Agendas – A Concept to Guide 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

We present the concept of an agenda. This concept serves to represent process knowledge in the area of software development. 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 some validation conditions that help detect errors. We present example agendas and discuss the distinguishing features of the agenda concept in detail.

Agendas provide methodological support to their users, make development knowledge explicit and thus comprehensible, and contribute to a standardization of software development activities and products. Agendas are flexible and useful in many different contexts and lay the basis for powerful machine support in software development.

1 Re-use of Development Knowledge

Software development comprises a number of development activities, the result of each of which is an artifact, such as a requirements document, a formal specification, program code, and test cases. Experienced software engineers have over time acquired problem-related fine-grained knowledge how to perform the various development activities.

To date, such expert knowledge is rarely made explicit. This forces each software engineer to gain experience from scratch. Previously acquired knowledge is not re-used to support software processes and not 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.

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.

This work presents the concept of an *agenda*. An agenda gives guidance on how to perform a specific software development activity. Whereas concrete agendas are very much oriented on the

activity to be supported, the general concept of an agenda is not specialized to a programming paradigm such as object-orientedness or an activity such as software design, as is the case for design patterns and architectural styles. Agendas can be used for structuring quite different activities and in different contexts. We have set up and used agendas that support requirements engineering, specification acquisition, software design using architectural styles, and developing code from specifications [Hei97].

Agendas 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. Moreover, when the semantics of the developed artifact is taken into account, stronger machine support, e.g., by theorem provers, becomes possible.

A major drawback of formal techniques, however, is that they are not easy to apply. Users of formal techniques need an appropriate education. They have to deal with lots of details, and often they are left alone with a mere formalism without any guidance on how to use it. For the application of formal techniques in software engineering, the means for mastering complexity and for finding common patterns in different products and different processes are at least as important as in classical software engineering. Agendas tackle these problems.

In the following, we introduce the concept of an agenda in more detail and give examples of concrete agendas (Section 2). Then we discuss the distinguishing features of agendas in Section 3. Related work is discussed in Section 4, and a summary of what has been achieved concludes the paper (Section 5).

2 Agendas

We first introduce the concept of an agenda in general and then present as examples two agendas that support the development of formal specifications for safety-critical software.

2.1 General Concept

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, similar to the general process proposed by the spiral model of software engineering. Agendas are presented as tables, see e.g. Tables 1 and 2, together with a dependency graph of the steps, see e.g. Figure 2.

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. The purpose of the artifact is always clear in the context of an agenda, because the agendas represent specific development knowledge. When formal techniques are applied, the validation conditions can be expressed and proven in a formal way. Since the verification 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.

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 (which nevertheless requires creativity) 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. This product should then be subject to further validation, taking the specific application into account.

Following an agenda gives no guarantee of success. Agendas cannot replace creativity, but they can tell the software engineer what needs to be done and can help avoid omissions and inconsistencies. Their use lies in an improvement of the quality of the developed products and the possibility for reusing the knowledge incorporated in an agenda.

If formal techniques are applied, the development of agendas that support some software development task needs collaboration between experts on formal techniques and those parties who will be applying a formal technique. In a knowledge engineering process, experts and software engineers jointly define agendas for the development task. It is the responsibility of the software engineers to make their knowledge explicit and to identify the steps that must be taken when performing the development task. Software Engineers and formal techniques experts can then work together and relate the results of the different steps identified to the formalism to be used.

2.2 Agendas for Formally Specifying Safety-Critical Software

In this section, we present two concrete agendas that support the formal specification of software for safety-critical applications. Because we want to give the readers a realistic impression of agendas, we present the agendas 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. To understand the main points of this paper, however, it is not necessary to understand every detail of this section. We will refer to the agendas presented here to illustrate the specific features of agendas discussed in Section 3.

