engineering aims at producing software systems in a systematic and cost-effective way. Two different aspects are of importance here: first, the *process* that is followed when producing a piece of software, and second, the various intermediate *products* that are developed during that process, e.g., requirements documents, formal specifications, program code, or test cases.

To date, research on the process aspects of software engineering concentrates on the management of large software projects, whereas research on the product aspects of software engineering concentrates on developing appropriate languages to express the various software artifacts, e.g., object-oriented modeling languages, architectural description languages, specification or programming languages.

The work presented in this paper is intended to fill a gap in current software engineering technology: it introduces concepts to perform the *technical* parts of

software processes in a systematic way. By ensuring that the developed products fulfill certain pre-defined quality criteria, our concepts also establish an explicit link between processes and products.

We wish to systematically exploit existing software development knowledge, i.e., the problem-related fine-grained knowledge acquired by experienced software engineers that enables them to successfully produce the different software engineering artifacts. To date, such expert knowledge is rarely made explicit. As a consequence, it cannot be re-used to support software processes and cannot be employed to educate novices. Making development knowledge explicit, on the other hand, would

- support re-use of this knowledge,
- improve and speed up the education of novice software engineers,
- lead to better structured and more comprehensible software processes,
- make the developed artifacts more comprehensible for persons who have not developed them,
- allow for more powerful machine support of development processes.

Agendas and *strategies* help achieve these goals. An agenda gives guidance on how to perform a specific software development activity. It informally describes the different steps to be performed. Agendas can be used to structure quite different activities in different contexts.

Strategies are a formalization of agendas. They aim at machine supported development processes. The basic idea is to model software development tasks as problem solving processes. Strategies can be implemented and supplied with a generic architecture for systems supporting strategy-based problem solving.

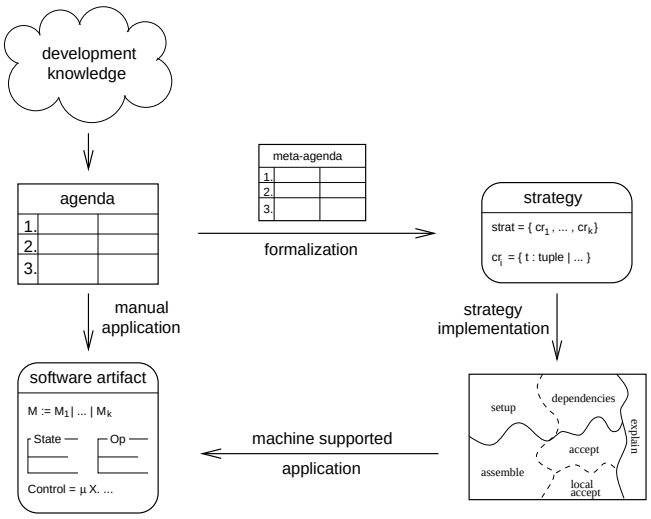

Fig. 1. Relation between agendas and strategies

Figure 1 shows the relation between agendas and strategies. First, the development knowledge used by experienced software engineers must be made explicit. Expressed as an agenda, it can be employed to develop software artifacts independently of machine support. If specialized machine support is sought for, the agenda can be formalized as a strategy. Such a formalization can be performed systematically, following a meta-agenda. Implemented strategies provide machine support for the application of the formalized knowledge to generate software artifacts. In general, the steps of an agenda correspond to subproblems of a strategy.

Agendas and strategies are especially suitable to support the application of formal techniques in software engineering. Formal techniques have the advantage that one can positively guarantee that the product of a development step enjoys certain semantic properties. In this respect, formal techniques can lead to an improvement in software quality that cannot be achieved by traditional techniques alone.

In the following two sections, we present agendas and strategies in more detail. Related work is discussed in Section 4, and conclusions are drawn in Section 5.

2 Agendas

An agenda is a list of steps to be performed when carrying out some task in the context of software engineering. The result of the task will be a document expressed in a certain language. Agendas contain informal descriptions of the steps. With each step, schematic expressions of the language in which the result of the activity is expressed can be associated. The schematic expressions are instantiated when the step is performed. The steps listed in an agenda may depend on each other. Usually, they will have to be repeated to achieve the goal.

Agendas are not only a means to guide software development activities. They also support quality assurance because the steps of an agenda may have validation conditions associated with them. These validation conditions state necessary semantic conditions that the artifact must fulfill in order to serve its purpose properly. When formal techniques are applied, the validation conditions can be expressed and proven formally. Since the validation conditions that can be stated in an agenda are necessarily application independent, the developed artifact should be further validated with respect to application dependent needs.

2.1 An Agenda for Formally Specifying Safety-Critical Software

To illustrate the agenda concept, we present a concrete agenda that supports the formal specification of software for safety-critical applications. Because we want to give the readers a realistic impression of agendas, we present the agenda unabridged and give a brief explanation of the important aspects of software system safety and the language and methodology we use to specify safety-critical software.

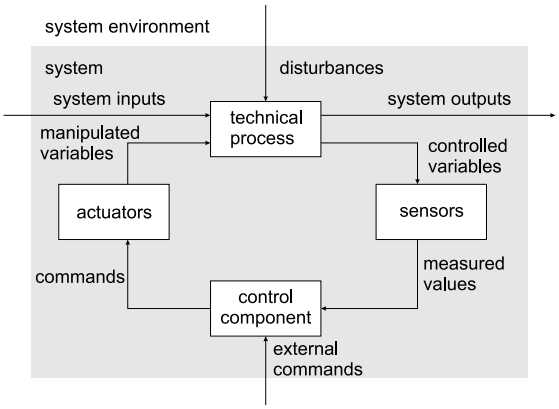

Fig. 2. System Model

The systems we consider in the following, see Figure 2, consist of a technical process that is controlled by dedicated system components being at least partially realized by software. Such a system consists of four parts: the *technical process*, the *control component*, *sensors* to communicate information about the current state of the technical process to the control component, and *actuators* that can be used by the control component to influence the behavior of the technical process.

