Methodology and Machine Support for the Application of Formal Techniques in Software Engineering

von Dr. rer.nat. Maritta Heisel

Habilitation am Fachbereich Informatik der TU Berlin

Lehrgebiet: Informatik

Eröffnung des Verfahrens:	5. Februar 1997
Verleihung der Lehrbefähigung:	18. Juni 1997
Aushändigung der Urkunde:	

1997
D83

Acknowledgments

I am grateful to Stefan Jähnichen for his constant encouragement and for urging me to direct my research toward practically relevant subjects.

Thomas Santen has accompanied my research for several years now and has shown an untiring willingness to discuss strategies and to read all the research documents I have written. He has read and commented on all the chapters of this work and has made useful suggestions for improving the presentation of my results.

Graeme Smith and Carsten Sühl's careful reading of Chapter 3 helped put the finishing touch to the agendas presented there.

Frances Paulisch, Patricia Johann, Ellen Siegel, Graeme Smith and Debora Weber-Wulff have helped polish up the English.

Finally, I would like to thank Nicole Lévy, Jeanine Souquières and Carsten Sühl for the work we have done together.

Contents

Chapter 1

Methodological Support for the Application of Formal Techniques

Methodological support for the application of formal techniques in software engineering is the motto for this entire work. We use the term "formal techniques" instead of the more common term "formal methods", because we find the term "formal method" to be a misnomer. A formal notation with a mathematically rigorous semantics is often called a formal method. In comparison with notational and semantic issues, methodological aspects are frequently neglected in the research on formal techniques. In our opinion, this fact is one of the greatest obstacles that hinders the transfer of formal techniques from academic environments into software engineering practice. The word "technique" does not suggest that there exists a method for guiding the application of the formalism in question.

The aim of this work is to demonstrate how formal techniques can be profitably employed in software engineering. We do not treat the whole software engineering process and consider all known formal techniques, but show areas where formal techniques can improve the quality of products or processes in software engineering. For this purpose, we describe some important and typical situations and show what can be gained by applying formal techniques in these situations. Examples are the development of safety-critical systems, where formal specification techniques contribute to the overall system safety, and software architectures, where a formal characterization makes it possible to reuse previously acquired design knowledge in a semantically sound way.

Using formal techniques, we can positively guarantee that the product of a development step of the software engineering process 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. However, formal techniques are no panacea. Even if a program is proven correct with respect to its specification, this does not mean that it will perform to the satisfaction of its users. The specification may not capture the requirements adequately, the performance of the program may be unsatisfactory, or the compiler or the operating system that are needed to execute the program may contain errors. Hence, informal methods as they are applied in traditional software engineering – and especially informal validation techniques such as testing – are still indispensable, and we propose to *complement* traditional techniques by formal ones, not to *replace* them.

1.1 Benefits of Formal Techniques

There are three areas where the benefits of formal techniques become particularly apparent: quality improvement, machine support, and reuse.

Quality improvement

Since the artifacts of the software engineering process that are expressed formally have a well-defined and unambiguous semantics, this semantics can be used to assess their quality. Not only syntactic, but also semantic properties of the artifact can be checked and eventually guaranteed. This offers a strong potential for quality improvement. If semantic properties of an artifact are specified and demonstrated, then there are fewer possibilities for misconceptions, contradictions, and omissions than is the case for classical methods of quality assurance, such as reviews and testing. It could be demonstrated that a system is specified in such a way that certain safety constraints cannot be violated, or a program can be proven correct with respect to a formal specification.

Machine support

Supporting parts of the software development process by machine enforces the representation of the documents to be produced or processed in a formal syntax. If this syntax has no rigorous semantics, only limited machine support can be provided. Taking semantics into account, machine support can cover a wider range of activities, namely the ones that are concerned with those properties of a product which cannot be expressed with reference to its syntax alone. A formal, machine-supported proof of properties of the developed product can be conducted, or test cases can automatically be generated and the results evaluated.

Reuse

A necessary condition for reusing previously acquired knowledge about software engineering is that the knowledge is represented in some way. Using formal techniques to represent development knowledge supports reuse: First, a formal document that represents some knowledge has an unambiguous semantics. Hence, it is clear *what* the document represents, and the intended way of reuse and the situations in which the document can be reused can be expressed in an unambiguous manner. Second, reuse is normally achieved by constructing libraries that contain reusable entities. Finding an appropriate library item is difficult if no meaning is associated with it. Again, it is the fact that formally defined documents have a semantics that makes reuse more goal-directed and thus more promising. Software design principles can be represented by (formally defined) architectural styles, or program libraries can be annotated with specifications, which describe the function of the program at a much higher level of abstraction than the program itself.

1.2 Making Formal Techniques Applicable for Non-Experts

A major drawback of formal techniques 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.

While nothing can be done about the first two points, it is definitely possible to provide guidance for the users of formal techniques. Indeed, this is the main goal of the work presented here. We consider appropriate guidance as a necessary condition for the practical applicability of formal techniques. This work presents two concepts that realize such guidance. The first concept, called an *agenda*, is informal and supports the application of formal techniques without the need for machine support. The second concept is called a *strategy*. It is a formalization of agendas, and its purpose is to make the knowledge represented in agendas amenable to machine support.

1.2.1 Agendas

An agenda is a list of activities to be performed when carrying out some task in the context of software engineering. Agendas contain informal descriptions of the activities and sometimes schematic expressions of a formal notation that can be instantiated in carrying out the activity. The activities listed in an agenda may depend on each other. Usually, they will have to be repeated to achieve the goal, like in the spiral model of software engineering.

As one of the major reasons for applying formal techniques is to guarantee semantic properties of an artifact, the activities of an agenda may have validation conditions associated with them. These validation conditions state *necessary* semantic conditions, which 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 are defined to make explicit tried and tested approaches to tackle some particular class of problems. 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.

Following an agenda gives no guarantee of success. Agendas cannot replace creativity, but they can tell the user 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.

1.2.2 Strategies

Strategies are a formally defined concept. They model software development tasks as problem solving processes. A strategy specifies how to reduce a given problem to a number of subproblems, how to assemble the solution of the original problem from the solution to the subproblems, and what semantic conditions a solution must fulfill to be an acceptable solution to the problem. The definition of strategies is generic with respect to the definitions of problems, solutions, and acceptability. Hence, strategies can serve to formalize a wide variety of software engineering activities.

When comparing an agenda and a strategy that are defined to support the same development task, it turns out that most of the steps of the agenda correspond to subproblems generated by the strategy.

Strategies can be combined to perform larger and more sophisticated development steps. They can be implemented and supplied with a generic architecture for systems supporting strategy-based problem solving. Hence strategies lead to machine supported development processes. In comparison to agendas, the formalization provided by strategies further enhances product quality (e.g. by formal proofs) and reusability.

Agendas and strategies are designed to better exploit the potential benefits formal techniques can achieve than would be possible without methodological support. They lead their users through different stages of the development, relieve them of tedious bookkeeping tasks, and propose or enforce validations of the developed product.

The development of agendas and strategies 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 first knowledge engineering phase, experts and users jointly define agendas for the development task. It is the responsibility of the users to make their knowledge explicit and to identify the steps that must be taken when performing the development task. Users and formal techniques experts can then work together and relate the results of the different steps identified to the formalism to be used.

After the knowledge engineering phase that leads to agendas is finished, it is the exclusive task of the experts to formalize the agendas as strategies. Although it cannot be expected of sufficiently trained users of formal techniques to be able to *define* strategies, they should be able to successfully *work with* agendas or strategies.

1.3 Overview

This work consists of two parts. The first part is centered around the concept of an agenda. We present agendas for different development tasks and different contexts. The second part is centered around the concept of a strategy. In the first part, we provide methodological support for different areas of software engineering by defining agendas. In the second part, we define strategies corresponding to the agendas of the first part. These strategies make the application of the methods represented by the agendas supportable by machine. The chapters and their interrelations are illustrated in Figure 1.1.

The top layer of the figure shows the methodological part, where we set up agendas for different specification and design approaches. In the second part, whose subject is to support the application of formal techniques by machine, these agendas are mapped to the strategy framework shown in the bottom layer of the figure. The strategy framework consists of the formal definition of strategies, complemented by a system architecture. Chapters 6–9 present different *instantiations* of this generic framework. For the instantiation for program synthesis (Chapter 6), an implemented support system called Integrated Open Synthesis System (IOSS) exists.

The methodological part focuses particularly on formal specification techniques for several reasons. First, the presence of a formal specification is a necessary prerequisite for supporting other development activities with formal techniques. It is not only the basis for an implementation but also helps in maintaining the developed software.

Secondly, there is a tendency to perform large parts of the software development within the specification language. Specifications are subject to refinements aimed at making the transition from a refined design specification to an executable program almost a routine task.

Finally, formal specification techniques are nearest to practical application in industry.

Formal specification techniques can be applied in many domains and in many different ways. All these application areas have their own methodologies. Specifically, we consider the areas of safety-critical software and software architecture.

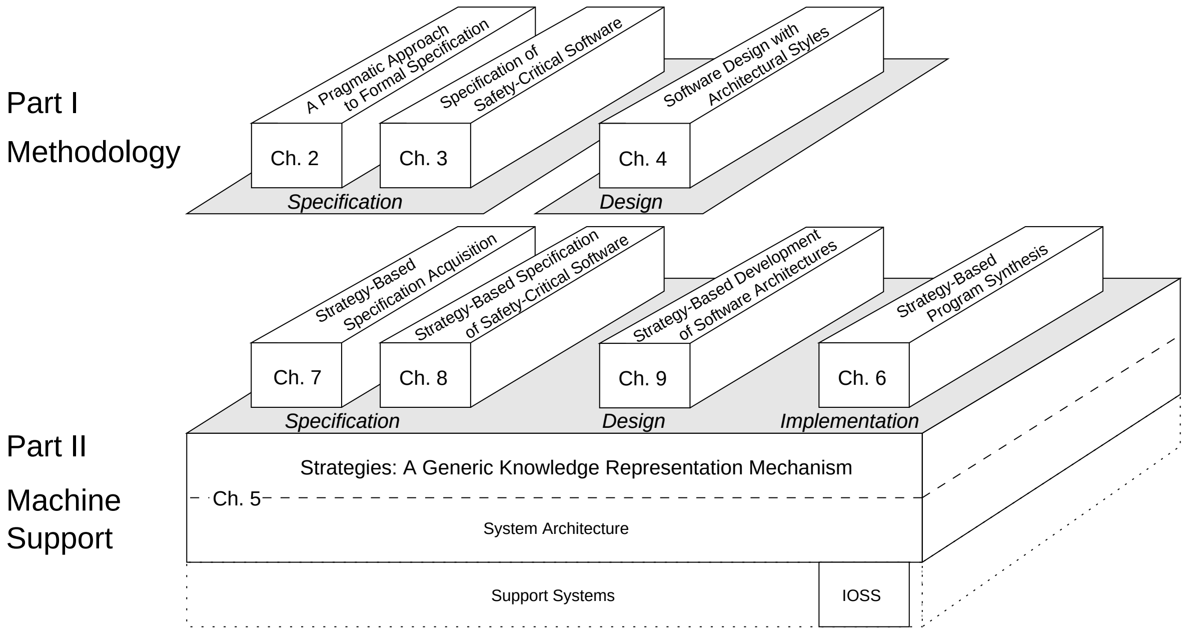

Part I
Methodology
A Pragmatic Approach to Formal Specification
Ch. 2
Specification of Safety-Critical Software
Ch. 3
Software Design with Architectural Styles
Ch. 4
Specification
Design
Strategy-Based Specification Acquisition
Ch. 7
Strategy-Based Specification of Safety-Critical Software
Ch. 8
Strategy-Based Development of Software Architectures
Ch. 9
Strategy-Based Program Synthesis
Ch. 6
Specification
Design
Implementation
Part II
Machine Support
Strategies: A Generic Knowledge Representation Mechanism
Ch. 5
System Architecture
Support Systems
IOSS

Figure 1.1: Overview of chapters and their interrelation

In most chapters, we use the specification language Z (Spivey, 1992b). A summary of the Z notation is given in Appendix A. While Z is useful in the contexts under consideration, it must be noted that our methodologies could also be defined and profitably applied in conjunction with other formal specification languages.

In the following, we outline the contents of each chapter.

Chapter 2

We describe the process of developing formal specifications in some generality and represent it as an agenda. This agenda shows how formal specification techniques can be integrated in traditional software development processes. It is relatively abstract because we cannot make specific assumptions about the kind of software system to be developed.

One step in the agenda is the transformation of (informal) requirements into a formal specification. To overcome some difficulties arising in the practical usage of formal specification techniques, we propose to perform this step in a pragmatic way. We argue that the transition from informal requirements to a formal specification should not be made too early, that it is not necessary to formally specify every detail, that different formalisms should be combined where appropriate, and that sometimes it may be useful not to adhere to limitations imposed by the formal specification language. This pragmatic approach also helps to deal with legacy systems.

The chapter shows that the application of formal techniques is not a question of “all or nothing”. Instead, there are several degrees of formality. Many of the benefits of formal techniques still occur when they are applied in a pragmatic way. This approach and the guidance provided by the agenda help to overcome some of the difficulties that arise when formal techniques are newly introduced into an organization.

Chapter 3

In this chapter, we no longer consider specification processes in general but concentrate on the specification of software for safety-critical applications. We consider the requirements a specification language must fulfill in order to be suitable for this purpose and show that a combination of the formal languages Z and real-time CSP fulfills these requirements. To obtain methodological support for the application of the combined language, we introduce a software model that is refined to different reference architectures representing frequently used designs of safety-critical systems. Specific agendas complement the reference architectures to guide specifiers in the development of software components for the architectures. These agendas are fairly detailed, and validation conditions are associated with many of the steps. The validation conditions are, of course, independent of a particular application. To further validate the developed specification, safety-related and liveness properties of the specification should be proven. A definition of refinement enables steps toward an implementation.

The chapter shows that the agendas guiding the development of an artifact in the software development process can be quite precise, if the context of development is limited to a certain application area. This is of great importance if non-experts are to apply formal techniques. We are convinced that following an agenda and showing all the associated validation conditions leads to a better quality of the developed specifications and the software implemented on their basis.

Chapter 4

Not only the specification, but also the design of software systems can be supported with formal techniques. *Software architectures* make design principles for software systems explicit. Classes of architectures that follow common principles are called *architectural styles*. In this chapter, we show how the formal description language LOTOS can be used to define software architectures and how patterns over LOTOS can serve to characterize architectural styles. The benefit of using LOTOS for architectural descriptions is not only the formality of the language, but also the availability of tools for analysis and animation.

We characterize styles by giving characteristics of the involved processes, a top-level communication pattern, and constraints that are sufficient conditions for a concrete architectural description to be an instance of a given style. The purpose of the style characterizations is not only to clarify the meaning of styles, but also to form the basis for defining agendas that support the development of concrete architectures. Three style characterizations and their corresponding agendas are presented and illustrated by an example.

With this chapter, we contribute to a systematic design of software systems and to a semantic foundation of architectural styles.

Chapter 4 concludes the first part of this work, which is concerned with the methodological aspects of formal techniques. In this part, we demonstrate that different development activities and different formalisms can be supported by agendas. The purpose of agendas is to contribute (i) to a better acceptance of formal techniques by software engineers by giving detailed guidance and (ii) to a better quality of the developed product by stating validation conditions. Thus, agendas help to apply formal techniques independently of machine support.

The second part is devoted to the development of concepts for the machine-supported application of formal techniques. Applying formal techniques with machine support further enhances their acceptance and the quality of the developed products, because, due to the amount of detail that must be handled, the application of formal techniques without machine support tends to be error-prone.

Chapter 5

This chapter presents a formal definition of strategies in Z. Strategies represent development knowledge used to perform different software engineering activities. The development of an artifact is modeled as a problem solving process. Hence, the definition of strategies is based on relations between *problems* and *solutions* to these problems. Since it is an important goal for us to guarantee semantic properties of the developed product, the solutions that are developed must always be *acceptable* for the corresponding problem. The definition of acceptability captures the semantic requirements on the solution.

Strategies can be combined to obtain more powerful strategies using *strategicals*, which are functions that take strategies as their arguments and yield strategies as their result. Moreover, strategies support stepwise automation of development tasks. We already noted that, because the concept of a strategy is generic, strategies can profitably be employed in several different phases of the software life cycle.

The definition of strategy is complemented by a generic system architecture that serves as a template for the implementation of support tools for strategy-based problem solving.

The strategy framework provides a uniform approach to the representation of software development knowledge and its machine supported application.

Chapter 6

This chapter presents an instance of the strategy framework that supports the synthesis of provably correct imperative programs. A methodology for program synthesis is not presented in the first part of this work. This topic is treated in detail in an earlier work (Heisel, 1992).

An implemented prototype system for strategy-based program synthesis exists, whose features and implementation are discussed. The instance of the strategy framework that supports program synthesis also serves to compare program synthesis with the activities that are formalized in the following chapters.

Chapter 7

Chapter 7 presents an instantiation of the strategy framework that supports the development of specifications in the language Z. In its generality, it matches Chapter 2, although the strategies presented focus only on one step of the agenda of Chapter 2. First, we introduce the notion of a specification style. These styles represent different approaches to the development of a formal specification. For each of the styles, we give a set of strategies associated with that style. An example of a specification acquisition shows how the development of a specification can be driven by styles. Finally, we sketch how different instances of the strategy framework can be combined.

Chapter 8

Chapter 8 presents a “meta-agenda” that shows how agendas can be formalized as strategies in a routine way. This meta-agenda is then used to define strategies for the agendas of Chapter 3.

Chapter 9

In Chapter 9, the agendas defined in Chapter 4 are transformed into strategies. Subsequently, the four instantiations of the strategy framework presented in Chapters 6–9 are compared. It turns out that strategies are powerful enough to represent several different development activities that use different formalisms. Chapter 10 summarizes and assesses what has been achieved.

In summary, the goal of this work is to show that the difficulties in introducing and applying formal techniques in software engineering are not insurmountable. By way of several important and practically relevant examples, we demonstrate that – for well-defined development activities – it is possible to supply comprehensive guidance to the users of formal techniques. It is no longer necessary for developers to be confronted with a mere formalism without guidelines on how to use it. Instead, they can use agendas that tell them what to do in which order and how to validate the developed product.

This work not only presents concrete methodologies that show precisely how to apply formal techniques in different contexts, but also gives special consideration to the definition of adequate concepts for the machine supported application of the developed methodologies. An implemented support system relieves the developers of tedious bookkeeping tasks and enforces the fulfillment of certain semantic conditions of the product, thus solving two important problems that normally arise in the use of formal techniques.

Part I

METHODOLOGY

Chapter 2

A Pragmatic Approach to Formal Specification

In this chapter, we argue that the application of formal specification techniques need not be a question of "all or nothing". The choice is not to either apply formal techniques completely rigorously and as intended by their designers, or not at all. The "all or nothing" standpoint would mean that the introduction of formal techniques necessitates a complete revision of the software development process. It is clear that organizations usually are neither willing nor capable to undergo such drastic changes.

Instead, we vote for a smooth introduction of formal techniques into software engineering. This can be achieved by taking a pragmatic attitude. To motivate what we mean by a pragmatic approach, we first discuss the benefits and drawbacks of formal specification. The pragmatic approach then consists of relaxing formal specification discipline in order to overcome the identified drawbacks. Our pragmatic approach to formal specification also helps to deal with legacy systems.

In theory, the advantages of formal specification techniques over conventional ones are well known:

- The problem is analyzed in more detail and thus better understood.
- The formal specification is an unambiguous and (hopefully) complete starting point for the implementation of a software system.
- The formal specification documents the behavior of the system.
- It can be used to select test cases and to determine if the results of the test cases coincide with the expected behavior.
- It makes maintenance and evolution of the system easier.
- Systems implemented from formal specifications often contain fewer defects.

In practice, however, formal specification techniques are not widely applied for the following reasons:

1. The formal nature of specification languages may make their use difficult, especially if the semantics is not easily comprehensible.

2. Formal specification languages can be as rich as programming languages and have a similarly steep learning curve.

3. There is no single specification language that is equally well suited for all kinds of systems and all aspects of an individual system, just as there is no one true programming language for all tasks.

4. It takes longer and is more expensive to develop a formal specification than to specify a system with conventional methods.

Are these really valid arguments against formal specification[1], and if so, what can be done to lessen their disadvantages? Point 4 cannot be regarded as a drawback because a greater effort in the earlier phases of software development may pay off in later phases and need not lead to an overall increase of costs (Houston and King, 1991).

The difficulties mentioned in points 1 and 2 cannot be completely overcome. But programming languages are formal languages too, and there is no argument against programming just because one has to learn one or more programming languages with non-trivial semantics. Specification languages with useful, semantically clean and intuitively clear concepts are needed. Such languages would make the introduction of formal specification techniques into software engineering practice much easier. Existing specification languages are not altogether bad, but all of them leave something to be desired, and unnecessarily so, as we have argued elsewhere (Heisel, 1995b).

Even carefully designed languages have their strengths and weaknesses. We cannot expect to find one single general-purpose specification language that suits all needs equally well. Hence, point 3 is a very serious one. One idea is to use several formalisms instead of one. When such a combination is done with care, a "hybrid" specification will be clearer, shorter and more comprehensible than a specification in only one language that is clumsy in parts because the language does not allow some relevant parts to be expressed elegantly and concisely. We would even go further and recommend not using formal specification techniques at all for those aspects of a system that just cannot be formally specified in a satisfactory way[2]. These aspects need not necessarily be non-functional, as shown in the example of Section 2.4.2.

With these ideas, we seek to bridge the gap between the theoretical benefits of and the practical problems with formal specification techniques. The choice is not to either apply them in a puristic way or not at all. Instead, there are varying degrees of formality.

However, to make formal specification techniques more widely applicable, it does not suffice to consider only those activities in software development that deal with formal objects. Before we can write down some formal text, we should have an idea of what we want to write down. This means that a detailed requirements elicitation is a very important prerequisite to making the application of formal specification techniques successful.

In Section 2.1, an overview is given of what we understand to be a pragmatic approach to formal specification when a new system is built. More often, however, legacy systems have to be used and maintained. Here, formal specification techniques can be of help, too, as is

[1]By formal we mean all those specifications where the language is given a formal semantics.

[2]What "satisfactory" exactly means depends not only on the problem and the language but also on the individual specifier.

No.	Phase	Validation
1	Define all relevant notions of the application domain.	
2	Define the requirements for the system to be built.	Every important aspect of the application domain and the system must be expressible. For each of the defined notions a statement for the system should be made.
3	Convert the requirements in a pragmatic way into a formal specification.	All relevant aspects of the system must be expressed appropriately. The specification must be more abstract than code.
4	Set up a mapping between the requirements and the formal specification.	Each requirement must show up in the specification. Each part of the specification must belong to a requirement.
5	Validate the specification.	Besides inspecting the specification, use as many of the following mechanisms as appropriate: checklists, animation, proof of properties, testing.

Table 2.1: Agenda for specification acquisition

explained in Section 2.2. In Sections 2.3 through 2.6, the various activities that make up the pragmatic approach to formal specification are presented in more detail and illustrated by examples. The main focus is on demonstrating how formal specification discipline can be relaxed in order to overcome the intrinsic problems of formal specification techniques as mentioned above. Finally, we discuss what is gained by using this pragmatic approach, and point out directions for further research. This chapter is an adaptation of the paper (Heisel, 1996b).

2.1 Specifying New Systems

In the following, we give an agenda for our approach, i.e. a list of the activities that have to be performed. Some of the activities are complemented by means for validating their results. The single steps should not be considered as isolated phases with no feedback between each other. They should be partially carried out in parallel and are likely to be repeated, as in the spiral model of software engineering (Boehm, 1988). A "later" activity can reveal errors or omissions in an "earlier" phase. A overview of the agenda is given in Table 2.1. The dependencies between the phases are shown in Figure 2.1. Phase i depends on phase j if the result of phase j is needed to perform phase i. We now explain the phases one by one.

Phase 1 *Define all relevant notions of the application domain.*

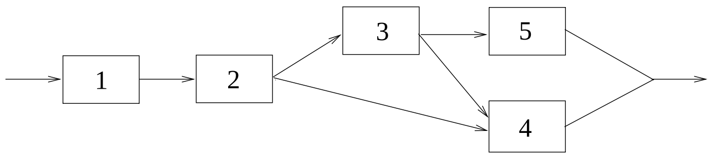


Figure 2.1: Dependencies of phases

It must be possible to talk about all relevant aspects of the system. For this purpose, all phenomena that might be of interest must be given names and be informally described as precisely as possible. Jackson and Zave (Jackson and Zave, 1995) call this a *designation set.* In the context of software architectures, one may define *criteria* that are relevant for these systems, see Section 2.3 and (Heisel and Krishnamurthy, 1995a).

Phase 2 *Define the requirements for the system to be built.*

The notions defined in Phase 1 provide a *language* in which the requirements can be expressed. If some requirement or some phenomenon concerning the system cannot be expressed, then the application domain was not investigated carefully enough, and we have to go back to Phase 1. Conversely, in order not to forget some requirements, we should make sure that for each of the relevant notions a statement for the new system is made (this may be the statement that some phenomenon will not be treated at all).

Phase 3 *Convert the requirements in a pragmatic way into a formal specification.*

By "pragmatic" we mean that we should not always adhere to the ideal of purely formal specification but take the freedom to make a specifier's life easier. In our past work on and with formal specification, we applied the following relaxations of formal specification discipline[3].

1. Combine different formal or semi-formal specification techniques if one formalism alone is not powerful enough to express all relevant parts of the specification elegantly.

2. If the specification would be as low-level as program code, refrain from specifying these details formally but use conventional specification techniques and document the program code in a particularly detailed way.

3. Ignore restrictions of the specification language if it is clear how to give the requirement a semantics.

Of course, all the above relaxations must be applied with great care because they are potentially dangerous: a specification where several formalisms are combined may be inconsistent. Incomplete formal specifications may result in an insufficient understanding of the parts not formally specified and thus lead to a wrong implementation. Indeed, this relaxation should mostly be applied for legacy systems, see Section 2.2. Finally, when we write down illegal expressions of a specification language, we must make sure that these expressions have a well-defined semantics, which should be explained carefully in the accompanying text of the formal specification. Section 2.4 presents some examples and sketches how to apply the relaxations "safely".

[3]Other persons may come up with different relaxations, according to their experience.

Phase 4 *Set up a mapping between the requirements and the formal specification.*

Such a mapping shows where and how each requirement is reflected in the formal specification. It helps to make the specification complete. When the system must be adjusted to new requirements, the specification should be changed before the code. The mapping shows where the changes have to be made. An example is given in Section 2.5.

Phase 5 *Validate the specification.*

This step is very important because, usually, customers only have a vague idea of what the system should do. Formal specifications can be validated more rigorously than informal ones because they can be checked for inconsistencies or incompleteness with formal proof techniques. Possible techniques to apply are animation, proof of properties, and testing.

Again, the different phases are not independent of each other. Especially Phase 5 will have an effect on Phases 1 and 2. After several rounds in a spiral consisting of the above phases, the specification should stabilize, with a high probability that the requirements are complete, the specification captures them adequately, and that customers and developers understand equally well what the system is supposed to do.

2.2 Dealing with Legacy Systems

Building a new system entirely from scratch is not always possible or desirable. Often, legacy systems have to be used and maintained. But also for existing code, it is very useful to have a formal specification. It documents the behavior of the system and helps in maintenance and evolution.

If the legacy system exhibits some unexpected behavior, it is more feasible to seek the explanation for the behavior in a formal specification than in the code because the specification is more abstract and usually much shorter. Using reverse engineering techniques, the formal specification can help in locating code that deals with a certain aspect of the given system. Thus, when a formal specification is available, it is no longer necessary to search through the entire code to make changes in a legacy system, see (Heisel and Krishnamurthy, 1995a) and Section 2.5.

To deal with legacy systems, the approach described in the previous section has to be adjusted. Phase 1 does not change. In Phase 2, not the requirements are formulated but the behavior of the system as far as it is known. Phases 3 and 4 are as before. In Phase 5, the formal specification is used to generate test cases. Hörcher and Peleska describe how this can be done systematically (Hörcher and Peleska, 1995). These test cases should be run in order to check if the results of Phases 2 and 3 coincide with the actual behavior of the system.

Phase 1 *Define all relevant notions of the application domain.*

Phase 2 *Describe the behavior of the legacy system as far as it is known.*

Phase 3 *Convert this description in a pragmatic way into a formal specification.*

Phase 4 *Set up a mapping between the description and the formal specification.*

Phase 5 *Use the formal specification to generate test cases and validate the assumptions.*

As when building new systems, it can be expected that the last phase reveals errors in earlier ones and that several iterations are necessary.

We see that specifying new systems and dealing with legacy systems are quite similar activities. The following sections that explain the various phases in more detail can thus serve to illustrate both of them. Of course, the specification of the legacy system exclusively documents its behavior. To understand the implementation with all its possible optimizations and "tricks", more effort is necessary.

2.3 Phases 1 and 2: Definitions and Requirements/Behavior

These phases consist mainly of informal activities. The resulting documents may be informal or semi-formal. They serve to form a basis for the development of a formal specification. Their purpose is to gain a thorough understanding of the problem domain and relate this to the requirements for or the description of a concrete system. Without such an understanding, it is hopeless to work on setting up formal specifications. But of course these phases are also important when traditional methods are applied.

We illustrate the informal definition of the notions relevant for a software system by the description of event-action systems (Krishnamurthy and Rosenblum, 1995). Such systems wait for events to occur (e.g. it is Friday 5 p.m. and the boss is logged in) and then take an action (e.g. send an email to all group members that there will be a group meeting at 5.30 p.m.). Such a pair, consisting of an event pattern and a corresponding action, is called *specification* and is not to be confused with formal specifications. The areas of application of event-actions systems include calendar and notification systems, computer network management, and software process automation. Some of the criteria that characterize event-action systems (Heisel and Krishnamurthy, 1995a) are[4]:

- Matching style
  Matching event patterns against occurring events is the heart of each event-action system. Event patterns can be complex expressions of so-called primitive events. When an event pattern is only partially matched, it is possible to either consider the whole event pattern as unmatched (transient matching) or mark the matched events and only wait for those events that were not yet matched (perpetual matching).

- Context
  This means that not only the event itself but also the context in which it occurs is taken into account, e.g. the event must be caused by a certain user or a certain machine.

- Grouping specifications
  Is it possible to group logically related specifications together and refer to them as a whole?

- Handling unmatchable specifications
  Can the system detect and eliminate specifications that can never be matched?

These criteria provide a language in which the requirements for newly designed event-action systems can be expressed (e.g. the system should consider context and allow for grouping

[4]We will give a formal characterization of the event-action architectural style in Chapter 4. The criteria we mention here will be part of the specification of the event manager component and the type that defines events.

specifications, but unmatchable specifications are not considered). They also contribute to make the requirements complete: for each criterion, a decision should be made. If the set of criteria is complete, then so are the requirements. We will come back to this example and show some of the formalizations of the criteria in Sections 2.4.2 and 2.5.

2.4 Phase 3: Pragmatic Approach

The purpose of relaxing formal specification discipline is to show that formal techniques need not be straightjackets that leave little freedom to their users. The relaxations may convince potential users to start experimenting with them.

2.4.1 Combining Different Specification Techniques

The combination of different specification techniques is certainly the most important means to make a specifier's life easier. It directly addresses the drawback mentioned in the beginning of this chapter that there is no single ideal specification language. We illustrate it by an example, where algebraic and model-based specifications are combined. In Chapter 3, this technique is also applied. There, Z and real-time CSP are used to specify different aspects of safety-critical systems.

The European Commission has defined the so-called *Information Technology Security Evaluation Criteria*, ITSEC (ITSEC, 1991). These criteria define security levels against which information technology systems can be evaluated. As an example, we consider the formalization of the ITSEC functionality class F-C1. Its description in natural language is as follows:

> *"The TOE shall be able to distinguish and administer access rights between each user and the objects which are subject to the administration of rights, on the basis of an individual user, or on the basis of membership of a group of users, or both. It shall be possible to completely deny users or user groups access to an object."*

Here 'TOE' means *Target of Evaluation*, i.e. the product to be evaluated. For the formal specification of security functionality classes, the clearest and most abstract specification is algebraic. For the systems to be certified, however, Z is more appropriate. It must be shown that the Z specification of a particular system is a correct "refinement" of the algebraic one. This can be established by working from both ends: doing algebraic refinements of the abstract specification, and performing "abstractions" on the Z specification until the refined algebraic specification and the abstracted Z specification can be related by a one-to-one mapping of the involved constructs.

To formally specify F-C1, we use the algebraic specification language PLUSS[5] (Bidoit et al., 1989). This language was designed to make formal specifications resemble natural-language text. Accordingly, names of functions or predicates may consist of several words. The argument positions are indicated by "_". Quantifiers and connectives have a nonstandard, yet obvious, syntax. The keyword **proc** indicates that the specification is generic, and the symbol "$*$" denotes the Cartesian product. Each axiom is given a name for later reference.

[5]Any other algebraic specification language could be used as well, as long as it allows for genericity, first-order formulas as axioms, and has a loose semantics.

The semantics of a PLUSS specification is loose, i.e. it consists of all Σ-algebras that satisfy the axioms, where Σ is the signature of the specification.

Taking the above description of the functionality class F-C1 as a starting point, we come up with the following PLUSS specification:

```
proc $F\text{-}C1(object, user, group)$
predicate
    $user\_allowed\ access\ to\_ : user * object$
    $user\_denied\ access\ to\_ : user * object$
    $group\_allowed\ access\ to\_ : group * object$
    $group\_denied\ access\ to\_ : group * object$
    $\_associated\ with\_ : user * group$
    $grant\_access\ to\_ : user * object$
axioms
$uc$ : $(user\ u\ allowed\ access\ to\ o\ \&\ user\ u\ denied\ access\ to\ o)$ is false
$gc$ : $(group\ g\ allowed\ access\ to\ o\ \&\ group\ g\ denied\ access\ to\ o)$ is false
$ga$ : $grant\ u\ access\ to\ o \Rightarrow$
    $((user\ u\ allowed\ access\ to\ o$ or
        $($exists $g_1 : group.(u\ associated\ with\ g_1\ \&\ group\ g_1\ allowed\ access\ to\ o)))$
    $\&\ user\ u\ denied\ access\ to\ o$ is false
    $\&\ ($exists $g_2 : group.(u\ associated\ with\ g_2$
        $\&\ group\ g_2\ denied\ access\ to\ o))$ is false $)$
where $u : user, g : group, o : object$
end $F\text{-}C1$
```

This specification clarifies the ambiguities contained in the natural language description, e.g. what happens if a user is allowed access individually who at the same time is member of a group that is denied access. It defines a global predicate *grant_access to_* deciding if a user is granted access to an object or not. Access may only be granted if there is positive but no negative information concerning the user. The user must either be granted access directly (predicate *user_allowed access to_*) or via a group (predicate *group_allowed access to_*). *Additionally*, the user may neither be denied access directly (formalized by the predicate *user_denied access to_*), nor be member of a group that is denied access (predicate *group_-denied access to_*). This formalization represents a "conservative" approach: in case of any doubt, access is denied. If the conservative policy is considered too restrictive, it is always possible to define the predicate *group_denied access to_* to be *false* everywhere.

Let us suppose that the access control mechanism of Unix is to be evaluated against the above requirement. In Unix, each file belongs to a user and is associated with exactly one group. The *permission mode* of a file consists of three parts: the access rights of the user, the group, and the others. The tokens `r,w,x` stand for the rights to read, write, or execute the file, respectively. The token "`-`" means that the corresponding access right is denied.

This means that the access information is stored locally with each Unix object, not globally as in the PLUSS specification. The most natural way to specify the Unix access control mechanism is not algebraically but model-based since the access information is stored in a global system state. A formal specification of the Unix access control mechanism in Z can be found in (Peleska, 1995).

Therefore, the task is to show that a model-based specification of a concrete access control mechanism captures security requirements that are specified algebraically, i.e. that it is a correct refinement. For algebraic as well as model-based specifications, there are notions of refinement. However, it is not clear what it means that a model-based specification is a refinement of an algebraic one. To make the transition between the algebraic and the model-based world as smooth as possible, we introduce an intermediate specification. It is stated in Z and is an abstraction of the Unix access control mechanism. The parameters of the PLUSS specification are introduced as basic types.

$$[USER, GROUP, OBJECT]$$

The access information is stored together with the respective object, like in Unix. We abstract, however, from the different possible access rights and treat users and groups symmetrically, as in the algebraic specification. The information about which groups a user is associated with is also localized.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{ZObject}} \\
o : OBJECT \\
pos\_user, neg\_user : \mathbb{F}\ USER \\
pos\_group, neg\_group : \mathbb{F}\ GROUP \\
\hline
pos\_user \cap neg\_user = \varnothing \\
pos\_group \cap neg\_group = \varnothing \\
\hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{User}} \\
userid : USER \\
assoc\_with : \mathbb{F}\ GROUP \\
\hline
\end{array}$$

The following schema specifies when a user is granted access to an object.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{GrantAccess}} \\
u? : User \\
zo? : ZObject \\
\hline
(u?.userid \in zo?.pos\_user \lor \\
\quad (\exists\, g_1 : GROUP \bullet (g_1 \in (u?.assoc\_with \cap zo?.pos\_group)))) \\
u?.userid \notin zo?.neg\_user \\
\neg\ (\exists\, g_2 : GROUP \bullet (g_2 \in (u?.assoc\_with \cap zo?.neg\_group))) \\
\hline
\end{array}$$

It must now be shown that if the Z schema grants a user access to an object then this must comply with the abstract predicate *grant_access to_*. The algebraic notion of refinement by model inclusion (Wirsing, 1990) requires us to do the following:

1. Provide a signature morphism from the abstract entities to the concrete ones.
2. Apply the signature morphism to the axioms of the abstract specification.
3. Show that the formulas obtained in this way are theorems of the concrete theory.

In this special case, the refinement is relatively simple. The algebraic specification consists solely of predicates. All of these except *grant_access to_* are "auxiliary predicates" that do not belong to the "user interface" of the specification. These auxiliary predicates are represented by sets in the abstract Z specification. This is justified because a predicate and its extension can be regarded equivalent. In the general case, however, more research is needed to define a mapping between the algebraic semantics based on universal algebras to the Z semantics based on sets. The signature morphism is as follows:

$$\begin{array}{rcl}
group & \mapsto & GROUP \\
user & \mapsto & User \\
object & \mapsto & ZObject \\
user\_allowed\ access\ to\_ & \mapsto & \lambda\, u : User; zo : ZObject \bullet u.userid \in zo.pos\_user \\
user\_denied\ access\ to\_ & \mapsto & \lambda\, u : User; zo : ZObject \bullet u.userid \in zo.neg\_user \\
group\_allowed\ access\ to\_ & \mapsto & \lambda\, g : GROUP; zo : ZObject \bullet g \in zo.pos\_group \\
group\_denied\ access\ to\_ & \mapsto & \lambda\, g : GROUP; zo : ZObject \bullet g \in zo.neg\_group \\
\_associated\ with\_ & \mapsto & \lambda\, u : User; g : GROUP \bullet g \in u.assoc\_with \\
grant\_access\ to\_ & \mapsto & GrantAccess
\end{array}$$

In this signature morphism, we have only changed the name of the predicate that makes up the user interface of the specification. The internal predicates have been replaced by their extensions. In this way, we have made sure that *GrantAccess* can be used instead of *grant_access to_* without any restrictions, provided *GrantAccess* fulfills the axioms stated for *grant_access to_*. This can easily be shown to be the case.