Although every software-based system potentially benefits from the application of formal methods, their use is particularly advantageous in the development of safety-critical systems. The potential damage operators and developers of a safety-critical system have to envisage in case of an accident may be much greater than the additional costs of applying formal methods in system development. It is therefore worthwhile to develop formal methods tailor-made for the development of safety-critical systems.

The system class we consider pertains to technical processes that have to be 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.

Most safety-critical systems are *reactive*. Hence, 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. These requirements lead us to use a combination of the process algebra real-time CSP and the model-based specification language Z. Readers not familiar with these languages may consult [Dav93] and [Spi92], respectively.

In [HS96, Hei97] 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 event *OpExecution*. 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 can be expressed as *reference architectures*.

We present agendas for two such reference architectures which cover frequently used design principles of safety-critical systems. The first architecture assumes that sensors are passive measuring devices. The second architecture assumes that sensors can cause interrupts in the control component. For both architectures, we assume that it is appropriate to distinguish several *operational modes* of the system. Within distinct modes, which can model different environmental or internal conditions, the behavior of the system – and thus of the control component – may be totally different.

2.2.1 Agenda for Passive Sensors Architecture

In the passive sensors architecture, all sensors are passive, i.e., they cannot trigger activities of the control component, and their measurements 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, and an inert gas release system, whose purpose is to detect and extinguish fire.

Figure 1 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.

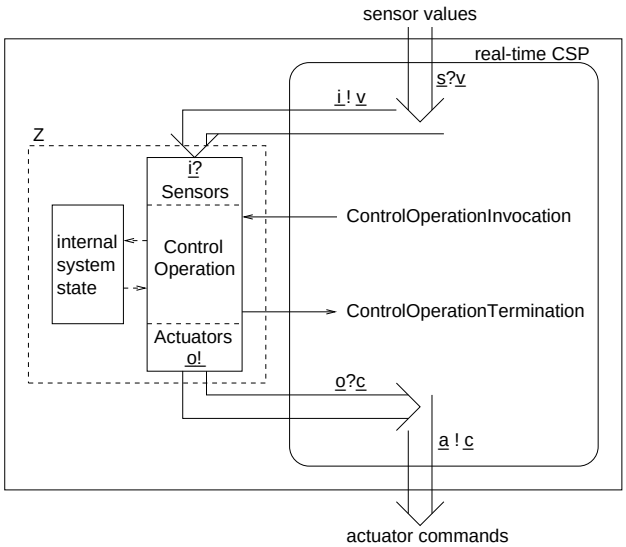

Figure 1: Software Control Component for Passive Sensors Architecture

The agenda for the passive sensors architecture is summarized in Tables 1 and 2. The dependencies between the steps are shown in Figure 2. The agenda gives instructions on how to proceed in the specification of a software-based control according to the chosen reference architecture. It consists of seven steps, some of which have validation obligations associated with them. As indicated by Figure 2, the steps need not be carried out exactly in the given order. Some of them are independent of each other.

No.	Step	Schematic Expressions	Validation Conditions
1	Model the sensor values and actuator commands as members of Z types.	$Type ::= value_1 \mid \ldots \mid value_n$ $Type ::= k \,..\, m$	
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.	*State* $mode : MODE$ $\ldots$ *integrity constraints* *InitState* $State'$ *initiality conditions*	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.	$Sensors \mathrel{\widehat=}$ $[State; in_1? : ST_1; \ldots; in_N? : S_N \mid$ *consistency conditions* *redundancy mechanisms*$]$ $Actuators \mathrel{\widehat=}$ $[State'; out_1! : AT_1; \ldots; out_M! : AT_M \mid$ *derivation of commands*$]$ *OpModeI* $\Delta State$ $Sensors; Actuators$ $\ldots$	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