Two aspects are important for the specification of software for safety-critical systems. First, it must be possible to specify behavior, i.e. how the system reacts to incoming events. Second, the structure of the system's data state and the operations that change this state must be specified. We use a combination of the process algebra real-time CSP [Dav93] and the model-based specification language Z [Spi92] to specify these different aspects.

In [Hei97,HS96] we have described the following principles of the combination of both languages in detail: For each system operation *Op* specified in the Z part of a specification, the CSP part is able to refer to the events *OpInvocation* and *OpTermination*. For each input or output of a system operation defined in Z, there is a communication channel within the CSP part onto which an input value is written or an output value is read from. The dynamic behavior of a software component may depend on the current internal system state. To take this requirement into account, a process of the CSP part is able to refer to the current internal system state via predicates which are specified in the Z part by schemas.

There are several ways to design safety-critical systems, according to the manner in which activities of the control component take place, and the manner in which system components trigger these activities. These different approaches to the design of safety-critical systems are expressed as *reference architectures*.

We present an agenda for a reference architecture where all sensors are passive, i.e., they cannot trigger activities of the control component, and their mea-

surements are permanently available. This architecture is often used for monitoring systems, i.e., for systems whose primary function is to guarantee safety. Examples are the control component of a steam boiler whose purpose it is to ensure that the water level in the steam boiler never leaves certain safety limits, or an inert gas release system, whose purpose is to detect and extinguish fire.

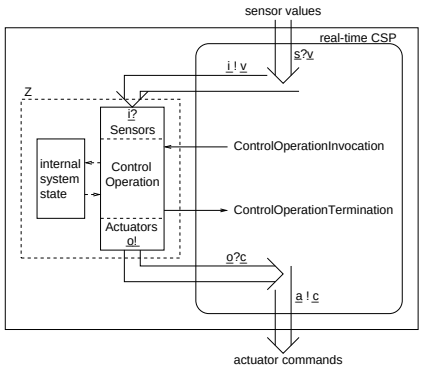

Fig. 3. Software Control Component for Passive Sensors Architecture

Figure 3 shows the structure of a software control component associated with the passive sensors architecture. Such a control component contains a single control operation, which is specified in Z, and which is executed at equidistant points of time. The sensor values $\underline{v}$ coming from the environment are read by the CSP control process and passed on to the Z control operation as inputs. The Z control operation is then invoked by the CSP process, and after it has terminated, the CSP control process reads the outputs of the Z control operation, which form the commands $\underline{c}$ to the actuators. Finally, the CSP control process passes the commands on to the actuators.

Agendas are presented as tables with the following entries for each step:

- a numbering for easy reference,
- an informal description of the purpose of the step,
- a schematic expression that proposes how to express the result of the step in the language used to express the document,
- possibly some informal or formal validation conditions that help detect errors.

The agenda for the passive sensors architecture is presented in Tables 1 and 2, where informal validation conditions are marked "$\circ$", and formal validation conditions are marked "$\vdash$". The dependencies between the steps are shown in Figure 4.

The agenda gives instructions on how to proceed in the specification of a software-based control component according to the chosen reference architecture. We briefly explain its steps.

No.	Step	Schematic Expressions	Validation Conditions
1	Model the sensor values and actuator commands as members of Z types.	$Type ::= \ldots$	
2	Decide on the operational modes of the system.	$MODE ::= Mode1 \mid \ldots \mid ModeK$	
3	Define the internal system states and the initial states.	*InternalSystemState* *mode* : *MODE* ... ... *InternalSystemStateInit* *InternalSystemState*′ ... *InternalSystemState*′;	∘ The internal system state must be an appropriate approximation of the state of the technical process. ⊢ The internal state must contain a variable corresponding to the operational mode. ∘ Each legal state must be safe. ⊢ There must exist legal initial states. ∘ The initial internal states must adequately reflect the initial external system states.
4	Specify an internal Z operation for each operational mode.	$Sensors \mathrel{\widehat=}$ [*InternalSystemState*; *in*1? : *SType*1; ... ; *inN*? : *STypeN* | ⟨*consistency / redundancy*⟩] $Actuators \mathrel{\widehat=}$ [*InternalSystemState*′; *out*1! : *AType*1; ... ; *outM*! : *ATypeM* | ⟨*derivation of commands*⟩] $OpModeJ \mathrel{\widehat=}$ [Δ*InternalSystemState*; *Sensors*; *Actuators* | ...]	⊢ The only precondition of the operation corresponding to a mode is that the system is in that mode. ⊢ For each operational mode and each combination of sensor values there must be exactly one successor mode. ⊢ Each operational mode must be reachable from an initial state. ⊢ There must be no redundant modes.

Table 1. Agenda for the passive sensors architecture, part 1

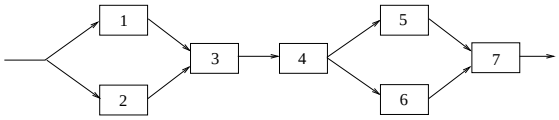

Fig. 4. Dependencies of steps of agenda for passive sensors architecture

Step 1 The defined types depend on the technical properties of the sensors and actuators. If the sensor is a thermometer, the corresponding type will be a subset of the integers. If the sensor can only distinguish a few values, the corresponding type will be an enumeration of these values. The same principles are applied to model the actuators.