In this example, the desire to combine different formalisms arose because the formal specifications had different levels of abstraction. For abstract properties, algebraic techniques are the most natural, and for the specification of concrete realizations, model-based techniques are appropriate.

Many more useful combinations of different specification techniques are conceivable. Weber (Weber, 1996) combines Statecharts (Harel, 1987) and Z for the design of safety-critical systems. Zave's and Jackson's (Zave and Jackson, 1993) multiparadigm specifications are combinations of partial specifications expressed in different languages. In Chapter 3, we present a combination of Z and real-time CSP in detail.

2.4.2 Leaving out Details

Formal specifications need not be useful in every situation; sometimes it is more important to keep the specification concise and easily comprehensible. In (Heisel and Krishnamurthy, 1995b), the specification of an existing event-action system called YEAST (Yet another Event Action Specification Tool) is given. As already mentioned in Section 2.3, matching of events against event-action-specifications is very important for such systems. Starting out from so-called primitive events, the event language of YEAST allows one to build complex event expressions, using the connectors *then* (sequencing), *and* (conjunction), and *or* (disjunction).

Matching of composite events can be defined in terms of matching of primitive events relatively easily. A specification of matching for primitive events, however, would be no more abstract than the code itself and would make the formal specification much longer and less comprehensible. Therefore, matching of primitive events is not included in the formal specification but only described in the system documentation. For the formal specification of matching, this means that we *declare* a matching predicate on primitive events, but do not *define* it.

$$\mid \ \_matches\_ : (EVENT \times TIME) \leftrightarrow (EVENT \times TIME)$$

The further specification of the matching process can now make use of this predicate, even though its meaning is not contained in the formal specification.

This relaxation to formal specification discipline can be applied in those cases where a formal specification would not be an abstract description of system behavior but only a different notation for a program.

2.4.3 Ignoring Restrictions

We expect that almost everybody who has some experience in formal specification techniques has encountered situations of surprise and annoyance, where it seemed obvious how to write down some formal expression, only that the designers of the formal notation had not foreseen the respective situation and hence excluded this possibility. Sometimes, it is advisable to disregard such language restrictions. We illustrate such a situation by a specification of the Unix file system, which will be presented in more detail in Section 7.4.

The Unix file system presents itself to the user as a tree where each node has a name and an arbitrary number of successors. A specification of such trees should be present in some library for re-use, where the content of the nodes (as opposed to their names) should be a generic parameter. It appears straightforward to define such trees as a free type in Z:

$$[NAME]$$

$$\begin{aligned} NAMED\_TREE[X] ::= {} & lf \langle\langle NAME \times X \rangle\rangle \\ \mid {} & node \langle\langle NAME \times \operatorname{seq} NAMED\_TREE[X] \rangle\rangle \end{aligned}$$

Although it is semantically sound as long as X is not instantiated with a type or set depending on $NAMED\_TREE$, this specification is invalid because free types in connection with genericity are not allowed in Z. For the subsequent specification, it has to be decided if this (unnecessary) restriction of the Z language should be ignored and the rest of the specification should use the generic free type definition. In this case, we decided against it because Z type checkers reject the above type definition; hence, tool support would be lost for the entire specification.

Instead, we chose an alternative definition based on sets. Named trees are finite partial functions from sequences of positive natural numbers into the Cartesian product $NAME \times X$.

$$\begin{aligned} & NAMED\_TREE[X] == \\ & \quad \{ f : \operatorname{seq} \mathbb{N}_1 \mathrel{+\!\!\!\!\!\twoheadrightarrow} NAME \times X \mid \\ & \qquad \langle\rangle \in \operatorname{dom} f \\ & \qquad \wedge (\forall path : \operatorname{seq}_1 \mathbb{N}_1 \mid path \in \operatorname{dom} f \bullet \\ & \qquad\quad front\ path \in \operatorname{dom} f \\ & \qquad\quad \wedge (last\ path \neq 1 \Rightarrow front\ path \frown \langle last\ path - 1 \rangle \in \operatorname{dom} f)) \} \end{aligned}$$

This definition models trees as functions mapping "addresses" to the content of the node under the respective address. Each node consists of a name and an item of the parameter type X. The empty sequence is the address of the root. The length of an address sequence coincides with the depth of the node in the tree. Hence, an address can only be valid if its *front* is also a valid address. The number i denotes the i-th subtree. If there is an i-th subtree for $i > 1$ then there must also exist an $i - 1$-th subtree.

In comparison to free types, this definition looks quite complicated. Although the operations on named trees can be defined elegantly, this shows how much more incomprehensible specifications can become when the specification language does not support the features best suited for the situation at hand.

To specify a function selecting a successor of a node with a given name, we gave the following specification:

$$
\begin{array}{|l}
[X] \\
\hline
\mathit{child\_named} : \mathit{NAMED\_TREE}[X] \times \mathit{NAME} \rightarrow\!\!\!\!\!\!+\; \mathit{NAMED\_TREE}[X] \\
\hline
\forall\, n : \mathit{NAME};\ t : \mathit{NAMED\_TREE}[X] \bullet \\
\qquad (n \in \mathit{names}(\mathit{subtrees}\ t) \Rightarrow (t, n) \in \mathrm{dom}(\mathit{child\_named})) \\
\qquad \land\ \mathit{child\_named}(t, n) \in \mathit{children}\ t \\
\qquad \land\ \mathit{name\_of\_tree}(\mathit{child\_named}(t, n)) = n \\
\hline
\end{array}
$$

Again, there are problems with genericity. The above specification of the function *child_named* is semantically invalid in Z because in the reference manual it is required that "the predicates must define the values of the constants uniquely for each value of the formal parameters.", (Spivey, 1992b, p. 80). This is not the case here, because if there is more than one child with the given name, *child_named* selects an *arbitrary* one. However, we do not see any difficulties with a definition like this. On the contrary, it has the advantage to give an implementor the greatest possible freedom: if it is more efficient to search the list of subtrees from the back to the front instead of vice versa, it should be possible to do so.

"Legal" possibilities would be to either define *child_named* as a relation instead of a function or give an unambiguous definition. Both of these do not cover our intention, namely to state that of the several functions satisfying the specification we do not care which one is implemented. Since this specification clearly has a a well-defined semantics and no type checker can find the "violation", it is possible to stick to this "illegal" specification without loosing tool support.

2.5 Phase 4: Mapping between Requirements/Description and Specification

The effort to record where in the specification the various requirements or features of a system are reflected is worthwhile, because such a mapping serves several purposes. First, it helps us understand the formal specification, starting from an intuitive understanding of the requirements or system features. Second, it may help to detect misconceptions in case the intuitive understanding of the system contradicts the corresponding parts in the formal specification. Third, changes in the system usually first manifest themselves in the requirements. The mapping helps us find out what effects the change in the requirements has on the formal specification and indirectly on the code. If a change is very hard to accomplish in the formal specification, it can be expected that changing the code would be difficult, too. Fourth, in case the correspondence between specification and code is not available, it helps to detect those parts of the code implementing a certain feature of the system.

To illustrate this mapping, we come back to the example of Sections 2.3 and 2.4.2, the YEAST case study. We demonstrate how the criterion of grouping specifications is reflected in the formal specification. The schema defining the global system state is

$$
\begin{array}{|l}
\textit{SpecState} \\
\hline
\textit{specs} : \mathbb{F}\ \textit{Spec} \\
\textit{specMap} : \textit{LABEL} \rightarrowtail\!\!\!\!\!\!\!\!\;+\; \textit{Spec} \\
\textit{groups} : \textit{GNAME} \leftrightarrow \textit{LABEL} \\
\hline
\textit{specMap} = \{ s : \textit{specs} \bullet s.\textit{label} \mapsto s \} \\
\operatorname{ran} \textit{groups} \subseteq \operatorname{dom} \textit{specMap} \\
\hline
\end{array}
$$

Specifications can be referred to using labels. This is reflected by the injective partial function *specMap*. The *group* component of the schema records which specification belongs to which groups via labels.

Other components of the formal specification onto which the criterion of grouping specifications is mapped include all operations having arguments or results of type *GNAME* or changing the *groups* component. Once we know that *GNAME* and *group* represent the grouping mechanism, these other components can be found automatically because they are characterized purely syntactically. In case the grouping mechanism is to be changed, the mapping gives all parts of the formal specification that have to be considered.

YEAST was formally specified only after it has been implemented. Therefore, this case study serves well to illustrate the treatment of legacy systems. For these systems, the mapping between criteria (i.e. the informal description of the system) and the specification is especially useful because it helps to locate the criterion in the code[6].

In order to locate those parts of the YEAST code that implement the grouping mechanism, we used the mapping from the criterion to the formal specification (which points out all relevant user operations) to generate test cases that *should* trigger the code associated with specification grouping. All of these were executed with various inputs. Different reverse engineering tools made it possible to pinpoint the set of functions implementing specification groups. Only 22 functions out of 262 were executed more than once. Some of these were library and other utility functions. This left only a handful of functions that had to be considered. For more details, see (Heisel and Krishnamurthy, 1995a).

2.6 Phase 5: Validation of the Specification

For legacy systems, the validation is easier than for systems that are not yet implemented because the implementation is available. In principle, even a formal verification would be possible that establishes the consistency of the specification and the code.

For new systems, specifications can only be validated informally because no formal relation can be established between the necessarily informal requirements and a formal specification.

[6]For newly designed systems, the mapping between the formal specification and the code could be recorded from the beginning.

Validating specifications is very important because, as Brooks (Brooks, 1987) stated: “For the truth is, the client does not know what he wants”. As possible validation techniques we mentioned animation, proof of properties, and testing.

For animation, an executable prototype is built that makes it possible to “try things out”. This possibility is very helpful in the elicitation of the “real” requirements. Elsewhere (Heisel, 1995b), we have argued that formal specifications should be as abstract as possible and that they should not introduce any implementation bias. Such specifications are usually non-constructive and hence not executable. However, the possibility of animating the specification is so valuable that we consider it worthwhile to perform a few refinement steps in order to make the specification executable.

Proving properties of the specification (e.g. that two operations are inverses of each other) or showing that in certain situations something undesirable cannot happen also enhance confidence in the specification and contribute to its understanding. The problem with this technique may be to find the relevant properties to be proven. In a more concrete context, like the specification of safety-critical systems (see Chapter 3), or software design using architectural styles (see Chapter 4), detailed validation guidelines can be given.

Finally, specifications can be tested almost like code. Test cases (that must be selected anyway) can be used to check if the specification captures the expected behavior of the system for these cases.

2.7 Summary

Our work aims at supporting the introduction and application of formal techniques in system specification and development. We take a pragmatic viewpoint of formal techniques. They should not be applied under all circumstances but only in situations where it is clear what is gained by their application. To achieve this aim, we must (i) show how formal specification techniques can be integrated in the traditional software engineering process, and (ii) show how the undeniable drawbacks of formal techniques can be dealt with.

Integration in traditional software processes

The application of formal techniques can only be successful when it is well prepared. This means that those phases of the software process where formal techniques cannot be applied are not to be neglected. On the contrary, requirements analysis must be performed at least as thoroughly as in a traditional process. Our approach gives a guideline how to proceed: first create a suitable language, and then use it to express all relevant requirements or facts about the system. It clearly does not suffice to hand a formal language description to specifiers and then expect them to be able to write formal specifications. Not only the formal techniques themselves have to be learned, but also how formal techniques can smoothly be integrated in the process model to be followed.

Relaxations of formal specification discipline

Our pragmatic approach to using formal specification techniques directly addresses the problems described earlier.

- If a formal specification technique not suitable to specify certain aspects of a system, and is suitable for other aspects, then different formalisms should be combined.

- If formally defining every detail would enlarge the specification disproportionally and make it harder to comprehend, then some aspects of the system should not be formally specified at all.

- If it is clear how some aspect of the system could be specified but the chosen formalism is too weak, then appropriately commented "illegal" specifications should be considered.

The proposed relaxations are most valuable in situations where formal techniques would otherwise be rejected because a complete specification using a single formalism would be too long, too complicated, or too expensive to develop. Of course, all these relaxations have to be used with care. It must always be demonstrated that the resulting specifications are semantically sound.

One might object that such a "pragmatic" specification does not enjoy the advantages of formal specifications as enumerated at the beginning of this chapter any longer. Even if this were true, a "pragmatic" specification would still lead to a better analysis of the system, contain fewer ambiguities and better document the system behavior than an informal specification does. Even in connection with formal techniques, there are better choices than "all or nothing".

However, it need not always be true that "pure" specifications are superior to "pragmatic" ones. Consider the case of the embedded safety-critical systems, which we will discuss in detail in Chapter 3. Rejecting the combination of Z with another formalism means that important aspects of safety-critical systems cannot be formally specified at all or only in an unsatisfactory way.

In summary, our approach has the following benefits:

- An agenda makes explicit the tasks to be performed and their interdependencies.

- The relaxations of formal specification discipline recommended for transforming requirements into formal specifications (if necessary) contribute to making the specification less complicated and better comprehensible.

- Validating the specification helps to make the specification complete and conforming more closely to the wishes of the customers.

- When requirements change, the mapping between requirements and formal specification shows where changes have to be made in the formal specification. The corresponding code can be found with reverse engineering techniques.

2.8 Further Research

The pragmatic approach can be further elaborated in the following ways:

Method for requirements engineering. The elicitation of the requirements was only described very briefly. We intend to develop agendas for requirements engineering. They should be defined such that the application of formal specification techniques is well prepared.

Combination of algebraic and model-based specifications. The example of a combination of the algebraic language PLUSS and the model-based language Z we presented in Section 2.4.1 was a simple case, where we considered only predicates, which can be identified with their set-theoretic extensions. More research is needed to develop a general notion of refinement between algebraic and model-based languages.

Integration of semi-formal methods. In this and the following chapter, we consider the combination of different formal specification languages. It is also promising to consider the combination of semi-formal, e.g. graphical notations with formal ones. This would make it possible to gradually enrich traditional techniques with formal elements. The tasks here are to develop such combinations and to develop agendas for existing combinations, e.g. (Weber, 1996).

Other relaxations. More experience should be gained concerning useful relaxations of formal specification discipline.

Reverse engineering. To successfully adapt and maintain legacy systems, reverse engineering techniques must be applied. Methods and agendas for this purpose remain to be developed.

Chapter 3

Specification of Safety-Critical Software with Z and Real-Time CSP

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. These are systems whose malfunctioning can lead to accidents resulting in loss of property or danger for human lives. 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.

Most safety-critical systems are *reactive*. This means they do not just perform data transformations, like payroll systems. Instead, they are not intended to terminate, and their behavior depends on stimuli coming from the environment and their internal state which usually is an approximation of the state of the environment. Frequently, they have to fulfill real-time requirements.

From these characteristics, it follows that two aspects are important for the specification of software for safety-critical systems. First, it must be possible to specify behavior, i.e. what happens in the system in which order, how the system reacts to incoming events, and what signals it sends to the environment under which conditions. The specified behavior must additionally take place sufficiently fast. This is a crucial requirement for the system and thus should be expressed in the specification. Second, complementing the behavioral specification, the structure of the system's data state and the operations that change this state must be specified.

Both of these parts are of equal importance, and a specification that ignores one of them would not be satisfactory. Process algebras offer appropriate constructs to specify behavior. With some extensions, also real-time requirements can be expressed. Model-based specification languages are suitable to specify the data-oriented part of the system. Since they allow the legal states of the system to be explicitly and elegantly specified, they are to be preferred over algebraic languages in this context. A combination of a process algebra and a model-based specification language yields a suitable language for the specification of software for safety-critical systems.

We choose to combine the specification notation Z with the process algebra real-time CSP (Davies, 1993) which adds real-time constructs to CSP (Hoare, 1985). Both languages are fairly well known and frequently used. Other such combinations, however, (e.g. VDM (Jones, 1990) and CCS (Milner, 1980)) would also be conceivable. Although with the language LOTOS, a combined language already exists, we do not use it because the specification language contained in LOTOS is a simple algebraic language that is less appropriate than Z for the data-oriented part of the specification.

A combination of two different specification languages must be given a common semantics; otherwise, combined specifications cannot be regarded as completely formal. Once this is achieved, we obtain a specification language tailored for the modeling of safety-critical systems.

A mere language, however, does not suffice to improve product quality. A methodology for its application that provides specifiers with guidance on how to construct specifications is indispensable. We provide such a methodology by identifying frequently used designs of safety-critical systems. These designs are expressed as reference architectures, and for each architecture we give an agenda that can be followed to develop an instance of the architecture.

The validation of a specification is as important as a controlled process for its development. Therefore, our approach also contains guidelines for this purpose. In the agendas, general validation criteria are stated that are independent of concrete applications, referring only to the chosen architecture. Furthermore, two different kinds of properties are important for safety-critical systems, namely safety-related and liveness properties. Examples are assertions that the system cannot stay longer than a certain time in a certain state, that the violation of a safety constraint is noticed within some time limit, that certain conditions exclude each other, or that certain conditions always occur together. Our method encourages specifiers to identify and demonstrate such properties.

The main focus of this chapter is on the development of the specification and its validation. A formal specification is a necessary prerequisite for the usage of formal methods in the development process. Later phases like design and implementation can only be supported by formal methods in presence of a formal specification. But formal specifications are not only necessary for exploiting the benefits of formal techniques. Often, it is even sufficient to develop a formal specification and perform the subsequent development steps with traditional methods to obtain a considerable gain in product quality (Houston and King, 1991).

Nevertheless, our approach also offers formal support for the later phases of the software development process. A notion of refinement for specifications in the combined language is defined, and how code can be synthesized for the Z part of a combined specification is shown in Chapter 6 and Section 7.5.

In the following, we first make explicit what kind of system we want to specify with our methodology (Section 3.1). This gives us a structure and the vocabulary to adequately model this kind of system. We then present the common language to be used in Section 3.2. The reference architectures and the corresponding agendas presented in Sections 3.3 and 3.4 make use of the language. How to further refine the specifications developed with the help of the agendas is explained in Section 3.5. A discussion of the approach, a summary and further research directions conclude the chapter. This chapter further elaborates concepts presented in (Heisel, 1995a; Heisel, 1996a; Heisel and Sühl, 1996b; Heisel and Sühl, 1996a; Sühl, 1996).

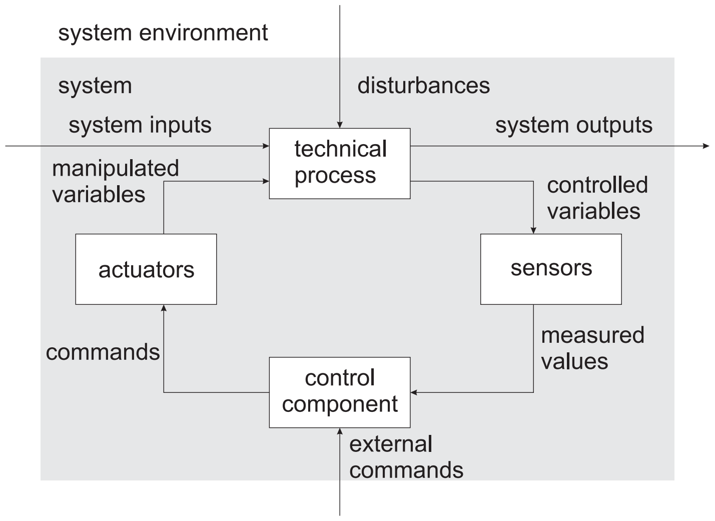


Figure 3.1: System Model

3.1 System Model

The purpose of the systems we want to consider is to control some technical process, where the control component is at least partially realized by software, see Figure 3.1 and (Leveson, 1995). 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.

The control component may consist of several sub-components, some of which can be realized with software. In the following, we focus on the software-based parts of the control component. A software-based control component affects certain process variables (*manipulated variables*) by sending commands to actuators. By evaluating the current state of certain process variables which are measured by sensors (*controlled variables*), the control component approximates the current state of the technical process to verify the effect of the commands sent to the actuators (feedback control) and to determine further commands to be sent. It is very important that the image of the state of the technical process that is built up in the software control component is sufficiently accurate and up-to-date. In the following, we will call this state the *internal* state, because it is internal to the software control component; the state of the technical process we will call the *external* state.

The behavior of the technical process does not only depend on internal conditions within the process, e.g. the state of the manipulated variables, but it is also influenced by external disturbances. The basic objective of process control is to achieve the process control function in spite of disturbances from the environment.

Safety can be defined as the property of a system to be free from accidents or losses (cf. (Leveson, 1995)). It follows that a software component which is considered in isolation cannot be unsafe because it is not directly able to cause a loss event. Safety is a property of a whole system in the context of its environment rather than a property of a separate system component. A method concerned with *software* development for safety-critical systems must aim at *system* safety and can only be evaluated in this respect. Hence, the following subsystems of a safety-critical system must be modeled to contribute to system safety:

- *all* parts of the process-control component, i.e. software components, mechanical and electrical components, and interfaces to human operators,
- sensors, determining the projection of the technical process state to the internal state of the control component, and
- actuators, which realize the execution of commands given by the control component within the technical process.

Another desirable property of software systems is *correctness*. What is the relationship between safety and correctness? The latter is defined as the property of a software component to fulfill the relation between inputs and outputs prescribed in the component specification.

One might consider safety to be a weaker requirement than correctness. Leveson (Leveson, 1986) states "We assume that, by definition, the correct states are safe." However, safety concerns have an influence on what is considered a correct state. For example, incorrect measurements of process variables or the failed realization of given commands by the actuators are normally not relevant in the context of correctness. To achieve system safety, on the other hand, the thorough examination of the above situations is a necessary condition.

This leads to differences in the modeling of a system. If correctness in the usual sense of the word is striven for, the environment in which the system operates, hardware failures, and the credibility of inputs are of no interest. In contrast, to achieve system safety, the environment must explicitly be modeled, too. It is necessary to try to detect hardware failures, and not only the specified *relation* between input and output values must be guaranteed. It must also be checked if the input values can represent a possible situation in the real world, e.g. by consistency checks on different sensors and by redundant arrangements of sensors.

The approaches to guarantee correctness on the one hand and safety on the other hand do not differ in a *technical*, but in a *methodological* way: in the end, safety requirements are expressed as functional requirements, and safety is guaranteed by developing software that is correct with respect to the safety requirements.

3.2 A Language to Specify Safety-Critical Software

In general, the control component of a safety-critical system is a *reactive* system, which is characterized as being mainly event-triggered. It continuously reacts to events occurring within the environment by invoking internal operations and subsequently emitting resulting events into the environment. Hence, two aspects of the software component must be specified in an adequate way: First, how does it react to events, and second, how is its internal state defined, and how can it evolve? Accordingly, we split the specification of a software component into two parts.

1. In the *dynamic* part the reactive behavior of the software component is specified, i.e. its reaction to the occurrence of events within the technical process (detected by sensors) which is realized by invoking internal operations and giving commands to the actuators. In this part, real-time requirements and the ordering of events are crucial.

2. In the *functional* part the invariant properties and the structure of the possible internal system states, i.e. data structures, are specified, as well as the operations applied to these states . Operations on the internal state are defined by relations between inputs, outputs, and the internal system states before and after the execution of the respective operation.

The specification languages Z and real-time CSP provide constructs to adequately express both aspects. Before we can show how the two languages are combined, we give a brief description of the language real-time CSP.

3.2.1 The Language Real-Time CSP

Real-time CSP (Davies, 1993; Davies and Schneider, 1995) is a language to model the communicating behavior of real-time systems. It adds real-time constructs to CSP (Hoare, 1985; Hinchey and Jarvis, 1995). The following definitions are mostly taken from (Davies and Schneider, 1995).

A *process* denotes the behavior pattern of a component of a real-time system. *Events* mark important points in the history of a process at which a process may communicate or interact. The set of events a process can engage in is called its *alphabet*.

Events are atomic and instantaneous, i.e. their occurrence takes no time. Processes need not compete for resources. This property is called maximal parallelism. The property of maximal progress says that internal events occur as soon as possible. Communication between processes is synchronous, i.e. each communication event requires the simultaneous participation of the involved processes.

Syntax of Real-Time CSP

Process expressions of real-time CSP are built from the following syntactic constructs:

Prefix: $a \rightarrow P$ first accepts event a and subsequently behaves as process P.

External Choice: $P \Box Q$ behaves either identical to process P or Q where the environment might influence this choice by accepting a certain initial event.

Internal Choice: $P \sqcap Q$ behaves either identical to process P or Q where the environment can neither influence nor observe the choice.

Channel Input: $c?x \rightarrow P(x)$ first is ready to receive an arbitrary value x from channel c and afterwards behaves as the parameterized process $P(x)$.

Channel Output: $c!v \rightarrow P$ first is ready to write the value v to the channel c and subsequently behaves equal to process P.

Parallel Composition: $P \parallel Q$ has the processes P and Q as parallel subprocesses.

Hiding: $P \setminus A$ behaves as P, except that all events contained in the event set A are hidden from the environment, i.e. they become internal.

Sequential Composition: $P; Q$ first behaves as process P until its termination and afterwards behaves as process Q.

Conditional: (if b then P else Q fi) behaves as P if the predicate b is true. Otherwise it behaves as Q.

Interrupt: $P \triangle Q$ behaves as P until the environment offers an initial event of Q. From then on, P is discarded and the process behaves as Q.

Atomic Process: *Skip* accepts the termination event before releasing control; *Stop* never engages in any event.

Wait: *Wait* t does not accept any event for the first t time units and afterwards is ready to accept the termination event before releasing control.

Timeout: $P \triangleright\{t\}\ Q$ is initially prepared to behave as P, but if no events have occurred within t time units, it begins to behave as Q instead.

Timed-Interrupt: $P \swarrow\{t\}\ Q$ behaves as P for t time units, and then behaves as Q.

Semantics of Real-Time CSP

Different semantic models have been developed for CSP. Each process is associated with the set of *observations* that can be made during its execution. In the *traces model*, an observation is a sequence of events:

$$traces : CSP \longrightarrow \mathbb{P}(\mathrm{seq}\ EVENTS)$$

In this model, however it is impossible to specify non-deterministic behavior: the processes $P \Box Q$ and $P \sqcap Q$ cannot be distinguished. This leads to the *failures-divergences model*, where two semantic functions are defined:

$$failures : CSP \longrightarrow \mathbb{P}(\mathrm{seq}\ EVENTS \times \mathbb{P}\ EVENTS)$$

maps a process onto a set of pairs, consisting of a trace and a set of events, called a *refusal*. A pair (tr, ref) is a failure of process P if P may perform tr and then refuse every event in ref.

$$divergences : CSP \longrightarrow \mathbb{P}(\mathrm{seq}\ EVENTS)$$

A trace tr is a *divergence* of P if P may engage in an unbounded sequence of internal events after performing tr.

For real-time CSP, a *timed failures model* is defined. This model associates with each process term a set of *timed failures* which represents possible observations of the process. A timed failure consists of a *timed trace* and a *timed refusal*. A timed trace is a sequence of *timed events*, where each timed event is a pair of an event and the time instant at when it was observed. A timed refusal is a set of timed events. In the case when the corresponding timed trace has been observed, an event can be refused by the system at a time instant if the corresponding pair is a member of the timed refusal.

$$timed\,failures : RT\_CSP \longrightarrow \mathbb{P}(\mathrm{seq}\ TimedEvents \times \mathbb{P}\ TimedEvents)$$

Behavioral Specifications

Two different styles of specifying the reactive behavior of systems in real-time CSP can be distinguished. First, a term of the syntax of real-time CSP can be given to model the dynamic behavior in a constructive manner which is amenable to further refinement. Second, predicates can be used to constrain the set of possible behaviors. This is a more abstract way of specification. Both approaches are semantically equivalent and can thus be combined arbitrarily. The syntax for behavioral specifications is

$$\mathit{Process}\ \mathsf{sat}\ \mathit{Spec}(o)$$

Its semantics is that all observations associated with the process must satisfy the specified predicate:

$$\begin{array}{l}\mathit{Process}\ \mathsf{sat}\ \mathit{Spec}(o) \Leftrightarrow \\ \quad \forall\, o : \mathit{OBSERVATIONS} \bullet o \in \mathit{semantic\,function}\,[\![\mathit{Process}]\!] \Rightarrow \mathit{Spec}(o)\end{array}$$

Specification of Timing Properties

The following specification macros allow one to state requirements concerning timed observations in a concise way. We would like to express that some event happens at some time instant, or that the process or its environment are prepared to engage in an event at some time instant.

The function σ yields the set of events occurring in a trace or refusal. With $\uparrow$, we restrict the traces or refusals we consider to a certain time interval. We can then give a formal definition of an event e happening at time t or being refused at this time, relative to a trace tr and refusal ref. To make assertions about a process, we must universally quantify over the timed failures associated with the process.

$$\begin{array}{l} e\ \mathsf{at}\ t\ (tr, ref) \Leftrightarrow e \in \sigma(tr \uparrow [t, t]) \\ e\ \mathsf{ref}\ t\ (tr, ref) \Leftrightarrow e \in \sigma(ref \uparrow [t, t]) \end{array}$$

If a process accepts event e at time t, this is expressed by the predicate

$$e\ \mathsf{live}\ t\ (tr, ref) \Leftrightarrow e\ \mathsf{at}\ t\ (tr, ref) \lor \neg\ (e\ \mathsf{ref}\ t\ (tr, ref))$$

The event happens at time t or it is not refused by the process.

An event e is accepted by the environment of the process at time t if it happens at t or is refused by the process.

$$e\ \mathsf{open}\ t\ (tr, ref) \Leftrightarrow e\ \mathsf{at}\ t\ (tr, ref) \lor e\ \mathsf{ref}\ t\ (tr, ref)$$

This concludes our presentation of real-time CSP. We now define a software model that leads to a suitable combination of the two languages.

3.2.2 Software Model

To achieve a suitable combination of both parts of the formal specification of a software component formulated in Z and real-time CSP, we propose the software model shown in Figure 3.2.

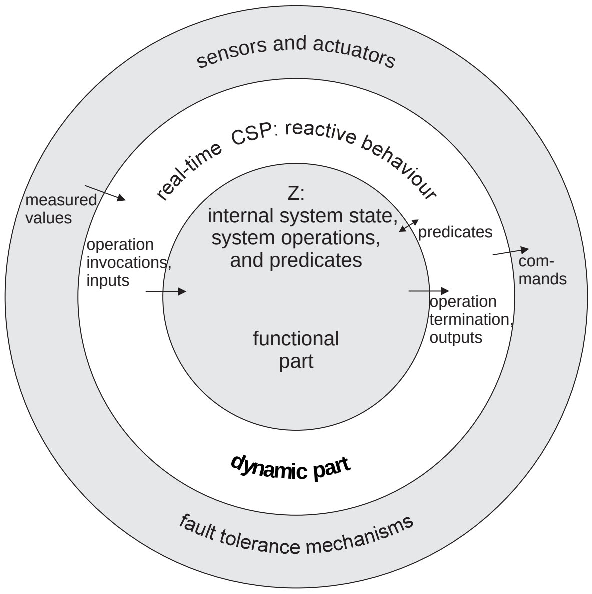


Figure 3.2: Software Model

1. The innermost component which is expressed in Z specifies the functional aspects, i.e. the structure and the properties of the valid internal system states as well as the requirements for system operations.

2. Around this innermost component, a CSP process specifies the reactive behavior, i.e. the absorption of values provided by the sensors, the invocation and termination of internal operations, and the transmission of the operation results to the actuators.

3. The outermost component models the required behavior of the sensors and actuators. It offers the possibility to specify fault tolerance mechanisms, e.g. the redundant arrangement of sensors and actuators. The sensors and actuators can be modeled in Z or in real-time CSP. This depends on the particular application[1].

Both the Z specification and the sensors and actuators form the environment of the CSP process.

Informally, the relation between the elements of the Z part and the CSP part of a formal specification can be explained as follows, see also Figure 3.3. For each system operation *Op* specified in the Z part which is intended to be externally available, the CSP part is able to refer to the events *OpInvocation* and *OpTermination*, whose occurrences represent the invocation of the system operation *Op* by the software component and its termination, respectively. The two events mark the execution interval of an operation. This makes it possible to specify requirements for the maximal duration in terms of assumptions about the environment which constrain the availability of these events, see Section 3.3. Alternatively, if the duration of the execution is negligible, only one event *OpExecution* is used to represent the execution of the operation *Op*.

[1]In Section 3.3.2, a redundancy mechanism is specified in Z. In Section 3.4, priorities on sensor messages are specified in real-time CSP.

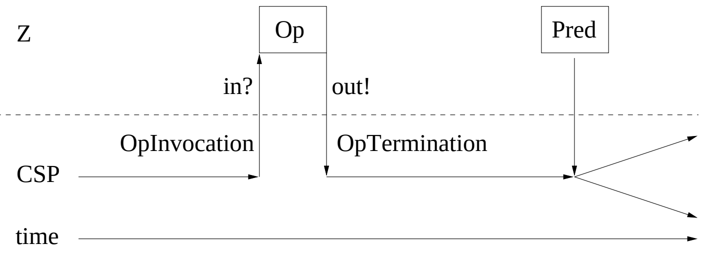


Figure 3.3: Connection between Z and CSP

For each input *in*? : *Type* of a system operation *Op*, there is a communication channel *in* within the CSP part onto which an input value possibly derived from sensor measurements is written before operation invocation. The alphabet of this channel is identical to the type of the operation input.

Analogously, for each output *out*! : *Type* of a system operation, there is a communication channel *out* in the CSP part from which the output value of the operation is read after termination and possibly used to derive commands to the respective actuator.

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.

These links between both parts contribute to a clear separation of the functional aspects from the dynamic aspects of the system.

The connection between the CSP part and the specification of the intended behavior of the sensors and actuators is as follows. The CSP part is linked with every sensor via a communication channel from which the measured values of the respective sensor are read. Analogously, the CSP part is connected to every actuator via a communication channel onto which the commands to the respective actuator are written.

Furthermore, the specification of communication channels in terms of CSP processes makes it feasible to model aspects of a distributed communication, for example the delay of transmission or the redundant arrangement of unreliable communication channels.

3.2.3 Common Semantic Model

In this section, we outline the formal definition of the semantics associated with a combined specification as explained informally in the previous sections. The basis of this definition is the semantic function of the timed failures model of real-time CSP, which we mentioned in Section 3.2.1.

To define a semantic function for the combined language consisting of Z and real-time CSP, we must define the set of all possible observations of a system specified in the combined language. In this context, apart from traces and refusals, a third component is of importance, namely the evolution of the internal system state within the observation interval. Hence an observation for a combined specification is a tuple consisting of a timed trace, a timed refusal, and a *timed state*. A timed state is defined as a function that maps every time instant of the observation interval to the observed system state.

The Z part of a specification is characterized by a state schema *State*, an initial state schema *InitState*, a set of external operation schemas $Op1, \ldots, OpN$, and a set of predicates on the internal system state $Pred1, \ldots, PredM$. The CSP part of a specification is characterized by a term of real-time CSP and a predicate of the timed failures model. The set *RESTR_RTCSP_PROCESS* contains all process terms of real-time CSP that do not allow subprocesses to perform an event concerning the execution of a system operation (and consequently to cause a state change) in parallel with other subprocesses that either perform an operation event or evaluate a predicate on the internal system state[2]. Furthermore, the set *TF_PREDICATE* contains all predicates of the timed failures model. Thus the signature of our semantic function is as follows.

$$\begin{aligned} &\mathit{timed\,failures\,states} : \mathit{SCHEMA} \times \mathit{SCHEMA} \times \mathbb{P}\,\mathit{SCHEMA} \times \\ &\quad \mathbb{P}\,\mathit{SCHEMA} \times \mathit{RESTR\_RTCSP\_PROCESS} \times \mathit{TF\_PREDICATE} \rightarrowtail\!\!\!\!\!\!+ \\ &\quad \mathbb{P}((\mathrm{seq}\ \mathit{TimedEvents} \times \mathbb{P}\,\mathit{TimedEvents}) \times (\mathit{TIME} \rightarrowtail\!\!\!\!\!\!+\ \mathit{STATES})) \end{aligned}$$

A possible observation $((tr, ref), tstate)$ of the behavior of the specified system can be interpreted in the following sense: the timed failure (tr, ref) consisting of the timed trace tr and the timed refusal ref is defined by the semantic function *timed failures* as a possible observation of the CSP process, and the timed state *tstate* maps each instant of the observation interval to an internal system state. This internal system state must be one of the states that can be reached at the respective time instant, starting from an initial state and proceeding in accordance with the operation events as well as their assigned input and output values which are recorded in the timed trace tr up to the considered time instant. The formal definition of the function *timed failures states* can be found in (Sühl, 1996).

3.3 The Passive Sensors Architecture

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. We express two of these different approaches to the design of safety-critical systems as *reference architectures*. The first architecture, which is the subject of this section, assumes that sensors are passive measuring devices. The second architecture, presented in Section 3.4, assumes that sensors can cause interrupts in the control component.

For each of the reference architectures, we define an agenda. The agendas describe the steps to be taken to specify software control components suitable for the reference architectures. They provide schematic expressions of Z or real-time CSP that only need to be instantiated, and state validation obligations that should be fulfilled. Our general approach to the specification of safety-critical software is to first decide on the architecture of the system for which a software control component must be specified, and then to follow the steps of the corresponding agenda.

The two architectures we present cover frequently used design principles of safety-critical systems. However, the aim of this work is not to completely cover the area of specification of safety-critical systems, but to show that detailed guidance can be provided to specifiers of software systems, under the condition that special contexts are considered, which stem from particular application areas. Consequently, other reference architectures for the design of safety-critical systems are useful, too. An example is a reference architecture for distributed systems, where each component may have its own private state. These additional

[2]We do not give a formal definition of *RESTR_RTCSP_PROCESS* but only note that in the specifications developed with our method at most one process of a parallel composition refers to the Z part of the specification.

reference architectures could be supported in a similar way as the ones presented in this chapter. Furthermore, concrete safety-critical systems need not be "pure" instances of predefined architectures. When necessary, reference architectures can be combined as appropriate.