No.	Step	Schematic Expressions	Validation Conditions
5	Define the Z control operation.	$ControlOperation$ $\Delta SystemState$ $Sensors; Actuators$ --- $mode = Mode1 \Rightarrow OpMode1$ $\land \ldots \land$ $mode = ModeK \Rightarrow OpModeK$	
6	Specify the control process in real-time CSP.	$ControlComponent \mathrel{\widehat{=}} SystemInitExecution \rightarrow ControlComponent_{READY}$ $ControlComponent_{READY} \mathrel{\widehat{=}} \mu X \bullet$ $((sensor1?valueS1 \rightarrow input1!valueS1 \rightarrow Skip \parallel \ldots \parallel$ $sensorN?valueSN \rightarrow inputN!valueSN \rightarrow Skip);$ $ControlOperationInvocation \rightarrow ControlOperationTermination \rightarrow$ $(output1?valueA1 \rightarrow actuator1!valueA1 \rightarrow Skip \parallel \ldots \parallel$ $outputM?valueAM \rightarrow actuatorM!valueAM \rightarrow Skip)$ $\parallel$ $Wait\ INTERVAL); X$	
7	Specify further requirements if necessary.		

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

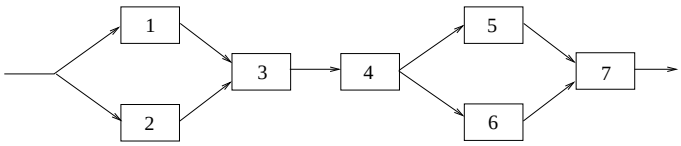

Figure 2: Dependencies of steps of agenda for passive sensors architecture

Since the specification of safety-critical software is not the main subject of this paper, we do not explain the steps, the schematic expressions, and the validation conditions in detail. Some aspects of the agenda will be discussed in Section 3. We only note that

- the agenda is fairly detailed,
- 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 verification conditions that help avoid common errors are tailored for the reference architecture and the structure of its corresponding specification.

2.2.2 Agenda for Active Sensors Architecture

An active sensor controls a certain variable of the technical process and independently reports certain changes of the controlled variable to the control component at arbitrary time instants. Such a report immediately triggers a handling operation within the control component. The active sensors architecture is applicable to systems with only active sensors.

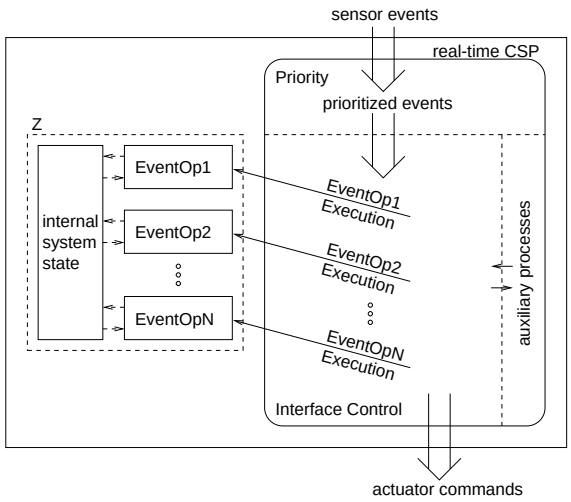

Figure 3: Software Control Component for Active Sensors Architecture

Figure 3 shows the structure of a software control component associated with the active sensors architecture. The CSP part of such a control component consists of three parallel processes. A *Priority* process receives the sensor events from the environment. If several events occur at the same time, this process defines which of these events is treated with priority. Depending on the prioritized events passed on from the *Priority* process, an *InterfaceControl* process invokes a Z operation to update the internal state of the software controller. The Z operations do not correspond to operational modes, as in the passive sensors architecture, but to events that cause transitions between internal modes. The *InterfaceControl* process is also responsible for sending

actuator commands to the environment. Finally, there may be auxiliary processes that interact only with the *InterfaceControl* process, not with the environment or with the *Priority* process. The parallel composition of the auxiliary processes forms the third subprocess of the control component.

The active sensors architecture is suitable for systems whose purpose is different from merely ensuring safety of a technical process by monitoring it, but which continuously have to react to user commands or other stimuli from the environment. Examples are microwave ovens, gas burners, or railroad crossings.