Step 2 We assume that the controller is always in one of the operational modes $Mode1, \ldots, ModeK$ that are defined with respect to the needs of the technical process. The operational modes are defined as an enumeration type in Z.

Step 3 Here, the legal internal states of the software component must be defined by means of a Z schema. The components of the internal state must be defined such that, for each time instant, they approximate the state of the technical process in a sufficiently accurate way. The state invariant defines the relations between the components. It comprises the safety-related requirements as well as the functional properties of the legal states. Initial states must be specified, too.[1]

Step 4 We must now specify how the state of the system can evolve. When new sensor values are read, the internal state must be updated accordingly. Each internal Z operation $OpModeJ$ specifies the successor mode of the current mode $ModeJ$ and the commands that have to be given to the actuators, according to the sensor values. It is normally useful to define separate schemas for the sensor values and actuator commands according to the following schematic expressions *Sensors* and *Actuators*. The internal Z operations then import these schemas.

Step 5 The central control operation defined in Z is a case distinction according to the operational modes. For this operation, we give a schematic expression to be instantiated. By importing the schemas *Sensors* and *Actuators* the operation has all inputs from the sensors at its disposition, and it is guaranteed that all actuator commands are defined. The inputs and the current operational mode determine the successor mode which is specified by the internal operations $OpModeI$.

Step 6 For the specification of the control process in real-time CSP, we again can provide schematic expressions to aid building the specification. First, the system must be initialized, establishing an initial state. Then, the recursive process $ControlComponent_{READY}$ is executed. Before invoking the control operation, all associated input values are read from the respective

[1] The schema decoration S' of a schema S is obtained by replacing all declared variables $v_1, v_2, \ldots$ in S by their "primed" versions $v_1', v_2', \ldots$. S and S' denote the state before and after execution of an operation, respectively.

sensor channels ($sensor1, \ldots, sensorN$) in parallel. This is modeled using the parallel composition operator $\|$. When the control operation has terminated, all output values are written to the respective actuator channels ($actuator1, \ldots, actuatorM$) in parallel. The process *Wait INTERVAL* does not accept any event for *INTERVAL* time units and afterwards is ready to accept the termination event before releasing control. The constant *INTERVAL* must be chosen small enough, so that it is guaranteed that the internal system state is always sufficiently up-to-date.

No.	Step	Schematic Expressions
5	Define the Z control operation.	*Control* $\Delta InternalSystemState$ $Sensors;\ Actuators$ --- $mode = Mode1 \Rightarrow OpMode1$ $\wedge \ldots \wedge$ $mode = ModeK \Rightarrow OpModeK$
6	Specify the control process in real-time CSP.	$ControlComponent \mathrel{\widehat{=}} SystemInitExec \rightarrow ControlComp_{READY}$ $ControlComp_{READY} \mathrel{\widehat{=}} \mu\, X \bullet$ $((sensor1?valueS1 \rightarrow in1!valueS1 \rightarrow Skip \parallel \ldots \parallel$ $sensorN?valueSN \rightarrow inN!valueSN \rightarrow Skip);$ $ControlInvocation \rightarrow ControlTermination \rightarrow$ $(out1?valueA1 \rightarrow actuator1!valueA1 \rightarrow Skip \parallel \ldots \parallel$ $outM?valueAM \rightarrow actuatorM!valueAM \rightarrow Skip)$ $\parallel Wait\ INTERVAL);\ X$
7	Specify further requirements if necessary.	

Table 2. Agenda for the passive sensors architecture, part 2

Usually, different phases can be identified for processes expressed as an agenda. The first phase is characterized by the fact that high-level decisions have to be taken. For these decisions, no validation conditions can be stated. In our example, these are the Steps 1 and 2. In the second phase, the language templates that can be proposed are fairly general (for example, we cannot say much more than that schemas should be used to define the internal system states and the initial states), but it is possible to state a number of formal and informal validation conditions. In our example, the second phase consists of Steps 3 and 4. In the third and last phase of an agenda, the parts of the document developed in the earlier phases are assembled. This can be done in a routine or even completely automatic way. Consequently, no validation conditions are necessary for this phase. In our example, the third phase consists of Steps 5 and 6. Step 7

allows specifiers to add specification text, if this is necessary for the particular application. The example shows that

- the agenda is fairly detailed and provides non-trivial methodological support,
- the structure of the specification need not be developed by the specifier but is determined by the agenda,
- the schematic expressions proposed are quite detailed,
- the validation conditions that help avoid common errors are tailored for the reference architecture and the structure of its corresponding specification.

2.2 Agenda-Based Development

In general, working with agendas proceeds as follows: first, the software engineer selects an appropriate agenda for the task at hand. Usually, several agendas will be available for the same development activity, which capture different approaches to perform the activity. This first step requires a deep understanding of the problem to be solved. Once the appropriate agenda is selected, the further procedure is fixed to a large extent. Each step of the agenda must be performed, in an order that respects the dependencies of steps. The informal description of the step informs the software engineer about the purpose of the step. The schematic language expressions associated with the step provide the software engineer with templates that can just be filled in or modified according to the needs of the application at hand. The result of each step is a concrete expression of the language that is used to express the artifact. If validation conditions are associated with a step, these should be checked immediately to avoid unnecessary dead ends in the development. When all steps of the agenda have been performed, a product has been developed that can be guaranteed to fulfill certain application-independent quality criteria.