For both reference architectures we present, 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 the control component – may be totally different. For instance, an elevator might assume different operational modes when moving up, moving down, not moving with the door open, and not moving with the door closed. Depending on the mode, the reaction of the elevator to a pressed button will be different.

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 (see (Heisel, 1996a; Sühl, 1996)), and an inert gas release system, whose purpose is to detect and extinguish fire. We present a specification of the latter as an example of the passive sensors architecture in Section 3.3.2.

Figure 3.4 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.

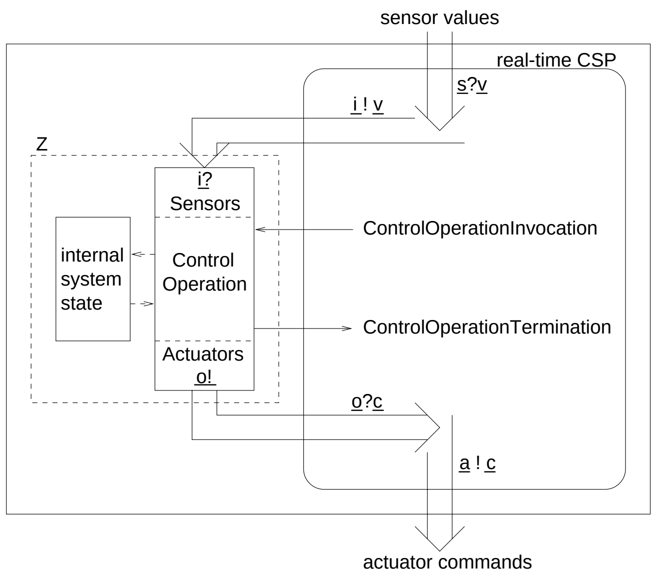


Figure 3.4: Software Control Component for Passive Sensors Architecture

The passive sensors architecture is suitable only for systems where all actuators are able to perform the commands given by the control component at arbitrary time instants, and for which it can be guaranteed that executing the control operation at equidistant time instants suffices to obtain a sample rate that is high enough to provide all relevant information about changes of the external system state.

3.3.1 Agenda for the Passive Sensors Architecture

An agenda gives instructions on how to proceed in the specification of a software-based control component for systems as described in Section 3.1, according to a chosen reference architecture. It consists of several steps, some of which have validation obligations associated with them. The steps need not be carried out exactly in the given order. Some of them are independent of each other. In the following, we give a graphical representation of the dependencies between the different steps for each agenda we present.

The validation obligations given in an agenda state only validation mechanisms that are independent of concrete applications. Usually, the specifications developed according to an agenda should be validated further, taking the specifics of the application into account. Such further validation will usually consist of mathematical proofs, where safety-related as well as liveness properties of the specification are demonstrated.

The agenda for the passive sensors architecture is summarized in Table 3.1. The dependencies between the steps are shown in Figure 3.5.

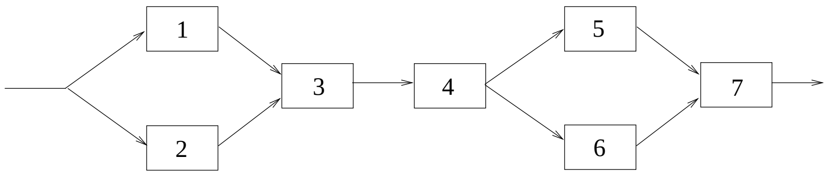


Figure 3.5: Dependencies of steps of agenda for passive sensors architecture

We now explain the steps one by one.

In the first step, we model the sensor values and actuator commands, i.e. the interface of the software component with its environment. Since for the architecture of passive sensors, the sensors are mere measuring devices, the measurement values of the sensors must be modeled as members of appropriate types in Z. These types coincide with the alphabets of the channels that are used by the CSP control process to read the sensor values coming from the environment of the software component. Analogously, we have to define types that model the actuator commands, which are an output of the Z control operation. Again, these types coincide with alphabets of communication channels in of the CSP control process.

Step 1 *Model the sensor values and actuator commands as members of Z types.*

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.

No.	Step	Validation Conditions
1	Model the sensor values and actuator commands as members of Z types.	
2	Decide on the operational modes of the system.	
3	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. 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.	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.
5	Define the Z control operation.	
6	Specify the control process in real-time CSP.	
7	Specify further requirements if necessary.	

Table 3.1: Agenda for the passive sensors architecture

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.

Step 2 *Decide on the operational modes of the system.*

The operational modes are defined as an enumeration type in Z. If possible, a fail-safe mode should be defined. The system can then switch to this mode when safety can no longer be guaranteed.

$$MODE ::= Mode1 \mid \ldots \mid ModeK$$

Next, the legal internal states of the software component must be defined by means of a Z schema.

Step 3 *Define the internal system states and the initial states.*

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 internal state must contain all information that is relevant to control the technical process. The specifier must decide if the sensor values are components of the state or if they are incorporated only indirectly as inputs of operations. The components must suffice to characterize the operational modes of the system.

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. It can be set up by one party, treating functional as well as safety-related requirements. Another possibility is to set up two specifications, a functional and a safety specification by different parties and then show that the safety requirements are entailed by the functional specification. The latter approach can be used to double-check the safety requirements, or it may be enforced by certification procedures or safety standards.

Initial states must be specified, too. Here, the specifier must decide if it is necessary to define the initial states in terms of real sensor values or if default values for the components suffice.

A suitable definition of the internal state is crucial for the safety of the system. Therefore, we have several validation obligations.

Validation Condition 3.1 *The internal system state must be an appropriate approximation of the state of the technical process.*

This condition cannot be proven formally. It is a reminder for the specifier to carefully reconsider the definition. The specifier must be convinced that the internal state represents all important aspects of the state of the technical process.

Validation Condition 3.2 *The internal state must contain a variable corresponding to the operational mode.*

This variable *mode* will be used to define the Z control operation.

Validation Condition 3.3 *Each legal state must be safe.*

This condition need only be shown if the state invariant does not directly contain the safety-related requirements, as in the second scenario discussed previously, where the safety and functional requirements are set up by different parties.

Validation Condition 3.4 *There must exist legal initial states.*

If this condition were not true, our state definition would be erroneous.

Validation Condition 3.5 *The initial internal states must adequately reflect the initial external system states.*

System safety can only be guaranteed if, on initialization of the system, the internal state faithfully reflects reality.

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.

Step 4 *Specify an internal Z operation for each operational mode.*

Each of these Z operations specifies the successor mode of the current mode 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.

$$\begin{aligned} \mathit{Sensors} \mathrel{\widehat{=}} [&\mathit{SystemState};\ \mathit{input}1? : \mathit{SType}1;\ \ldots;\ \mathit{inputN}? : \mathit{STypeN} \mid \\ &\langle \mathit{consistency\ conditions\ /\ redundancy\ mechanisms} \rangle\,] \end{aligned}$$

The types of the different inputs have been defined in Step 1. If necessary, the predicate part of the schema should contain the specification of consistency checks concerning the sensor measurements in relation to the internal system state. Therefore, the *Sensors* schema may import the state schema *SystemState*. Moreover, in the *Sensors* schema, redundancy mechanisms can be specified, e.g., the arrangement of several identical sensors and the derivation of a unique value from a set of measured values of the same controlled variable.

$$\begin{aligned} \mathit{Actuators} \mathrel{\widehat{=}} [&\mathit{SystemState}';\ \mathit{output}1! : \mathit{AType}1;\ \ldots;\ \mathit{outputM}! : \mathit{ATypeM} \mid \\ &\langle \mathit{derivation\ of\ commands} \rangle\,] \end{aligned}$$

In the *Actuators* schema, the derivation of commands from the current internal system state, i.e., the state after the internal operation has terminated, is specified.

Validation Condition 4.1 *The only precondition*[3] *of the operation corresponding to a mode is that the system is in that mode.*

This condition requires that the specifier not only takes the normal functioning of the system into account, but also considers the situation where an inconsistency between the sensor values and the internal system state is detected. We recommend to define a consistency condition for each operational mode and then define the internal operation by case distinction on this consistency condition.

Validation Condition 4.2 *For each operational mode and each combination of sensor values there must be exactly one successor mode.*

There must be *at least* one successor mode, because otherwise situations could arise that are not taken care of by the specification of the control component. The internal state would no longer faithfully reflect the external state, and safety could no longer be guaranteed.

There must be *at most* one successor mode, because otherwise the system would be nondeterministic. Although determinism is not a necessary condition for safety, in the most cases it will enhance comprehensibility of the specification.

[3]The formal definition of a precondition in Z also includes the state invariant and the requirement that the inputs are members of the specified sets. These conditions, however, are preconditions of all operations working on the system state and legal inputs. Therefore, we do not state them explicitly.

Validation Condition 4.3 *Each operational mode must be reachable from an initial state.*

Unreachable modes indicate a specification error. Either the mode is not necessary and should be eliminated, or the mode transition relation – which results from the definition of the internal operations – is erroneous.

Validation Condition 4.4 *There must be no redundant modes.*

There should be no two modes that cannot be distinguished, i.e. where the system behaves identically. Eliminating redundant modes makes the specification simpler and more comprehensible.

The central control operation defined in Z is a case distinction according to the operational modes.

Step 5 *Define the Z control operation.*

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*.

$$
\begin{array}{|l}
\hline
\textit{ControlOperation} \\
\Delta \textit{SystemState} \\
\textit{Sensors}; \textit{Actuators} \\
\hline
\textit{mode} = \textit{Mode}1 \Rightarrow \textit{OpMode}1 \\
\wedge \ldots \wedge \\
\textit{mode} = \textit{ModeK} \Rightarrow \textit{OpModeK} \\
\hline
\end{array}
$$

This concludes the specification of the functional view of the system using Z. It remains to define the behavior of the system using real-time CSP.

Step 6 *Specify the control process in real-time CSP.*

Again, we can provide schematic expressions to aid building the specification. First, the system must be initialized, establishing an initial state. If the initial state schema requires sensor values, these have to be read first. We give the schematic expression for the case where the initialization is performed with default values.

$$\textit{ControlComponent} \mathrel{\widehat{=}} \textit{SystemInitExecution} \rightarrow \textit{ControlComponent}_{READY}$$

The behavior of the process $\textit{ControlComponent}_{READY}$ (see Figure 3.4) is cyclic and is modeled by a recursive process definition. Before invoking the control operation, all associated input values are read from the respective 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.

$$
\begin{aligned}
&ControlComponent_{READY} \mathrel{\widehat{=}} \mu\, X \bullet \\
&\quad ((sensor1?valueS1 \rightarrow input1!valueS1 \rightarrow Skip \parallel \ldots \parallel \\
&\quad sensorN?valueSN \rightarrow inputN!valueSN \rightarrow Skip); \\
&\quad ControlOperationInvocation \rightarrow ControlOperationTermination \rightarrow \\
&\quad (output1?valueA1 \rightarrow actuator1!valueA1 \rightarrow Skip \parallel \ldots \parallel \\
&\quad outputM?valueAM \rightarrow actuatorM!valueAM \rightarrow Skip) \\
&\quad \parallel \\
&\quad Wait\ INTERVAL); X
\end{aligned}
$$

If the control operation is not time-critical, we can write *ControlOperationExecution* instead of *ControlOperationInvocation* $\rightarrow$ *ControlOperationTermination*.

The constant *INTERVAL* must be chosen small enough, so that it is guaranteed that the internal system state is always sufficiently up-to-date.

The execution time of the process $ControlComponent_{READY}$ is the maximum of the execution time of the Z control operation and the constant *INTERVAL*. Since we want the control operation to be executed every *INTERVAL* time units, we must state the requirement that the execution time of the control operation is at most *INTERVAL* time units. We can formally express this requirement by limiting the maximal time distance between the invocation and the termination of the control operation to *INTERVAL* time units. Moreover, the invocation of the control operation must be possible at any time. This is expressed as a predicate *EnvironmentalAssumption*[4]:

$$
\begin{aligned}
&EnvironmentalAssumption \mathrel{\widehat{=}} \\
&\quad (\forall\, t : [0, \infty); tr : \mathrm{seq}\ TimedEvents; ref : \mathbb{P}\ TimedEvents \mid \\
&\qquad (tr, ref) \in timed\,failures\, [\![ControlComponent]\!] \bullet \\
&\quad SystemInitExecution\ \mathsf{open}\ t\ (tr, ref) \land \\
&\quad ControlOperationInvocation\ \mathsf{open}\ t\ (tr, ref) \land \\
&\quad ControlOperationInvocation\ \mathsf{at}\ t\ (tr, ref) \Rightarrow (\exists\, t' : (t, t + INTERVAL] \bullet \\
&\qquad \forall\, t'' : [t', t + INTERVAL] \bullet ControlOperationTermination\ \mathsf{open}\ t''(tr, ref)))
\end{aligned}
$$

Furthermore, we require that each sensor is always able to send a measured value to the controller and that each actuator is always able to receive an arbitrary command from the controller:

[4] Recall that the environment of the real-time CSP process consists of the sensors and actuators and the Z part of the specification.

$$
\begin{array}{l}
\mathit{SensorActuatorAssumption} \mathrel{\widehat=} \\
\quad (\forall\, t : [0, \infty);\ tr : \mathrm{seq}\ \mathit{TimedEvents};\ \mathit{ref} : \mathbb{P}\ \mathit{TimedEvents} \mid \\
\quad\quad (tr, \mathit{ref}) \in \mathit{timed\,failures}\, [\![\mathit{ControlComponent}]\!] \bullet \\
\quad (\exists\, \mathit{value} : \mathit{SType}1 \bullet \mathit{sensor}1.\mathit{value}\ \mathsf{open}\ t\ (tr, \mathit{ref})) \wedge \\
\quad \ldots \\
\quad (\exists\, \mathit{value} : \mathit{STypeN} \bullet \mathit{sensorN}.\mathit{value}\ \mathsf{open}\ t\ (tr, \mathit{ref})) \\
\quad \wedge \\
\quad (\forall\, \mathit{value} : \mathit{AType}1 \bullet \mathit{actuator}1.\mathit{value}\ \mathsf{open}\ t\ (tr, \mathit{ref})) \wedge \\
\quad \ldots \\
\quad (\forall\, \mathit{value} : \mathit{ATypeM} \bullet \mathit{actuatorM}.\mathit{value}\ \mathsf{open}\ t\ (tr, \mathit{ref})))
\end{array}
$$

To summarize, the specification of the control process in real-time CSP consists of the definition of the process *ControlComponent* and the behavioral specifications *Environmental-Assumption* and *SensorActuatorAssumption*. We see that both styles of specifying behavior in real-time CSP (as process terms and as predicates) are useful and should be combined as appropriate.

Step 7 *Specify further requirements if necessary.*

Additional requirements depending on particular applications can be stated in this step.

3.3.2 Example: The Inert Gas System

The following case study is a variant of a specification problem used in (McDermid and Pierce, 1995). The software controller of an inert gas release system, which is operated from the control room of a plant, is to be specified. The task of this system is to detect fire in one of the machine rooms of the plant and to extinguish a detected fire with the help of inert gas. The sensors are passive, always allowing the controller to request the current value of the controlled process variable. The only control operation is executed at equidistant time instants.

Step 1: Model the sensors values and actuator commands as members of Z types

To detect the event of a fire in a certain machine room, the software controller of the inert gas system makes use of two redundantly arranged sensors ($\mathit{fire\_detector}1?$, $\mathit{fire\_detector}2?$) that are able to detect the presence of smoke. The controller only assumes the existence of fire if both sensors report smoke simultaneously. This redundancy mechanism will be defined in Step 5. The gas sensor ($\mathit{gas\_detector}?$) serves to observe if inert gas really escapes into the machine room after the gas release has been initiated. Thus, it realizes feedback control. For these sensors, we define the type $\mathit{DETECTION\_STATUS}$.

There are two banks of extinguishant, bank A and bank B. By means of a bank selector switch ($\mathit{bank\_selector}?$) within the control room, the operator is able to select one of them or to deselect both by choosing the *INHIBIT* position of the switch. This yields the type $\mathit{BANK\_SELECTOR\_STATUS}$.

Inside the control room there is an inhibit switch ($\mathit{inhibit\_button}?$), which – if in the inhibit position – prevents a fire alarm being automatically triggered somewhere in the plant. The

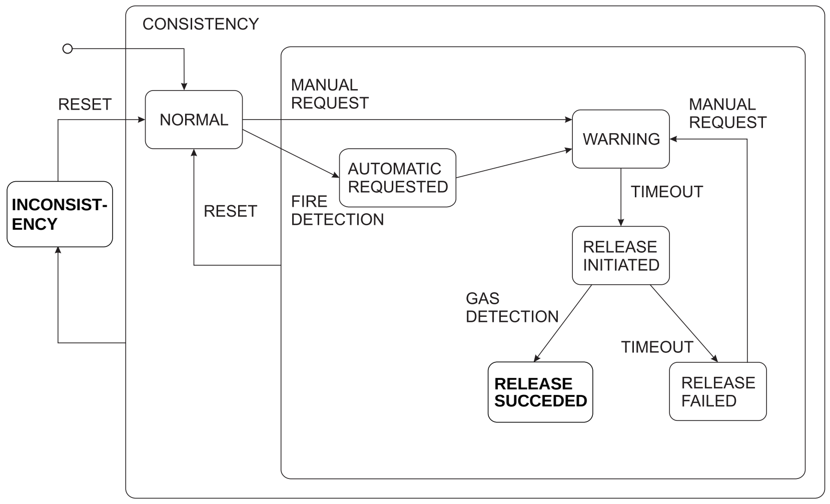


Figure 3.6: Operational modes of inert the gas system

operator can by-pass such a global inhibit by pressing the request button (*request_button?*). Moreover, the operator can abort the release of gas and reset the inert gas system at any time by pressing the reset button situated in the control room (*reset_button?*). The states of buttons are modeled by the type *BUTTON_STATUS*.

The controller guides the escape of inert gas from the two banks by means of two actuators (*release_bank_A!*, *release_bank_B!*). Their status is captured by the type *OPEN_CLOSED*. To inform the persons in the machine room that a release of gas will take place soon, that a release of gas is currently happening, that a release has taken place recently, or that a gas leak was detected, a warning light (*warning_light!*) can change between the states *ON*, *OFF*, and *FLASHING*. The warning beeper (*warning_beeper!*) serves a similar purpose acoustically.

This yields the following type definitions:

$$DETECTION\_STATUS ::= DETECTION \mid NO\_DETECTION$$

$$BANK\_SELECTOR\_STATUS ::= BANK\_A \mid BANK\_B \mid INHIBIT$$

$$BUTTON\_STATUS ::= PRESSED \mid NOT\_PRESSED$$

$$OPEN\_CLOSED ::= OPEN \mid CLOSED$$

$$LIGHT\_STATUS ::= ON \mid OFF \mid FLASHING$$

$$BEEP\_STATUS ::= BEEPING \mid NOT\_BEEPING$$

Step 2: Decide on the operational modes of the system

The operational modes of the inert gas system are defined in the following data type *MODE* and the possible transitions between them are depicted in Figure 3.6.

$$\begin{aligned} MODE ::= {} & NORMAL \mid AUTOMATIC\_REQUESTED \\ \mid {} & WARNING \mid RELEASE\_INITIATED \mid RELEASE\_FAILED \\ \mid {} & RELEASE\_SUCCEEDED \mid INCONSISTENCY \end{aligned}$$

Step 3: Define the internal system states and the initial states

We define the abstract states of the software controller in the schema *InertGasSystem*. The main component is the state variable *mode* representing the current operational mode. Furthermore, the controller must have at its disposal two timer components which are initialized with the duration of the warning period or the checking period and subsequently decrease their values until reaching zero. Hence, we need some more global definitions before we can define the state schema.

The constant *WARNING_DURATION* represents the duration of an interval after an automatic or manual request during which the persons in the machine room are warned before the release of inert gas actually takes place. The duration of the period during which the system tries to detect escaping gas after the initiation of gas release before assuming a failure is represented by the constant *CHECK_DURATION*. Finally, the length of the time interval between two consecutive executions of the control operation is characterized by the constant *EXECUTION_INTERVAL*.

$$
\begin{array}{|l}
WARNING\_DURATION : \mathbb{N}_1 \\
CHECK\_DURATION : \mathbb{N}_1 \\
EXECUTION\_INTERVAL : \mathbb{N}_1 \\
\hline
WARNING\_DURATION \bmod EXECUTION\_INTERVAL = 0 \\
CHECK\_DURATION \bmod EXECUTION\_INTERVAL = 0
\end{array}
$$

It is required that the warning and check durations are multiples of the time distance between two consecutive executions of the control operation.

The timer components are represented by the state variables *warning_timer* and *check-_timer*, respectively. The other state components define the current states of the actuators as assumed by the controller.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\_\_ InertGasSystem \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
mode : MODE \\
warning\_timer : 0 \mathrel{..} WARNING\_DURATION \\
release\_check\_timer : 0 \mathrel{..} CHECK\_DURATION \\
release\_bank\_A, release\_bank\_B : OPEN\_CLOSED \\
warning\_light : LIGHT\_STATUS \\
warning\_beeper : BEEP\_STATUS \\
\hline
mode \neq RELEASE\_INITIATED \Rightarrow \\
\quad release\_bank\_A = release\_bank\_B = CLOSED \\
mode = WARNING \Leftrightarrow warning\_timer > 0 \\
mode = RELEASE\_INITIATED \Leftrightarrow release\_check\_timer > 0 \\
\quad \Leftrightarrow warning\_light = ON \\
mode \notin \{WARNING, RELEASE\_INITIATED, INCONSISTENCY\} \\
\quad \Leftrightarrow warning\_light = OFF \\
\quad warning\_light = FLASHING \Leftrightarrow mode \in \{WARNING, INCONSISTENCY\} \\
mode = NORMAL \Leftrightarrow warning\_beeper = NOT\_BEEPING \\
\hline
\end{array}
$$

Only in the mode *RELEASE_INITIATED* a release of gas from bank A or bank B is possible if the selector switch is in the corresponding position. The warning timer is only set (i.e.

has a strictly positive value) in the mode *WARNING*, and the release check timer is only set in the mode *RELEASE_INITIATED*. The warning light is *ON* when inert gas is released, and it is flashing in the warning period before the gas release and in the *INCONSISTENCY* mode. The warning beeper is always beeping outside the *NORMAL* mode.

The initial operational mode of the system is the *NORMAL* mode.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{InertGasSystemInit}} \\ \hline
\textit{InertGasSystem}' \\ \hline
\textit{mode}' = \textit{NORMAL} \\ \hline
\end{array}$$

We now have to show some validation conditions. As already mentioned, condition 3.1 is a matter of judgment. As required by validation condition 3.2, the state schema contains the variable *mode* corresponding to the operational mode of the system. Since the purpose of the whole system is to detect unsafe situations in the plant and to deal with these, the safety-related requirements coincide with the functional requirements. Hence, condition 3.3 is trivially satisfied. The initialization of the system with the *NORMAL* mode is consistent with the state invariant. Therefore, validation condition 3.4 is fulfilled. It is safe to initialize the system in this way, because a possible fire during initialization of the system will be detected as soon as the first sensor values are delivered. Hence, validation condition 3.5 is also satisfied.

Step 4: Specify an internal Z operation for each operational mode

As recommended, we first define schemas for the sensors and actuators, where we use the names for the sensors and actuators as introduced in Step 1.

There is a consistency condition between the current state of the controller, i.e. the current operational mode, and the incoming sensor values. An inconsistency exists if and only if the controller is not in the mode *RELEASE_INITIATED* (and consequently not releasing inert gas) but the gas sensor is reporting the detection of escaping gas. This is represented by the component *consistency*. This condition allows the controller to detect leaks in the gas banks. To define *consistency*, we need the type *YES_NO*.

$$\textit{YES\_NO} ::= \textit{YES} \mid \textit{NO}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Sensors}} \\ \hline
\textit{InertGasSystem} \\
\textit{bank\_selector}? : \textit{BANK\_SELECTOR\_STATUS} \\
\textit{request\_button}? : \textit{BUTTON\_STATUS} \\
\textit{reset\_button}? : \textit{BUTTON\_STATUS} \\
\textit{inhibit\_button}? : \textit{BUTTON\_STATUS} \\
\textit{fire\_detector}1?, \textit{fire\_detector}2? : \textit{DETECTION\_STATUS} \\
\textit{gas\_detector}? : \textit{DETECTION\_STATUS} \\
\textit{fire\_detector} : \textit{DETECTION\_STATUS} \\
\textit{consistency} : \textit{YES\_NO} \\ \hline
\textit{fire\_detector} = \textit{DETECTION} \Leftrightarrow \\
\qquad \textit{fire\_detector}1? = \textit{fire\_detector}2? = \textit{DETECTION} \\
\textit{consistency} = \textit{NO} \Leftrightarrow \\
\qquad \textit{mode} \neq \textit{RELEASE\_INITIATED} \land \textit{gas\_detector}? = \textit{DETECTION} \\ \hline
\end{array}$$

All outputs of the system are collected in the *Actuators* schema. The operator of the inert gas system is informed about the state of the system (*mode*!).

$$
\begin{array}{|l}
\textit{Actuators} \\
\hline
InertGasSystem' \\
release\_bank\_A!, release\_bank\_B! : OPEN\_CLOSED \\
warning\_light! : LIGHT\_STATUS \\
warning\_beeper! : BEEP\_STATUS \\
mode! : MODE \\
\hline
release\_bank\_A! = release\_bank\_A' \\
release\_bank\_B! = release\_bank\_B' \\
warning\_light! = warning\_light' \\
warning\_beeper! = warning\_beeper' \\
mode! = mode' \\
\hline
\end{array}
$$

We are now able to define the operations for the modes shown in Figure 3.6 one by one.

Under normal environmental conditions, i.e., if there has been no fire detection by the sensors and no manual request by the operator, the controller is in the mode *NORMAL*. No inert gas is released, and the visual or auditory signals are switched off.

$$
\begin{array}{|l}
\textit{OpNormal} \\
\hline
\Delta InertGasSystem \\
Sensors; Actuators \\
\hline
mode = NORMAL \\
\\
(consistency = NO \Rightarrow mode' = INCONSISTENCY) \\
(consistency = YES \Rightarrow \\
\quad (reset\_button? = PRESSED \Rightarrow mode' = NORMAL) \wedge \\
\quad (reset\_button? = NOT\_PRESSED \Rightarrow \\
\quad\quad (request\_button? = PRESSED \Rightarrow mode' = WARNING) \wedge \\
\quad\quad (request\_button? = NOT\_PRESSED \Rightarrow \\
\quad\quad\quad (fire\_detector = DETECTION \Rightarrow \\
\quad\quad\quad\quad mode' = AUTOMATIC\_REQUESTED) \wedge \\
\quad\quad\quad (fire\_detector = NO\_DETECTION \Rightarrow mode' = NORMAL)))) \\
\hline
\end{array}
$$

If both redundantly arranged sensors report the detection of smoke, the controller changes to the mode *AUTOMATIC_REQUESTED*. If the request button is pressed in the mode *NORMAL* there is a transition into the mode *WARNING*. If the reset button is pressed, a transition from any mode to the mode *NORMAL* is the consequence.

In the case of an automatic request, the *WARNING* mode can only be entered if the global inhibit switch is not set.

$$\begin{array}{l}
\textit{OpAutomaticReq} \\
\hline
\Delta \textit{InertGasSystem} \\
\textit{Sensors}; \textit{Actuators} \\
\hline
\textit{mode} = \textit{AUTOMATIC\_REQUESTED} \\[6pt]
(\textit{consistency} = \textit{NO} \Rightarrow \textit{mode}' = \textit{INCONSISTENCY}) \\
(\textit{consistency} = \textit{YES} \Rightarrow \\
\quad (\textit{reset\_button}? = \textit{PRESSED} \Rightarrow \textit{mode}' = \textit{NORMAL}) \wedge \\
\quad (\textit{reset\_button}? = \textit{NOT\_PRESSED} \Rightarrow \\
\quad\quad (\textit{inhibit\_button}? = \textit{PRESSED} \Rightarrow \textit{mode}' = \textit{AUTOMATIC\_REQUESTED}) \wedge \\
\quad\quad (\textit{inhibit\_button}? = \textit{NOT\_PRESSED} \Rightarrow \\
\quad\quad\quad \textit{mode}' = \textit{WARNING} \wedge \textit{warning\_timer}' = \textit{WARNING\_DURATION}))) \\
\hline
\end{array}$$

In the mode *WARNING* which lasts exactly *WARNING_DURATION* time units, the warning light is flashing to inform the persons in the machine room about the following release of inert gas to give them the possibility to leave the danger area. After this warning period has elapsed, there is a transition into the mode *RELEASE_INITIATED*. At each execution of the control operation in the mode *WARNING*, the warning timer either has to be reduced by *EXECUTION_INTERVAL* or, if the controller leaves the *WARNING* mode, has to be set to zero to fulfill the state invariant.

$$\begin{array}{l}
\textit{OpWarning} \\
\hline
\Delta \textit{InertGasSystem} \\
\textit{Sensors}; \textit{Actuators} \\
\hline
\textit{mode} = \textit{WARNING} \\
(\textit{warning\_timer}' = \textit{warning\_timer} - \textit{EXECUTION\_INTERVAL} \\
\quad \vee\ \textit{warning\_timer}' = 0) \\
(\textit{consistency} = \textit{NO} \Rightarrow \textit{mode}' = \textit{INCONSISTENCY}) \\
(\textit{consistency} = \textit{YES} \Rightarrow \\
\quad (\textit{reset\_button}? = \textit{PRESSED} \Rightarrow \textit{mode}' = \textit{NORMAL}) \wedge \\
\quad (\textit{reset\_button}? = \textit{NOT\_PRESSED} \Rightarrow \\
\quad\quad (\textit{warning\_timer} - \textit{EXECUTION\_INTERVAL} > 0 \Rightarrow \\
\quad\quad\quad \textit{mode}' = \textit{WARNING}) \wedge \\
\quad\quad (\textit{warning\_timer} - \textit{EXECUTION\_INTERVAL} = 0 \Rightarrow \\
\quad\quad\quad \textit{mode}' = \textit{RELEASE\_INITIATED} \wedge \\
\quad\quad\quad (\textit{release\_bank\_A}' = \textit{OPEN} \Leftrightarrow \textit{bank\_selector}? = \textit{BANK\_A}) \wedge \\
\quad\quad\quad (\textit{release\_bank\_B}' = \textit{OPEN} \Leftrightarrow \textit{bank\_selector}? = \textit{BANK\_B}) \wedge \\
\quad\quad\quad \textit{release\_check\_timer}' = \textit{CHECK\_DURATION}))) \\
\hline
\end{array}$$

In the *RELEASE_INITIATED* mode, inert gas is released either from bank A or bank B, or no gas is released if the bank selector switch is in the *INHIBIT* position. The alarm light is *ON* to indicate the potential danger. During a period of *CHECK_DURATION* time units it is tested if inert gas is indeed escaping into the machine room. The detection of escaping gas by the respective sensor will cause a transition into the mode *RELEASE_SUCCEEDED*. If there is no gas detection within this period, a change into the mode *RELEASE_FAILED* results. At each execution of the control operation in the mode *RELEASE_INITIATED* the check timer either has to be reduced by *EXECUTION_INTERVAL* or, if the controller leaves the *RELEASE_INITIATED* mode, has to be set to zero to fulfill the state invariant.

$$
\begin{array}{|l}
\textit{OpReleaseInitiated} \\
\hline
\Delta \textit{InertGasSystem} \\
\textit{Sensors}; \textit{Actuators} \\
\hline
\textit{mode} = \textit{RELEASE\_INITIATED} \\
(\textit{release\_check\_timer}' = \textit{release\_check\_timer} - \textit{EXECUTION\_INTERVAL} \\
\quad \lor \textit{release\_check\_timer}' = 0) \\
(\textit{consistency} = \textit{NO} \Rightarrow \textit{mode}' = \textit{INCONSISTENCY}) \\
(\textit{consistency} = \textit{YES} \Rightarrow \\
\quad (\textit{reset\_button}? = \textit{PRESSED} \Rightarrow \textit{mode}' = \textit{NORMAL}) \land \\
\quad (\textit{reset\_button}? = \textit{NOT\_PRESSED} \Rightarrow \\
\qquad (\textit{gas\_detector}? = \textit{DETECTION} \Rightarrow \textit{mode}' = \textit{RELEASE\_SUCCEEDED}) \land \\
\qquad (\textit{gas\_detector}? = \textit{NO\_DETECTION} \Rightarrow \\
\qquad\quad (\textit{release\_check\_timer} - \textit{EXECUTION\_INTERVAL} > 0 \Rightarrow \\
\qquad\qquad \textit{mode}' = \textit{RELEASE\_INITIATED} \land \\
\qquad\qquad \textit{release\_bank\_A}' = \textit{release\_bank\_A} \land \\
\qquad\qquad \textit{release\_bank\_B}' = \textit{release\_bank\_B}) \land \\
\qquad\quad (\textit{release\_check\_timer} - \textit{EXECUTION\_INTERVAL} = 0 \Rightarrow \\
\qquad\qquad \textit{mode}' = \textit{RELEASE\_FAILED})))) \\
\hline
\end{array}
$$

Being in the mode *RELEASE_FAILED* indicates that either the chosen bank is empty or defective or that the bank selector switch is in the *INHIBIT* position. Therefore the operator must have the possibility to change the selector position and to repeat the process of gas release. This is done by pressing the request button, which causes a transition into the mode *WARNING*.

$$
\begin{array}{|l}
\textit{OpReleaseFailed} \\
\hline
\Delta \textit{InertGasSystem} \\
\textit{Sensors}; \textit{Actuators} \\
\hline
\textit{mode} = \textit{RELEASE\_FAILED} \\
(\textit{consistency} = \textit{NO} \Rightarrow \textit{mode}' = \textit{INCONSISTENCY}) \\
(\textit{consistency} = \textit{YES} \Rightarrow \\
\quad (\textit{reset\_button}? = \textit{PRESSED} \Rightarrow \textit{mode}' = \textit{NORMAL}) \land \\
\quad (\textit{reset\_button}? = \textit{NOT\_PRESSED} \Rightarrow \\
\qquad (\textit{request\_button}? = \textit{PRESSED} \Rightarrow \textit{mode}' = \textit{WARNING}) \land \\
\qquad (\textit{request\_button}? = \textit{NOT\_PRESSED} \Rightarrow \textit{mode}' = \textit{RELEASE\_FAILED}))) \\
\hline
\end{array}
$$

The system stays in mode *OpReleaseSucceeded* until the reset button is pressed or an inconsistency between the internal system state and the sensor values is detected.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\_\, \mathit{OpReleaseSucceeded}\,\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
\Delta \mathit{InertGasSystem} \\
\mathit{Sensors}; \mathit{Actuators} \\
\hline
\mathit{mode} = \mathit{RELEASE\_SUCCEEDED} \\
(\mathit{consistency} = \mathit{NO} \Rightarrow \mathit{mode}' = \mathit{INCONSISTENCY}) \\
(\mathit{consistency} = \mathit{YES} \Rightarrow \\
\quad (\mathit{reset\_button}? = \mathit{PRESSED} \Rightarrow \mathit{mode}' = \mathit{NORMAL}) \wedge \\
\quad (\mathit{reset\_button}? = \mathit{NOT\_PRESSED} \Rightarrow \mathit{mode}' = \mathit{RELEASE\_SUCCEEDED})) \\
\hline
\end{array}
$$

If the controller notices an inconsistency in an arbitrary mode it immediately changes to the mode *INCONSISTENCY*. No inert gas is released in this mode and the flashing warning light and warning beep alert the persons in the machine room.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\_\, \mathit{OpInconsistency}\,\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
\Delta \mathit{InertGasSystem} \\
\mathit{Sensors}; \mathit{Actuators} \\
\hline
\mathit{mode} = \mathit{INCONSISTENCY} \\
(\mathit{consistency} = \mathit{NO} \Rightarrow \mathit{mode}' = \mathit{INCONSISTENCY}) \\
(\mathit{consistency} = \mathit{YES} \Rightarrow \\
\quad (\mathit{reset\_button}? = \mathit{PRESSED} \Rightarrow \mathit{mode}' = \mathit{NORMAL}) \wedge \\
\quad (\mathit{reset\_button}? = \mathit{NOT\_PRESSED} \Rightarrow \mathit{mode}' = \mathit{INCONSISTENCY})) \\
\hline
\end{array}
$$

Analysis of these schemas shows that they all contain case distinctions according to the consistency of the internal state with the sensor values and the environmental conditions as represented by the sensor values. Hence, validation conditions 4.1 and 4.2 are easily verified. Inspection of the Z operations also shows that they faithfully represent the state transition diagram of Figure 3.6. Therefore, conditions 4.3 and 4.4 are also fulfilled.