An overview of the agenda is given in Table 3. The dependencies between the steps are shown in Figure 4. For reasons of simplicity, we do not show the schematic expressions that are associated with the different steps.

No	Step	Validation Condition
1	Model the sensors and actuators as sets of CSP events or Z types.	
2	Decide on auxiliary processes.	
3	Decide on the operational modes of the system and the initial modes.	
4	Set up a mode transition relation, specifying which events relate which modes.	All events identified in Step 1 and all modes defined in Step 3 must occur in the transition relation. The omission of a mode-event pair from the relation must be justified. All modes must be reachable from an initial mode.
5	Define the internal system states and the initial states.	The internal system state must be an appropriate approximation of the state of the technical process. Each legal state must be safe. There must exist legal initial states. For each initial internal state, the controller must be in an initial mode.
6	Specify a Z operation for each mode transition contained in the mode transition relation.	These operations must be consistent with the mode transition relation.
7	Define the auxiliary processes identified in Step 2.	The alphabets of these processes must not contain external events or events related to the Z part of the specification.
8	Specify priorities on events (optional).	The priorities must not be cyclic.
9	Specify the interface control process.	All prioritized external events and all internal events must occur as initial events of the branches of the interface control process. The preconditions of the invoked Z operations must be satisfied.
10	Define the overall control process.	The auxiliary processes must communicate with the interface control process.
11	Define further requirements or environmental assumptions if necessary.	

Table 3: Agenda for the Active Sensors Architecture

This agenda is even more elaborated than the one for the passive sensors architecture. Again, it is not intended that the reader understand all details of the agenda. Instead, we want to give

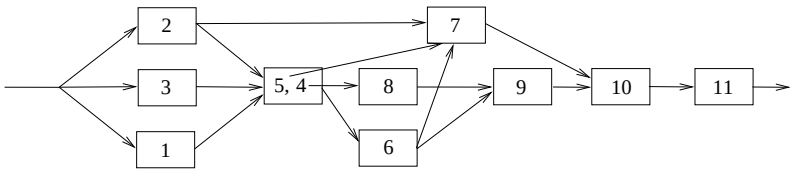

Figure 4: Dependencies of steps

an impression of the kind of guidance offered by agendas and the preciseness in which the steps can be expressed and validated. In the next session, we will discuss some aspects of the agendas presented here in more detail.

3 Discussion of Agendas

Agendas have a number of distinguishing features that we discuss one by one and illustrate them using the examples of Section 2.2.

Selecting an agenda corresponds to a high-level decision. Agendas provide detailed guidance for performing well-defined tasks in a given context. Once an agenda is chosen, the process to be followed and its resulting product are clearly structured. Selecting an appropriate strategy for a given task, however, is a high-level decision that must be taken according to the needs of the particular application at hand.

For example, when a software controller for a safety-critical system has to be developed, the decision as to which agenda to apply depends on the technical properties of the system and the sensors to be used. In Section 2.2, we have informally described the kinds of systems for which the agendas are suitable.

Another example is the design of a software system following some architectural style. Agendas to design systems according to the styles *repository*, *pipe/filer*, and *event-action* are defined in [Hei97]. Before such an agenda can be profitably used, an appropriate architectural style for the system to be implemented must be selected. This selection depends on the purpose and the features of the system. For each agenda, we can give rules of thumb when it is suitable. However, this does not replace creativity of software engineers.

Agendas provide non-trivial methodological support for software development activities. The agendas presented in Section 2.2 clearly show that agendas can have a non-trivial content and provide substantial guidance to developers.

When comparing the agendas for the active and the passive sensors architectures, we notice that in the passive sensors architecture, one Z operation per *operational mode* is developed, see Step 4 of Table 1. In the active sensors architecture, on the other hand, one Z operation for each possible *event* is developed, see Step 6 of Table 3. This decision is not obvious, and following an agenda spares specifiers the labor to reflect this decision over and over again for each new system to be specified.