Agenda-based development of software artifacts has a number of characteristics:

- **Agendas make software processes explicit, comprehensible, and assessable.**
  Giving concrete steps to perform an activity and defining the dependencies between the steps make processes explicit. The process becomes comprehensible for third parties because the purpose of the various steps is described informally in the agenda.
- **Agendas standardize processes and products of software development.**
  The development of an artifact following an agenda always proceeds in a way consistent with the steps of the agenda and their dependencies. Thus, processes supported by agendas are standardized. The same holds for the products: since applying an agenda results in instantiating the schematic expressions given in the agenda, all products developed with an agenda have the same structure.

- **Agendas support maintenance and evolution of the developed artifacts.**
  Understanding a document developed by another person is much less difficult when the document was developed following an agenda than without such information. Each part of the document can be traced back to a step in the agenda, which reveals its purpose. To change the document, the agenda can be "replayed". The agenda helps focus attention on the parts that actually are subject to change. In this way, changing documents is greatly simplified, and it can be expected that maintenance and evolution are less error-prone when agendas are used.
- **Agendas are a promising starting point for sophisticated machine support.**
  First, agendas can be formalized and implemented as strategies, see Section 3. But even if a formal representation of development knowledge is not desired, agendas can form the basis of a process-centered software engineering environment (PSEE) [GJ96]. Such a tool would lead its users through the process described by the agenda. It would determine the set of steps to be possibly performed next and could contain a specialized editor that offers the user the schematic language expressions contained in the agenda. The user would only have to fill in the undefined parts. Furthermore, an agenda-based PSEE could automatically derive the validation obligations arising during a development, and theorem provers could be used to discharge them (if they are expressed formally).

We have defined and used agendas for a variety of software engineering activities that we supported using different formal techniques. These activities include (for more details on the various agendas, the reader is referred to [Hei97]):

- Requirements engineering
  We have defined two different agendas for this purpose. The first supports requirements elicitation by collecting possible events, classifying these events, and expressing requirements as constraints on the traces of events that may occur. Such a requirements description can subsequently be transformed into a formal specification. The second agenda places requirements engineering in a broader context, taking also maintenance considerations into account. This agenda can be adapted to maintain and evolve legacy systems.
- Specification acquisition in general
  There exist several agendas that support the development of formal specifications without referring to a specific application area (such as safety-critical systems). The agendas are organized according to *specification styles* that are language-independent to a large extent.
- Specification of safety-critical software
  Besides the agenda presented in Section 2.1, more agendas for this purpose can be found in [HS97,GHD98].
- Software design using architectural styles
  In [HL97], a characterization of three architectural styles using the formal

description language LOTOS is presented. For each of these styles, agendas are defined that support the design of software systems conforming to the style.

- Object-oriented analysis and design
  An agenda for the object-oriented *Fusion* method [CAB+94] makes the dependencies between the various models set up in the analysis and design phases explicit and states several consistency conditions between them.
- Program synthesis
  We have defined agendas supporting the development of provably correct programs from first-order specifications. Imperative programs can be synthesized using Gries' approach [Gri81], and functional programs can be synthesized using the KIDS approach [Smi90].

3 The Strategy Framework

In the previous section, we have introduced the agenda concept and have illustrated what kind of technical knowledge can be represented as agendas. Agendas are an informal concept whose application does not depend on machine support. They form the first layer of support for systematic software development.

We now go one step further and provide a second layer with the strategy framework. In this layer, we represent development knowledge *formally*. When development knowledge is represented formally, we can reason about this knowledge and prove properties of it. The second aim of the strategy framework is to support the application of development knowledge by machine in such a way that *semantic* properties of the developed product can be guaranteed.

In the strategy framework, a development activity is conceived as the process of constructing a solution for a given problem. A strategy specifies how to reduce a given problem to a number of subproblems, and how to assemble the solution of the original problem from the solution to the subproblems. The solution to be constructed must be *acceptable* for the problem. Acceptability captures the semantic requirements concerning the product of the development process. In this respect, strategies can achieve stronger quality criteria than is intended, e.g., by CASE. The notion of a strategy is *generic* in the definition of problems, solutions and acceptability.

How strong a notion of acceptability can be chosen depends on the degree of formality of problems and solutions. For program synthesis, both problems and solutions can be formal objects: problems can be formal specifications, solutions can be programs, and acceptability can be the total or partial correctness of the program with respect to the specification. For specification acquisition, on the other hand, we might wish to start from informal requirements. Then problems consist of a combination of informal requirements and pieces of a formal specification. Solutions are formal specifications, and a solution is acceptable with respect to a problem if the combination of the pieces of formal specification contained in the problem with the solution is a semantically valid specification. This notion of acceptability is necessarily weaker than the one for program synthesis, because the adequacy of a formal specification with respect to informal

requirements cannot be captured formally. Only if the requirements are also expressed formally, a stronger notion of acceptability is possible for specification acquisition.

The strategy framework is defined in several stages, leading from simple mathematical notions to an elaborated architecture for systems supporting strategy-based problem solving. In the first stages, strategies are defined as a purely declarative knowledge representation mechanism. Experience has shown that formal knowledge representation mechanisms are (i) easier to handle and (ii) have a simpler semantics when they are declarative than when they are procedural. As for strategies, (i) agendas can be transformed into strategies in a routine way (see Section 1), and (ii) the relational semantics of strategies supports reasoning about and combination of strategies. Further stages gradually transform declaratively represented knowledge into executable constructs that are provided with control structures to guide an actual problem solving process. Figure 5 shows the different stages.