Step 5: Define the Z control operation

This step simply consists of the instantiation of the generic schema given in the agenda.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\_\, \mathit{ControlOperation}\,\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
\Delta \mathit{InertGasSystem} \\
\mathit{Sensors}; \mathit{Actuators} \\
\hline
\mathit{mode} = \mathit{NORMAL} \Rightarrow \mathit{OpNormal} \\
\mathit{mode} = \mathit{AUTOMATIC\_REQUESTED} \Rightarrow \mathit{OpAutomaticReq} \\
\mathit{mode} = \mathit{WARNING} \Rightarrow \mathit{OpWarning} \\
\mathit{mode} = \mathit{RELEASE\_INITIATED} \Rightarrow \mathit{OpReleaseInitiated} \\
\mathit{mode} = \mathit{RELEASE\_FAILED} \Rightarrow \mathit{OpReleaseFailed} \\
\mathit{mode} = \mathit{RELEASE\_SUCCEEDED} \Rightarrow \mathit{OpReleaseSucceeded} \\
\mathit{mode} = \mathit{INCONSISTENCY} \Rightarrow \mathit{OpInconsistency} \\
\hline
\end{array}
$$

Step 6: Specify the control process in real-time CSP

According to Step 6 of the agenda, we get the top-level process

$$ControlComponent \mathrel{\widehat=} InertGasSystemInitExecution \rightarrow ControlComponent_{READY}$$

The definition of the process $ControlComponent_{READY}$ also follows the schematic expression given on page 43 (in the version with a single event *ControlOperationExecution*), with the minor syntactic difference that we define separate processes *SensorInputs* and *ActuatorOutputs*.

$$\begin{array}{l}
ControlComponent_{READY} \mathrel{\widehat=} \mu\, X \bullet \\
\quad (SensorInputs; \\
\quad (ControlOperationExecution \rightarrow \\
\quad ActuatorOutputs) \\
\quad \| \\
\quad Wait\ EXECUTION\_INTERVAL);\ X
\end{array}$$

The process *SensorInputs* specifies the reading of sensor values before executing of the control operation. All sensor values are read in parallel from the corresponding communication channels. These values are subsequently written to the channels having the identical names as the inputs of the operation schema.

$$\begin{array}{l}
SensorInputs \mathrel{\widehat=} \\
\quad bank\_selector\_sensor?bs\_status \rightarrow bank\_selector!bs\_status \rightarrow Skip \\
\| \\
\quad request\_button\_sensor?rq\_status \rightarrow request\_button!rq\_status \rightarrow Skip \\
\| \\
\quad reset\_button\_sensor?rs\_status \rightarrow reset\_button!rs\_status \rightarrow Skip \\
\| \\
\quad inhibit\_button\_sensor?ib\_status \rightarrow inhibit\_button!ib\_status \rightarrow Skip \\
\| \\
\quad fire\_detector1\_sensor?fd1\_status \rightarrow fire\_detector1!fd1\_status \rightarrow Skip \\
\| \\
\quad fire\_detector2\_sensor?fd2\_status \rightarrow fire\_detector2!fd2\_status \rightarrow Skip \\
\| \\
\quad gas\_detector\_sensor?gd\_status \rightarrow gas\_detector!gd\_status \rightarrow Skip
\end{array}$$

The process *ActuatorOutputs* is defined analogously.

$$\begin{array}{l}
ActuatorOutputs \mathrel{\widehat=} \\
\quad release\_bank\_A?rbA\_status \rightarrow release\_bank\_A\_actuator!rbA\_status \rightarrow Skip \\
\| \\
\quad release\_bank\_B?rbB\_status \rightarrow release\_bank\_B\_actuator!rbB\_status \rightarrow Skip \\
\| \\
\quad warning\_light?wl\_status \rightarrow warning\_light\_actuator!wl\_status \rightarrow Skip \\
\| \\
\quad warning\_beeper?wb\_status \rightarrow warning\_beeper\_actuator!wb\_status \rightarrow Skip \\
\| \\
\quad mode?m\_status \rightarrow mode\_output!m\_status \rightarrow Skip
\end{array}$$

As prescribed in the agenda, we define a predicate *EnvironmentalAssumption* concerning the environment. We require that the control operation and the initialization operation may be executed at arbitrary time instants by the controller.

$$\begin{array}{l}
\mathit{EnvironmentalAssumption} \mathrel{\widehat{=}} \\
\quad (\forall\, t : [0, \infty);\ tr : \mathrm{seq}\ \mathit{TimedEvents};\ \mathit{ref} : \mathbb{P}\ \mathit{TimedEvents} \mid \\
\qquad (tr, \mathit{ref}) \in \mathit{timed\,failures}\ [\![\mathit{ControlComponent}]\!] \bullet \\
\quad \mathit{InertGasSystemInitExecution}\ \mathsf{open}\ t\ (tr, \mathit{ref}) \wedge \\
\quad \mathit{ControlOperationExecution}\ \mathsf{open}\ t\ (tr, \mathit{ref}))
\end{array}$$

To express the assumption that each sensor is always able to send a measured value to the controller and that every actuator is always able to receive an arbitrary command from the controller, we again instantiate the schematic expression given in the agenda.

$$\begin{array}{l}
\mathit{SensorActuatorAssumption} \mathrel{\widehat{=}} \\
(\forall\, t : [0, \infty);\ tr : \mathrm{seq}\ \mathit{TimedEvents};\ \mathit{ref} : \mathbb{P}\ \mathit{TimedEvents} \mid \\
\quad (tr, \mathit{ref}) \in \mathit{timed\,failures}\ [\![\mathit{ControlComponent}]\!] \bullet \\
(\exists\, \mathit{value} : \mathit{BANK\_SELECTOR\_STATUS} \bullet \\
\quad (\mathit{bank\_selector\_sensor.value})\ \mathsf{open}\ t\ (tr, \mathit{ref})) \wedge \\
(\exists\, \mathit{value} : \mathit{BUTTON\_STATUS} \bullet (\mathit{request\_button\_sensor.value})\ \mathsf{open}\ t\ (tr, \mathit{ref})) \wedge \\
(\exists\, \mathit{value} : \mathit{BUTTON\_STATUS} \bullet (\mathit{reset\_button\_sensor.value})\ \mathsf{open}\ t\ (tr, \mathit{ref})) \wedge \\
(\exists\, \mathit{value} : \mathit{BUTTON\_STATUS} \bullet (\mathit{inhibit\_button\_sensor.value})\ \mathsf{open}\ t\ (tr, \mathit{ref})) \wedge \\
(\exists\, \mathit{value} : \mathit{DETECTION\_STATUS} \bullet (\mathit{fire\_detector1\_sensor.value})\ \mathsf{open}\ t\ (tr, \mathit{ref})) \wedge \\
(\exists\, \mathit{value} : \mathit{DETECTION\_STATUS} \bullet (\mathit{fire\_detector2\_sensor.value})\ \mathsf{open}\ t\ (tr, \mathit{ref})) \\
\wedge \\
(\forall\, \mathit{value} : \mathit{OPEN\_CLOSED} \bullet (\mathit{release\_bank\_A\_actuator.value})\ \mathsf{open}\ t\ (tr, \mathit{ref})) \wedge \\
(\forall\, \mathit{value} : \mathit{OPEN\_CLOSED} \bullet (\mathit{release\_bank\_B\_actuator.value})\ \mathsf{open}\ t\ (tr, \mathit{ref})) \wedge \\
(\forall\, \mathit{value} : \mathit{LIGHT\_STATUS} \bullet (\mathit{warning\_light\_actuator.value})\ \mathsf{open}\ t\ (tr, \mathit{ref})) \wedge \\
(\forall\, \mathit{value} : \mathit{BEEP\_STATUS} \bullet (\mathit{warning\_beeper\_actuator.value})\ \mathsf{open}\ t\ (tr, \mathit{ref})) \wedge \\
(\forall\, \mathit{value} : \mathit{MODE} \bullet (\mathit{mode\_actuator.value})\ \mathsf{open}\ t\ (tr, \mathit{ref})))
\end{array}$$

This concludes the specification of the inert gas system. Step 7 is not necessary. This example shows that – once a suitable architecture and the necessary operating modes are chosen – the specification can be set up in a fairly routine way.

Further validation of the specification

Apart from the general validation criteria, we state safety-related and liveness properties following from the specification.

The specification guarantees the following safety-related properties:

1. Gas leaks are detected and result in an alarm.

2. A gas release can only take place if both of the smoke sensors detect smoke.

3. If a fire is detected, the persons in the danger area have *WARNING_DURATION* time units to be evacuated before gas is released. They are visually and acoustically warned during that time.

4. If a fire is detected but the release of gas is not successful, this can be noticed by the operator after *CHECK_DURATION* time units.

The specification guarantees the following liveness properties:

5. After an unsuccessful gas release, the operator can change the bank selector switch and manually try to release gas.
6. The system can be brought back to normal operation at any time by pressing the reset button.
7. At every time instant the operator is able to by-pass a global inhibit (inhibit button is set) for a certain machine room by pressing the corresponding request button for this room.

We do not give proofs of these properties here because they follow directly from the specification.

3.4 The Active Sensors Architecture

This architecture models systems with only active sensors, i.e., all sensors control a certain variable of the technical process and independently report certain changes of the controlled variable to the control component at arbitrary time instants. Such a report immediately triggers the execution of a handling operation within the control component.

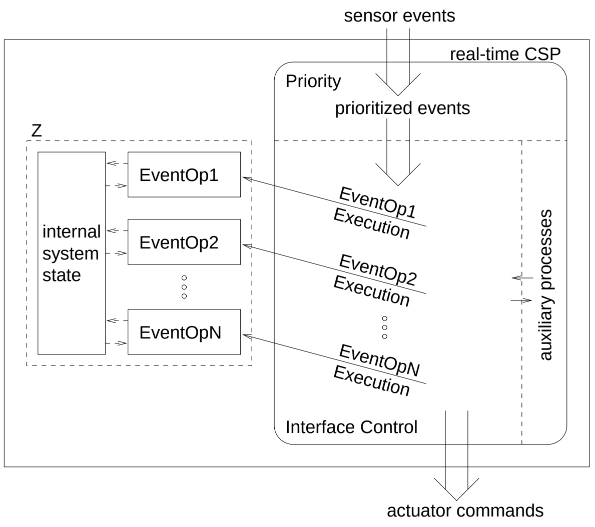


Figure 3.7: Software Control Component for Active Sensors Architecture

Figure 3.7 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 elevators (Sühl, 1996), microwave ovens (Heisel, 1995a), or the gas burner presented in Section 3.4.2.

As for the passive sensors architecture, we first give an agenda and then, following the agenda, we specify an example system.

3.4.1 Agenda

An overview of the agenda is given in Table 3.2. The dependencies between the steps are shown in Figure 3.8. Steps 4 and 6 are drawn in the same box because they both depend on Steps 1-3. Step 9 depends only on Step 4, whereas Step 7 depends on Steps 4, 6, and 5. Step 8 depends on Steps 2, 6 and 7.

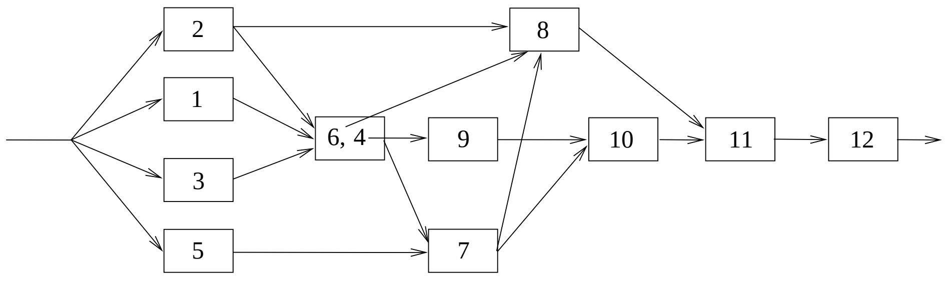


Figure 3.8: Dependencies of steps

Step 1 *Model the sensors as CSP events or members of Z types.*

In the active sensors architecture, sensors trigger operations of the control component. If a sensor carries a measured value, it is modeled as in the passive sensors architecture. If a sensor just carries boolean information (i.e., something happens or not), it is modeled as a CSP event, without a corresponding communication channel. This step yields a set of events *External_Events*. If a sensor delivers values of type T over communication channel c, then the set *External_Events* contains events of the form $c?v$, for all $v \in T$.

Step 2 *Decide on auxiliary processes.*

One can regard these auxiliary processes as subcomponents of the controller that do not need a state. Examples are timers that are controlled by the software component, and beepers that have to beep for a number of time units, and are automatically switched off afterwards.

No.	Step	Validation Condition
1	Model the sensors as CSP events or members of 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 successor mode for a mode-event pair must be justified. All modes must be reachable from an initial mode, and there must be no redundant modes.
5	Model the actuator commands as members of Z types or CSP events.	
6	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.
7	Specify a Z operation for each event that can cause a mode transition.	These operations must be consistent with the mode transition relation.
8	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.
9	Specify priorities on events if necessary.	The priorities must not be cyclic.
10	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 interface control process must be deterministic. The preconditions of the invoked Z operations must be satisfied.
11	Define the overall control process.	The auxiliary processes must communicate with the interface control process.
12	Define further requirements or environmental assumptions if necessary.	

Table 3.2: Agenda for the active sensors architecture

The events that are used in the auxiliary processes to communicate with the rest of the control component must be defined. For timers, these usually are the events *start_timer*, *stop-_timer* and *timer_elapsed*. This step yields a set of events *Internal_Events*.

Step 3 *Decide on the operational modes of the system and the initial modes.*

This step resembles Step 2 of the passive sensors architecture, with the difference that the modes must be identified in which the controller can be initialized. If possible, a fail-safe mode should be defined. This step yields an enumeration type *MODE* as in the passive sensors architecture.

Step 4 *Set up a mode transition relation, specifying which events relate which modes.*

This transition relation can be defined in Z, or it can be given as a state transition diagram. For each operational mode m and each event e (which can be internal or external), it must be decided on the successor modes that are possible when event e occurs in mode m. It should also be specified what happens when the sensors report an event that normally cannot happen in the respective mode (e.g., if the operational mode assumes a door to be open, but the *open_door* occurs). Hence, the mode transition relation should be made as complete as possible, and a justification should be given, if for a pair (m, e) no successor mode is defined.

Validation Condition 4.1 *All events identified in Step 1 and all modes defined in Step 3 must occur in the transition relation.*

Validation Condition 4.2 *The omission of a successor mode for a mode-event pair must be justified.*

Furthermore, all modes should be necessary.

Validation Condition 4.3 *All modes must be reachable from an initial mode, and there must be no redundant modes.*

No matter if it is directly expressed in Z or given as a state transition diagram, Step 4 yields a relation

$$\mid \; transition : (MODE \times (External\_Events \cup Internal\_Events)) \leftrightarrow MODE$$

to which we will refer in the following.

Step 5 *Model the actuator commands as members of Z types or CSP events.*

Those commands that are not determined by Z operations must be modeled as events. The others are modeled as members of Z types, as in the passive sensors architecture.

Step 6 *Define the internal system state and the initial state.*

This step can be performed in the same way as for the passive sensors architecture. We also have the same validation conditions, plus the condition that the initial internal states must correspond to some initial mode.

Validation Condition 6.1 *The internal system state must be an appropriate approximation of the state of the technical process.*

Validation Condition 6.2 *Each legal state must be safe.*

Validation Condition 6.3 *There must exist legal initial states.*

Validation Condition 6.4 *For each initial internal state, the controller must be in an initial mode.*

Step 7 *Specify a Z operation for each event that can cause a mode transition.*

In contrast to the passive sensors specification, we have several externally available Z operations. They do not correspond to the operational modes but to the events that cause transitions between them, i.e., to the events $e \in \operatorname{ran}(\operatorname{dom} \mathit{transition})$ of the transition function defined in Step 4. If (m_1, e) and (m_2, e) for $m_1 \neq m_2$ are both in $\operatorname{dom} \mathit{transition}$, it suffices to define one operation that treats the occurrence of event e.

As in the passive sensors architecture, it is useful to define a schema *Actuators* that specifies how the actuator commands that are determined by Z operations are derived from the internal state. Since the sensors are no longer necessarily modeled as Z types, it is possible that the Z operations import only the state and the actuator schemas.

Validation Condition 7.1 *These operations must be consistent with the state transition relation.*

The precondition of the operation corresponding to event e must be true for all operational modes m with $(m, e) \in \operatorname{dom} \mathit{transition}$. Furthermore, the successor states defined in the operation must be consistent with the state transition relation.

Step 8 *Define the auxiliary processes identified in Step 2.*

This step can be performed by defining process terms or by specifying predicates that restrict the behavior of the respective processes. Timers can be defined as processes beginning with a *start_timer* event, followed by a *Wait* process and a *timer_elapsed* event. This process can be interrupted at any time by a *stop_timer* event.

$$\begin{aligned} &\mathit{Timer} \mathrel{\widehat=} \mu\, X \bullet \\ &\quad (\mathit{start\_timer} \to \mathit{Wait}\ \mathit{duration};\ \mathit{timer\_elapsed} \to X) \mathbin{\triangle} (\mathit{stop\_timer} \to X) \end{aligned}$$

The auxiliary processes should neither receive external sensor messages nor invoke Z operations or depend on the internal system state. They should exclusively interact with the *InterfaceControl* process, see Figure 3.7.

Validation Condition 8.1 *The alphabets of these processes must not contain external events or events related to the Z part of the specification.*

Step 9 *Specify priorities on events if necessary.*

To determine if priorities are necessary, we have to analyze the state transition diagram. If more than one event can occur at the same time when the system is in a certain operational mode, it must be decided how the system reacts when several events occur simultaneously. Usually, the event with the highest importance for safety will be treated, whereas the other ones will be ignored.

Technically, this means to define derived events and a process *Priority* that relates the original events with the derived ones[5]. If we have a high priority event *high* and a low priority event *low*, then the system will only react to the event *low* if *high* does not occur at the same time. Therefore, an event *excl_low* is derived that occurs at time t exactly when *low* but not *high* occurs at time t:

$$\alpha Priority \mathrel{\widehat{=}} \{high, low, excl\_low\}$$

$$\begin{array}{l}
Priority \ \mathsf{sat} \ \forall\, t : [0, \infty);\ tr : \operatorname{seq} TimedEvents;\ ref : \mathbb{P}\ TimedEvents \mid \\
\qquad\quad (tr, ref) \in timedfailures\, [\![Priority]\!] \bullet \\
\qquad high \ \mathsf{live}\ t\ (tr, ref) \land low \ \mathsf{live}\ t\ (tr, ref) \land \\
\qquad (excl\_low \ \mathsf{live}\ t\ (tr, ref) \Leftrightarrow low \ \mathsf{at}\ t\ (tr, ref) \land \neg\, high \ \mathsf{at}\ t\ (tr, ref))
\end{array}$$

This definition can easily be extended for several events of lower or higher priority, or several degrees of priority. Basically, *Priority* implements a partial order on events.

Validation Condition 9.1 *The priorities must not be cyclic.*

Step 10 *Specify the interface control process.*

The interface control process handles the prioritized events coming from the sensors. According to the internal or external events that occur, it triggers the execution of Z operations and sends events to actuators or auxiliary processes. The syntactic form of the process is an external choice of prioritized events. Each branch of the external choice should be robust, i.e., if the sensors send signals that contradict the internal state of the system, then the system must handle the faulty situation consistently with the state transition relation of Step 4.

[5]This approach to handling priorities was developed by C. Sühl.

The interface control process that is executed after the system is initialized can be defined by

```
$InterfaceControl_{READY} \mathrel{\widehat=} \mu X \bullet$
    $event_1 \rightarrow$
    if $\langle consistency\ condition \rangle$
    then $\langle execute\ event_1 - Z - operation \rangle\ \langle send\ events \rangle; X$
    else $\langle emergency\ shutdown \rangle\ \langle send\ events \rangle; Stop$ fi
$\Box$
    $event_2 \rightarrow \ldots$
$\Box$
    $\ldots$
$\Box$
    $event_n \rightarrow \ldots$
```

This form is only possible if there is a fail-safe state. Then the system can shut down when an inconsistency is detected.

All branches are defined similarly. If a branch consists exclusively of events (which take no time), *Wait* processes must be introduced to model the time one execution of the $InterfaceControl_{READY}$ takes.

To express the consistency conditions, predicates on the current internal system state must be defined in Z.

The following validation conditions relate the results of different steps of the agenda.

Validation Condition 10.1 *All prioritized external events and all internal events must occur as initial events of the branches of the interface control process.*

To check this condition, the results of Steps 1 and 9 have to be considered.

Validation Condition 10.2 *The interface control process must be deterministic.*

This validation condition was already explained in Section 3.3. In Step 9, derived events were defined that guarantee that none of the events that guard the external choice can occur simultaneously.

Validation Condition 10.3 *The preconditions of the invoked Z operations must be satisfied.*

This is guaranteed by appropriate consistency conditions guarding the invocation of the Z operations in the interface control process. Moreover, we must check that in each branch of the interface control process the Z operation corresponding to the respective event, as defined in Step 7, is invoked.

Step 11 *Define the overall control process.*

The process combines the processes defined in Steps 8, 9 and 10. Let $Aux_1, \ldots, Aux_k$ be the auxiliary processes defined in Step 8. Then

$$ControlComponent \mathrel{\widehat=} (InterfaceControl \parallel Aux_1 \parallel \ldots \parallel Aux_k) \setminus Internal\_Events$$

$$\begin{array}{l} InterfaceControl \mathrel{\widehat=} SystemInitExecution \rightarrow \\ \quad (InterfaceControl_{READY} \parallel Priority) \setminus \\ \qquad (\alpha Priority \setminus (External\_Events \cup Internal\_Events)) \end{array}$$

where *SystemInitExecution* establishes an initial internal system state. The internal events are hidden from the environment, and the prioritized events newly introduced in the alphabet of the *Priority* process are hidden from the the other components of the controller (and hence from the environment).

Validation Condition 11.1 *The auxiliary processes must communicate with the interface control process.*

Technically, this means that the alphabets of $(Aux_1 \parallel \ldots \parallel Aux_k)$ and *InterfaceControl* have a non-empty intersection.

Step 12 *Define further requirements or environmental assumptions if necessary.*

Usually, these will be assumptions on the environment and real-time requirements on the execution time of Z operations.

3.4.2 Example: A Gas Burner

The specification of the software controller of a gas burner illustrates the active sensors architecture. The specification is a simplified version of the case study presented by Ravn et al. (Ravn et al., 1993). There are two actuators: The gas actuator controls the emission of gas and receives commands to start or stop emitting gas at arbitrary time instants from the controller. The ignition actuator can ignite escaping gas at arbritrary time instants. There are two sensors. The thermometer sensor measures the temperature in the vicinity of the burner and actively reports to the controller decreases below and increases above certain points, indicating a disappearance or an appearance of the flame, respectively. Users can request the controller to activate or deactivate the burner via the thermostat sensor. These sensor reports cause an immediate reaction of the controller. The gas burner system is safety critical, because a persistent escape of unburned gas or a failure to realize a request to deactivate the flame can lead to major accidents.

Step 1: Model the sensors as CSP events or members of Z types.

According to the informal description of the system, the controller must react to the following external events, the first two coming from the thermostat, the other two generated by the thermometer sensor:

$$External\_Events = \{heat\_off\_request, heat\_on\_request, flame\_on, flame\_off\}$$

Step 2: Decide on auxiliary processes.

For safety reasons, we need two timers. First, a user's request to start the gas burner may be served only after a delay of $DELAY\_DURATION$ time units to ensure that two different attempts to ignite gas are sufficiently separated. Secondly, after the controller has tried to activate the burner by starting the emission of gas and activating the ignition, it must be checked whether or not the ignition was successful, because a persistent escape of unburned gas is dangerous to the environment. The second timer produces an alarm event after $CHECK\_DURATION$ time units, unless a flame is detected and the timer is reset. As a result of this step, we have

$$\begin{aligned} Internal\_Events = \{&start\_timer1, stop\_timer1, timer1\_elapsed, \\ &start\_timer2, stop\_timer2, timer2\_elapsed\} \end{aligned}$$

Steps 3: Decide on the operational modes of the system and the initial modes.

The operational modes are defined by the type

$$MODE ::= IDLE \mid DELAY \mid IGNITION \mid BURNING \mid SHUT\_DOWN$$

The initial mode will be mode $IDLE$.

Step 4: Set up a mode transition relation, specifying which events relate which modes.

The possible mode transitions are illustrated in Figure 3.9.

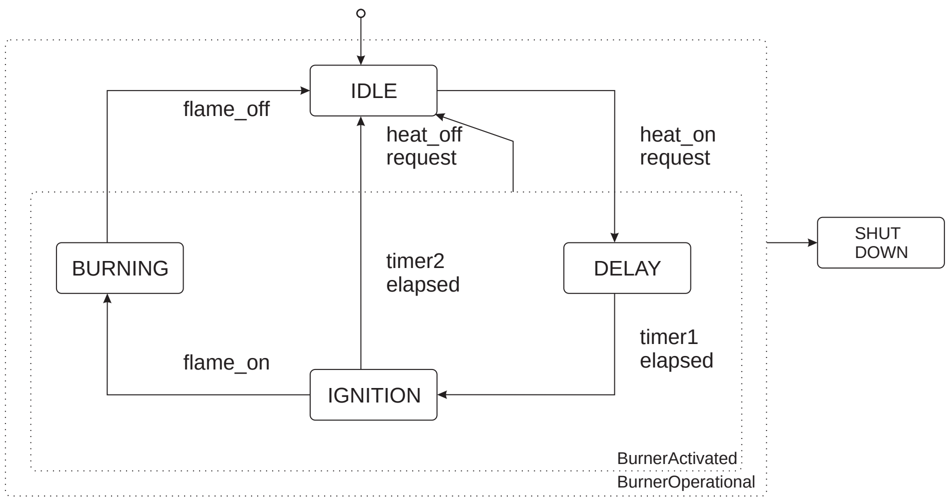


Figure 3.9: Mode transitions of gas burner

In the $IDLE$ mode the controller waits for an activation request without emitting or igniting gas. With an activation request the controller changes to the $DELAY$ mode, waiting for $DELAY\_DURATION$ time units. This delay is realized by a process $Timer1$ identified in Step 2. After changing to mode $IGNITION$, the controller tries to activate the burner by starting the emission of gas and activating the ignition source. $Timer2$ is set. If a flame is detected within $CHECK\_DURATION$ time units, the controller changes to the $BURNING$

mode in which gas further escapes but the source of ignition is switched off. Otherwise the controller returns to the *IDLE* mode. When the flame disappears in the *BURNING* mode, the controller changes to the *IDLE* mode. Furthermore, a request to deactivate the gas burner causes a change to the *IDLE* mode from every other mode with priority. Each event not explicitly shown in the diagram causes the system to enter the *SHUT_DOWN* mode.

The validation conditions 4.1 through 4.3 can easily be checked: All members of *External_Events* and *MODE* occur in Figure 3.9, and for each mode-event pair, a successor mode is defined. Each mode can be reached and is distinguished from the others.

Step 5: Model the actuator commands as members of Z types or CSP events.

The commands to the actuators depend on the operational mode of the gas burners and hence will be determined by Z operations. Therefore, the actuator commands are modeled as members of Z types. Since both commands are binary, one type for both actuators suffices.

$$YES\_NO ::= YES \mid NO$$

Step 6: Define the internal system states and the initial states.

The abstract internal system state of the gas burner controller is specified by the schema *GasBurner*. There is one major system variable *mode* representing the current operational mode. The other system variables *gas*, *ignition*, and *flame* can be deduced from this system variable.

$$\begin{array}{|l}
\multicolumn{1}{l}{GasBurner} \\
\hline
mode : MODE \\
gas, ignition, flame : YES\_NO \\
\hline
gas = YES \Leftrightarrow mode \in \{IGNITION, BURNING\} \\
flame = YES \Leftrightarrow mode = BURNING \\
ignition = YES \Leftrightarrow mode = IGNITION \\
\hline
\end{array}$$

After initialization the controller is in the *IDLE* mode.

$$GasBurnerInit \mathrel{\widehat{=}} [\, GasBurner' \mid mode' = IDLE \,]$$

Only the specifier can assert that condition 6.1 is fulfilled. Since the safety conditions for the gas burner cannot be expressed statically but only with respect to the duration of certain conditions, validation condition 6.2 is vacuously fulfilled (the state at one time instant cannot be unsafe); we will prove a safety property later. Since the initial state is consistent with the state invariant and puts the system into the *IDLE* mode, which was distinguished as the initial mode, validation conditions 6.3 and 6.4 are satisfied.

Step 7: Specify a Z operation for each event that can cause a mode transition.

We start with the definition of the *Actuators* schema. The commands to the actuators can directly be deduced from the current internal system state.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Actuators}} \\
\hline
\mathit{GasBurner}' \\
\mathit{gas}!, \mathit{ign}! : \mathit{YES\_NO} \\
\hline
\mathit{gas}! = \mathit{gas}' \land \mathit{ign}! = \mathit{ignition}' \\
\hline
\end{array}$$

The system operations are straightforward. They follow the state transition diagram of Figure 3.9.

$$\mathit{HeatOnRequest} \mathrel{\widehat=} [\,\Delta \mathit{GasBurner}; \mathit{Actuators} \mid \mathit{mode} = \mathit{IDLE} \land \mathit{mode}' = \mathit{DELAY}\,]$$

$$\begin{aligned}&\mathit{HeatOffRequest} \mathrel{\widehat=} [\,\Delta \mathit{GasBurner}; \mathit{Actuators} \mid \\ &\quad \mathit{mode} \in \{\mathit{DELAY}, \mathit{IGNITION}, \mathit{BURNING}\} \land \mathit{mode}' = \mathit{IDLE}\,]\end{aligned}$$

$$\mathit{Ignition} \mathrel{\widehat=} [\,\Delta \mathit{GasBurner}; \mathit{Actuators} \mid \mathit{mode} = \mathit{DELAY} \land \mathit{mode}' = \mathit{IGNITION}\,]$$

$$\mathit{IgnitionOK} \mathrel{\widehat=} [\,\Delta \mathit{GasBurner}; \mathit{Actuators} \mid \mathit{mode} = \mathit{IGNITION} \land \mathit{mode}' = \mathit{BURNING}\,]$$

$$\mathit{IgnitionFailure} \mathrel{\widehat=} [\,\Delta \mathit{GasBurner}; \mathit{Actuators} \mid \mathit{mode} = \mathit{IGNITION} \land \mathit{mode}' = \mathit{IDLE}\,]$$

$$\mathit{FlameFailure} \mathrel{\widehat=} [\,\Delta \mathit{GasBurner}; \mathit{Actuators} \mid \mathit{mode} = \mathit{BURNING} \land \mathit{mode}' = \mathit{IDLE}\,]$$

$$\mathit{ShutDown} \mathrel{\widehat=} [\,\Delta \mathit{GasBurner}; \mathit{Actuators} \mid \mathit{mode}' = \mathit{SHUT\_DOWN}\,]$$

These mode transitions are consistent with the state transition diagram of Figure 3.9, as required in validation condition 7.1.

Step 8: Define the auxiliary processes identified in Step 2.

We define two timers according to the schematic definition of Step 8, page 58.

$$\begin{aligned}&\alpha \mathit{Timer}1 \mathrel{\widehat=} \{\mathit{start\_timer}1, \mathit{stop\_timer}1, \mathit{timer}1\_\mathit{elapsed}\} \\ &\alpha \mathit{Timer}2 \mathrel{\widehat=} \{\mathit{start\_timer}2, \mathit{stop\_timer}2, \mathit{timer}2\_\mathit{elapsed}\}\end{aligned}$$

$$\begin{aligned}&\mathit{Timer}1 \mathrel{\widehat=} \mu\, X \bullet \\ &\quad (\mathit{start\_timer}1 \rightarrow \mathit{Wait}\ \mathit{DELAY\_DURATION}; \mathit{timer}1\_\mathit{elapsed} \rightarrow X) \\ &\ \triangle\ (\mathit{stop\_timer}1 \rightarrow X)\end{aligned}$$

$$\begin{aligned}&\mathit{Timer}2 \mathrel{\widehat=} \mu\, X \bullet \\ &\quad (\mathit{start\_timer}2 \rightarrow \mathit{Wait}\ \mathit{CHECK\_DURATION}; \mathit{timer}2\_\mathit{elapsed} \rightarrow X) \\ &\ \triangle\ (\mathit{stop\_timer}2 \rightarrow X)\end{aligned}$$

where

$$\begin{array}{|l}
\mathit{DELAY\_DURATION}, \mathit{CHECK\_DURATION} : \mathbb{N}_1 \\
\hline
\mathit{DELAY\_DURATION} > \mathit{CHECK\_DURATION}
\end{array}$$

The alphabets of the timers do not contain any event of the set *External_Events* or any events related to the Z part of the specification. Hence, validation condition 8.1 is fulfilled.

Step 9: Specify priorities on events if necessary.

The request to turn off the flame has priority over all other events. The events *heat_off_request* and *heat_on_request* cannot occur simultaneously, because they are signals of the same sensor. Hence, we get the following *Priority* process:

$$\begin{aligned}\alpha Priority \mathrel{\widehat{=}} \{&heat\_off\_request, flame\_on, flame\_off, timer1\_elapsed, timer2\_elapsed,\\ &excl\_flame\_on, excl\_flame\_off, excl\_timer1\_elapsed, excl\_timer2\_elapsed\}\end{aligned}$$

$$\begin{aligned}
&Priority\ \mathsf{sat}\ \forall\, t : [0, \infty);\ tr : \mathrm{seq}\ TimedEvents;\ ref : \mathbb{P}\ TimedEvents \mid\\
&\qquad (tr, ref) \in timed\,failures\,[\![Priority]\!] \bullet\\
&\quad heat\_off\_request\ \mathsf{live}\ t\ (tr, ref) \land\\
&\quad flame\_on\ \mathsf{live}\ t\ (tr, ref) \land flame\_off\ \mathsf{live}\ t\ (tr, ref) \land\\
&\quad timer1\_elapsed\ \mathsf{live}\ t\ (tr, ref) \land timer2\_elapsed\ \mathsf{live}\ t\ (tr, ref)\\
&\quad \land\\
&\quad (excl\_flame\_on\ \mathsf{live}\ t\ (tr, ref) \Leftrightarrow\\
&\qquad flame\_on\ \mathsf{at}\ t\ (tr, ref) \land \neg\, heat\_off\_request\ \mathsf{at}\ t\ (tr, ref)) \land\\
&\quad (excl\_flame\_off\ \mathsf{live}\ t\ (tr, ref) \Leftrightarrow\\
&\qquad flame\_off\ \mathsf{at}\ t\ (tr, ref) \land \neg\, heat\_off\_request\ \mathsf{at}\ t\ (tr, ref)) \land\\
&\quad (excl\_timer1\_elapsed\ \mathsf{live}\ t\ (tr, ref) \Leftrightarrow\\
&\qquad timer1\_elapsed\ \mathsf{at}\ t\ (tr, ref) \land \neg\, heat\_off\_request\ \mathsf{at}\ t\ (tr, ref)) \land\\
&\quad (excl\_timer2\_elapsed\ \mathsf{live}\ t\ (tr, ref) \Leftrightarrow\\
&\qquad timer2\_elapsed\ \mathsf{at}\ t\ (tr, ref) \land \neg\, heat\_off\_request\ \mathsf{at}\ t\ (tr, ref))
\end{aligned}$$

The priorities defined here are not cyclic, as required by validation condition 9.1.

Step 10: Specify the interface control process

The main control process is specified by the process $InterfaceControl_{READY}$. It follows the schematic expression given previously for Step 10 on page 60, where ϵ is the response time for the technical components.

```
$InterfaceControl_{READY} \mathrel{\widehat{=}} \mu\, X \bullet$
    $heat\_off\_request \to Wait\ \epsilon;$
        if $\neg\, BurnerIsDeactivated$
        then $stop\_timer1 \to stop\_timer2 \to HeatOffRequestExecution$
            $\to ActuatorControl;\ X$
        else $ShutDownExecution \to ActuatorControl;\ Stop$ fi
$\Box$
    $heat\_on\_request \to Wait\ \epsilon;$
        if $BurnerIsDeactivated$
        then $start\_timer1 \to HeatOnRequestExecution \to ActuatorControl;\ X$
        else $ShutDownExecution \to ActuatorControl;\ Stop$ fi
$\Box$
    $excl\_flame\_on \to Wait\ \epsilon;$
        if $IgnitionIsActivated$
        then $stop\_timer2 \to IgnitionOKExecution \to ActuatorControl;\ X$
        else $ShutDownExecution \to ActuatorControl;\ Stop$ fi
$\Box$
```

```
    $excl\_flame\_off \to Wait\ \epsilon;$
        if $FlamePresent$
        then $FlameFailureExecution \to ActuatorControl;\ X$
        else $ShutDownExecution \to ActuatorControl;\ Stop$ fi
$\Box$
    $excl\_timer1\_elapsed \to Wait\ \epsilon;\ start\_timer2 \to IgnitionExecution$
    $\to ActuatorControl;\ X$
$\Box$
    $excl\_timer2\_elapsed \to Wait\ \epsilon;\ IgnitionFailureExecution \to ActuatorControl;\ X$
```

The controller reacts to prioritized events. Predicates are used to check for inconsistencies between the sensor values and the internal system state. To process the events, Z operations are executed and events are sent to the environment of the process. The outputs to the actuators are defined by the process *ActuatorControl*. We still need to define the Z predicates and the process *ActuatorControl*:

$$BurnerIsDeactivated \mathrel{\widehat=} [\, GasBurner \mid mode = IDLE \,]$$
$$BurnerIsActivated \mathrel{\widehat=} [\, GasBurner \mid mode \in \{DELAY, IGNITION, BURNING\} \,]$$
$$IgnitionIsActivated \mathrel{\widehat=} [\, GasBurner \mid ignition = YES \,]$$
$$FlamePresent \mathrel{\widehat=} [\, GasBurner \mid flame = YES \,]$$

```
$ActuatorControl \mathrel{\widehat=}$
    $gas?Gas \to gas\_command!Gas \to Skip$
$\parallel$
    $ignition?Ignition \to ignition\_command!Ignition \to Skip$
```

Conditions 10.1 and 10.2 are satisfied by the definitions of *External_Events*, *Internal_Events* and *Priority*. Validation condition 10.3 is shown by inspecting the predicates and the preconditions of the Z operations that are executed after the consistency predicate has been checked.

Step 11: Define the overall control process

The dynamic behavior of the gas burner controller is defined by the real-time CSP process *GasBurnerControl*. It is an instantiation of the schema given in the agenda on page 61.

$$GasBurnerControl \mathrel{\widehat=} (InterfaceControl \parallel Timer1 \parallel Timer2) \setminus Internal\_Events$$
$$InterfaceControl \mathrel{\widehat=} GasBurnerInitExecution \to (InterfaceControl_{READY} \parallel Priority)$$
$$\quad \setminus \{excl\_flame\_on, excl\_flame\_off, excl\_timer1\_elapsed, excl\_timer2\_elapsed\}$$

Step 12: Define further requirements or environmental assumptions if necessary.

The behavior of the environment of the burner is constrained by the following predicate.

```
$EnvironmentalAssumption \mathrel{\widehat=}$
    $(\forall\, t : [0, \infty);\ tr : \mathrm{seq}\ TimedEvents;\ ref : \mathbb{P}\ TimedEvents \mid$
        $(tr, ref) \in timed\,failures\, [\![ GasBurnerControl ]\!] \bullet$
        $(\forall\, op : Operations \bullet op\ \mathsf{open}\ t\ (tr, ref)) \land$
        $\neg\ (heat\_on\_request\ \mathsf{open}\ t(tr, ref) \land heat\_off\_request\ \mathsf{open}\ t(tr, ref)))$
```

where

$$\begin{aligned}
\mathit{Operations} \mathrel{\widehat{=}} \{&\mathit{HeatOnRequestExecution}, \mathit{HeatOffRequestExecution},\\
&\mathit{IgnitionExecution}, \mathit{IgnitionOKExecution}, \mathit{IgnitionFailureExecution},\\
&\mathit{FlameFailureExecution}, \mathit{GasBurnerInitExecution}\}
\end{aligned}$$

The execution of the system operations must be under exclusive control of the control process. Requests to activate and to deactivate the gas burner must not occur simultaneously.

Proof of a safety constraint

During each interval of *DELAY_DURATION* time units unburned gas may escape for at most $\mathit{CHECK\_DURATION} + \epsilon$ time units. We prove this constraint for our specification.

According to the definition of the state schema *GasBurner*, unburned gas can escape only in the *IGNITION* mode. Hence, within an interval of at most *DELAY_DURATION* time units, the total length of all subintervals that begin with the event *IgnitionExecution* and end with an event from the set of operations (marking the end of the ignition phase) must not exceed $\mathit{CHECK\_DURATION} + \epsilon$ time units.