We have validated the reference architectures and agendas of Section 2.2 by several non-trivial case studies, including the specification of a steam boiler [Hei96, Süh96], an inert gas system [Hei97], a microwave oven [Hei95], an elevator [Süh96], a gas burner [HS96], and a railway crossing [HS97].

Currently, more agendas for the specification of safety-critical embedded software are developed in the German project ESPRESS [ESP], which is a joint project with partners from industry, research institutions, and universities. In this project, a combination of Z and statecharts [Har87] is used instead of the combination of Z and CSP described in Section 2.2.

Validation conditions avoid errors and make reviews cheaper. The validation conditions are a very important aspect of agendas. Not only do we want to develop software artifacts in a structured way, but also should the developed artifacts be of good quality.

In the case studies we have performed with agendas, the validation conditions have indeed revealed errors. For example, the specification of a microwave oven was developed with the agenda for the active sensors architecture shown in Table 3. It turned out that a validation condition of Step 5 was not satisfied, because the initial state of the oven was defined in such a way that safety could not be guaranteed.

Clearly, the errors revealed by failing to demonstrate validations conditions of an agenda can only be of an application-independent nature. Checking the validation conditions cannot guarantee e.g., that a system is adequately modeled by a developed specification, but many common errors are can be found and eliminated nevertheless. As reported by Heitmeyer et al. [HJL96], in the certification of the Darlington plant (which cost $ 40M), "the reviewers spent too much of their time and energy checking for simple, application-independent properties." To improve this situation, Heitmeyer et al. have implemented a tool that performs consistency checks. Since this tool is not tailored for any application domain, it can only check very general consistency conditions. In comparison, the validation conditions provided by agendas are much more to the point, such that more specific tool support for checking validation conditions generated by agendas is conceivable.

Agendas make software processes explicit, comprehensible, and assessable. Obviously, 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. Moreover, the purpose of all developed parts of the artifact becomes clear, because they are linked to steps of the agenda. Agendas can be subject to evaluation by review, i.e. by inspection by domain experts, and by test, i.e. by using the agenda to develop an artifact and then inspect the result of the development process. If the result is unsatisfactory, then the agenda should be revised.

For example, the agendas of Section 2.2 emerged from discussions with experts from software certification authorities and were refined by performing various case studies.

Agendas standardize processes and products of software development. Agendas structure development processes. 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.

For example, when developing a specification with the passive sensors agenda of Tables 1 and 2, all seven steps given there will be carried out in an order consistent with Figure 2, and the resulting specification will have the form shown in Figure 1.

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.

For example, when a new operational mode must be introduced in a specification developed with the passive sensors agenda of Tables 1 and 2, a replay of the agenda reveals that the type introduced in Step 2 must be changed, that the integrity constraints contained in the *State* schema defined in Step 3 must be considered once more, and so on.

Agendas are a promising starting point for sophisticated machine support. Agendas can be formalized and implemented as *strategies* [Hei97]. A meta-agenda (that proposes steps how to transform an agenda into a strategy) makes the formalization of an agenda a routine task. Strategies are complemented with a generic system architecture that describes how to implement support systems for strategy-based development activities. A prototype system for program synthesis exists that validates the generic system architecture [HSZ95].

If a formalization is not striven for, 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).

Agendas can be profitably employed for many different activities and using different languages. We have defined and used agendas for a variety of software engineering activities that we supported using different formal techniques. These activities include:

- **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 constraints on the traces of events that may occur. Such a requirements description can subsequently be transformed into a formal specification expressed in an existing specification language. 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 [SH96]. The agendas have been used to support specification acquisition in the languages Z and PLUSS (an algebraic specification language) [BGM89].

- **Specification of safety-critical software**
  The corresponding agendas have been presented in Section 2.2.