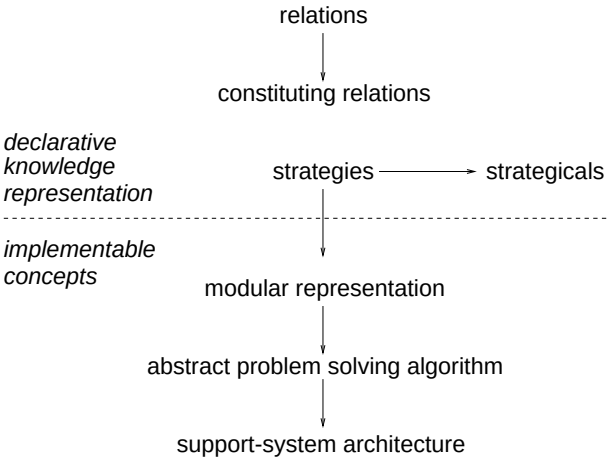

Fig. 5. Stages of definitions

The basic stage consists in defining a suitable notion of *relation*, because, formally, strategies establish a relation between a problem and the subproblems needed to solve it, and between the solutions of the subproblems and the final solution. Relations are then specialized to problem solving, which leads to the definition of *constituting relations*. *Strategies* are defined as sets of constituting relations that fulfill certain requirements. In particular, they may relate problems only to acceptable solutions. *Strategicals* are functions combining strategies; they make it possible to define more powerful strategies from existing ones.

To make strategies implementable, they are represented as *strategy modules*, which rely on constructs available in programming languages. In particular, relations are transformed into functions. The next step toward machine support

consists in defining an *abstract problem solving algorithm*. This algorithm describes the manner in which strategy-based problem solving proceeds and can be shown to lead to acceptable solutions. The *generic system architecture* provides a uniform implementation concept for practical support systems.

In the following, we sketch the definitions of the strategy framework (for details, see [Hei97]). Subsequently, we discuss its characteristics. Strategies, strategicals, and strategy modules are formally defined in the language Z [Spi92]. This does not only provide precise definitions of these notions but also makes reasoning about strategies possible.

3.1 Relations

In the context of strategies, it is convenient to refer to the subproblems and their solutions by *names*. Hence, our definition of strategies is based on the the notion of relation as used in the theory of relational databases [Kan90], instead of the usual mathematical notion of relation. In this setting, relations are sets of tuples. A tuple is a mapping from a set of *attributes* to *domains* of these attributes. In this way, each component of a tuple can be referred to by its attribute name. In order not to confuse these domains with the domain of a relation as it is frequently used in Z, we introduce the type *Value* as the domain for all attributes and define tuples as finite partial functions from attributes to values: $tuple : \mathbb{P}(Attribute \nrightarrow Value)$, where $\mathbb{P}$ is the powerset operator. 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.

$$relation : \mathbb{P}(\mathbb{P}\, tuple)$$

$$\forall\, r : relation \bullet \forall\, t_1, t_2 : r \bullet \operatorname{dom} t_1 = \operatorname{dom} t_2$$

3.2 Constituting Relations

Constituting relations specialize relations for problem solving. Attributes can either be *ProblemAttribute*s or *SolutionAttribute*s, whose values must be *Problem*s or *Solution*s, respectively. The types *Problem* and *Solution* are generic parameters.

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

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

Acceptability, the third generic parameter, is a relation between problems and solutions: $\_acceptable\_for\_ : Solution \leftrightarrow Problem$. By default, we use the distinguished attributes *P_init* and *S_final* to refer to the initial problem and its final solution.

The schemes of constituting relations are divided into *input attributes IA* and *output attributes OA*. The constituting relations restrict the values of the output attributes, given the values of the input attributes. Thus, they determine an order on the subproblems that must be respected in the problem solving process. Based on the partitioning of schemes, it is possible to define a dependency relation on constituting relations. A constituting relation cr_2 directly depends on another such relation cr_1 ($cr_1 \sqsubset_d cr_2$) if one of its input attributes is an output attribute of the other relation: $OA\ cr_1 \cap IA\ cr_2 \neq \emptyset$. For any given set *crs* of constituting relations, a *dependency relation* $\sqsubset_{crs}$ is defined to be the transitive closure of the direct dependency relation it determines.

A set of constituting relations defining a strategy must conform to our intuitions about problem solving. Among others, the following conditions must be satisfied:

- The original problem to be solved must be known, i.e. *P_init* must always be an input attribute.
- The solution to the original problem must be the last item to be determined, i.e. *S_final* must always be an output attribute.
- Each attribute value except that 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.
- The dependency relation on the constituting relations must not be cyclic.

Finite sets of constituting relations fulfilling these and other requirements are called *admissible*. For a complete definition of admissibility, see [Hei97].

Example. For transforming the agenda presented in Section 2.1 into a strategy, we must first define suitable notions of problems, solutions, and acceptability. A problem pr : *SafProblem* consists of three parts: the part $pr.req$ contains an informal requirements description, the part $pr.context$ contains the specification fragments developed so far, and the part $pr.to\_develop$ contains a *schematic* Z-CSP expression that can be instantiated with a concrete one. This schematic expression specifies the syntactic class of the specification fragment to be developed, as well as how the fragment is embedded in its context. Solutions are syntactically correct Z-CSP expressions, and a solution *sol* : *SafSolution* is acceptable for 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-CSP specification.