The following argument has two parts. First, we show that every period within the *IGNITION* mode lasts at most $\mathit{CHECK\_DURATION} + \epsilon$ time units, which is essentially the purpose of the *Timer*2 process. Second, we show that two different periods in the *IGNITION* mode are separated by at least *DELAY_DURATION* time units.

Let t be an arbitrary point in time.

$\mathit{IgnitionExecution}\ \mathsf{at}\ t$

$\vdash \qquad [\mathit{InterfaceControl}_{READY}]$

$\mathit{start\_timer}2\ \mathsf{at}\ t$

$\vdash \qquad [\mathit{Timer}2]$

$(\exists\, t' : (t, t + \mathit{CHECK\_DURATION}] \bullet \mathit{stop\_timer}2\ \mathsf{at}\ t') \lor$
$\mathit{timer}2\_\mathit{elapsed}\ \mathsf{at}\ (t + \mathit{CHECK\_DURATION})$

First case:

$\exists\, t' : (t, t + \mathit{CHECK\_DURATION}] \bullet \mathit{stop\_timer}2\ \mathsf{at}\ t'$

$\vdash \qquad [\mathit{InterfaceControl}_{READY}, \mathit{Priority}]$

$\exists\, t' : (t, t + \mathit{CHECK\_DURATION}] \bullet$
$\quad \mathit{IgnitionOKExecution}\ \mathsf{at}\ t' \lor \mathit{HeatOffRequestExecution}\ \mathsf{at}\ t'$

Second case:

$\mathit{timer}2\_\mathit{elapsed}\ \mathsf{at}\ (t + \mathit{CHECK\_DURATION})$

$\vdash \qquad [\mathit{InterfaceControl}_{READY}, \mathit{Priority}]$

$\mathit{IgnitionFailureExecution}\ \mathsf{at}\ (t + \mathit{CHECK\_DURATION} + \epsilon)$
$\lor\ \mathit{HeatOffRequestExecution}\ \mathsf{at}\ (t + \mathit{CHECK\_DURATION} + \epsilon)$
$\lor\ \mathit{ShutDownExecution}\ \mathsf{at}\ (t + \mathit{CHECK\_DURATION} + \epsilon)$

This argument shows that in all cases, the *IGNITION* mode is left after at most $CHECK\_DURATION + \epsilon$ time units.

To show that two different periods in the *IGNITION* mode are separated by at least *DELAY_DURATION* time units, we consider two arbitrary points in time, $t1$ and $t2$, at which the event *IgnitionExecution* occurred.

$$
\begin{array}{ll}
& t1 < t2 \land IgnitionExecution \text{ at } t1 \land IgnitionExecution \text{ at } t2 \land \\
& (\forall t : (t1, t2) \bullet \neg\, IgnitionExecution \text{ at } t) \\
\vdash & \hfill [\text{possible mode transitions, } InterfaceControl_{READY}] \\
& \exists t : (t1, t2) \bullet start\_timer1 \text{ at } t \land mode = DELAY \qquad (\text{at } t) \\
& (\forall t : (t1, t2) \bullet \neg\, IgnitionExecution \text{ at } t) \\
\vdash & \hfill [Timer1] \\
& \exists t : (t1, t2) \bullet (\neg\, (\exists t' : [t, t + DELAY\_DURATION) \bullet timer1\_elapsed \text{ at } t')) \land \\
& (\forall t : (t1, t2) \bullet \neg\, IgnitionExecution \text{ at } t) \\
\vdash & \hfill [InterfaceControl_{READY}] \\
& \exists t : (t1, t2) \bullet (\neg\, (\exists t' : [t, t + DELAY\_DURATION) \bullet IgnitionExecution \text{ at } t')) \land \\
& (\forall t : (t1, t2) \bullet \neg\, IgnitionExecution \text{ at } t) \\
\vdash & \hfill [\text{predicate logic}] \\
& (\forall t' : (t1, t1 + DELAY\_DURATION) \bullet \neg\, IgnitionExecution \text{ at } t') \\
\vdash & \\
& t2 \geq t1 + DELAY\_DURATION
\end{array}
$$

As a consequence, there is at most one subinterval of $CHECK\_DURATION + \epsilon$ time units during which unburned gas escapes in any interval of at most *DELAY_DURATION* time units. Thus the safety constraint is guaranteed by the software controller.

3.5 Refinement

To make stepwise refinement possible for our combined language, we have to define what it means for a specification of a reactive system consisting of a Z part and a real-time CSP part to be refined by another such combination. To this end, we can make use of the existing refinement notions of Z and real-time CSP.

The essential idea of refinement in Z is that abstract data structures are transformed into more concrete data structures. The relations between abstract and concrete data types have to be formally defined by a so-called *abstraction* relation. For every operation on the abstract system state, a corresponding operation on the concrete system state has to be defined.

The concrete operation must satisfy two conditions: First, if the corresponding abstract operation is applicable to an abstract state (i.e. its precondition is fulfilled) then the corresponding concrete operation must be applicable to all concrete states that are related to the abstract state. Second, if the execution of a concrete operation can result in a certain concrete state then there must exist an abstract state which is a possible result of the execution of the corresponding abstract operation and is related to the concrete state (Spivey, 1992b).

In real-time CSP every semantic model maps a term of the process syntax to a set of possible observations as described in Section 3.2.3. A process term is refined by another process term if each possible behavior of the latter is also a possible behavior of the former.

To refine a combined specification, either the Z part or the real-time CSP part is refined separately. For the Z part, however, the notion of refinement must be strengthened. In the definition of refinement in Z as described above, the refining operation can have a weaker precondition than the refined operation, i.e. it can be applicable to a system state to which the latter is not applicable.

If this were admitted, the refining specification would include such behaviors as possible observations that result from the application of the concrete operation to system states where the abstract operation is not applicable. These behaviors would not be observable in the refined specification. To avoid this violation to the notion of refinement, the definition of operation refinement is adapted in the sense that the precondition of the refining operation must be equivalent to the precondition of the refined operation. With this adaptation every isolated refinement of the Z or the real-time CSP part is a refinement of the combined specification.

3.6 Related Work

The use of model-based languages like Z or VDM (Jones, 1990) in the area of system safety is not uncommon. Several case studies use VDM, e.g. the British government regulations for storing explosives (Mukherjee and Stavridou, 1993), a railway interlocking system (Hansen, 1994), and a water-level monitoring system (Williams, 1994). Mukherjee's and Stavridou's as well as Hansen's work, however, focus on adequately modeling safety requirements, independently of the question of whether software is employed or not. Consequently, they do not discuss issues specific to the construction of safety-critical software.

Jacky (Jacky, 1995) uses Z to define a framework for safety-critical systems that emphasizes safety interlocking. McDermid and Pierce (McDermid and Pierce, 1995) define a graphical notation based on a variant of statecharts (Harel, 1987) that is translated into Z for the purpose of mechanical validation. This notation is used to specify and develop software for programmable logic controllers. Halang and Krämer (Halang and Krämer, 1994) also focus on programmable logic controllers. They describe a development process, from the formalization of requirements to the testing of the constructed program. They use the specification language Obj and the Hoare calculus, and their choice is motivated by the available tool support. The specification language Obj is weaker than our combination of Z and real-time CSP because Obj only allows conditional equations to be stated.

The work presented here is distinguished from these approaches in that it is intended to be used for systems where the exclusive use of model-based or algebraic specification languages does not lead to satisfactory results. The expressive power of these languages does not suffice to specify the behavior of sophisticated real-time systems adequately. Other researchers share our goal to provide more powerful constructs to express behavioral and real-time requirements.

Ravn et al. (Ravn et al., 1993) use the duration calculus to express functional requirements and safety constraints. The duration calculus is a specialized formalism designed to express requirements on the duration of states. These durations are expressed as integrals. In contrast, our approach uses less specialized formalisms that are more easily accessible and more widely used. Weber (Weber, 1996) combines Z and statecharts for purposes similar to ours. Since statecharts are a semi-formal specification technique, the resulting specification is not completely formal. Using a formal language like CSP, however, yields completely formal combined specifications, as shown in Section 3.2.3.

Like our work, Moser's and Melliar-Smith's approach to the formal verification of safety-critical systems (Moser and Melliar-Smith, 1990) comprises the specification, design and implementation phases. They use a reliability model for the processors that execute the program. This enables them to take computer failures into account, an aspect we do not address. On the other hand, their approach does not cover the validation of the top-level specification, an issue that we pay particular attention to.

3.7 Summary

The approach presented in this chapter concentrates on the specification of software for safety-critical applications. It cannot guarantee that a system whose software controller is implemented from a specification developed according to our approach is free from accidents for several reasons: first, our validation criteria cannot rule out errors in the specification completely; second, there may be errors in the implementation; third, there may be errors in the compiler or the operating system that are used to run the software; fourth, nothing can be guaranteed about the hardware. For instance, our method does not take processor failures into account. This last limitation cannot be overcome by means concerning the software alone. Instead, fault tolerance methods like redundancy have to be applied.

We can expect, however, that there are fewer errors in the specification and the implementation of safety-critical software if our approach is applied. With the work presented here, we have provided an elaborate methodology for the formal specification of software for safety-critical applications:

- The system model underlying most of these applications is taken into account by explicitly referring to it in the methodology. It provides a suitable structure and nomenclature to model safety-critical systems.

- Two formal languages are combined according to the needs arising in the development of safety-critical systems. Each of the languages in isolation would not be satisfactory; in combination, however, they provide adequate constructs for the specification of safety-critical software components. Both languages are well-established.

- The combined language is given a common semantics, making combined specifications completely formal and providing a basis for formal proof.

- A software model for the combined use of the two languages is defined, yielding a general framework for the modeling and specification of control components for safety-critical systems.

- This model is further refined into reference architectures that capture frequent designs of safety-critical systems. For each of these architectures, an agenda is given. Safety-related considerations are of particular importance in the agendas.

- The agendas provide detailed guidance, not only for developing specifications of software controllers matching the reference architectures, but also for validating the developed specifications. To complement the application-independent validation criteria of the agendas, we propose to demonstrate safety-related and liveness properties that necessarily are application-dependent.

- The agendas are sufficiently detailed to allow them to be formalized and their application by machine to be supported, as demonstrated in Chapter 8.
- Not only the specification phase but also the later phases in software development are supported: a notion of refinement for combined specifications is defined, and the semi-automatic synthesis of programs for the Z part of the specification is possible, as described in Chapters 6 and 7.

3.8 Further Research

In the future, we intend to improve the technical basis of our approach and further elaborate the methodological part.

Calculus. We want to develop a common calculus for Z and real-time CSP that will allow us to perform formal proofs on and refinements of combined specifications. An implementation of this calculus would provide machine support for discharging validation conditions.

Synthesis for CSP. We want to investigate how program synthesis for the CSP part of a specification can be supported; of special importance are the real-time requirements.

Partial verification. For relatively small systems, a complete formal treatment certainly can be recommended because the control software is relatively simple. The cost for a formal safety proof would be much less than potential damages. For larger systems, however, a complete formal treatment might not be feasible. In this case, one would formalize and prove only selected properties of the system and treat the other requirements with traditional techniques (*partial verification*, (Leveson, 1991)). When this approach is taken, all of the software modules still have to be considered. To reduce cost further, one might exclude those parts of the software from the verification process that can be guaranteed to be of no importance for safety. We want to develop guidance on how to guarantee (i) that a requirement is not safety-critical, and (ii) that a part of the software is not safety-critical.

Formalism independence. We want to show that our approach can also be used with other formalisms than the ones chosen here. We believe that the agendas can easily be adapted. Another formalism for specifying behavior is Petri Nets. We intend to replace real-time CSP by Petri Nets and investigate how the agendas have to be changed.

Other architectures. We want to consider more reference architectures, especially for distributed systems.

Structural aspects of specifications. The combination of Z and real-time CSP does not support modularization very well. Language constructs to modularize specifications are needed.

Chapter 4

Software Design Using Architectural Styles

Architectural styles are a mechanism to make system design knowledge explicit and thus amenable to reuse. They characterize designs in terms of the components of a system and the connectors that enable communication between components (Abowd et al., 1993). An important question in the field of software architectures is how to represent styles in such a way that unambiguous criteria can be stated to decide whether a given design conforms to some style. A second question is how a style representation can help to develop concrete architectures.

Informal circle-and-line drawings have shown their limitations and, today, the need for formal languages to represent software architectures has been recognized. New languages for architectural descriptions have been developed but they are still in a maturing phase and few are provided with tools (Clements, 1996).

In this chapter, we address the questions of representing architectural styles and supporting the development of style instances in three steps: first, we demonstrate that LOTOS (Bolognesi and Brinksma, 1987) is a suitable language to express architectural designs. Second, we contribute to a clarification of the meaning of architectural styles by characterizing such styles as LOTOS patterns. Third, we present agendas to support designers in the development of concrete software architectures.

LOTOS as an Architectural Description Language

LOTOS is a formal description language designed to specify open distributed systems. It consists of a process algebra similar to CSP (Hoare, 1985) or CCS (Milner, 1980), and an algebraic specification language that allows the equational definition of data types. Using LOTOS to express architectural designs has several advantages:

- LOTOS consists of two parts, an algebraic specification language to define data, and a process algebra to define the behavior of a system. Hence, the communication between system components in an architecture can be described using the process algebraic parts of LOTOS, and the algebraic specification language can be used to specify the data transformations that are performed by the system.

- Architectural descriptions in LOTOS are formal and hence have an unambiguous semantics. They can be subject to proofs and analyses.

- The use of LOTOS makes it possible to use existing tools, like CADP (Caesar/Aldebaran Distribution Package) (Fernandez et al., 1992), for analysis and animation of architectures defined with it.
- LOTOS is an ISO standard. The use of a standardized language relieves system designers of the burden to learn an extra architectural description language. These can be quite rich and complex, see e.g. (Luckham et al., 1995).

Style Characterizations

We characterize an architectural style by (i) requirements on the processes specifying the components of a system, (ii) a communication pattern defining its top-level behavior, and (iii) constraints, which provide sufficient conditions for an architectural description to be an instance of the style. These conditions can be checked mechanically.

Our style characterizations clarify the meaning of an architectural style by making its essence explicit.

Design Support

Starting from the style characterizations, we can define agendas for the development of instances of the styles. The validation conditions associated with the agendas refer to the different parts of the style characterizations. Concrete architectures can be developed recursively in such a way that subsystems of a system can again be instances of styles. Furthermore, the architectural descriptions can be analyzed and animated using existing tools. No new tools need to be developed.

In Section 4.1, we explain the general approach we take to express architectural designs in LOTOS and styles as LOTOS patterns. The approach is illustrated by characterizing three architectural styles: repository (Section 4.2), pipe/filter (Section 4.3) and event-action (Section 4.4). In Section 4.5, we present three different designs for a robot, following the three architectural styles. The tool CADP is used to compare the alternative designs. The concluding sections discuss our approach in the context of related work and give directions for further research. This chapter extends the work presented in (Heisel and Lévy, 1997).

4.1 Expressing Architectural Designs and Styles with LOTOS

Architectural designs and styles are usually described in terms of *components* and *connectors* between them. In our approach to represent architectural styles, system components are modeled as *processes*. These processes usually perform some data transformation. They can consist of another architectural description, representing the design of a subsystem. In this way, hierarchical composition of architectures is possible. Connectors are not separate syntactic entities but are realized by the kind of communication that takes place between the component processes.

A LOTOS specification is composed of interacting processes. They can be parameterized by abstract data types. A process can exchange typed values with another process and call functions to transform data. Communication between processes in LOTOS is synchronous, i.e. two processes must participate in a common action at the same time. *Gates* are used to synchronize processes and to exchange data. To synchronize, two processes must contain an action via the same gate `g`. To exchange data, one of them must contain an action `g ? v: t` which reads a value `v` of type `t` via gate `g`. The other process must contain an action `g ! exp` that writes a value `exp` of type `t` onto the gate `g`. It is also possible to read or write more than one value in the same action.

We use this kind of communication by rendez-vous to describe the communication between the components of a system. In LOTOS, data are described using abstract data types with conditional equations and an initial semantics. Abstract data types are used for describing process parameters and values exchanged by the processes via gates.

Each architectural description must be a valid LOTOS expression, regardless of the style it belongs to. It consists of two parts. The *behavior* part describes the overall behavior of the architecture, i.e. the interaction of its parts. The *local definitions* part contains the definition of the processes involved in the behavior part and the necessary definitions of abstract data types. The syntactic structure of an architectural description is

`behaviour` ***`behav_expr`*** `where` ***`local_def_list`***

LOTOS *patterns* are obtained from LOTOS by abstraction, i.e. by replacing concrete LOTOS expressions by metavariables. Both parts of an architectural description, i.e., ***`behav_expr`*** as well as ***`local_def_list`***, can be subject to abstraction. In the following, concrete LOTOS expressions are set in `teletype`, and metavariables are set in ***`italics teletype`***.

A characterization of an architectural style consists of

- **component characteristics**, which describe properties of the involved component processes;
- a **communication pattern**, which characterizes the top-level behavior of the system by a LOTOS pattern;
- **constraints**, which, when fulfilled, guarantee that an architectural description conforms to the style.

Such representations make style characteristics explicit and form the basis for the definition of agendas. In the following, we present characterizations of three architectural styles.

4.2 Repository Style

Garlan and Shaw (Garlan and Shaw, 1993; Shaw and Garlan, 1996) describe the repository style as follows:

> " In a repository style there are two distinct kinds of components: a central data structure represents the current state, and a collection of independent components operate on the central data store."

In a first step, we characterize the style with LOTOS patterns. Then we define an agenda that gives guidelines for the development of concrete architectures conforming to the repository style.

4.2.1 Style Characterization

In our modeling, we suppose that the central data structure – the *shared memory* – contains data accessible via indices, which select parts of the stored data.

Component Characteristics

We consider three kinds of components operating on the shared memory: components that only read (part of) the memory, components that only change the memory, and components that do both. There is no interaction between components: they behave independently and only communicate with the repository and the environment.

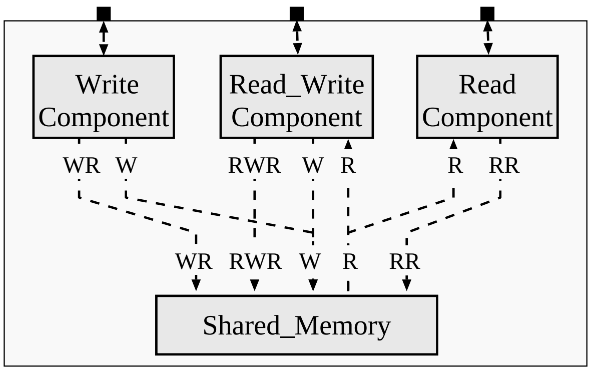


Figure 4.1: General view of repository style architecture

The three kinds of components are illustrated in Figure 4.1, where the system interface is represented by black squares. If a component wants to change the shared memory, it sends the message `WR` (write request). This causes the shared memory to set a lock. Only then can the new value be passed, using the gate `W` (write). If a component wants to read the shared memory, it sends the message `RR` (read request). If no lock is set the value is passed via the gate `R` (read). It may happen that a value to be written into the shared memory depends on a value that was read previously. In this case, no other write operation should be allowed between the read and the write action. For this purpose, the message `RWR` (read/write request) is used.

Each process sending a request must also send a unique identification. This prevents other processes from accessing the memory during a transaction. The process implementing the shared memory is defined as follows:

```
process Shared_Memory [RR, R, WR, W, RWR]
                        (sm: shared_memory, is_locked: BOOL, for_whom: id): noexit :=
      [ is_locked = false ]
        -> (  RR ? who: id;
              R  ? who: id ? j : index ;
              R  ! who     ! get(sm, j);
              Shared_Memory [RR, R, WR, W, RWR] (sm, false, for_nobody)
           []
              WR ? who: id;
              Shared_Memory [RR, R, WR, W, RWR] (sm, true, who)
           []
```

```
             RWR ? who: id;
             Shared_Memory [RR, R, WR, W, RWR] (sm, true, who) )
  [] [ is_locked = true ]
       -> (  W ? who: id ? j : index ? nv: value [who=for_whom] ;
             Shared_Memory [RR, R, WR, W, RWR] (store(sm, j, nv), false, for_nobody)
          []
             R ? who: id ? j : index ;
             R ! who ! get(sm, j);
             W ? who: id ? nv: value [who=for_whom];
             Shared_Memory [RR, R, WR, W, RWR] (store(sm, j, nv), false, for_nobody))
endproc
```

The process Shared_Memory has the gates RR, R, WR, W, RWR and the parameters sm representing the memory, is_locked and for_whom. It does not terminate, as indicated by the keyword noexit. If the lock is not set, either a read request can be served, or the lock can be set because of a write or read/write request. If the lock is set, either a new value and an index are read via the gate W, or the part of the repository stored under index j is output on gate R, followed by reading a new value via gate W. These actions can only take place if the same process that sent the request participates in them, as expressed by the guard [who=for_whom]. The new value of the shared memory becomes the new parameter of the process, and the lock is reset[1]. The constant *for_nobody* indicates that access to the shared memory is not reserved for a particular process.

The process Shared_Memory is the same for all instantiations of the repository architecture, except for the type of information to be stored, and the types used as indices and for the identification of components. The type *shared_memory*, which represents the shared memory, has to be defined algebraically. We need an initial value *init*, a function *store* changing the shared memory, and a function *get* reading it. The type definition should follow the schema

```
   type   SHARED_MEMORY
   is  INDEX, VALUE
       sorts
        shared_memory
       opns
        init  : -> shared_memory
        store : shared_memory, index, value -> shared_memory
        get  : shared_memory, index -> value
       eqns
          forall sm: shared_memory, j1, j2: index, v1:  value
          ofsort value
            get(store(sm, j1, v1), j1) = v1;
            not(equiv(j1, j2)) => get(store(sm, j1, v1), j2) = get (sm, j2);
            get(init, j1) = ...;
   endtype
```

where *equiv* is the equality on indices.

[1]To keep our presentation concise, we do not allow parallel write or read/write actions on different parts of the shared memory, i.e. on different indices. The definition of such an optimization is straightforward.

The types ***id***, ***index*** and ***value*** of the values that can be stored under an index are also defined algebraically. For the type ***id***, we need an initial value ***for_nobody***, as explained previously.

Each repository architecture consists of a process `Shared_Memory` as defined above and an arbitrary number of independent components. Each of these is either a *read process*, a *write process* or a *read/write process*.

A read process does not use the gates `WR, W, RWR` and contains an arbitrary (positive) number of read behaviors but neither write nor read/write behaviors. A read behavior is defined by the pattern

```
RR ! me ;
R  ! me ! ind ;
R  ? who: id ? v : value [who = me]
```

where *me* is the identification of the process and ***ind*** is the index to be read.

A write process does not use the gates `RR, R, RWR` and contains an arbitrary (positive) number of write behaviors but neither read nor read/write behaviors. A write behavior is defined by the pattern

```
WR ! me ;
W  ! me ! ind ! v
```

where *v* is the new value to be stored under index ***ind***.

A read/write process may use three behavioral patterns. It contains at least one read/write behavior, or read as well as write behaviors. A read/write behavior is defined by the pattern

```
RWR ! me ;
R  ! me ! ind ;
R  ? who: id ? v : value [who = me]
```

followed, in all the branches of the process, by writing access to the shared memory according to the pattern

```
W  ! me ! ind ! nv
```

for the same index ***ind*** and a value ***nv***. This condition can be syntactically decided as follows: Let *R/W_Comp* be a process whose behavior part contains as a sub-expression the first part of the read/write pattern, composed via "`;`" with another behavior expression *B*. We now define a predicate *write_pattern* that is a sufficient condition for *B* to contain the second part of the pattern in each of its branches. This is done by a case distinction of the syntactic form of *B*:

exit: *write_pattern*(`exit(...)`) = *false*

stop: *write_pattern*(`stop`) = *false*

action prefix: $B = g\,\alpha_1 \ldots \alpha_n$ `;` *B'*:
If $g\,\alpha_1 \ldots \alpha_n =$ `W ! me ! ind ! nv` then *write_pattern*(*B*) = *true*;
else *write_pattern*(*B*) = *write_pattern*(*B'*)

choice or disabling: *B* = *B1* `[]` *B2* or *B* = *B1* `[>` *B2*:
write_pattern(*B*) = *write_pattern*(*B1*) $\wedge$ *write_pattern*(*B2*)

parallel or sequential composition: *B* = *B1 op B2*
with $op \in \{$`|[g1, ..., gn]|`, `|||`, `||`, `>>`$\}$:
write_pattern(*B*) = *write_pattern*(*B1*) ∨ *write_pattern*(*B2*)

hiding: *B* = `hide` *g1, ..., gn* `in` *B'*:
if `W` $\in$ {*g1, ..., gn*} then *write_pattern*(*B*) = *false*;
else *write_pattern*(*B*) = *write_pattern*(*B'*)

process instantiation: *B* = *P*`[`*g1, ..., gn*`](...)`:
if `W` $\notin$ {*g1, ..., gn*} or *P* invokes ***R/W_Comp*** then *write_pattern*(*B*) = *false*;
else *write_pattern*(*B*) = *write_pattern*(*PB*`[`*g1/g1', ...,gn/gn'*`]`),
where *PB*`[`*g1/g1', ...,gn/gn'*`]` is the behavior expression that defines process *P*, where the formal gates *g1', ..., gn'* are replaced by the actual gates *g1, ..., gn*.

Note that the condition for process instantiation may be too restrictive if *P* first contains a write pattern and only then invokes ***R/W_Comp***. Hence, the predicate *write_pattern* is only a sufficient condition that the process following the first part of a read/write pattern contains the second part of the pattern in each of its branches.

The occurrence of the patterns for read and write processes and the first part of a read/write pattern in a process definition can easily be checked syntactically; hence, we have shown that we can define sufficient conditions for the component processes to fulfill the component characteristics, and that these conditions can be checked syntactically.

Communication Pattern

The communication between the shared memory and the independent components is expressed by the following pattern, where for better readability we use "..." instead of an inductive definition:

```
hide   RR, R, WR, W, RWR in
Shared_Memory [RR, R, WR, W, RWR] (init of shared_memory, false, for_nobody)
       |[ RR, R, WR, W, RWR ]|
   (Component_1[gate_list_1]
       |||
   ...
       |||
    Component_n [gate_list_n] )
```

All components behave independently of each other (the operator `|||` involves no communication at all). For every ***Component_i***, its ***gate_list_i*** must contain the gates `RR` and `R` if it is a read process and `WR` and `W` if it is a write process. A read/write process may contain `RR, R` as well as `WR, W`, or `RWR`, `R` and `W`. The repository and the independent components must synchronize on these gates, as expressed by the synchronization list `|[ RR, R, WR, W, RWR ]|`. The `hide` clause hides communications via the gates `RR, R, WR, W, RWR` from the environment.

Constraints

Constraints are expressed in terms of the two parts of an architectural description, namely ***behav_expr*** and ***local_def_list***, as introduced in Section 4.1. For the repository style, we have the constraints that the ***behav_expr*** must conform to the communication pattern given above, and that each process occurring in ***behav_expr***, except `Shared_Memory`, must be a read, a write or a read/write process as defined in the process characteristics.

4.2.2 Agendas

The steps that lead to a repository architecture are summarized in Table 4.1. They have to be performed in the given order, as indicated in Figure 4.2. The guidelines for the development of architectural descriptions are mostly captured in the style characterization. Hence, the agenda contains only a few steps.

No.	Step	Validation Conditions
1	Define the types ***shared_memory***, ***id***, ***index*** and ***value***.	The type ***shared_memory*** must be defined according to the schema given in Section 4.2.1. The type ***id*** must contain a constant ***for_nobody***.
2	Define the component processes.	Each component process must be either a read, a write, or a read/write process.
3	Assemble the overall architectural description according to the communication pattern of the repository style.	The processes must communicate with the shared memory according to their being a read, write or read/write process, as described in the communication pattern.

Table 4.1: Agenda for the repository architectural style

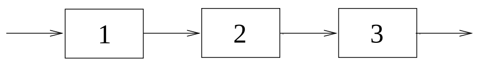


Figure 4.2: Dependencies of steps for developing repository architectures

Step 1 *Define the types* ***shared_memory***, ***id***, ***index*** *and* ***value***.

First, we must decide what kind of information is to be stored in the repository and how it can be accessed.

Step 2 *Define the component processes.*

For defining a single component process, we give an agenda in Table 4.2. The steps of this agenda must be performed in the given order. The second step is very general. This makes it possible to define one component of a repository architecture as a subsystem according to some other style. The agenda shown in Table 4.2 can be applied repeatedly.

Step 3 *Assemble the overall architectural description according to the communication pattern.*

This step can be performed in a routine way. It only must be guaranteed that the right gates are used for the communication between the repository and the component processes.

No.	Step	Validation Conditions
1	Decide if the component is a read, write, or read/write process.	
2	Define the component as a process.	The process definition must contain the patterns for the chosen kind of component.

Table 4.2: Agenda to develop components for a repository architecture

4.3 Pipe/Filter Style

The characteristics of the pipe/filter style are the following (Garlan and Shaw, 1993; Shaw and Garlan, 1996):

> "In a pipe and filter style each component has a set of inputs and a set of outputs. A component reads streams of data on its inputs and produces streams of data on its outputs, [...] Components are termed "filters". The connectors of this style serve as conduits for the streams, transmitting outputs of one filter to inputs of another. Hence connectors are termed "pipes". [...] filters must be independent entities: in particular, they should not share state with other filters. "

Garlan et al. (Garlan et al., 1996) additionally state the topological constraint that pipes are directional and that at most one pipe can be connected to a given "port" of a filter. Figure 4.3 shows an example of a pipe/filter architecture. As can be seen, a filter (in this case `Filter_3`) may have several incoming and several outgoing pipes. Cycles are also allowed, see (Garlan and Shaw, 1993).

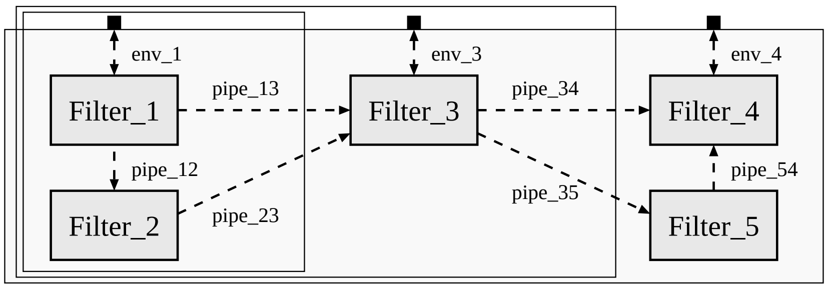


Figure 4.3: A pipe/filter architecture

4.3.1 Style Characterization

In the LOTOS characterization of this style, a pipe between two filters is a synchronous communication via some gate.

Component Characteristics

A filter is modeled by a process that takes its inputs from the incoming pipes, transforms them according to its task, and delivers the results via the outgoing pipes. Communication with the environment is also possible.

Hence, a component of this style is not characterized by some specific behavior but by its gates. These are divided into the lists *in_pipe_list*, *out_pipe_list* and *env_gate_list*. A filter process does not write on gates of its *in_pipe_list* and does not read from gates of its *out_pipe_list*.

Communication Pattern

Two filters communicate via their common pipes. For example, the filters `Filter_1` and `Filter_2` in the smallest box of the architecture shown in Figure 4.3 exhibit the communication behavior

```
Filter_1[env_1, pipe_12, pipe_13]
   |[pipe_12]|
Filter_2[pipe_12, pipe_23]
```

When adding the third filter `Filter_3` synchronizing with the previous system via the pipes `pipe_13` and `pipe_23` connecting it to `Filter_1` and `Filter_2`, the following behavior is obtained:

```
( Filter_1 [env_1, pipe_12, pipe_13]
      |[pipe_12]|
  Filter_2 [pipe_12, pipe_23]  )
      |[pipe_13, pipe_23]|
  Filter_3 [env_3, pipe_13, pipe_23, pipe_34, pipe_35]
```

Hence, the general communication pattern of a pipe/filter architecture has the form

```
hide pipe_list_1, pipe_list_2, ... pipe_list_n-1 in
   ( ... ((Filter_1[gate_list_1] |[pipe_list_1]| Filter_2[gate_list_2])
                |[pipe_list_2]| Filter_3[gate_list_3])
            ...
                |[pipe_list_n-1]|
                Filter_n[gate_list_n])
```

We have used "..." again instead of an inductive definition for better comprehensibility of the pattern.

Constraints

Again, we state the constraints in terms of the top-level behavior *behav_expr* and the *local-_def_list*:

- All synchronization lists (i.e. the values given to *pipe_list_1*, ..., *pipe_list_n-1*) occurring in *behav_expr* are disjoint. This means that a pipe connects only two filters.
- Each gate occurring in some synchronization list of *behav_expr* occurs exactly twice in the gates of the processes *Filter_1*, ..., *Filter_n* defined in *local_def_list*. This means that a pipe cannot be re-used as an external gate.

- Each of the processes that occur in ***behav_expr*** must conform to the characterization given above. The gates of a process representing pipes are exactly the ones that occur in some synchronization list ***pipe_list_1***, ..., ***pipe_list_n-1***. The direction of the pipe can be determined from the process definition.

Note that, in our definition, pipes and filters have no buffers like in (Abowd et al., 1993), because – according to the synchronous communication of LOTOS – no data can be lost. The buffered version – which we consider to be closer to an implementation – could also be expressed in LOTOS.

4.3.2 Agendas

The agenda to develop a software architecture according to the pipe/filter style is shown in Table 4.3. The steps must be performed in the order given in the agenda.

No.	Step	Validation Conditions
1	Define the filters one by one.	Each filter must fulfill the conditions stated in the component characteristics part of the style characterization.
2	Assemble the filters according to the pattern given in the communication pattern part of the style characterization.	The architectural description must fulfill the constraints stated in the constraints part of the style characterization.

Table 4.3: Agenda for the pipe/filter architectural style

Again, we define an agenda that helps to define a single filter process and that can be applied repeatedly to perform Step 1 of the agenda for the pipe/filter style in Table 4.4. The dependencies of the steps are shown in Figure 4.4.

No.	Step	Validation Conditions
1	Decide on the pipes that connects the filter with other filters.	
2	Decide on the gates of the filter with the environment.	
3	Define the filter as a process.	The process must fulfill the conditions stated in the component characteristics part.

Table 4.4: Agenda to develop components for a pipe/filter architecture

4.4 Event-Action Style

According to Krishnamurthy and Rosenblum (Krishnamurthy and Rosenblum, 1995),

> “An event-action system is a software system in which events occurring in the environment of the system trigger actions in response to the events. The triggered actions may generate other events, which trigger actions, and so on.“

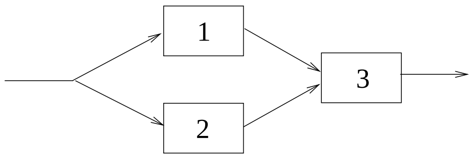


Figure 4.4: Dependencies of steps for developing a filter

Garlan and Shaw (Garlan and Shaw, 1993) mention that " The main invariant in this style is that announcers of events do not know which components will be affected by those events."

4.4.1 Style Characterization

An event architecture consists of components that react to events. When an event has happened, actions are carried out and other events may be sent. An event manager is responsible for distributing all events that have occurred to all components that have to react to that event. Figure 4.5 shows an example of an event architecture.

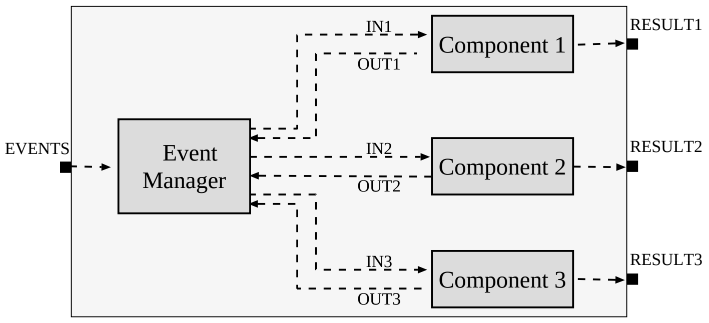


Figure 4.5: An event-action architecture

Component Characteristics

The event manager has the following form[2]:

```
process Event_Manager [EVENTS, IN_1, OUT_1,... IN_n, OUT_n] : func :=
   receive_event
   >> accept  e: event in
   trigger_actions
endproc
```

This definition consists of two processes, *receive_event* and *trigger_actions*, which are separated by >>. The accept clause means that an event *e* is passed from the process *receive_event* (via exit clauses) to the process *trigger_actions*. The event manager may terminate (*func* = exit) or not (*func* = noexit). The data type *event* must be defined algebraically. It can be structured to allow the handling of complex events.

[2]In this definition, there is only one gate *EVENTS*. The pattern can easily be generalized to allow for several external gates.

In the process *receive_event*, the event manager reads incoming events, either from the environment via the gate *EVENTS* or from some other component via some gate *OUT_i*. The process *receive_event* must contain the following pattern:

```
   EVENTS ? e: event; exit(e)
[] OUT_1  ? e: event; exit(e)
[] ...
[] OUT_n  ? e: event; exit(e)
```

In the process *trigger_actions*, the event manager distributes the event to the various components that define the actions to be taken in reaction to the event, according to some predicates *p_j*. The process *trigger_actions* must contain the pattern

```
      [p_1(e)]  -> IN_1,1 ! e ; ... IN_1,n1 ! e ;
                   Event_Manager [EVENTS, IN_1, OUT_1, ... IN_n, OUT_n]
   [] ...
   [] [p_k(e)]  -> IN_k,1 ! e ; ... IN_k,nk ! e ;
                   Event_Manager [EVENTS, IN_1, OUT_1, ... IN_n, OUT_n]
```

Each event-action architecture consists of a process `Event_Manager` as described above and an arbitrary number of independent components. Each such component *Component_i* has a gate *IN_i* and contains an action

```
IN_i ? e: event
```

If the component generates events, it has a gate *OUT_i*, which is used to send events to the event manager. In this case, the process behavior contains actions of the form:

```
OUT_i ! e
```

The process does not write on *IN_i* and does not read from *OUT_i*.

Communication Pattern

The communication between the event manager and the independent components takes place according to the pattern

```
hide IN_1, OUT_1, ... IN_n, OUT_n in
    Event_Manager [EVENTS, IN_1, OUT_1, ... IN_n, OUT_n]
       |[IN_1, OUT_1, ... IN_n, OUT_n]|
  ( Component_1[IN_1, OUT_1, env_gate_list_1]
       |||
    ...
       |||
    Component_n [IN_n, OUT_n, env_gate_list_n] )
```

Constraints

The *behav_expr* and *local_def_list* making up the architectural description of an event-action system must satisfy the following constraints:

- *behav_expr* must conform to the communication pattern given above.
- Each of the processes that occurs in *behav_expr*, except `Event_Manager`, must conform to the description given in the component characterization.