- **Software design using architectural styles**
  In [HL97], a characterization of three architectural styles using the formal description language LOTOS [BB87] is presented. For each of these styles, agendas are defined that support the design of software systems conforming to the style.

- **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].

For more details on the various agendas, the reader is referred to [Hei97].

4 Related Work

In Section 1, we have already mentioned design patterns [GHJV95] and architectural styles [SG96]. 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.

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, e.g., programs, requirements documents, or designs, each of which can contain schematic parts. The programming task is performed by "inspection"– i.e., by choosing an appropriate cliché and customizing it by combining it with other clichés, instantiating its schematic parts, and making structural changes to it. These activities are performed using high-level editing commands. In comparison to clichés, agendas are more process-oriented.

Wile [Wil83] presents the development language Paddle, which is similar in many ways to conventional programming languages. Paddle's control structures are called *goal structures*, and its programs provide a means of expressing developments, i.e., 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].

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 for design methods. IBIS representing heuristics tend to be specialized for particular application domains. The strategy framework, in contrast, aims at representing general, domain independent problem solving knowledge.

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.

5 Conclusions

In the preceding Sections, 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 the development and propose or enforce validations of the developed product. Following an agenda, software development tasks can be developed 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 under development to be used in a relatively large project (ESPRESS), where industrial-size case studies will be performed.

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 even more activities (e.g., testing) and specific contexts, e.g., object-oriented software development.

Acknowledgment. Thanks to Thomas Santen whose suggestions helped improve the presentation of this work.

References

[BB87] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14:25–59, 1987.

[BGM89] M. Bidoit, M.-C. Gaudel, and A. Mauboussin. How to make algebraic specifications more understandable: An experiment with the PLUSS specification language. *Science of Computer Programming*, 12:1–38, 1989.

[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.

[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.

[ESP] ESPRESS. Engineering of safety-critical embedded systems. Project description: http://www.first.gmd.de/~espress.

[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*, number 4 in Trends in Software, chapter 2, pages 25–52. Wiley, 1996.

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

[Har87] David Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[Hei95] Maritta Heisel. Six steps towards provably safe software. In G. Rabe, editor, *Proceedings of the 14th International Conference on Computer Safety, Reliablity and Security (SAFECOMP), Belgirate, Italy*, pages 191–205, London, 1995. Springer.

[Hei96] Maritta Heisel. An approach to develop provably safe software. *High Integrity Systems*, 1(6):501–512, 1996.

[Hei97] Maritta Heisel. *Improving Software Quality with Formal Methods: Methodology and Machine Support*. Habilitation Thesis, TU Berlin, 1997. submitted.

[HJL96] C. Heitmeyer, R. Jeffords, and B. Lebaw. Automated consistency checking of requirements specifications. *ACM Transactions on Software Engineering and Methodology*, 5(3):231–261, July 1996.

[HL97] Maritta Heisel and Nicole Lévy. Using LOTOS patterns to characterize architectural styles. In M. Bidoit and Max Dauchet, editors, *Proceedings TAPSOFT'97*, LNCS 1214, pages 818–832. Springer, 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*, pages 31–45. Springer, 1996.

[HS97] Maritta Heisel and Carsten Sühl. Methodological support for formally specifying safety-critical software. In *Proceedings 16th International Conference on Computer Safety, Reliability and Security*. Springer, 1997. to appear.

[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*, number 4 in Trends in Software, chapter 2, pages 1–24. Wiley, 1996.

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

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

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

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

[SH96] Jeanine Souquières and Maritta Heisel. Expression of style in formal specification. In W. B. Samson, editor, *Proceedings Software Quality Conference*, pages 56–65, ISBN 1 899796 02 9, 1996. University of Abertay Dundee.

[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.

[Süh96] Carsten Sühl. Eine Methode für die Entwicklung von Softwarekomponenten zur Steuerung und Kontrolle sicherheitsrelevanter Systeme. Master's thesis, Technical University of Berlin, 1996.

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