3.3 Strategies

We define strategies as admissible sets of constituting relations that fulfill certain conditions. Let $strat = \{cr_0, \ldots, cr_{max}\}$ and $scheme_s\ strat = scheme\ cr_0 \cup \ldots \cup scheme\ cr_{max}$. The set *strat* is a strategy if it is admissible and

- the set $scheme_s\ strat$, contains the attributes *P_init* and *S_final*,

– for each problem attribute a of $scheme_s\ strat$, a corresponding solution attribute, called $sol\ a$, is a member of the scheme, and vice versa,
– if a member of the relation $\bowtie\ strat$[2] contains acceptable solutions for all problems except P_init, then it also contains an acceptable solution for P_init. Thus, if all subproblems are solved correctly, then the original problem must be solved correctly as well.

$$
\begin{array}{|l}
strategy : \mathbb{P}(\mathbb{F}\ const\_rel) \\
\hline
\forall\ strat : strategy \bullet \\
\quad admissible\ strat \land \\
\quad \{P\_init, S\_final\} \subseteq scheme_s\ strat \land \\
\quad (\forall\ a : ProblemAttribute \bullet a \in scheme_s\ strat \Leftrightarrow sol\ a \in scheme_s\ strat) \land \\
\quad (\forall\ res : \bowtie\ strat \bullet \\
\qquad (\forall\ a : subprs_s\ strat \bullet (res\ (sol\ a))\ acceptable\_for\ (res\ a)) \\
\qquad \Rightarrow (res\ S\_final)\ acceptable\_for\ (res\ P\_init))
\end{array}
$$

The last condition guarantees that a problem that is solved exclusively by application of strategies is solved correctly. This condition requires that strategies solving the problem directly must produce only acceptable solutions. Figure 6 illustrates the definition of strategies, where arrows denote the propagation of attribute values.

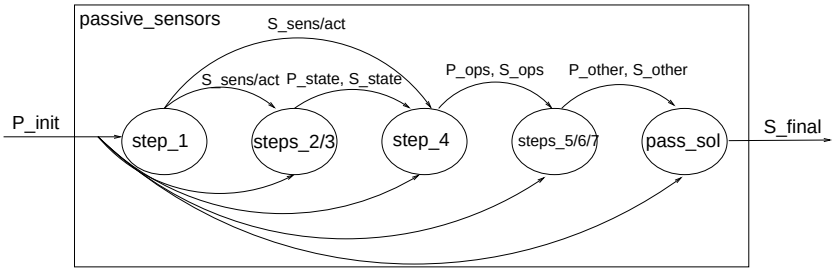

Fig. 6. Strategy for passive sensors

Example. When transforming an agenda into a strategy, we must decide which of the steps of the agenda will become subproblems of the strategy. If the result of a step consists in a simple decision or can be assembled from already existing partial solutions, then no subproblem corresponding to the step is necessary.

[2] A *join* $\bowtie$ combines two relations. The scheme of the joined relation is the union of the scheme of the given relations. On common elements of the schemes, the values of the attributes must coincide. The operation $\bowtie$ denotes the join of a finite sets of relations.

Considering the agenda of Section 2.1, we decide that Steps 2, 5, and 6 need not become subproblems. Hence, we can define

$$passive\_sensors = \{step\_1, steps\_2/3, step\_4, steps\_5/6/7, pass\_sol\}$$

Figure 6 shows how attribute values are propagated. The constituting relation $step_1$, for example, has as *P_init* as its only input attribute, and *P_sens/act* and *S_sens/act* as its output attributes. The requirements *P_sens/act.req* consist of the requirements *P_init.req* with the addition "Model the sensor values and actuator commands as members of Z types." (see Table 1). The context *P_sens/act.context* is the same as for *P_init*, and *P_sens/act.to_develop* consists of the single metavariable type_defs : *Z-ax_def*, which indicates that axiomatic Z definitions have to be developed. For the solution *S_sens/act* of problem *P_sens/act*, the only requirement is that it be acceptable. The other constituting relations are defined analogously. The complete strategy definition can be found in [Hei97].

3.4 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 combining lower-level ones or to restrict the set of applicable strategies, thus contributing to a larger degree of automation of the development process.

Three strategicals are defined [Hei97] 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 stepwise repetition of a strategy. Such a strategical is useful in the context of specification acquisition, where often several items of the same kind need to be developed. To increase applicability of the REPEAT strategical, we also define a LIFT strategical that transforms a strategy for developing one item into a strategy for developing several items of the same kind.

3.5 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. The implementation of a strategy should be a module with a clearly defined interface to other strategies and the rest of the system.

Because strategies are defined as relations, 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. A type *ExtInfo* is used to select one of these combinations, thus transforming relations into functions. Such external information can be derived from user input or can be computed automatically. A strategy module consists of the following items:

- the set *subp* : $\mathbb{P}$ *ProblemAttribute* of subproblems it produces,

- a dependency relation $\_depends\_on\_ : ProblemAttribute \leftrightarrow ProblemAttribute$ on these subproblems,
- for each subproblem, a procedure $setup : tuple \times ExtInfo \nrightarrow Problem$ that defines it, using the information in the initial problem and the subproblems and solutions it depends on, and possibly external information,
- for each solution to a subproblem, a predicate $local\_accept : tuple \leftrightarrow Solution$ that checks whether or not the solution conforms to the requirements stated in the constituting relation of which it is an output attribute,
- a procedure $assemble : tuple \times ExtInfo \nrightarrow Solution$ describing how to assemble the final solution, and
- a test $accept\_ : \mathbb{P}\, tuple$ of acceptability for the assembled solution.

Optionally, an *explain* component may be added that explains *why* a solution is acceptable for a problem, e.g., expressed as a correctness proof.

3.6 An Abstract Problem Solving Algorithm

The abstract problem solving algorithm consists of three functions, called *solve*, *apply*, and *solve_subprs*. The function *solve* has a problem *pr* as its input. To solve this problem, a strategy *strat* must be selected from the available strategies. The function *apply* is called that tries to solve the problem *pr* with strategy *strat*. If this is successful, then the value of the attribute *S_final* obtained from the tuple yielded by *apply* is the result of the *solve* function. Otherwise, another trial is made, using a different strategy.