4.4.2 Agendas

Like a repository architecture, an event-action architecture consists of a distinguished component – the event manager – and a number of other components that perform independently of each other and communicate only with the event manager and the environment. Hence, the agendas are similar to the agendas for the repository style. The top-level agenda is given in Table 4.5. It must be processed in the given order.

No.	Step	Validation Conditions
1	Define the type *event*.	
2	Define pairs, consisting of a predicate on the type *event* and a process defining the corresponding action.	Each action process must communicate with the event manager and define the reaction to the events that fulfill the defined predicate.
3	Define the process `Event_Manager` and assemble the overall architectural description according to the communication pattern.	The definition of the event manager must conform to the pattern given in the component characteristics, and it must be consistent with Step 2.

Table 4.5: Agenda for the event-action architectural style

The validation condition of Step 3 means that for each pair (*pred*, *Component*) defined in Step 2, the process *Component* must be notified exactly of all events satisfying the predicate *pred*, i.e., the definition of the process `Event_Manager` must contain the expression

```
[pred(e)]  ->  ... IN_k ! e ; ... ;
               Event_Manager [EVENTS, IN_1, OUT_1, ... IN_n, OUT_n]
```

where *IN_k* is the gate used by process *Component* to communicate with the event manager.

The agenda to develop one component is given in Table 4.6. The steps must be performed in the given order. Again, the generality of the second step makes it possible to define one component of a repository architecture as a subsystem according to some other style.

4.5 Example: Design of a Robot

We illustrate the development of instances of architectural styles by designing a robot. This robot can make the movements shown in Figure 4.6: it can advance by moving its right or its left leg; it can stand still; and it can smile or not. In the following, we develop three

No.	Step	Validation Conditions
1	Decide on the events to be treated.	
2	Define the action to be taken as a process.	The process definition conforms to the component characteristics given in the style characterization.

Table 4.6: Agenda to develop components for an event-action architecture

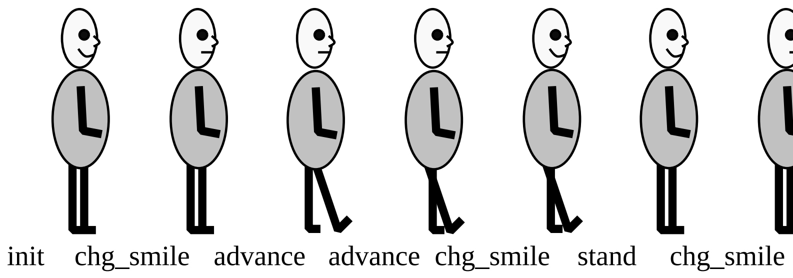


Figure 4.6: The movements of the robot

alternative designs, one for each style presented previously. These three designs use the same robot definition.

The robot can be modeled as an automaton with three states: `standing`, `left_up` and `right_up` as shown in Figure 4.7. To each state a boolean value is associated indicating whether the robot is smiling or not. The initial state is standing and smiling.

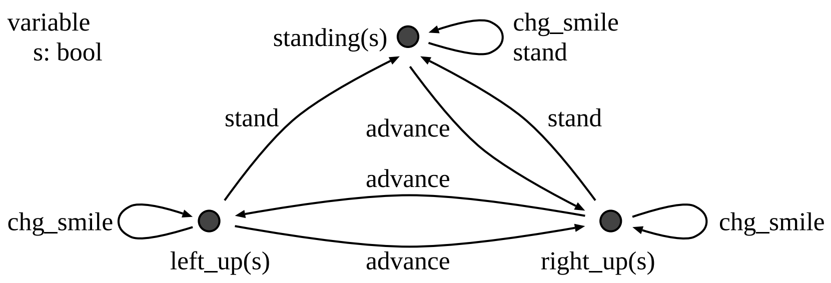


Figure 4.7: The robot automaton

The robot is defined by the abstract data type `robot` where the states are defined as constants and the movements as transitions from one state to another, except for smiling, which is defined by a boolean value: `true` for smiling. For each state a predicate is defined deciding if the robot is in this state.

```
type   ROBOT
is     BOOLEAN
     sorts robot
     opns
       standing    : bool -> robot
       left_up     : bool -> robot
       right_up    : bool -> robot
       is_standing, is_left_up, is_right_up    : robot -> bool
```

```
          the_smile                                  : robot -> bool
          init                                       :       -> robot
          stand, advance, chg_smile                  : robot -> robot
        eqns
         forall roro: robot,  s : bool
         ofsort  bool
           is_standing(standing(s)) = true;
           is_standing(left_up(s))  = false;
           is_standing(right_up(s)) = false;
           is_left_up(standing(s))  = false;
           is_left_up(left_up(s))   = true;
           is_left_up(right_up(s))  = false;
           is_right_up(standing(s)) = false;
           is_right_up(left_up(s))  = false;
           is_right_up(right_up(s)) = true;
           the_smile(standing(s))   = s;
           the_smile(left_up(s))    = s;
           the_smile(right_up(s))   = s;
         ofsort  robot
           init = standing(true);
           stand(roro) = standing(the_smile(roro));
           advance (standing(s))   = right_up (s);
           advance (left_up(s))    = right_up (s);
           advance (right_up(s))   = left_up (s);
           chg_smile (standing(s)) = standing(not(s));
           chg_smile (left_up(s))  = left_up (not(s));
           chg_smile (right_up(s)) = right_up (not(s));
   endtype
```

The movements are defined by the type `mvt` with three constants `m_stand, m_advance` and `m_chg_smile`:

```
   type mvt
   is   boolean
        sorts mvt
        opns
          m_stand, m_advance, m_chg_smile : -> mvt
   endtype
```

The robot will be asked to execute several movements collected in a list. This list is defined by an abstract data type `m_list` with a constant `empty`, a function `add` adding an element to the end of the list, a function `rm_first` removing the first element of a list, a function `first` selecting the first element of a list, and a predicate `is_empty`. A constant `init_list` is used to define the list of movements initially given to the robot.

```
   type M_LIST
   is   BOOLEAN, MVT
        sorts  m_list
        opns
          empty_list :             -> m_list
          add        : m_list, mvt -> m_list
          rm_first   : m_list      -> m_list
          is_empty   : m_list      -> bool
          first      : m_list      -> mvt
          init_list  :             -> m_list
        eqns
          forall m:mvt, lm: m_list
          ofsort m_list
            rm_first (add (empty_list, m)) = empty_list;
            not (is_empty (lm)) => rm_first (add (lm, m)) =
                                       add (rm_first (lm), m);
            init_list = add(add(add(add(add(add(empty_list,
                        m_chg_smile), m_advance), m_advance), m_chg_smile),
                        m_stand), m_chg_smile);

          ofsort Bool
            is_empty (empty_list)  = true;
            is_empty (add (lm, m)) = false;
          ofsort mvt
            first (add (empty_list, m)) = m;
            not (is_empty (lm)) => first (add (lm, m)) = first (lm);
   endtype
```

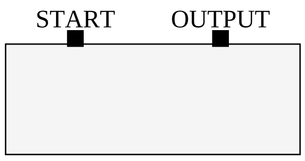


Figure 4.8: Interface of robot design

We have the same interface shown in Figure 4.8 for all architectures. The input is the initial state of the robot together with the movements to be performed. Therefore, a data type *value* must be defined as the Cartesian product of the two types `robot` and `m_list`. Its constructor function is called `make`, and the two selector functions are called `the_robot` and `the_list`. This data type will be defined in Section 4.5.1. The gate `START` is used to start the simulation, yielding in the following top-level behavior:

```
START  !make(init of robot,init_list); exit
  |[START]|
(  behav_expr  )
```

The three architectures will result in different definitions of *behav_expr* and the associated *local_def_list*.

4.5.1 The robot design using the repository style

Our first robot design follows the repository style. Step 1 of the agenda shown in Table 4.1 tells us to first give type specifications for *shared_memory*, *id*, *index* and *value*.

The shared memory is to hold the current state of the robot and the list of movements to be executed, i.e. items of type *value*. As we just have one value (one robot and its list of movements) to be stored in the shared memory, we only need one index, which we call `index1`. This yields the following type definitions.

```
type   SHARED_MEMORY
is  INDEX, VALUE
    sorts
     shared_memory
    opns
     init  : -> shared_memory
     store : shared_memory, index, value -> shared_memory
     get   : shared_memory, index -> value
    eqns
       forall sm: shared_memory, j1, j2: index, v1:  value
       ofsort value
         get(store(sm, j1, v1), j1) = v1;
         not(equiv(j1, j2)) => get(store(sm, j1, v1), j2) = get (sm, j2);
         get(init, j1) = make(init of robot, empty);
endtype

type   INDEX
is  BOOLEAN
    sorts
       index
    opns
     index1  : -> index
     equiv: index,index -> bool
    eqns
       forall j1, j2: index
       ofsort bool
     equiv(index1,index1) = true;
endtype

type   IDENTIFIER
is  BOOLEAN
    sorts
       id
    opns
       for_nobody   : -> id
       id_init_sm   : -> id
       id_stand     : -> id
       id_advance   : -> id
       id_chg_smile : -> id
endtype

type   VALUE
is  ROBOT, M_LIST
    sorts
       value
    opns
     make     : robot, m_list -> value
     the_robot: value -> robot
     the_list : value -> m_list
    eqns
       forall roro: robot, ml: m_list
       ofsort robot
         the_robot(make(roro, ml)) = roro;
       ofsort m_list
         the_list(make(roro, ml)) = ml;
endtype
```

It can easily be checked that the validation conditions associated with Step 1 of the agenda are fulfilled: The type `shared_memory` is defined according to the schema given in Section 4.2.1, and the type `id` contains a constant `for_nobody`.

Next, as required in Step 2 of the agenda, we define the component processes of the architecture. First, we need a write process `Init_sm` that writes the initial state and the initial list of movements into the shared memory.

Furthermore, we need three independent components `Stand`, `Chg_Smile` and `Advance` to execute the corresponding movements. These components try in parallel to access the shared memory in order to execute the movement they are responsible for. Therefore, they all are read/write processes. Each of them first reads the list of movements, denoted by the variable `ml`. If the first movement is the one it is responsible for, the movement is executed, the robot state changed (variable `roro`) and the rest of the movement list is written back in the shared memory. If the movement cannot be executed by the component that has been granted access, it writes back the unchanged state in order to unlock the shared memory. This architecture is illustrated in Figure 4.9.

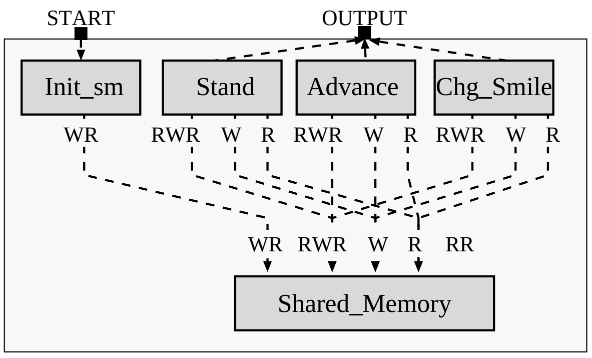


Figure 4.9: The repository architecture for the robot

The component processes are defined as follows.

```
   process  Init_sm [START,  W, WR] : exit
   :=
      START ? vv: value;
      WR ! id_init_sm;
      W  ! id_init_sm ! index1 ! vv;
      exit
   endproc

   process  Stand [OUTPUT, R, W, RWR] : exit
   :=
           RWR ! id_stand;
           R ! id_stand ! index1 ;
           R ? for_whom: id ? v: value [for_whom=id_stand];
           (let   ml: m_list = the_list(v),
            roro: robot = the_robot(v)
         in  [is_empty(ml)= true ] -> W ! id_stand ! v ; exit
         []  [is_empty(ml)= false] ->
             (   [first(ml) equal m_stand = true ]
                   -> OUTPUT ! make(stand(roro), rm_first(ml)) ;
                      W  ! id_stand ! make(stand(roro), rm_first(ml)) ;
                      Stand [OUTPUT, R, W, RWR]
            []  [first(ml) equal m_stand = false]
                   -> W ! id_stand ! v ;
                      Stand [OUTPUT, R, W, RWR]
             ))
endproc

process  Advance [OUTPUT, R, W, RWR] : exit
:=
        RWR ! id_advance;
        R ! id_advance ! index1 ;
        R ? for_whom: id ? v: value [for_whom=id_advance];
        (let   ml: m_list = the_list(v),
              roro: robot = the_robot(v)
         in  [is_empty(ml)= true ] -> W ! id_advance ! v ; exit
         []  [is_empty(ml)= false] ->
          (  [first(ml) equal m_advance = true ]
                 -> OUTPUT ! make(advance(roro), rm_first(ml)) ;
                    W  ! id_advance ! make(advance(roro), rm_first(ml)) ;
                    Advance [OUTPUT, R, W, RWR]
         []  [first(ml) equal m_advance = false]
                 -> W ! id_advance ! v ;
                    Advance [OUTPUT, R, W, RWR]
         ))
endproc

process  Chg_Smile [OUTPUT, R, W, RWR] : exit
:=
        RWR ! id_chg_smile;
        R ! id_chg_smile ! index1 ;
        R ? for_whom: id ? v: value [for_whom=id_chg_smile];
        (let   ml: m_list = the_list(v),
              roro: robot = the_robot(v)
         in  [is_empty(ml)= true ] -> W ! id_chg_smile ! v ; exit
         []  [is_empty(ml)= false] ->
          (  [first(ml) equal m_chg_smile = true ]
                 -> OUTPUT ! make(chg_smile(roro), rm_first(ml)) ;
                    W  ! id_chg_smile ! make(chg_smile(roro), rm_first(ml)) ;
                    Chg_Smile [OUTPUT, R, W, RWR]
         []  [first(ml) equal m_chg_smile = false]
                 -> W ! id_chg_smile ! v ;
                    Chg_Smile [OUTPUT, R, W, RWR]
         ))
endproc
```

The process `Init_sm` conforms to the pattern for write processes. The other component processes conform to the pattern for read/write processes. They first lock the shared memory and read its content. Subsequently, in each branch of the choice, a write action is performed. Hence, the validation conditions associated with Steps 2 of the agendas shown in Tables 4.1 and 4.2 are fulfilled. This concludes Step 2 of the agenda for the repository architecture. Step 3 yields the overall behavior of the repository robot and the concrete definition of the process SM:

```
   START  !make(init of robot,init_list); exit
   |[START]|
   (
   hide   RR, R, WR, W, RWR in

      SM [RR, R, WR, W, RWR] (init of shared_memory, false, for_nobody )
      |[ RR, R, WR, W, RWR ]|
      (
        Init_sm [START, W, WR]
        |||
        Stand [OUTPUT, R, W, RWR]
        |||
        Chg_Smile [OUTPUT, R, W, RWR]
        |||
        Advance [OUTPUT, R, W, RWR]
      )
    )
where

   process SM [RR, R, WR, W, RWR] (sm:shared_memory, is_locked:BOOL, for_whom: id)
                                                                      : noexit
   :=
      [ is_locked = false ]
        -> ( RR ? who: id;
             R  ? who: id ? j1 : index ;
             R  ! who     ! get(sm, j1);
             SM [RR, R, WR, W, RWR] (sm, false, for_nobody)
             []
             WR ? who: id;
             SM [RR, R, WR, W, RWR] (sm, true, who)
             []
             RWR ? who: id;
             SM [RR, R, WR, W, RWR] (sm, true, who) )
      []
      [ is_locked = true ]
        -> ( W ? who: id ? j1 : index ? nv: value [who=for_whom] ;
             SM [RR, R, WR, W, RWR] (store(sm, j1, nv), false, for_nobody)
             []
             R ? who: id ? j1 : index ;
             R ! who ! get(sm, j1);
             W ? who: id ? nv: value [who=for_whom];
             SM [RR, R, WR, W, RWR] (store(sm, j1, nv), false, for_nobody) )
   endproc
```

Again, syntactic checks show that the validation condition associated with Step 3 of the agenda of Table 4.1 is fulfilled: The communication of the component processes takes place as described in the communication pattern described in Section 4.2.1.

This architecture has the disadvantage that the system implementation must guarantee fairness, i.e. each component must be given the chance to access the shared memory. Otherwise, an infinite number of unsuccessful accesses is possible, and the system does not terminate.

4.5.2 The robot design using the pipe/filter style

In the pipe/filter modeling, we can make sure that each component is given the possibility to execute its movement if required. The idea is to have a line of filters. Each filter inspects the movement list. If it can execute the movement, it does so and hands the new robot state and the new movement list to the next filter. Otherwise, it passes on the unchanged data. Again, we need an initializing component, called here `Init_pf`. The architecture is shown in Figure 4.10. It also shows the gates that are needed for the processes.

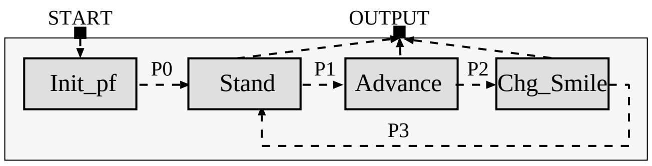


Figure 4.10: The pipe/filter architecture of the robot

For each process, we now proceed according to the agenda of Table 4.4. Process `Init_pf` is connected via pipe `P0` with the other filters and via `START` with the environment. Its definition is

```
process  Init_pf [START, P0] : exit
:=  START ? vv: value;
    P0 ! vv ;
    exit
endproc
```

For the other processes, we proceed analogously and get the definitions

```
process Stand [P0, P1, P3, OUTPUT] : exit
:=
    (P0 ? vv: value; exit (vv) )
    []
    (P3 ? vv: value; exit (vv) )
    >> accept v : value in
    (let ml: m_list = the_list(v), roro: robot = the_robot(v)
     in [is_empty(ml)= true ] -> (exit)
     [] [is_empty(ml)= false] ->
           (  [first(ml) equal m_stand = true ]
                 ->   OUTPUT ! make(stand(roro), rm_first(ml)) ;
                      P1 ! make(stand(roro), rm_first(ml)) ;
                      Stand [P0, P1, P3, OUTPUT]
            [] [first(ml) equal m_stand = false]
                 ->   P1 ! v ;
                      Stand [P0, P1, P3, OUTPUT]   ))
endproc
```

```
process Advance [P1, P2, OUTPUT] : exit
:=
    P1 ? v: value;
    (let ml: m_list = the_list(v), roro: robot = the_robot(v)
     in [is_empty(ml)= true ] -> (exit)
     [] [is_empty(ml)= false] ->
           (  [first(ml) equal m_advance = true ]
                 ->   OUTPUT ! make(advance(roro), rm_first(ml)) ;
                      P2 ! make(advance(roro), rm_first(ml)) ;
                      Advance [P1, P2, OUTPUT]
            [] [first(ml) equal m_advance = false]
                 ->   P2 ! v ;
                      Advance [P1, P2, OUTPUT]   ))
endproc


process Chg_Smile [P2, P3, OUTPUT] : exit
:=
    P2 ? v: value;
    (let ml: m_list = the_list(v), roro: robot = the_robot(v)
     in [is_empty(ml)= true ] -> (exit)
     [] [is_empty(ml)= false] ->
           (  [first(ml) equal m_chg_smile = true ]
                 ->   OUTPUT ! make(chg_smile(roro), rm_first(ml)) ;
                      P3 ! make(chg_smile(roro), rm_first(ml)) ;
                      Chg_Smile [P2, P3, OUTPUT]
            [] [first(ml) equal m_chg_smile  = false]
                 ->   P3 ! v ;
                      Chg_Smile [P2, P3, OUTPUT]   ))
endproc
```

This concludes Step 1 of the agenda given in Table 4.3. Each of the processes conforms to the component characteristics of Section 4.3.1. According to the style characterization, the overall behavior of the process is

```
hide P0, P1, P2, P3 in
    (  Init_pf [START, P0]
          |[ P0 ]|
       Stand [P0, P1, P3, OUTPUT]
          |[ P1, P3 ]|
       Advance [P1, P2, OUTPUT]
          |[ P2 ]|
       Chg_Smile [P2, P3, OUTPUT]   )
```

This result of Step 2 fulfills the constraints stated in Section 4.3.1: The synchronization lists `[ P0 ]`, `[ P1, P3 ]` and `[ P2 ]` are pairwise disjoint. Gate `P0` occurs exactly in the gate lists of the processes `Init_pf` and `Stand`. Gate `P1` occurs exactly in the gate lists of the processes `Stand` and `Advance`. Gate `P2` occurs exactly in the gate lists of the processes `Advance` and `Chg_Smile`. Finally, gate `P3` occurs exactly in the gate lists of the processes `Stand` and `Chg_Smile`. Pipe `P0` has the direction shown in Figure 4.10 because `Init_pf` only writes on it and `Stand` only reads from it. For the other pipes, the conditions that the gates of a process representing pipes are exactly the ones that occur in some synchronization list and that the direction of the pipe can be determined from the process definition can be checked analogously.

This solution is better than the repository architecture because it always terminates. It is not ideal, however, because each component must inspect the data, even if it cannot process them.

4.5.3 The robot design using the event-action style

The event-action style can be used to overcome the disadvantages of the previous two architectures. The event manager inspects the movement list and passes on the data only to the component that can process them. Events are items of type *value*. The initial state of the robot and the movement list are given to the event manager. An initialization component is not required. This architecture is shown in Figure 4.11.

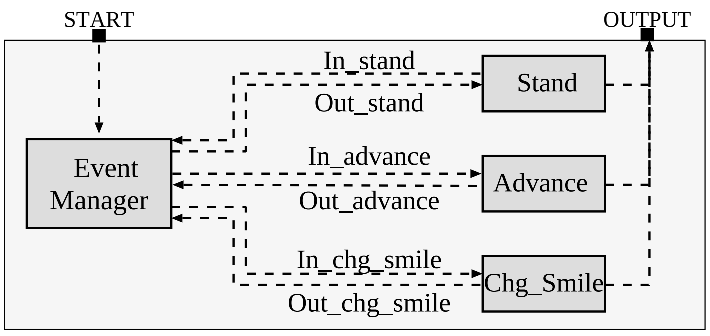


Figure 4.11: The event-action architecture for the robot

Step 1 of the agenda for the event-action style shown in Table 4.5 has already been performed in Section 4.5.1. According to the agenda of Table 4.6, we decide that the action `Stand` must be invoked for all events `v` where `the_list(v)` is a list that starts with the movement `m_stand`. For the actions `Advance` and `Chg_Smile` we have analogous predicates. These independent components communicate with the event manager via the gates `In_stand`, `In_advance` and `In_chg_smile`, respectively. They pass their result on to the event manager for further processing. Their definition is

```
process  Stand [OUTPUT, In_stand, Out_stand] : noexit
:=
        In_stand ? v: value;
        (let ml: m_list = the_list(v),
           roro: robot = the_robot(v)
        in OUTPUT ! make(stand(roro), rm_first(ml));
           Out_stand ! make(stand(roro), rm_first(ml));
           Stand [OUTPUT, In_stand, Out_stand]
        )
endproc

process  Advance [OUTPUT, In_advance, Out_advance] : noexit
:=
        In_advance ? v: value;
         (let ml: m_list = the_list(v),
           roro: robot = the_robot(v)
        in OUTPUT ! make(advance(roro), rm_first(ml));
           Out_advance ! make(advance(roro), rm_first(ml));
           Advance [OUTPUT, In_advance, Out_advance]
        )
endproc

process  Chg_Smile [OUTPUT, In_chg_smile, Out_chg_smile] : noexit
:=
        In_chg_smile ? v: value;
         (let ml: m_list = the_list(v),
           roro: robot = the_robot(v)
        in OUTPUT ! make(chg_smile(roro), rm_first(ml));
           Out_chg_smile ! make(chg_smile(roro), rm_first(ml));
           Chg_Smile [OUTPUT, In_chg_smile, Out_chg_smile]
        )
endproc
```

Each definition fulfills the validation condition of Step 2 of the agenda given in Table 4.6 by using appropriate gates to receive and send events. It remains to perform Step 3 of the agenda given in Table 4.5. The event manager is defined as follows.

```
process Event_Manager [START, In_stand, Out_stand, In_chg_smile,
                          Out_chg_smile, In_advance, Out_advance] : exit :=
      START         ? v: value; exit(v)
   [] Out_stand     ? v: value; exit(v)
   [] Out_advance   ? v: value; exit(v)
   [] Out_chg_smile ? v: value; exit(v)
   >> accept v: value in
      (let ml: m_list = the_list(v), roro: robot = the_robot(v)
       in [is_empty(ml)= true ] -> (exit)
       []([is_empty(ml)= false] ->
             (   [first(ml) = m_stand]
                   -> In_stand ! v ;
                      Event_Manager [START, In_stand, Out_stand,
                      In_chg_smile, Out_chg_smile, In_advance, Out_advance]
              [] [first(ml) = m_advance]
                   -> In_advance ! v ;
                      Event_Manager [START, In_stand, Out_stand,
                      In_chg_smile, Out_chg_smile, In_advance, Out_advance]
              [] [first(ml) = m_chg_smile]
                   -> In_chg_smile ! v ;
                      Event_Manager [START, In_stand, Out_stand,
                      In_chg_smile, Out_chg_smile, In_advance, Out_advance]  )))
endproc
```

The event manager contains the patterns required in Section 4.4.1. It invokes the actions exactly under the conditions stated defined in Step 2 of the agenda of Table 4.5: when the first movement to be executed is `m_stand`, then the event/value is passed via gate `In_stand` to the process `Stand`, and similarly for the other movements.

In accordance with the event-action style characterization, we have the following overall behavior:

```
  hide In_stand, Out_stand, In_chg_smile, Out_chg_smile, In_advance, Out_advance in
       Event_Manager [START, In_stand, Out_stand, In_chg_smile,
                      Out_chg_smile, In_advance, Out_advance]
         |[In_stand, Out_stand, In_chg_smile, Out_chg_smile, In_advance, Out_advance]|
     ( Stand [OUTPUT, In_stand, Out_stand]
         |||
       Advance [OUTPUT, In_advance, Out_advance]
         |||
       Chg_Smile [OUTPUT, In_chg_smile, Out_chg_smile] )
```

Note that the components executing the movements are much simpler now than in the other architectures.

4.5.4 Comparing the three designs with Aldebaran

Under the assumption of fairness for the repository solution, all three LOTOS specifications exhibit the same behavior to the environment, i.e. a user cannot distinguish implementations of the three architectures. This can be shown using CADP (Caesar/Aldebaran Distribution Package) (Fernandez et al., 1992). The tool generates the same following automata minimized with respect to safety equivalence (Fernandez, 1989) (i.e. internal transitions are not considered) for all the three architectures, where we use the movement list shown in Figure 4.6.

```
des (0,7,8)
(0,"START !MAKE (STANDING (TRUE),
                 ADD (ADD (ADD (ADD (ADD (ADD (EMPTY, M_CHG_SMILE),
                 M_ADVANCE), M_ADVANCE), M_CHG_SMILE), M_STAND), M_CHG_SMILE))",
    2)
(2,"OUTPUT !MAKE (STANDING (FALSE),
                 ADD (ADD (ADD (ADD (ADD (EMPTY,
                 M_ADVANCE), M_ADVANCE), M_CHG_SMILE), M_STAND), M_CHG_SMILE))",
    3)
(3,"OUTPUT !MAKE (RIGHT_UP (FALSE),
                 ADD (ADD (ADD (ADD (EMPTY,
                 M_ADVANCE), M_CHG_SMILE), M_STAND), M_CHG_SMILE))",
    4)
(4,"OUTPUT !MAKE (LEFT_UP (FALSE),
                 ADD (ADD (ADD (EMPTY, M_CHG_SMILE), M_STAND), M_CHG_SMILE))",
    5)
(5,"OUTPUT !MAKE (LEFT_UP (TRUE),
                 ADD (ADD (EMPTY, M_STAND), M_CHG_SMILE))",
    6)
(6,"OUTPUT !MAKE (STANDING (TRUE),
                 ADD (EMPTY, M_CHG_SMILE))",
    7)
(7,"OUTPUT !MAKE (STANDING (FALSE),
                 EMPTY)",
    1)
```

This labeled transition system has to be interpreted as follows: the initial state is 0, there are 8 states and 7 transitions. State 1 is a sink state that is entered when the list of movements is empty. The first transition is from State 0 to State 2, and it is performed when the automaton outputs the message STANDING (FALSE) and the rest of the movement list. This means that during this transition the robot has moved to the state where it still stands but is no longer smiling. This corresponds to the first movement of the list, M_CHG_SMILE, starting from the initial state, standing and smiling.

The labeled transition systems, minimized with respect to safety equivalence, are the same for the three architectures:

```
% aldebaran -sequ robot_repository robot_pipe_filter
TRUE
% aldebaran -sequ robot_repository robot_event
TRUE
% aldebaran -sequ robot_pipe_filter robot_event
TRUE
```

Stepwise execution of the three alternative architectures is also possible. This shows that existing LOTOS tools can help to animate and compare architectural descriptions, thus providing valuable support for their validation.

4.6 Related Work

This work is not the first to formally characterize architectural styles or to use a process algebra to specify the behavioral aspects of software architectures. Abowd, Allen and Garlan (Abowd et al., 1993) use the specification language Z to formally define architectural styles. Concrete designs, however, are described in a different language. Thus, there is no direct way from a style definition to an instance of the style. Fiadeiro and Maibaum (Fiadeiro and Maibaum, 1996) conceive architectural styles as well as architectural description languages as categories. Their work is language-independent and aims more at a categorical foundation of software architecture than detailed guidance for designers.

Allan and Garlan (Allan and Garlan, 1994) use CSP to formalize architectural connection. In their approach, *connectors* are defined as processes. In contrast to our work where *components* are modeled as processes, this yields several de-centralized behaviors in one architectural description instead of one central behavioral description characterizing the whole system, as proposed in this work. Moriconi and Qian (Moriconi and Qian, 1994) use CSP to show that an architectural description is a correct refinement of another. Both of these approaches are not concerned with architectural styles but with architectural descriptions in general. These need not conform to any style.

4.7 Summary

Considering the different style characterizations given in this chapter, we notice that there are two styles (repository and event-action) that contain a distinguished component (Shared-_Memory and Event_Manager, respectively). This results in a relatively detailed characterization of the other components of the architecture because one can state requirements concerning the communication of the other components with the distinguished one. Further constraints are not necessary. In contrast, the pipe/filter style does not have a distinguished component. This allows only for a weak characterization of the components, but leads to non-trivial constraints concerning the communication between the different components.

Formal descriptions of architectural styles and concrete architectural designs are important because only architectural descriptions with a formal semantics allows us to precisely answer the questions stated by Clements (Clements, 1996) on software architecture: What are the components? How do they behave? What do the connections mean?

In particular, the results of this chapter are:

- We have shown that LOTOS is a language suitable to express individual architectures, and that LOTOS patterns in combination with constraints are suitable to characterize architectural styles.
- The style characterizations provide a semantic foundation of architectural styles. Furthermore, they yield sufficient conditions for a given concrete architectural description to conform to the style.
- Agendas that are based on the style characterizations support the development of instances of the styles.
- The formal nature of the architectural descriptions and the availability of tools makes it possible to formally analyze and to animate the architectural descriptions. In our example, we have demonstrated how different designs can be compared.
- Our approach to expressing architectural descriptions allows for hierarchical composition of such descriptions.
- Substyles of given architectural styles can be defined by adding further constraints or adding further detail to the patterns of a style characterization.

Of the "hot research areas" identified by Garlan, Allen and Ockerbloom (Garlan et al., 1995), our work addresses *architectural description*, *formal underpinnings* and *role of tools and environments*. This is achieved using only a single formalism (and patterns of it) and existing tools.

4.8 Further Research

The work presented here forms the basis for future work in several directions:

Architecture refinement. A notion of architecture refinement should be defined, based on the notion of behavioral equivalence in LOTOS.

Other styles. Besides the three styles characterized here, other architectural styles, e.g. client-server, are of importance. To make our approach more broadly applicable, these should also be characterized and provided with agendas.

Alternatives for the algebraic specification language. The example has shown that the data type parts of the architectural descriptions are somewhat lengthy. Case studies should be performed to investigate what the architectural descriptions of more realistic systems look like. If appropriate, one could consider replacing the algebraic specification language of LOTOS by Z or another more powerful language.

Part II

MACHINE SUPPORT

Chapter 5

Strategies – A Generic Knowledge Representation Mechanism for Software Development Activities

All efforts to automate software engineering activities and to reuse previously gained experience must be based on representations of the knowledge possessed by software engineers, which are easily implementable on machines, and complemented by some process model describing how to make use of the represented knowledge.

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

Strategies are used to describe possible steps that may be taken during the development of an artifact of the software engineering process. A strategy might, for example specify how to decompose a system design to guarantee a particular property, how to conduct a data refinement, or how to implement a particular class of algorithms. This is the kind of knowledge often described in text books or the agendas presented in the first part of this work. The ability to decide which strategy may successfully be applied in a particular situation, on the other hand, typically requires human intuition and a deep understanding of the problem under consideration. While heuristics for selecting strategies are hardly mechanizable, strategies themselves can, in fact, be implemented.

The basic idea underlying strategies is to conceive software engineering activities as problem solving processes in which, for each given development problem, an *acceptable* solution has to be constructed. The notion of acceptability captures semantic requirements that the developed products must fulfill. The notion of strategy is *generic*. Its generic parameters are the notions of problems, solutions, and acceptability. This means that strategies can be used to formalize a variety of software development activities.

In problem solving with strategies, problems are solved by reducing them to a number of subproblems, which are in turn solved by applying strategies. The problem solving process terminates when the generated subproblems are so simple that they can be solved directly.

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

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

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

- We give a modular representation of strategies. This representation maps without difficulty to encapsulation mechanisms of modern programming languages.
- An abstract problem solving algorithm describes how development activities with strategies can be carried out by machine (where appropriate, user interaction will be necessary).
- The parts of the problem solving algorithm in which user interaction can be replaced by automatic procedures are clearly identified, making stepwise automation possible.
- A generic system architecture provides detailed concepts for implementing support systems for strategy-based problem solving.

We formally define strategies, strategicals, strategy modules, and the abstract problem solving algorithm in the language Z (Spivey, 1992b). This provides precise definitions of these notions, and thus supports reasoning about strategies.

The rest of the chapter is organized as follows. We present a formal definition of strategies in the specification language Z in Section 5.1. Section 5.2 introduces strategicals that can be used to define more powerful strategies from simpler ones. Steps toward an implementation of strategies are taken in Section 5.3. The system architecture described in Section 5.4 takes user needs into account. In Section 5.5, we compare strategy-based problem solving with tactical theorem proving and other related work. Finally, we summarize in Section 5.6 and give directions for further research in Section 5.7. The results of this chapter are based on the publications (Heisel, 1994; Heisel et al., 1995b; Heisel et al., 1995a; Heisel, 1996c)

5.1 Formal Definition of Strategies

As discussed earlier, strategies are used to describe possible steps that can be taken during the development of an artifact of the software engineering process. Strategies are based on the reduction of problems to subproblems from whose solutions the strategy synthesizes a solution to the original problem. The solutions to subproblems are obtained by strategy applications as well. Finally, the strategy tests if the synthesized solution to the original problem is acceptable according to some pre-defined notion of acceptability. In general, the subproblems generated by a strategy are neither independent of one another nor of solutions to other subproblems. Hence, the order in which the various subproblems can be set up and solved is restricted.

We first define the general notion of *relation* and then introduce specific relations called *constituting relations*. Strategies are then defined as sets of constituting relations, which relate problems to the subproblems needed to solve them, and relate their ultimate solutions to the solutions of those subproblems. The formal definition of a strategy is expressed in the specification language Z (Spivey, 1992b).

5.1.1 Definition of Database Relations

Since, in the context of strategies, it is convenient to refer to the subproblems and their solutions by *names*, our definition of strategies is based on the the notion of a relation, as used in the theory of relational databases (Kanellakis, 1990). In this setting, relations are sets of tuples. A tuple is a mapping from a set of *attributes* to the 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 the domains of attributes with the domains of relations as typically used in Z, we introduce the type *Value* to denote attribute values.

$$[\mathit{Attribute}, \mathit{Value}]$$

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

$$\mathit{tuple} : \mathbb{P}(\mathit{Attribute} \twoheadrightarrow\!\!\!\!\!+ \; \mathit{Value})$$

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

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

$$\mathit{scheme} == (\lambda\, r : \mathit{relation} \bullet \bigcup\{t : r \bullet \operatorname{dom} t\})$$

The usual notions of domain restriction and domain subtraction for relations are also needed for the relations used in database theory.

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

$$(\_⩤_r\_) == (\lambda\, \mathit{attrs} : \mathbb{F}\ \mathit{Attribute};\ r : \mathit{relation} \bullet \{t : r \bullet \mathit{attrs} ⩤ t\})$$

Here, $\lhd$ restricts the domain of a relation to its left argument, and $⩤$ subtracts its left argument from the domain of the relation. The operator $\mathbb{F}\, S$ denotes the set of final subsets of S.

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

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

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

The join operation is associative and commutative, and so can be extended to finite sets of relations.

$$\bowtie : \mathbb{F}\ relation \longrightarrow relation$$

$$\begin{aligned}
&\forall rels : \mathbb{F}\ relation \bullet \\
&\quad (rels = \varnothing \land \bowtie rels = \varnothing) \\
&\quad \lor \\
&\quad (\exists r : relation; rels' : \mathbb{F}\ relation \mid r \notin rels' \land rels = \{r\} \cup rels' \bullet \\
&\qquad \bowtie rels = r \bowtie (\bowtie rels'))
\end{aligned}$$

5.1.2 Problems, Solutions, Acceptability

Problems and solutions are generic parameters for strategies. The sets *Problem* and *Solution* are defined to be subsets of *Value*. Acceptability is a relation between problems and solutions.

$$Problem, Solution : \mathbb{P}\ Value$$

$$\_acceptable\_for\_ : Solution \leftrightarrow Problem$$

The sets *ProblemAttribute* and *SolutionAttribute* are countable, disjoint subsets of *Attribute*.

$$\begin{aligned}
&ProblemAttribute : \mathbb{P}\ Attribute \\
&SolutionAttribute : \mathbb{P}\ Attribute
\end{aligned}$$

$$\begin{aligned}
&ProblemAttribute \cap SolutionAttribute = \varnothing \\
&ProblemAttribute \rightarrowtail\!\!\!\!\twoheadrightarrow \mathbb{N} \neq \varnothing \\
&SolutionAttribute \rightarrowtail\!\!\!\!\twoheadrightarrow \mathbb{N} \neq \varnothing
\end{aligned}$$

We use the distinguished attributes *P_init* and *S_final* to refer to the initial problem and its final solution. Moreover, we assume a bijective correspondence *cor* between problem and solution attributes.

$$\begin{aligned}
&P\_init : ProblemAttribute \\
&S\_final : SolutionAttribute \\
&cor : ProblemAttribute \rightarrowtail\!\!\!\!\twoheadrightarrow SolutionAttribute
\end{aligned}$$

$$cor\ P\_init = S\_final$$

5.1.3 Constituting Relations

Each strategy will be defined to be a set of *constituting relations*, representing the dependencies between the subproblems generated by that strategy from any given problem. Their schemes consist of arbitrary attributes for problems and solutions, and are divided into *input attributes* and *output attributes*. Constituting relations restrict the values of output attributes, in relation to given values of the input attributes. Thus, they determine orderings on subproblems that must be respected during the problem solving process.

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

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

Given some relation, it will often be necessary to refer the problem or solution attributes of its scheme.

$$\mathit{problem\_attrs} == (\lambda\, r : \mathit{relation} \bullet \mathit{scheme}\ r \cap \mathit{ProblemAttribute})$$

$$\mathit{solution\_attrs} == (\lambda\, r : \mathit{relation} \bullet \mathit{scheme}\ r \cap \mathit{SolutionAttribute})$$

$$\mathit{subprs} == (\lambda\, r : \mathit{relation} \bullet \mathit{problem\_attrs}\ r \setminus \{\mathit{P\_init}\})$$

$$\mathit{partsols} == (\lambda\, r : \mathit{relation} \bullet \mathit{solution\_attrs}\ r \setminus \{\mathit{S\_final}\})$$

The functions *subprs* and *scheme* are also for sets of relations:

$$\begin{array}{|l}
\mathit{subprs}_s : \mathbb{F}\, \mathit{relation} \longrightarrow \mathbb{P}\, \mathit{ProblemAttribute} \\
\mathit{scheme}_s : \mathbb{F}\, \mathit{relation} \longrightarrow \mathbb{P}\, \mathit{Attribute} \\
\hline
\forall\, \mathit{crs} : \mathbb{F}\, \mathit{relation};\ t : \mathit{tuple} \bullet \\
\qquad \mathit{subprs}_s\ \mathit{crs} = \mathit{subprs}(\bowtie \mathit{crs}) \land \\
\qquad \mathit{scheme}_s\ \mathit{crs} = \mathit{scheme}(\bowtie \mathit{crs})
\end{array}$$

It is now possible to define dependency relations on constituting relations. One constituting relation directly depends on another such relation if one of its input attributes is an output attribute of the other relation; in this way, each set of constituting relations determines a direct dependency relation. The depending constituting relation is considered “larger” than the one on which it depends. For any given set *crs* of constituting relations, one *dependency relation* determined by *crs* is defined to be the transitive closure of the direct dependency relation it determines.

$$\begin{array}{|l}
\_ \sqsubset_d \_ : \mathit{const\_rel} \leftrightarrow \mathit{const\_rel} \\
\sqsubset : \mathbb{P}\, \mathit{const\_rel} \longrightarrow (\mathit{const\_rel} \leftrightarrow \mathit{const\_rel}) \\
\hline
\forall\, \mathit{cr}_1, \mathit{cr}_2 : \mathit{const\_rel};\ \mathit{crs} : \mathbb{P}\, \mathit{const\_rel} \bullet \\
\qquad ((\mathit{cr}_1 \sqsubset_d \mathit{cr}_2 \Leftrightarrow \mathit{OA}\ \mathit{cr}_1 \cap \mathit{IA}\ \mathit{cr}_2 \neq \emptyset) \land \\
\qquad (\sqsubset (\mathit{crs}) = \{\mathit{cr}_1, \mathit{cr}_2 : \mathit{crs} \mid (\exists\, \mathit{chain} : \mathrm{seq}\, \mathit{crs} \bullet \mathit{head}\ \mathit{chain} = \mathit{cr}_1 \land \\
\qquad\qquad\qquad \mathit{last}\ \mathit{chain} = \mathit{cr}_2 \land (\forall\, i : 1 \mathinner{.\,.} \#\mathit{chain} - 1 \bullet \mathit{chain}\ i \sqsubset_d \mathit{chain}(i+1)))\}))
\end{array}$$

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

A set of constituting relations defining a strategy must conform to our intuitions about problem solving, namely:

1. The original problem to be solved must be known, i.e. *P_init* must always be an input attribute.
2. The solution to the original problem must be the last item to be determined, i.e. *S_final* must always be an output attribute.
3. 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.
4. Each attribute value should be determined only once, i.e. the sets of output attributes of all constituting relations must be disjoint.
5. Each solution to a subproblem is used further, i.e., it occurs as an input attribute of some constituting relation. (For the subproblems, it is not necessary to state such a requirement, because they are guaranteed to be used further to generate the solutions to the subproblems.)
6. Each solution must directly depend on the corresponding problem, i.e. if a solution attribute is an output attribute of a constituting relation, then the corresponding problem attribute must occur in the scheme of this constituting relation. Each subproblem must therefore be set up before it is solved.
7. The dependency relation on the constituting relations must not be cyclic.

Finite sets of constituting relations fulfilling these requirements are called *admissible*. In the following formal definition of admissibility, each line of the predicate part of the axiomatic box formalizes one of the previous requirements. The inverse of a relation (or function) r is denoted $r^\sim$.

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

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

Lemma 1

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

Proof

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

■

5.1.4 Strategies

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

1. The set $scheme_s\ strat$ contains the attributes P_init and S_final.
2. For each problem attribute of $scheme_s\ strat$, the corresponding solution attribute is a member of the scheme, and vice versa.
3. If a member of the relation $\bowtie strat$ contains acceptable solutions for all problems except P_init, then it also contains an acceptable solution for P_init. Thus, if all subproblems are solved correctly, then the original problem must be solved correctly as well.

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.

As with admissible constituting relations, each of the above requirements that a strategy must satisfy corresponds to one conjunct in the formal definition.

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

Figure 5.1 illustrates the definition of strategies. Here, arrows denote the propagation of attribute values.

Note that the values of the output attributes of a constituting relation need not be independent. Strategies will usually be defined in such a way that a subproblem and its corresponding solution are output attributes of the same constituting relation, and the solution fulfills certain requirements derived from the problem.

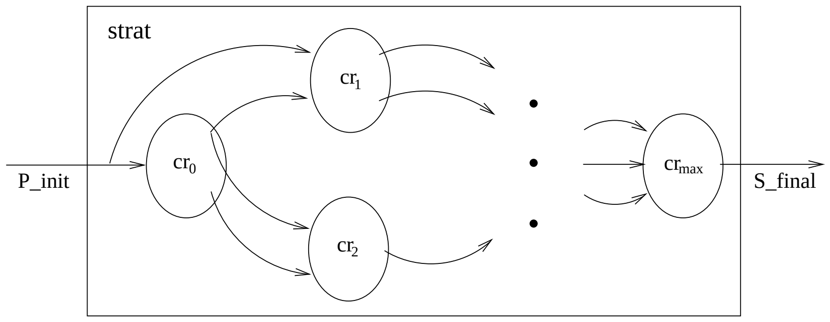


Figure 5.1: Definition of strategies

From the definition of strategies it follows that there is at least one member of a strategy that has *P_init* as its only input attribute. This means that the problem solving process can actually be initiated. Moreover, there is exactly one maximal member in a strategy that has *S_final* as its only output attribute, and this member depends on all of the other members of the strategy. In the following, we will often make use of these properties of the maximal constituting relation.

Lemma 2

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

Proof

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

■

Renaming attributes (except *P_init* and *S_final*) does not change the semantic content of a strategy. Hence, we can define an equivalence relation *_equiv_* on strategies, which will be used in Section 5.2.1:

$$\begin{array}{|l} \_equiv\_ : strategy \leftrightarrow strategy \\ \hline \forall\, strat_1, strat_2 : strategy \bullet \\ strat_1\ equiv\ strat_2 \\ \Leftrightarrow \\ (\exists\, f : scheme_s\ strat_1 \rightarrowtail\!\!\!\!\twoheadrightarrow scheme_s\ strat_2 \mid f(P\_init) = P\_init \land f(S\_final) = S\_final \bullet \\ \quad \forall\, cr : strat_1 \bullet \exists_1\, cr' : strat_2 \bullet \\ \qquad IA\ cr' = \{a : IA\ cr \bullet f\ a\} \land OA\ cr' = \{a : OA\ cr \bullet f\ a\} \land \\ \qquad (\forall\, t : tuple \bullet t \in cr \Leftrightarrow \{a : Attribute;\ v : Value \mid a \mapsto v \in t \bullet f\ a \mapsto v\} \in cr')) \end{array}$$

The definitions presented in this section comprise the theoretical foundation of our approach. The next sections show how strategies can be combined and how they can be represented to facilitate their implementation.

5.2 Strategicals

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

We define three strategicals that are useful in different contexts. The THEN strategical composes two strategies. Applications of this strategical can be found in program synthesis. The REPEAT strategical allows stepwise repetition of a strategy. Such a strategical is useful in the context of specification acquisition, where several items of the same kind often 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.

5.2.1 The THEN Strategical

The idea of this strategical is to replace one subproblem p generated by strategy $strat_1$ by the subproblems generated by strategy $strat_2$. The effect of the THEN strategical is the same as that obtained by first reducing a problem with $strat_1$ and then reducing some generated subproblem p by $strat_2$. The difference is that p and its corresponding solution $cor\,p$ are not generated explicitly. This is illustrated in Figure 5.2.

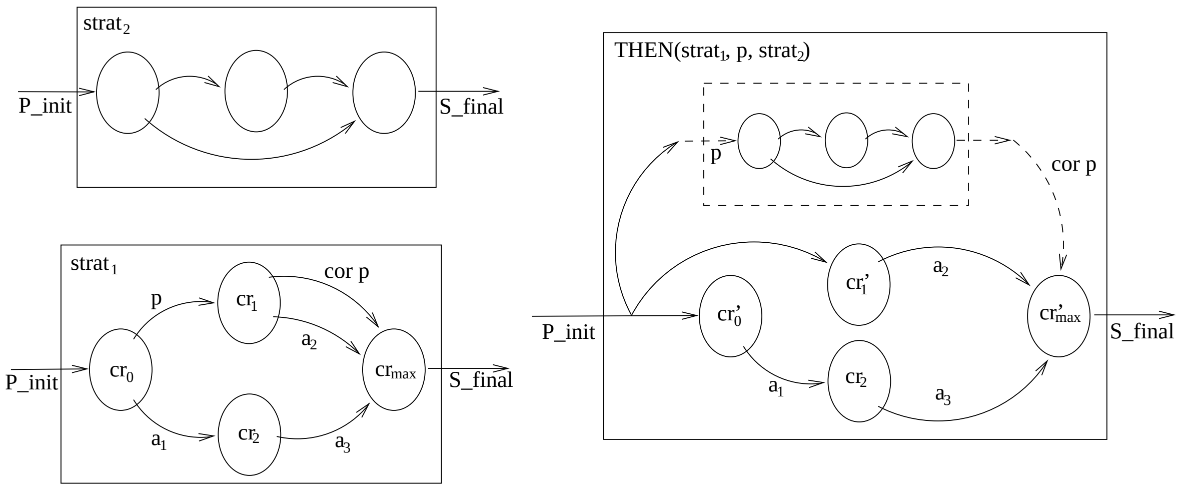


Figure 5.2: The THEN strategical

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

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

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

In defining $\textsc{Then}(strat_1, p, strat_2)$, we must first guarantee that the sets of subproblems generated by $strat_1$ and $strat_2$ are disjoint by choosing a strategy $strat_2'$ that is equivalent to $strat_2$ and fulfills this requirement. Then, in $strat_2'$, *P_init* is replaced by p and *S_final* is replaced by *cor p* using the function *replace* which replaces attribute a_{old} by attribute a_{new} in relation r.

$$\begin{array}{|l}
replace : Attribute \times Attribute \times relation \longrightarrow relation \\
\hline
\forall\, a_{old}, a_{new} : Attribute;\ r, r' : relation \bullet \\
\quad r' = replace(a_{old}, a_{new}, r) \Leftrightarrow \\
\qquad (a_{old} \notin scheme\ r \land r' = r) \lor \\
\qquad (a_{old} \in scheme\ r \land r' = \{t : r \bullet (\{a_{old}\} \ntriangleleft t) \cup \{a_{new} \mapsto t\ a_{old}\}\})
\end{array}$$

Each of the resulting constituting relations is then transformed using the function $transform_{Then}$.

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

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

Lemma 3

$$\begin{aligned}&\forall\, \mathit{strat}_1, \mathit{strat}_2 : \mathit{strategy};\, p : \mathit{ProblemAttribute} \mid (\mathit{strat}_1, p, \mathit{strat}_2) \in \mathrm{dom}\,\textsc{Then} \bullet\\ &\qquad \textsc{Then}(\mathit{strat}_1, p, \mathit{strat}_2) \in \mathit{strategy}\end{aligned}$$

The next lemma states that $\textsc{Then}(\mathit{strat}_1, p, \mathit{strat}_2)$ conforms to our intuition: its join contains exactly those tuples, which can also be obtained by joining strat_1 and $\mathit{strat}'_{2,r}$ (where $\mathit{strat}'_{2,r}$ is defined as before) and then dropping the values of p and $\mathit{cor}\, p$.

Lemma 4

$$\begin{aligned}&\bowtie (\textsc{Then}(\mathit{strat}_1, p, \mathit{strat}_2)) = \{p, \mathit{cor}\, p\} \lhd_r (\bowtie (\mathit{strat}_1 \cup \mathit{strat}'_{2,r}))\\ &\textit{where } \mathit{strat}'_{2,r} \textit{ is defined as in the definition of } \textsc{Then}\end{aligned}$$

Finally, Lemma 5 states that Then does not introduce any new dependencies, i.e., if two constituting relations of $\textsc{Then}(\mathit{strat}_1, p, \mathit{strat}_2)$ are dependent, also their untransformed versions are.

Lemma 5

$$\begin{aligned}&\mathit{cr}'_t \sqsubset_{\textsc{Then}(\mathit{strat}_1, p, \mathit{strat}_2)} \mathit{cr}_t \;\Rightarrow\; \mathit{cr}' \sqsubset_{\mathit{strat}_1 \cup \mathit{strat}'_{2,r}} \mathit{cr}\\ &\textit{where } \mathit{strat}'_{2,r} \textit{ is defined as in the definition of } \textsc{Then} \textit{ and}\\ &\mathit{cr}_t = \mathit{transform}_{\mathit{Then}}(\mathit{cr}, \mathit{strat}_1 \cup \mathit{strat}'_{2,r}, p) \wedge \mathit{cr}'_t = \mathit{transform}_{\mathit{Then}}(\mathit{cr}', \mathit{strat}_1 \cup \mathit{strat}'_{2,r}, p)\end{aligned}$$

Proof

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

$$\mathit{cr}'_t \sqsubset_d \mathit{cr}_t \Rightarrow \mathit{cr}' \sqsubset_{\mathit{strat}_1 \cup \mathit{strat}'_{2,r}} \mathit{cr}$$

The relation $\mathit{cr}_t \sqsubset_d \mathit{cr}'_t$ means $\mathit{OA}\, \mathit{cr}'_t \cap \mathit{IA}\, \mathit{cr}_t \neq \emptyset$. According to the definition of $\mathit{transform}_{\mathit{Then}}$, this is equivalent to

$$\begin{aligned}&(\mathit{OA}\, \mathit{cr}' \setminus \{p, \mathit{cor}\, p\})\\ &\qquad \cap\, ((\mathit{IA}\, \mathit{cr} \cup \mathit{scheme}(\bowtie \{r : \mathit{strat}_1 \cup \mathit{strat}'_{2,r} \mid r \sqsubset_{\mathit{strat}_1 \cup \mathit{strat}'_{2,r}} \mathit{cr}\})) \setminus \{p, \mathit{cor}\, p\})\\ &\neq \emptyset\end{aligned}$$

This condition is equivalent to

$$\begin{aligned}&\exists\, a : \mathit{Attribute} \mid a \notin \{p, \mathit{cor}\, p\} \bullet\\ &\qquad a \in \mathit{OA}\, \mathit{cr}' \wedge a \in (\mathit{IA}\, \mathit{cr} \cup \mathit{scheme}(\bowtie \{r : \mathit{strat}_1 \cup \mathit{strat}'_{2,r} \mid r \sqsubset_{\mathit{strat}_1 \cup \mathit{strat}'_{2,r}} \mathit{cr}\}))\end{aligned}$$

If $a \in \mathit{OA}\, \mathit{cr}' \cap \mathit{IA}\, \mathit{cr}$, we immediately have $\mathit{cr}' \sqsubset_d \mathit{cr}$ and hence $\mathit{cr}' \sqsubset_{\mathit{strat}_1 \cup \mathit{strat}'_{2,r}} \mathit{cr}$. Otherwise,

$$\exists\, \overline{\mathit{cr}} : \{r : \mathit{strat}_1 \cup \mathit{strat}'_{2,r} \mid r \sqsubset_{\mathit{strat}_1 \cup \mathit{strat}'_{2,r}} \mathit{cr}\} \bullet a \in \mathit{scheme}\, \overline{\mathit{cr}}$$

Since the sets of all output attributes of $\mathit{strat}_1 \cup \mathit{strat}'_{2,r}$ except p and $\mathit{cor}\, p$ are disjoint, and since $a \notin \{p, \mathit{cor}\, p\}$, a cannot be an output attribute of $\overline{cr}$. Hence, $a \in \mathit{IA}\, \overline{cr}$. This yields $\mathit{OA}\, cr' \cap \mathit{IA}\, \overline{cr} \neq \varnothing$. It follows that

$$cr' \sqsubset_d \overline{cr}, \text{ where } \overline{cr} \sqsubset_{\mathit{strat}_1 \cup \mathit{strat}'_{2,r}} cr$$

which finishes the proof of Lemma 5.

□

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

$$\begin{aligned} &\mathit{scheme}_s(\text{THEN}(\mathit{strat}_1, p, \mathit{strat}_2)) \\ &\quad = (\mathit{scheme}_s\, \mathit{strat}_1 \setminus \{p, \mathit{cor}\, p\}) \cup (\mathit{scheme}_s\, \mathit{strat}'_2 \setminus \{P\_\mathit{init}, S\_\mathit{final}\}) \end{aligned}$$

where $(\mathit{scheme}_s\, \mathit{strat}_1 \setminus \{p, \mathit{cor}\, p\}) \cap (\mathit{scheme}_s\, \mathit{strat}'_2 \setminus \{P\_\mathit{init}, S\_\mathit{final}\}) = \varnothing$. It follows that the conditions 1 and 2 of the definition of a strategy are fulfilled.

The admissibility of $\text{THEN}(\mathit{strat}_1, p, \mathit{strat}_2)$ is fulfilled, as the following argumentation shows:

- Since $\mathit{transform}_{\mathit{Then}}$ does not change the property of P_init or S_final being an input or output attribute of some $cr \in \mathit{strat}_1$, and since these two attributes do no longer occur in $\mathit{strat}'_{2,r}$, the requirements 1 and 2 of the admissibility definition are fulfilled.

- The function $\mathit{transform}_{\mathit{Then}}$ removes the attributes p and $\mathit{cor}\, p$ from the input and output attributes of the constituting relation supplied as its first argument. Since these attributes are no longer in the scheme of $\text{THEN}(\mathit{strat}_1, p, \mathit{strat}_2)$, removing them does not destroy satisfiability of requirements 3 and 4. Hence, the requirements also hold for the transformed constituting relations.

- Condition 5 holds because it holds for strat_1 as well as strat'_2, and because the attribute $\mathit{cor}\, p$ that replaces S_final in strat'_2 is an input attribute of a constituting relation of strat_1.

- The condition 6 is not changed by $\mathit{transform}_{\mathit{Then}}$ because the attributes p and $\mathit{cor}\, p$ are always removed pairwise from the schemes of the constituting relations.

- We show condition 7 by contraposition. If there were cyclic dependencies in $\text{THEN}(\mathit{strat}_1,$ $p, \mathit{strat}_2)$, then, by Lemma 5, there would also be a cyclic dependency in $\mathit{strat}_1 \cup \mathit{strat}'_{2,r}$. Since strat_1 and $\mathit{strat}'_{2,r}$ both cannot contain cycles (because strat_1 and strat'_2 are strategies), a cycle in $\mathit{strat}_1 \cup \mathit{strat}'_{2,r}$ must contain members of both strat_1 and $\mathit{strat}'_{2,r}$. It follows without loss of generality that

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

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

$$\begin{array}{l} chain\ i \in strat_1 \wedge chain(i+1) \in strat'_{2,r} \wedge \\ chain\ k \in strat'_{2,r} \wedge chain(k+1) \in strat_1 \end{array}$$

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

For index i, $cor\ p \in OA(chain\ i) \cap IA(chain(i+1))$ is impossible because, in $strat'_{2,r}$, $cor\ p$ is always an output attribute (it takes the role of S_final in $strat'_2$). For index k, $p \in OA(chain\ k) \cap IA(chain(k+1))$ is impossible because, in $strat'_{2,r}$, p is always an input attribute (it takes the role of P_init in $strat_{2'}$). It follows that

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

Since $p \in OA(chain\ i)$, we get $\neg\ (cr_{cor\ p} \sqsubset_{strat_1} chain\ i)$, where $cr_{cor\ p}$ is the unique constituting relation of $strat_1$ that contains $cor\ p$ as an output attribute. Since i was chosen to be minimal, $cr \sqsubset_{strat_1} chain\ i$ holds. It follows that $\neg\ (cr_{cor\ p} \sqsubset_{strat_1} cr)$ holds, since otherwise, by transitivity of $\sqsubset_{strat_1}$, we would have $cr_{cor\ p} \sqsubset_{strat_1} cr \sqsubset_{strat_1} chain\ i$.

Since $cor\ p \in IA(chain(k+1))$, we have $cr_{cor\ p} \sqsubset_{strat_1} chain(k+1)$. Index k was chosen to be maximal, which yields $chain(k+1) \sqsubset_{strat_1} cr$. By transitivity of $\sqsubset_{strat_1}$, we then get $cr_{cor\ p} \sqsubset_{strat_1} cr$. But this is a contradiction because we had already concluded that $\neg\ (cr_{cor\ p} \sqsubset_{strat_1} cr)$. This completes the proof that $\textsc{Then}(strat_1, p, strat_2)$ contains no cycles.

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

□

To prove Lemma 4, we first show that

$$\begin{array}{l} \forall\ cr : strat_1 \cup strat'_{2,r} \mid cr \neq cr_{max} \bullet cr \sqsubset_{strat_1 \cup strat'_{2,r}} cr_{max} \\ \text{where } cr_{max} == (\mu\ r : strat_1 \mid S\_final \in OA\ r) \end{array}$$

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

This gives us

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

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

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

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

which holds because all $cr_t : \textsc{Then}(strat_1, p, strat_2)$ are defined to be $\{p, cor\,p\} \lhd_r \bowtie crs'$ for some $crs' \subseteq strat_1 \cup strat'_{2,r}$. By the definition of $\bowtie$, if a tuple is in the join of a set of relations, then its appropriate restriction is in every subset of that set. This concludes the proof of Lemma 4 (and hence of Lemma 3).

$\blacksquare$

An example of a strategy defined with THEN is given in Section 6.2.4.

5.2.2 The REPEAT Strategical

The first argument of this strategical is the strategy *strat* to be repeated. Repetition here means that a subproblem p generated by *strat* should again be reduced by a finite iteration of *strat* or another strategy *terminate* that does not generate new subproblems. The attribute p and the strategy *terminate* are the other arguments of REPEAT. A strategy defined with REPEAT does not itself perform an iteration but only one step of an iteration. How often *strat* is iterated is decided elsewhere, e.g. by the user of an implemented system. The strategy $\textsc{Repeat}(strat, p, terminate)$ is distinguished from *strat* only in that one if its constituting relations is restricted, as shown in Figure 5.3.

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

$$
\begin{array}{|l}
\textsc{Repeat} : strategy \times ProblemAttribute \times strategy \pfun strategy \\
\hline
\forall\, strat, terminate : strategy;\ p : ProblemAttribute \bullet \\
((strat, p, terminate) \in \operatorname{dom} \textsc{Repeat} \\
\qquad \Rightarrow p \in subprs_s\ strat \land scheme_s\ terminate = \{P\_init, S\_final\}) \\
\land \\
(\mathbf{let}\ cr_p == (\mu\, cr : strat \mid cor\,p \in OA\,cr) \bullet \\
(\mathbf{let}\ cr_{rep} == \{t : cr_p \mid \{P\_init \mapsto t\,p, S\_final \mapsto t(cor\,p)\} \in \bowtie terminate \\
\qquad\qquad\qquad\qquad\qquad \lor \\
\qquad\qquad\qquad\qquad\qquad (\exists\, n : \mathbb{N}_1;\ ts : \operatorname{seq}(\bowtie strat) \mid \#ts = n \bullet \\
\qquad\qquad\qquad\qquad\qquad\qquad (ts\,1)\ P\_init = t\,p \land (ts\,1)\ S\_final = t(cor\,p) \land \\
\qquad\qquad\qquad\qquad\qquad\qquad (\forall\, i : 2 \mathrel{..} n \bullet (ts\ i)\ P\_init = (ts(i-1))\ p \land \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (ts\ i)\ S\_final = (ts(i-1))\ (cor\,p)) \land \\
\qquad\qquad\qquad\qquad\qquad\qquad \{P\_init \mapsto (ts\ n)\ p, S\_final \mapsto (ts\ n)(cor\,p)\} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \in \bowtie terminate)\} \bullet \\
IA\ cr_{rep} = IA\ cr_p \land \\
\textsc{Repeat}(strat, p, terminate) = ((strat \setminus \{cr_p\}) \cup \{cr_{rep}\})))
\end{array}
$$

Whenever $\textsc{Repeat}(strat, p, terminate)$ is defined, it yields a strategy:

Lemma 6

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

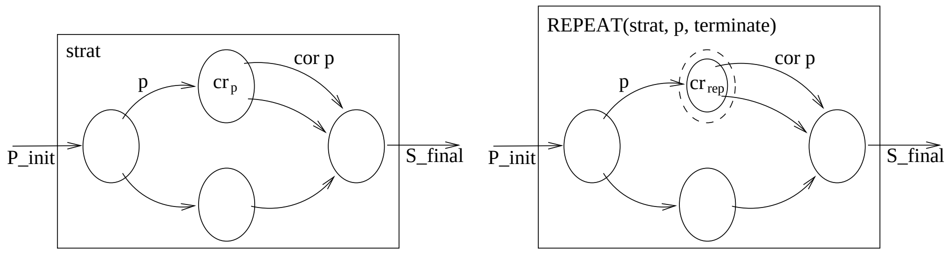


Figure 5.3: The Repeat strategical

Proof

Since Repeat(*strat*, *p*, *terminate*) is distinguished from *strat* only by additional requirements on membership in its constituting relation cr_p, it follows immediately that Repeat(*strat*, *p*, *terminate*) is a strategy.

■

5.2.3 The Lift Strategical

It is possible that a strategy must be changed to make the Repeat strategical applicable. In specification acquisition, for instance, we are face with the problem of defining a list of Z operations, and we might wish to solve this problem by repeatedly applying a strategy *define_schema* that defines one schema. As it is, this strategy cannot serve as an argument to Repeat, because it defines only one schema and not a list of schemas – it cannot be applied to the subproblems it generates, namely the problems to define the declaration part and to define the predicate part of the schema. We must first "lift" the strategy if we want to generate a list of schemas rather than a single schema. The "lifted" strategy will generate in addition to the problems generated by its argument strategy, one problem, which will be used for the repetition. In addition to a strategy to be repeated, the Lift strategical requires the following arguments:

1. a function *p_down*, which converts a "bigger" problem (e.g., that of developing a list of schemas), into a "smaller" one (e.g., that of developing one schema),
2. an injective function *p_combine*, which combines the original problem and a partial solution to yield a new problem, and
3. a function *s_combine*, which combines the solutions of "bigger" and "smaller" problems.

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

$$\begin{aligned}&\forall\, pr : Problem;\ sol, sol' : Solution \mid\\ &\quad sol'\ acceptable\_for\ p\_down\ pr \wedge sol\ acceptable\_for\ p\_combine(pr, sol') \bullet\\ &\quad s\_combine(sol', sol)\ acceptable\_for\ pr\end{aligned}$$

The Lift strategical is illustrated in Figure 5.4.

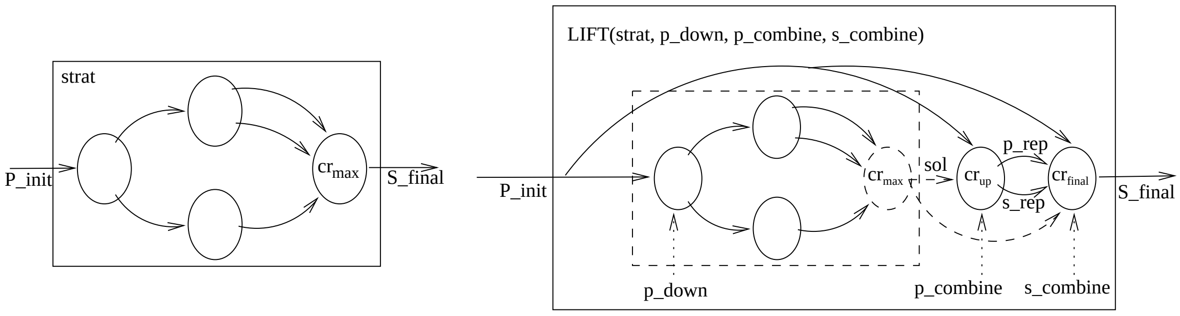


Figure 5.4: The LIFT strategical

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

As for the THEN strategical, we need to transform the constituting relations of *strat*. Since LIFT(*strat*, *p_down*, *p_combine*, *s_combine*) solves a "bigger" problem, this problem must be transformed by *p_down* into a "smaller" one in order to ensure that *strat* is applicable. The transformation concerns only those constituting relations whose schema contains *P_init*.

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

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

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

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

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

Lemma 7

$$
\begin{array}{l}
\forall\, \mathit{strat} : \mathit{strategy};\ \mathit{p\_down} : \mathit{Problem} \pfun \mathit{Problem}; \\
\quad \mathit{p\_combine} : \mathit{Problem} \times \mathit{Solution} \pinj \mathit{Problem}; \\
\quad \mathit{s\_combine} : \mathit{Solution} \times \mathit{Solution} \pfun \mathit{Solution} \mid \\
\qquad (\mathit{strat}, \mathit{p\_down}, \mathit{p\_combine}, \mathit{s\_combine}) \in \mathrm{dom}\, \textsc{Lift} \bullet \\
\qquad \textsc{Lift}(\mathit{strat}, \mathit{p\_down}, \mathit{p\_combine}, \mathit{s\_combine}) \in \mathit{strategy}
\end{array}
$$

Proof

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

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

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

  holds because $cr \sqsubset_{strat} cr_{max}$ for $cr : strat \setminus \{cr_{max}\}$ and $IA\ cr_{max} \subseteq IA\ cr_{up}$. Thirdly, we have $cr_{up} \sqsubset_d cr_{final}$, and for $cr : crs_{new}$, $(cr_{up} \sqsubset_{\textsc{Lift}(strat,\ldots)} cr)$ is impossible because $OA\ cr_{up} \cap scheme_s\ crs_{new} = \emptyset$. Finally, cr_{final} does not depend on any other constituting relation because *S_final* does not occur in the scheme of any constituting relation other than cr_{final}.

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

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

which, according to the requirements on *p_down*, *p_combine* and *s_combine* is sufficient to conclude that $s\_combine(sol, t\ s\_up) = t\ S\_final\ acceptable\_for\ t\ P\_init$ holds.

∎

An example of a strategy defined with LIFT and REPEAT is given in Section 7.3.2.

We have now defined strategies formally, and so have specified methods for combining simpler strategies into more powerful ones. Thus far, strategies have been described in a purely declarative manner. Our goal, however, is to make strategies *applicable* for problem solving, and so, we must ultimately take a more much procedural view of them.

5.3 Problem Solving With Strategies

Strategies and strategicals, as they have been defined thus far, are the conceptual basis for strategy-based problem solving. To make strategies applicable mechanically, we must take two further steps: First, we represent strategies as modules that are implementable using the encapsulation constructs offered by modern programming languages. Secondly, we present an abstract algorithm describing the manner in which strategy-based problem solving is to proceed. This algorithm will be expressed as a set of algebraically defined Z functions, intended to denote recursive algorithms easily implementable in a functional programming language. If the algorithm yields a solution to a given input problem, then this solution will be acceptable.

5.3.1 Modular Representation of Strategies

To render strategies implementable, we must find suitable representations for them, which are closer to the constructs provided by programming languages than are the relations of database theory. Implementations of strategies should be independent of each other with a uniform interface between them. In an implemented support system for strategy-based problem solving, the implementation of a strategy is a module with a clearly defined interface to other strategies, as well as the rest of the system. A strategy module comprises the following items:

- the set *subp* of subproblems it produces,
- the dependency relation *_depends_* on them and their solutions,
- for each subproblem, a procedure *setup* that defines it, using the information in the initial problem and the subproblems and solutions it depends on,
- for each solution to a subproblem, a predicate *local_accept* that checks whether or not the solution conforms to the requirements stated in the constituting relation of which it is an output attribute,
- a procedure *assemble* describing how to assemble the final solution, and
- a test *accept* of acceptability for the assembled solution.

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

In the following, we define a number of functions, each of which has a strategy as its argument and yields one of the pieces of information described before. That is, each of these functions defines one component of a strategy module for its argument strategy.

The function $subprs_s$ introduced in Section 5.1.4 yields the subproblems generated by a strategy. The dependency relation must be defined on pairs of problems instead of pairs of constituting relations:

$$\begin{array}{|l}
Depends : strategy \longrightarrow (ProblemAttribute \longleftrightarrow ProblemAttribute) \\
\hline
\forall\, strat : strategy;\ p_1, p_2 : ProblemAttribute \mid \{p_1, p_2\} \subseteq scheme_s\ strat \bullet \\
\quad (\mathbf{let}\ cr_1 == (\mu\, r : strat \mid p_1 \in OA\, r); \\
\qquad cr_2 == (\mu\, r : strat \mid p_2 \in OA\, r) \bullet \\
\qquad\quad (p_1, p_2) \in Depends(strat) \Leftrightarrow cr_1 \sqsubset_{strat} cr_2)
\end{array}$$

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

$[\mathit{ExtInfo}]$

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

```
Setup : strategy ⟶ (ProblemAttribute ⇸ (tuple × ExtInfo ⇸ Problem))
─────────────
∀ strat : strategy; p : ProblemAttribute •
    dom(Setup(strat)) = subprs_s strat ∧
    (p ∈ subprs_s strat ⇒
        (∃₁ cr_p : strat | p ∈ OA cr_p •
            ∀ t : tuple; i : ExtInfo | (t, i) ∈ dom(Setup(strat)(p)) •
                dom t ⊇ IA cr_p ∧
                ((IA cr_p ◁ t) ∈ (IA cr_p)◁_r cr_p ⇒
                    (IA cr_p ◁ t) ∪ {p ↦ (Setup(strat)(p))(t, i)}
                                  ∈ (IA cr_p ∪ {p})◁_r cr_p)))
```

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

```
Local_Accept : strategy ⟶ (SolutionAttribute ⇸ (tuple ⟷ Solution))
─────────────
∀ strat : strategy; s : SolutionAttribute •
    dom(Local_Accept(strat)) = partsols strat ∧
    (s ∈ partsols(⋈ strat) ⇒
        (∃₁ cr_s : strat | s ∈ OA cr_s •
            ∀ t : tuple; sol : Solution | (t, sol) ∈ Local_Accept(strat)(s) •
            (let inp == IA cr_s ∪ {cor~s} •
                dom t ⊇ inp ∧
                ((inp ◁ t) ∈ inp ◁_r cr_s ⇒
                    (inp ◁ t) ∪ {s ↦ sol} ∈ (inp ∪ {s}) ◁_r cr_s))))
```

The conditions for the *Assemble* function can be expressed similarly.

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

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

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

A Z specification does not specify what happens if a function is applied to an argument that does not lie in the domain of the function, and so an algorithm implementing the function could reasonably either fail to terminate or report failure. For our problem solving algorithm, we will require that failure is reported whenever a problem cannot be set up, a solution cannot be assembled properly, or a partial solution is determined not to be locally acceptable. This is achieved by defining free types into which problems, solutions, and tuples are embedded, and which contain error values indicating that some of the previous functions are undefined. Thus, partial functions are made total by allowing them to return members the free types other than problems, solutions, or tuples.

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

Strategy modules are algorithmic descriptions of strategies. A strategy module is obtained by applying the functions $subprs_s$, *Depends*, *Setup*, *Local_Accept*, *Assemble* and *Accept* to a strategy *strat*, and making total the functions being the results of *Setup* and *Assemble*. An error value is returned if and only if the corresponding partial function is undefined. Strategy modules are defined as schema types that resemble record types in programming languages. The components of schema types are selected with the dot notation, e.g., for a strategy module *sm*, we write *sm.subp* to denote the subproblems generated by the strategy.