The function *apply* first calls another function *solve_subprs* to solve the subproblems generated by the strategy *strat*. It then sets up the final solution and checks it for acceptability. If the acceptability test fails, *apply* yields a distinguished failure element. Otherwise, it yields a tuple that lies in $\bowtie$ *strat* (see Section 3.3).

The function *solve_subprs* has as its arguments the tuple consisting of the attribute values determined so far, and a set of subproblems still to be solved. It applies *solve* recursively to all subproblems contained in its second argument.

Problem solving with strategies usually requires user interaction. For the functions *solve*, *apply*, and *solve_subprs*, user interaction is simulated by providing them with an additional argument of type seq *UserInput*, where the type *UserInput* comprises all possible user input. User input must be converted into external information, as required by the strategy modules. To achieve this, we use *heuristic functions*. Heuristic functions are those parts of a strategy implementation that can be implemented with varying degrees of automation. It is also possible to automate them gradually by replacing, over time, interactive parts with semi- or fully automatic ones.

It can be proven 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, then this solution is acceptable.

3.7 Support-System Architecture

We now define a system architecture that describes how to implement support systems for strategy-based problem solving. Figure 7 gives a general view of the architecture which is described in more detail in [HSZ95]. This architecture is a sophisticated implementation of the functions given in the last section. We introduce data structures that represent the state of the development of an artifact. This ensures that the development process is more flexible than would be possible with a naive implementation of these functions in which all intermediate results would be buried on the run-time stack. It is not necessary to first solve a given subproblem completely before starting to solve 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 insofar as they have been determined. Links between siblings represent dependencies on other problems or solutions.

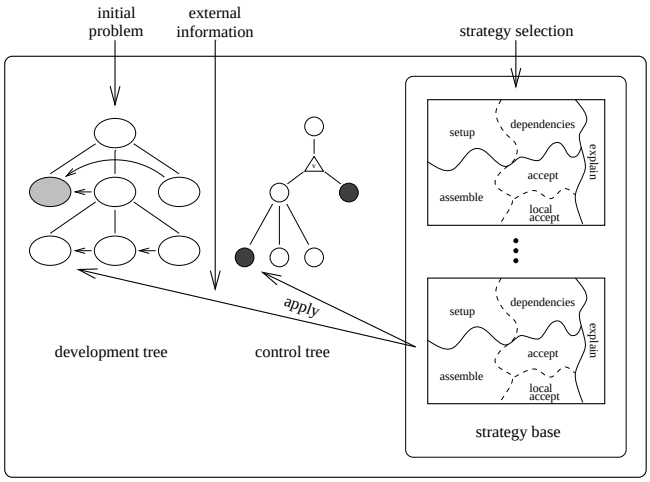

Fig. 7. General view of the system architecture

The data in the control tree are concerned only with the future development. Its nodes represent uncompleted tasks and point to nodes in the development tree that do not yet contain solutions. The degrees of freedom in choosing 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 used in strategy-based problem solving via 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. Secondly, a strategy is selected from the strategy base. Applying the strategy to the problem entails extending the development tree with nodes for the new subproblems, installing the functions of the strategy module in these nodes, and setting up dependency links between them. The control tree must also be extended.

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 and tested for local acceptability, then the functions to assemble and accept a solution are called; if the assembling and accepting functions are successful, then the solution is recorded in the respective node of the development tree. Because the control tree contains only references to unsolved problems, it shrinks whenever a solution to a problem is produced, and the problem-solving process terminates when the control tree vanishes. The result of the process is not simply the developed solution – instead, it is a development tree where all nodes contain acceptable solutions. This data structure provides valuable documentation of the development process, which produced it, and can be kept for later reference.

A research prototype that was built to validate the concept of strategy and the system architecture developed for their machine-supported application. The program synthesis system IOSS (Integrated Open Synthesis System) [HSZ95] supports the development of provably correct imperative programs.

3.8 Discussion of Strategies

The most important properties of the strategy framework are:

- **Uniformity.** The strategy framework provides a uniform way of representing development knowledge. It is independent of the development activity that is performed and the language that is used. It provides a uniform mathematical model of problem solving in the context of software engineering.
- **Machine Support.** 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.** To guarantee acceptability of a solution developed with an implemented system, the functions *local_accept* and *accept* are the

only components that have to be verfied. Hence, also support systems that are not verified compeletly can be trustworthy.
- **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.

4 Related Work

Recently, efforts have been made to support re-use of special kinds of software development knowledge: *Design patterns* [GHJV95] have had much success in object-oriented software construction. They represent frequently used ways to combine classes or associate objects to achieve a certain purpose. Furthermore, in the field of software architecture [SG96], *architectural styles* have been defined that capture frequently used design principles for software systems. Apart from the fact that these concepts are more specialized in their application than agendas, the main difference is that design patterns and architectural styles do not describe *processes* but *products*.

Agendas have much in common with approaches to software process modeling [Huf96]. The difference is that software process modeling techniques cover a wider range of activities, e.g., management activities, whereas with agendas we always develop a document, and we do not take roles of developers etc. into account. Agendas concentrate more on technical activities in software engineering. On the other hand, software process modeling does not place so much emphasis on validation issues as agendas do.

Chernack [Che96] uses a concept called *checklist* to support inspection processes. In contrast to agendas, checklists presuppose the existence of a software artifact and aim at detecting defects in this artifact.

Related to our aim to provide methodological support for applying formal techniques is the work of Souquières and Lévy [SL93]. They support specification acquisition with *development operators* that reduce *tasks* to subtasks. However, their approach is limited to specification acquisition, and the development operators do not provide means to validate the developed specification.

Astesiano and Reggio [AR97] also emphasize the importance of method when using formal techniques. In the "method pattern" they set up for formal specification, agendas correspond to *guidelines*.

A prominent example of knowledge-based software engineering, whose aims closely resemble our own, is the Programmer's Apprentice project [RW88]. There, programming knowledge is represented by *clichés*, which are prototypical examples of the artifacts in question. The programming task is performed by

"inspection"– i.e., by choosing an appropriate cliché and customizing it. In comparison to clichés, agendas are more process-oriented.

Wile's [Wil83] development language Paddle provides a means of describing procedures for transforming specifications into programs. Since carrying out a process specified in Paddle involves executing the corresponding program, one 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 other, more recent approaches to represent software development processes by process programming languages [Ost87,SSW92].

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

The strategy framework uses ideas similar to tactical theorem proving, which has first been employed in Edinburgh LCF [Mil72]. *Tactics* are programs that implement "backward" application of logical rules. The goal-directed, top-down approach to problem solving is common to tactics and strategies. However, tactics set up all subgoals at once when they are invoked. Dependencies between subgoals can only be expressed schematically by the use of *metavariables*. Since tactics only perform goal reduction, there is no equivalent to the *assemble* and *accept* functions of strategies.

5 Conclusions

We have shown that the concept of an agenda bears a strong potential to

- structure processes performed in software engineering,
- make development knowledge explicit and comprehensible,
- support re-use and dissemination of such knowledge,
- guarantee certain quality criteria of the developed products,
- facilitate understanding and evolution of these products,
- contribute to a standardization of products and processes in software engineering that is already taken for granted in other engineering disciplines,
- lay the basis for powerful machine support.

Agendas lead software engineers through different stages of a development and propose validations of the developed product. Following an agenda, software development tasks can be performed in a fairly routine way. When software engineers are relieved from the task to find new ways of structuring and validating the developed artifacts for each new application, they can better concentrate on the peculiarities of the application itself.

We have validated the concept of an agenda by defining and applying a number of agendas for a wide variety of software engineering activities. Currently, agendas are applied in industrial case studies of safety-critical embedded systems in the German project ESPRESS [GHD98].

Furthermore, we have demonstrated that strategies are a suitable concept for the formal representation of development knowledge. The generic nature of strategies makes it possible to support different development activities. Strategicals contribute to the scalability of the approach. The uniform representation as strategy modules makes strategies implementable and isolates those parts that are responsible for acceptability and the ones that can be subject to automation.

The generic system architecture that complements the formal strategy framework gives guidelines for the implementation of support systems for strategy-based development. The representation of the state of development by the data structure of development trees contributes essentially to the practical applicability of the strategy approach.

In the future, we will investigate to what extent agendas are independent of the language which is used to express the developed artifact, and we will define agendas for other activities such as testing and specific contexts, e.g., object-oriented software development. Furthermore, we will investigate how different instances of the system architecture can be combined. This would provide integrated tool support for larger parts of the software lifecycle.

References

[AR97] E. Astesiano and G. Reggio. Formalism and Method. In M. Bidoit and M. Dauchet, editors, *Proceedings TAPSOFT'97*, LNCS 1214, pages 93–114. Springer-Verlag, 1997.

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

[CAB+94] D. Coleman, P. Arnold, St. Bodoff, Ch. Dollin, H. Gilchrist, F. Hayes, and P. Jeremaes. *Object-Oriented Development: The Fusion Method.* Prentice Hall, 1994.

[Che96] Yuri Chernack. A statistical approach to the inspection checklist formal synthesis and improvement. *IEEE Transactions on Software Engineering*, 22(12):866–874, December 1996.

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

[GHD98] Wolfgang Grieskamp, Maritta Heisel, and Heiko Dörr. Specifying safety-critical embedded systems with Statecharts and Z: An agenda for cyclic software components. In E. Astesiano, editor, *Proc. ETAPS-FASE'98*, LNCS 1382, pages 88–106. Springer-Verlag, 1998.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software.* Addison Wesley, Reading, 1995.

[GJ96] P. Garg and M. Jazayeri. Process-centered software engineering environments: A grand tour. In A. Fuggetta and A. Wolf, editors, *Software Process*, Trends in Software 4, chapter 2, pages 25–52. Wiley, 1996.

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

[Hei97] Maritta Heisel. *Methodology and Machine Support for the Application of Formal Techniques in Software Engineering.* Habilitation Thesis, TU Berlin, 1997.

[HL97] Maritta Heisel and Nicole Lévy. Using LOTOS patterns to characterize architectural styles. In M. Bidoit and M. Dauchet, editors, *Proceedings TAPSOFT'97*, LNCS 1214, pages 818–832. Springer-Verlag, 1997.

[HS96] Maritta Heisel and Carsten Sühl. Formal specification of safety-critical software with Z and real-time CSP. In E. Schoitsch, editor, *Proceedings 15th International Conference on Computer Safety, Reliability and Security (SAFECOMP)*, pages 31–45. Springer-Verlag London, 1996.

[HS97] Maritta Heisel and Carsten Sühl. Methodological support for formally specifying safety-critical software. In P. Daniel, editor, *Proceedings 16th International Conference on Computer Safety, Reliability and Security (SAFECOMP)*, pages 295–308. Springer-Verlag London, 1997.

[HSZ95] 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.

[Huf96] Karen Huff. Software process modelling. In A. Fuggetta and A. Wolf, editors, *Software Process*, Trends in Software 4, chapter 2, pages 1–24. Wiley, 1996.

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

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

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

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

[SG96] Mary Shaw and David Garlan. *Software Architecture.* IEEE Computer Society Press, Los Alamitos, 1996.

[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.

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

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

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