StrategyModule

$$
\begin{array}{l}
subp : \mathbb{P}\ ProblemAttribute \\
\_depends\_on\_ : ProblemAttribute \leftrightarrow ProblemAttribute \\
setup : ProblemAttribute \mathrel{+\!\!\!\rightarrow} (tuple \times ExtInfo \longrightarrow total\_P) \\
local\_accept : SolutionAttribute \mathrel{+\!\!\!\rightarrow} (tuple \leftrightarrow Solution) \\
assemble : tuple \times ExtInfo \longrightarrow total\_S \\
accept : \mathbb{P}\ tuple
\end{array}
$$

$$
\begin{array}{l}
\exists\, strat : strategy \bullet \\
(subp = subprs_s\ strat \wedge \\
(\_depends\_on\_) = Depends\ strat \wedge \\
local\_accept = Local\_Accept\ strat \wedge \\
accept = Accept\ strat \wedge \\
(\forall\, p : ProblemAttribute;\ t : tuple;\ i : ExtInfo \bullet \\
\quad (((t,i) \in \mathrm{dom}((Setup\ strat)(p)) \Leftrightarrow setup(p)(t,i) \in \mathrm{ran}\ ok\_P) \wedge \\
\quad ((t,i) \in \mathrm{dom}((Setup\ strat)(p)) \Rightarrow setup(p)(t,i) = ok\_P((Setup\ strat)(p)(t,i))) \wedge \\
\quad ((t,i) \in \mathrm{dom}(Assemble\ strat) \Leftrightarrow assemble(t,i) \in \mathrm{ran}\ ok\_S) \wedge \\
\quad ((t,i) \in \mathrm{dom}(Assemble\ strat) \Rightarrow assemble(t,i) = ok\_S((Assemble\ strat)(t,i)))))
\end{array}
$$

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

5.3.2 An Abstract Problem Solving Algorithm

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

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

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

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

The set *available_strategies* denotes the set of all available strategy modules. The function *choice* is used to select, from among the available strategies, a strategy to solve the given problem.

```
available_strategies : ℙ StrategyModule
choice : Problem × (ℙ StrategyModule) × UserInput ⇸ StrategyModule
────────
∀ p : Problem; sms : ℙ StrategyModule; inp : UserInput | sms ≠ ∅ •
      choice(p, sms, inp) ∈ sms
```

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

```
solve : Problem × seq UserInput ⇸ (Solution × (seq UserInput))
────────
∀ pr : Problem; input_list : seq UserInput •
solve(pr, input_list) =
      (let sm == choice(pr, available_strategies, head input_list) •
      (let t == apply(pr, sm, tail input_list) •
      if first t = fail_t then solve(pr, second t)
      else ((ok_t~(first t)) S_final, second t)))
```

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

```
apply : Problem × StrategyModule × seq UserInput ⇸ (total_t × seq UserInput)
────────
∀ p : Problem; sm : StrategyModule; input_list : seq UserInput •
apply(p, sm, input_list) =
      (let s == solve_subprs({P_init ↦ p}, sm.subp, sm, input_list) •
      if first s = fail_t then (fail_t, second s)
      else (let tup == ok_t~(first s);
                  input_list' == second s •
            (let ext_info == heuristic_function(sm, S_final)(tup, head input_list') •
            (let final_solution == sm.assemble(tup, ext_info) •
            if final_solution = fail_S then (fail_t, tail input_list')
            else (let s' == tup ∪ {S_final ↦ ok_S~final_solution} •
                  if s' ∉ sm.accept then (fail_t, tail input_list')
                  else (ok_t(s'), tail input_list'))))))
```

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

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

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

Lemma 8

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

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

Lemma 9

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

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

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

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

Proof

Lemma 8 follows from Lemma 9, and the observation that *solve* is defined in such a way that the first component of its result is a tuple t belonging to the strategy implemented by the chosen strategy module *sm*.

$\square$

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

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

$\square$

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

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

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

$\blacksquare$

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

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

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

In this section, we have transformed purely declarative strategies into more procedural representations of strategies and used problem solving functions show how such representations are used to perform strategy-based development. These functions have, however, been defined in such a way that demonstrating acceptability of developed solutions is facilitated. They are therefore very abstract and do not take adequate user support into account, as is necessary for implemented support systems.

5.4 System Architecture

We now define a system architecture that describes how to implement support systems for strategy-based problem solving. By contrast with the functions of the previous section, this system architecture takes the user into account and allows for much more flexibility in the problem solving process than does the abstract algorithm of Section 5.3.2.

The definition of strategies is parameterized by the notions of problem, solution, and acceptability, leading to a *generic* system architecture supporting strategy-based development processes. Figure 5.5 gives a general view of the architecture.

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. The system architecture can be used to advantage, so that 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.

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 according to the dependencies between the produced subproblems.

As soon as the solution to a subproblem generated by the strategy has been developed, it is tested for local acceptability. Checking each solution for local acceptability immediately

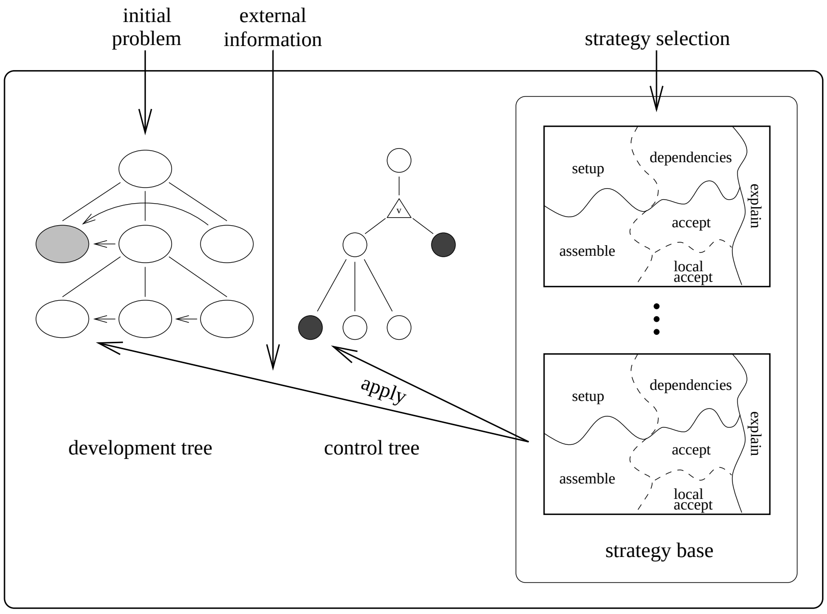


Figure 5.5: General view of the system architecture

after its construction ensures that the user is informed of a failure at the earliest possible moment.

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.

In the following, we describe the data structures of the generic system architecture and the data and control flow in more detail.

5.4.1 The Structure of Development and Control Trees

We describe the internal structure of the development and the control tree, and their interaction with the strategy base.

Development Tree

Figure 5.6 sketches a development tree. The arcs indicate that a problem is reduced by a strategy. The number of successor nodes of a node coincides with the number of subproblems generated by the strategy. Pointed arcs between sibling nodes indicate that the problem corresponding to the node from where the arc starts depends on the problem or solution of the node where it points to. The shaded node indicates that the corresponding problem has already been solved.

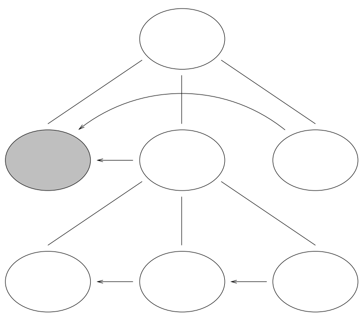


Figure 5.6: A development tree

In the development whose corresponding development tree is shown in Figure 5.6, the original problem was reduced by a strategy producing three subproblems. The problem corresponding to the leftmost successor of the root node must be solved first, whereas the other two subproblems are independent of one another. After having solved the first problem directly (the shaded node has no children), we reduced the problem corresponding to the second successor of the root node by another strategy, which generates three subproblems that have to be solved in a fixed order from left to right.

Two strategies are involved in processing one node of the development tree: a *creating* and a *reducing* strategy. Figure 5.7 shows the internal structure of a node of the development tree and its relation to the creating and reducing strategies. The flow of information is indicated by pointed arcs.

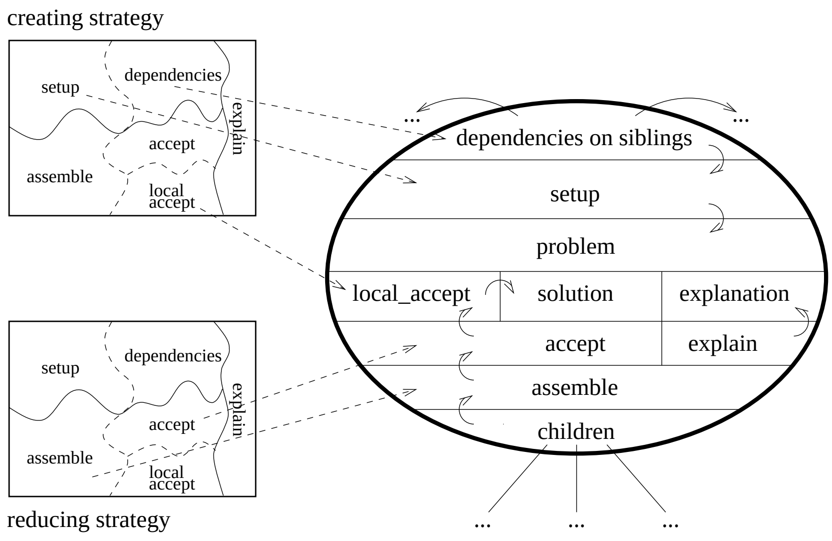


Figure 5.7: Structure of a node in the development tree

A node contains a problem and its solution, and references to its children and to siblings it depends on. Furthermore, it contains the functions needed to set up the problem and determine its solution. These functions stem from the strategy modules involved.

Let a particular node belong to the subproblem *p* of the creating strategy. Then the corresponding strategy module provides the set-up function *setup p*. The dependency pointers are obtained from the *_depends_on_* component of the strategy module.

The reducing strategy produces the children of a development node. It is therefore responsible to provide the functions that build the solution, check its acceptability and possibly provide an explanation. The same strategy plays a dual role for the children nodes: for them it is the creating strategy.

The development tree as a data structure contains all information about the process, the unsolved problems and the result of the current development. Thus it is the basis to browse and provide views of developments, switch between developments, and analyze them for replay and reuse.

Control Tree

The purpose of the control tree is to keep track of unsolved problems and their dependencies. It provides a basis to choose the next problem to reduce. Figure 5.8 shows how the nodes of the control tree point to unsolved nodes in the development tree.

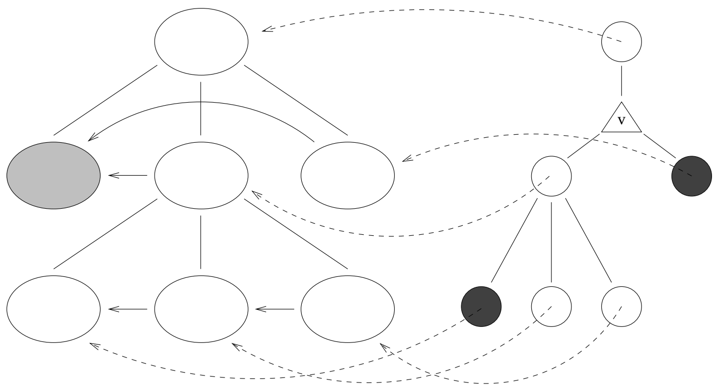


Figure 5.8: Relation between development and control trees

There are two kinds of branchings in the control tree that stem from the dependencies between the development nodes. They indicate whether siblings have to be solved in a fixed left-to-right order or if they may be solved in an arbitrary order. The “normal” branching in the left subtree of the control tree in Figure 5.8 represents a fixed order in which the problems have to be solved. On the other hand, the triangle v in the upper branching represents an arbitrary order for the two children of the root. The leaves of the control tree point to unreduced problems. The shaded leaves may be tackled in the next step.

As far as possible, selection of the next problem should be left to the developer. When selecting a strategy to reduce a particular problem, it is usually not obvious if the strategy will succeed in producing a solution. Therefore developers might try to tackle the “hardest” subproblem first and reduce it until they can decide if a solution is possible. Then they might concentrate on the next “hard” problem in some other branch of the development. In this way, the architecture supports focusing the development on the critical tasks first.

All information the control tree represents is contained in the development tree. Still, for efficiency reasons, it is useful to maintain control information explicitly. The development tree grows with each strategy application while the control tree shrinks whenever a solution is found. Without an explicit control tree, the set of reducible nodes would have to be recomputed for each strategy application.

5.4.2 Data Flow

The data flow diagram in Figure 5.9 describes how the global data structures are manipulated. The main control flow is a loop of strategy applications. Upon each entrance of the loop body, a backtrack point is set. The strategy application cycle consists of selecting a problem and a strategy, reducing that problem by the strategy, and assembling solutions.

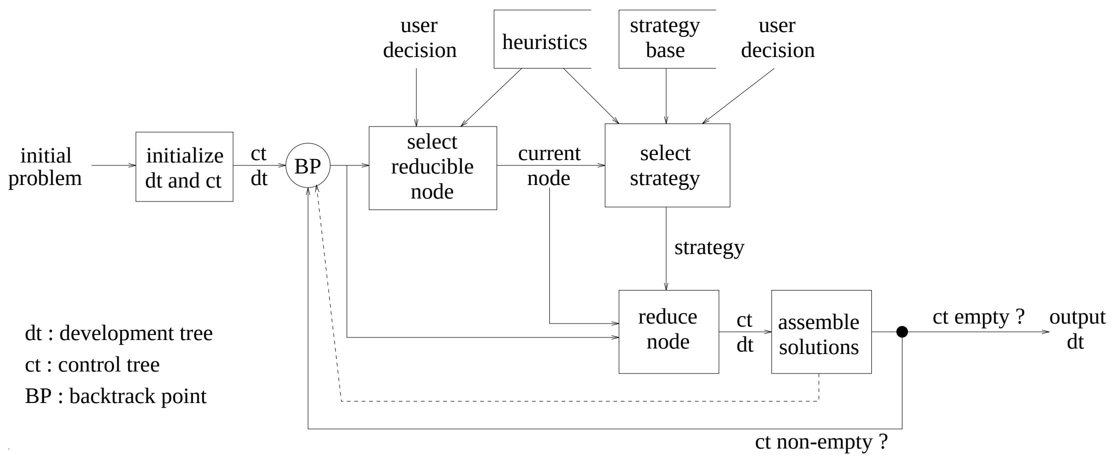


Figure 5.9: Data flow diagram for the architecture

Node selection

The set of reducible leaves can be determined by considering the control tree's two kinds of branchings. The reducible leaves of a tree with normal root branching are the reducible leaves of the leftmost subtree. For a triangle branching, they are the *union* of the sets of reducible leaves of all subtrees. Users may choose from the set of reducible leaves. The chosen node becomes the *current node*. It is possible to enhance flexibility of node selection and try to set up problems that depend on incomplete solutions. Such problems are not in the set of reducible leaves determined from the control tree.

Strategy selection

Like selecting a node, choosing a strategy is typically a user decision, which may be assisted by heuristics. For example, some strategies are applicable only to problems with certain properties. One heuristic might be to search the strategy base for strategies particularly suited for the current problem.

Node reduction

Node reduction extends the development tree and the control tree at the current node according to the selected strategy. The strategy module's *subpr* and *_depends_on_* components provide information how many children nodes must be created and which dependency pointers between them have to be established. The function *setup p* and the predicate *local_accept*(*cor p*) are entered in the children nodes for each subproblem p, and according to the role as reducing strategy for the current node, the *assemble*, *accept* and *explain* functions are entered in that node.

Solution assembly

After node reduction, the extended development and control trees are searched for solutions to assemble. If the selected strategy creates no subproblems, the solution to the current node can be immediately determined: *assemble* is called for the current node, and the *accept* test of the reducing strategy and the *local_accept* test of the creating strategy are applied. If one of them fails, the most recent cycle of problem selection and strategy application is undone. The system backtracks to the state of development before selection of the current node, symbolized by the dashed arrow in Figure 5.9.

If the solution is acceptable, *explain* fills in the *explanation* field of the current node (cf. Figure 5.7). The current node of the control tree is deleted. If the parent node of the deleted one has no other children, the process of solution assembly is recursively applied to that node.

Even if a solution is acceptable for the selected strategy, it may be inadequate as part of the solution to a problem higher up in the development tree. Any failure of an *accept* or *local_accept* predicate during recursive solution assembly therefore causes a backtrack, where the most recent strategy application is undone.

Backtracking may be initiated by the users as well, e.g. if they decide that a strategy application leads nowhere because the generated subproblems cannot be solved. User-driven backtracking is possible during both node and strategy selection.

The loop of strategy applications terminates when the control tree is empty, yielding a development tree in which all nodes have successfully been solved. Its root contains the solution to the initial problem.

This architecture guarantees the greatest possible flexibility in strategy-based problem solving. The user can always obtain an overview of the state of development and the context in which a certain problem has to be solved. The modular implementation of the strategy base facilitates incorporating new strategies in a routine manner. The architecture is independent of the kind of development activity that is to be supported, and so can be re-used for different instantiations of the strategy framework.

5.5 Related Work

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

Knowledge-Based Software Engineering (KBSE)

This discipline seeks to support software engineering via the use of artificial intelligence techniques. It comprises a variety of approaches to specification acquisition and program synthesis, such as those discussed in (Lowry and Duran, 1989) and (Lowry and McCartney, 1991). The strategy framework could also be subsumed under this field, because a knowledge representation mechanism is its central concept.

A prominent example of KBSE, whose aims closely resemble our own, is the Programmer's Apprentice project (Rich and Waters, 1988). 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. The assumption underlying the Apprentice approach is that a library of prototypical examples provides better user support than the representation of general-purpose knowledge. Our position is to prefer general-purpose knowledge because clichés depend to a large extent upon the application domain. This makes it difficult to set up a cliché library, which is sufficiently complete that it does not need to be extended to accommodate each new problem to be solved.

Representation of Design and Process Knowledge

Wile (Wile, 1983) 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 (Osterweil, 1987; Shepard et al., 1992).

Potts (Potts, 1989) aims at capturing not only strategic but also heuristic aspects of design methods. He uses *Issue-Based Information Systems* (IBIS) (Conclin and Begeman, 1988) 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.

In the project KORSO (Broy and Jähnichen, 1995), the product of a development is described by a *development graph.* The nodes of the development graph are specification modules or program modules whose static composition and refinement relations are expressed using two kinds of vertices. There is no explicit distinction between "problem nodes," whose contents are not completely known, and "solution nodes". Unlike development trees, KORSO development graphs do not reflect single development steps. A branching in a development tree maps to a subgraph in a development graph in which process information – such as dependencies between subproblems – cannot be represented.

Tactical Theorem Proving.

Tactical theorem proving was first employed in Edinburgh LCF (Milner, 1972). The idea behind tactical theorem proving is to interactively construct goal-directed proofs by backward chaining from input goals to sufficient subgoals. Tactical theorem proving is also used in modern theorem provers, e.g., in the generic interactive theorem prover Isabelle (Paulson, 1994), in the verification system PVS (Dold, 1995), and in KIV (Heisel et al., 1988), the theorem proving shell underlying the program synthesis system IOSS that will be presented in Chapter 6. Unlike our system architecture, theorem proving systems like Isabelle or PVS usually do not maintain data structures equivalent to development trees, and so it becomes the users' responsibility to record their proof steps textually outside of the system.

Although the goal-directed, top-down approach to problem solving is common to both tactics and strategies, there are some important differences between them. Tactics are programs that implement "backward" application of logical rules. They are monolithic pieces of code, and all subgoals are set up at their invocation. Dependencies between subgoals can only be expressed by the use of *metavariables*, which allow one to leave "holes" in subgoals that can "filled" during proofs of other subgoals by unification on metavariables. Dependencies not schematically expressible using metavariables cannot be realized by tactics. Since tactics perform only goal reduction, in tactical theorem proving there are no equivalents of the *assemble* and *accept* functions of strategies. Such equivalents are not actually necessary for the tactic approach because problems and solutions are identical except for instantiation of metavariables. By contrast, problems and solutions of strategies may be expressed in different languages, and the composition of solutions by *assemble* may not be expressible schematically.

Another important difference between the strategy framework and tactical theorem proving concerns the role of search, on the one hand, and tacticals or strategicals, respectively, on the other. In tactical theorem proving, proof search is promising because the theorem is known and need not be constructed. The purpose of strategy-based development, however, is to construct an artifact of the software development process in the first place, and this renders searching a hopeless enterprise. Consequently, the OR and FAIL tacticals that are used to program search are unnecessary in the context of strategy-based development. In addition, the REPEAT construct is realized differently in the two frameworks. While a proper loop construct is necessary in search procedures, the REPEAT strategical performs only one step of a loop; its purpose is therefore to impose restrictions determining which strategies may actually be applied. Only the THEN tactical or strategical is useful in both paradigms. Its utility derives from the fact that larger steps can be performed in proofs and developments.

We conclude, therefore, that strategy-based development and tactical theorem proving – which are indeed based on similar ideas – are actually quite different in their practical applications.

5.6 Summary

The concept of a strategy is designed to provide machine support for the application of formal techniques in software engineering. Strategies serve to formally represent knowledge that is informally expressed as agendas or knowledge that is described in text books.The definition of strategies relies on the notion of a relation, which reflects the fact that different applications of the same strategy to a problem may lead to different subproblems and produce different solutions. Strategies do not necessarily permit full automation of a development task, but rather provide guidance for the development process and validation of the resulting product. Strategies also leave a considerable degree of freedom in their application.

The most important properties of the strategy framework are:

Methodological support

In using formal techniques, it is important not to leave developers with a mere formalism and no guidance how to use it. In contrast to other approaches, where tools deal with single documents and not with the process aspect of a development, the strategy framework aims at providing *methodological* support for software engineers. Making explicit not only dependencies, but also independencies, of problems in strategies allows for the greatest possible flexibility in the development process.

Genericity

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

Uniformity

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

Different instantiations of the strategy framework rely on the same principles. When conducting increasingly more development activities with strategies, software engineers can still use their previously acquired skills in strategy-based development; indeed, they need not learn entirely new ways to proceed in developing software. Finally, when we strive for integrated tool support for different software engineering activities, it is more promising to integrate different implemented instances of the strategy framework into a system with a wider range of support, than attempting to combine totally unrelated systems.

Reuse

Strategies make development knowledge explicit. Knowledge represented in terms of strategies can be communicated to others, can be enhanced according to new experiences and insights, and can be reused both in different developments and by different persons.

Machine support

The strategy framework provides concepts for machine-supported development processes, and the uniform modular representation of strategies makes these development concepts implementable. The general system architecture derived from the formal strategy framework gives guidelines for implementing 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 demonstrated by the existence of the implemented program synthesis system IOSS, which will be described in Chapter 6.

Documentation

The development tree not only supports the development process, but is also useful after the development is finished because it documents the manner in which solutions to input problems were developed and can so be used as a starting point for later modifications to the product.

Semantic properties

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

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

Formality

By defining strategies formally we were able to establish a theory of strategy-based problem solving. We proved that strategies and strategicals conform to our intuition about problem solving, and we also defined a problem solving algorithm that was proven to lead to acceptable solutions.

Stepwise automation

Introducing the concept of a heuristic function, and using such 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 the question of replacing user interaction by semi- or fully automatic procedures. Indeed, gradually automating software development processes amounts to making only local changes in heuristic functions.

Scalability

Using strategicals, increasingly more elaborate strategies can be defined, until strategies gradually approximate the sizes and kinds of development steps that are actually performed by software engineers. When combined with the availability of stepwise automation facilities, this contributes to the scalability of this particular approach to software development.

Customizibility

To incorporate a new method into a support system, the strategy base need only be extended by the new strategies. This involves only local changes, and does not affect existing components. Similar comments apply to the automation of parts of the development process.

More work is necessary if the notion of problem, solution, or acceptability has to be changed. In these cases, all strategies must be revised, but the clear modularization of strategy implementations still helps in identifying precisely which code requires changes.

A necessary prerequisite for any successful work with strategies is familiarity with the formalisms involved. To use an instantiation for specification acquisition, a good knowledge of the specification language to be used is necessary. To develop programs with IOSS, the user should be familiar with Gries' method for developing correct programs (Gries, 1981). But under no circumstances is it necessary that the user be a researcher in the area of formal techniques to profitably apply strategies.

5.7 Further Research

Future improvements will mainly concern the process of applying strategies and making the system architecture even more flexible and powerful. Of course, all further enhancements must be such that the acceptability of the generated solutions can always be guaranteed. In particular, we intend to work on the following topics:

Application of strategies defined with strategicals. The current definitions of strategy modules and of the abstract problem solving algorithms, and the current design of the system architecture assume that a strategy is defined as a set of constituting relations. To apply a strategy that is defined with strategicals, it would be necessary to "unfold" the definition of the strategy to obtain the set of its constituting relations. This unfolding would have to be done for each individual strategy, which is not economical. Instead, either algorithms should be developed that automatically do the unfolding, or the problem solving algorithms and the system architecture should be changed so that they can deal with strategies that are defined with strategicals, without the need to unfold the definitions of these strategies.

Incomplete solutions. The architecture as it is designed currently allows the users to reduce a problem only when all problems it depends on are completely solved. This style of problem solving is not always realistic. In specification acquisition, for example, it is unrealistic to assume that the subproblems generated by a strategy can be solved one after another (see Chapter 7). Hence, the process that implements problem solving with strategies must allow specifiers to work on problems even if the solutions on which they depend are not yet completely known. Technically, we can achieve this effect by propagating incomplete solutions. When the developer wants to work on a "later" subproblem, the *assemble* functions contained in the strategy modules (see Section 5.3.1) are executed, with dummy values in place of solutions, which have not yet been developed. As soon as a change in an earlier problem/solution occurs, the *assemble* functions must be re-executed to propagate the results of the changes into later problem definitions. When a subproblem is finally solved, both the *assemble* and *accept* functions must be executed.

Replay mechanisms. Sometimes a developer might want to change the solution to a previous subproblem, even if that solution is already complete and has been propagated into subsequent problems and solutions. Currently, changing completed solutions to subproblems is only possible by backtracking. But then, all intermediate steps that were performed after the revised solution was completed are lost and have to be repeated. To make revisions of solutions more comfortable, a replay mechanism is needed that automatically tries to repeat the development steps between the development of the revised solution and state of development before the revision.

Exploratory developments. The current architecture does not well support users who first want to try out several alternatives to solve a problem before deciding on the best way to solve it. Exploring different alternatives is only possible with backtracking or performing entirely different developments to solve the same problem. To support exploratory development, the system architecture could allow developers to introduce branches in the development, which would result in several "parallel" development trees for the same development. Finally, the developer could select the best development and discard the alternative development trees.

Reuse of developments. The strategy framework supports reuse of development knowledge that is represented in terms of strategies. To support also reuse of complete developments, we must develop mechanisms to combine previously generated development trees. Such combination mechanisms must ensure that, for example, different pieces of information that by chance have the same name are not identified.

Support more phases of software development. It is our aim to eventually support all phases of the software development process with strategies. Besides the four instantiations presented in this work, we specifically intend to investigate possible instantiations of the strategy framework for requirements engineering and for testing.

Integrate different instances of the strategy framework. For now, different instantiations of the strategy framework lead to different independent support systems for different software engineering activities. We are currently investigating ways in which different instances of the system architecture can be combined; first ideas are reported in Chapter 7. An integration of several instances of the strategy framework into a single support system would provide integrated tool support for larger parts of the software lifecycle.

Record design decisions. In the current system architecture, the development tree documents the development process. But it only documents *which* development steps have been taken, not *why* they were taken. It would be a simple adjustment to allow the users to give reasons for their choices of strategies in natural language, and store these reasons in the development tree.

Chapter 6

Strategy-Based Program Synthesis

This chapter presents an instantiation of the strategy framework that supports the synthesis of totally correct imperative programs from specifications that are expressed as formulas of first-order predicate logic. This instantiation is implemented in a prototype system IOSS (Integrated Open Synthesis System). The implementation of IOSS follows the system architecture presented in Section 5.4.

Other programming paradigms can also be supported by strategies: A different instantiation of the strategy framework, supporting the synthesis of functional instead of imperative programs, is described in (Heisel, 1994).

We first define the generic parameters of the strategy framework and then present some example strategies. After describing the implemented prototype system IOSS, we summarize and point out further research. Like Chapter 5, this chapter uses results from (Heisel, 1994; Heisel et al., 1995b; Heisel et al., 1995a; Heisel, 1996c).

6.1 Problems, Solutions, Acceptability and Explanations

In defining the generic parameters of the strategy framework, we use a Z-like notation, but we do not formalize the syntax and semantics of formulas and programs. For syntactic combinations of formulas, we use the subscript "s", e.g. $\wedge_s$ and $\Rightarrow_s$. To refer to the semantics of formulas, we use predicates like *valid* and *satisfiable*.

Problems

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

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

Solutions

Solutions are programs in an imperative Pascal-like language. Besides, solutions contain an *additional precondition* and an *additional postcondition*. (These conditions are additional to the pre- and postconditions of contained in a problem.) If the additional precondition is not equivalent to *true*, then the developed program can only be guaranteed to work if both the originally specified precondition and the additional precondition hold. The additional postcondition gives information about the behavior of the program, in that it describes *how* the goal is achieved by the program. To exclude trivial solutions, the additional precondition is required to be distinct from *false*.

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

Acceptability

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

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

The formula $pre \Rightarrow_s \langle prog\rangle post$ is a formula of dynamic logic (Goldblatt, 1982), a logic designed for proving properties of imperative programs. This formula denotes the total correctness of program *prog* with respect to precondition *pre* and postcondition *post*.

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

6.2 Strategies for Program Synthesis

We present four strategies. The first one replaces the goals of programming problems by stronger or equivalent ones. With the second, we can develop compound statements. The third can be used to develop loops. Using the THEN strategical, we combine these strategies to yield a fourth strategy, more powerful than either of its constituents, for developing loops together with their initialization.

The notation we use is semi-formal and resembles Z. The type *Value* denotes the disjoint union of the schema types *ProgProblem* and *ProgSolution*, whose members are denoted by *bindings*, i.e., by lists of pairs of the form *attribute* $\Rrightarrow$ *attribute value*.

6.2.1 The *strengthening* strategy

This strategy is used to incorporate knowledge about the data structures occurring in problem descriptions into the synthesis process. The idea is to replace the goal of a programming problem by a stronger one, i.e., a formula which entails the old goal in the model under consideration. Examples of the domain-specific knowledge that is used to strengthen goals are facts about the natural numbers, e.g., that the sum over an empty range of indices is zero.

The strategy produces one subproblem, which is a transformation of its input problem.

$$strengthening = \{str\_pr, str\_sol\}$$

where *str_pr* is defined by

$$
\begin{array}{l}
IA\ str\_pr = \{P\_init\} \\
OA\ str\_pr = \{P\_str, S\_str\} \\
str\_pr = \{\ t : scheme\ str\_pr \longrightarrow Value\ | \\
\quad \exists\, g, inv\_pr : First\_Order\_Formula\ | \\
\qquad free(inv\_pr) \cap t(P\_init).res = \emptyset) \land \\
\qquad valid(g \land_s inv\_pr \Rightarrow_s t(P\_init).goal) \bullet \\
\quad (\mathbf{let}\ var\_pr == (\mu\ var : First\_Order\_Formula\ | \\
\qquad\quad valid(t(P\_init).pre \Leftrightarrow_s inv\_pr \land_s var)) \bullet \\
\qquad t(P\_str) = \langle\, pre \Rrightarrow t(P\_init).pre, \\
\qquad\qquad goal \Rrightarrow g, \\
\qquad\qquad inv \Rrightarrow t(P\_init).inv, \\
\qquad\qquad res \Rrightarrow t(P\_init).res \cup (free(g) \setminus (t(P\_init).inp \cup t(P\_init).state)), \\
\qquad\qquad inp \Rrightarrow t(P\_init).inp, \\
\qquad\qquad state \Rrightarrow t(P\_init).state\,\rangle) \land \\
\quad t(S\_str)\ correct\_for\ t(P\_str)\}
\end{array}
$$

The constituting relation *str_pr* has *P_init* as its only input attribute. Its output attributes are the strengthened problem *P_str* and its solution *S_str*. We define constituting relations by stating conditions on their member tuples.

Note the existential quantifier in the definition. It indicates that external information is necessary to set up the problem for *P_str*. In the implemented strategy of IOSS, the user is asked to provide the stronger goal g. The system automatically determines the invariant parts *inv_pr* of the precondition (i.e., parts that contain no result variables), which may be necessary to show that solving the problem with goal g indeed suffices to solve the input problem of the strategy. The formula var_pr can be determined automatically from $t(P\_init).pre$ and inv_pr.

Hence, an existential quantifier in the definition of a constituting relation indicates that external information is necessary to define some of the attribute values. Sometimes, user interaction is needed to obtain the external information. In other cases, the external information can be computed automatically. The μ operator[1] indicates that the defined value can be determined uniformly from other known values.

The *strengthening* strategy generates the verification condition

$$g \wedge inv\_pr \Rightarrow t(P\_init).goal$$

This means that the conjunction of the new goal g and the selected parts inv_pr of the precondition must entail the original goal $t(P\_init).goal$ in the model under consideration. This model has to be specified by appropriate axioms. The rules of dynamic logic that guarantee correctness of the developed programs can be found in (Heisel, 1994).

The value of P_str depends of the value of P_init and the external information. Only the *goal* and *res* components of the value of P_str differ from those of P_init. The component $t(P\_str).goal$ is the new goal g, and any variables newly introduced in g are classified as result variables. The value of S_str must be acceptable for the value of P_str.

The second constituting relation of the *strengthening* strategy defines the final solution, i.e., the value of S_final.

$$\begin{array}{l} IA\ str\_sol = \{S\_str\} \\ OA\ str\_sol = \{S\_final\} \\ str\_sol = \{\, t : scheme\ str\_sol \longrightarrow Value \mid t(S\_final) = t(S\_str)\} \end{array}$$

The solution to the original problem coincides with the solution to the strengthened problem.

6.2.2 The *protection* strategy

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

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

where $prot\_first$ is defined by

[1] In contrast to the definition of the μ operator in Z, we do not the require the predicate to determine a unique value. We interpret μ as the Hilbert operator of higher-order logic, which selects one fixed value of the predicate's extension.

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

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

Again, an existential quantifier indicates that external information is necessary to set up the problem for *P_first*. In the implemented strategy of IOSS, the user is asked to indicate the goal for the first problem. The constituting relation *prot_second* is defined by

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

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

The result variables for *P_second* are the same as those of the original problem. Its input variables are the input variables of *P_init*, together with all variables newly introduced in solving *P_first* (these occur in *t*(*S_first*).*apo*). It is necessary to classify these variables because, in the definition of programming problems, we stated that each variable must be classified: $\textit{free}(\textit{pre} \wedge_s \textit{goal} \wedge_s \textit{inv}) \subseteq \textit{res} \cup \textit{inp} \cup \textit{state}$. The state variables again remain unchanged.

Not only is the solution *S_second* required to be acceptable for *P_second*, but the postcondition established by *S_first* must also entail the additional precondition of *S_second*. Note that this second condition is a local acceptability condition for *S_second*. If it is not fulfilled, then a precondition, which is necessary for the program contained in *S_second* to establish the postcondition stated in *P_second*, cannot be guaranteed, and *S_second*, although acceptable for *P_second* in isolation, cannot be part of a solution to the original problem *P_init*.

The constituting relation *prot_sol* specifies the way in which the final solution to a problem is assembled from the solutions of its subproblems. Here, the final program is the sequential composition of two programs developed in solving subproblems of the original problem.

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

6.2.3 The *loop* strategy

This strategy enables the construction of a **while** loop. It is applicable only when the goal contains no quantifiers because the goal serves as the termination test of the loop. The *loop* strategy generates exactly one subproblem and is given by

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

where *loop_body* is defined by

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

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

The goal of problem *P_loop* is then to decrease the bound function, i.e. to make progress towards termination of the loop, while maintaining the invariant. To record the value of the bound function, a new state variable t_0 is introduced. The input and result variables are the same for *P_loop* as for *P_init*.

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

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

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

6.2.4 A Combined Strategy

Gries' approach to the development of correct programs (Gries, 1981) deals primarily with the development of loops. For **while** loops, the approach can be expressed by the agenda shown in Table 6.1. Let a precondition P and a postcondition R be given. Recall that the formula $P \Rightarrow \langle prog \rangle R$ of dynamic logic denotes the total correctness of program *prog* with respect to the precondition P and the postcondition R. The program developed with the this agenda has the form *init*; **while not** C **do** *body* **od**. The verification conditions given in the agenda guarantee that the developed loop is totally correct with respect to the given pre- and postconditions.

Combining the strategies introduced in the previous sections, we can formalize the agenda of Table 6.1 as a strategy. First, the *strengthening* strategy must be applied to replace the goal of the original problem with the loop invariant and the negation of the loop condition. The conditions I and C must be supplied as external information, where, for the development of the invariant, the heuristics given by Gries (Gries, 1981) can be employed. Second, the *protection* strategy must be applied. The first statement of the compound generated by the *protection* strategy is the initialization of the loop that establishes the invariant. The second part of the compound consists of the loop itself which is developed with the *loop* strategy.

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

$$while = \text{THEN}(strenthening, P\_str, \text{THEN}(protection, P\_second, loop))$$

where *P_str* is the only subgoal generated by the *strengthening* strategy, and *P_second* is the problem to develop the second part of the compound generated by the *protection* strategy.

No.	Step	Verification Condition
1	Develop a loop invariant I by weakening the postcondition R appropriately.	$R \Rightarrow I$
2	Develop a loop condition C such that upon termination the desired result is true.	$\neg\, C \wedge I \Rightarrow R$
3	Develop the initialization *init* of the loop such that it establishes the invariant.	$P \Rightarrow \langle init \rangle I$
4	Develop a bound function *bf* on a set with a well-founded ordering, such that *bf* is bounded from below as long as the loop has not terminated.	$well\_founded\_ordering(0, \prec) \wedge (C \Rightarrow 0 \prec bf)$
5	Develop the loop body *body* such that the bound function is decreased while the invariant is maintained.	$I \Rightarrow \langle t0 := bf;\ body \rangle (I \wedge bf \prec t0)$

Table 6.1: Agenda for developing **while** loops

Strategies defined with THEN perform larger development steps than their component strategies. Thus, strategies can gradually approximate the complexity of development steps performed by human developers in practice. More strategies for program synthesis can be found in (Heisel, 1994).

6.3 IOSS: An Implemented Program Synthesis System

The program synthesis system IOSS is a research prototype that was built to validate the concept of strategy and the system architecture developed for their machine-supported application. Currently, it supports the application of the program development methods described in (Gries, 1981) and (Dershowitz, 1983). Because the interfaces of strategy modules are presented uniformly, Gries' and Dershowitz' methods can be combined freely.

IOSS is an instantiation of the architecture described in Section 5.4, which uses the instantiation given in Section 6.1. The basis for the implementation of IOSS is the *Karlsruhe Interactive Verifier* (KIV), a shell for the implementation of proof methods for imperative programs (Heisel et al., 1988).

After giving an overview of the strategy base of IOSS, we describe its graphical user interface and show how existing software tools have been reused and integrated to implement IOSS.

6.3.1 The Strategy Base

A number of interactive, semi-automatic and fully automatic strategies have been implemented. In the current version, they are oriented toward programming language constructs.

Three strategies solve a problem directly: one for developing the empty program `skip` (*skip* strategy), two for developing assignments (*manual assignment* and *automatic assignment* strategy).

Two strategies can be applied to modify a problem: the *strengthening* strategy, which we presented in Section 6.2.1, and the *state variable* strategy, which introduces a new state variable for some result variable.

Three strategies are available for developing compound statements: one corresponds to the rule for compound statements in the Hoare calculus (*intermediate assertion* strategy), the two others are based on Dershowitz' approach for conjunctive goals (Dershowitz, 1983). The *disjoint goal* strategy can be applied if the goal can be divided into two independent subgoals. Two subgoals are independent if the set of result variables that must be changed to achieve one of the subgoals are disjoint from the set of result variables that must be changed to achieve the other subgoal. The *protection* strategy presented in Section 6.2.2 can be applied when the subgoals are not independent as required for the disjoint goal strategy.

Two strategies can be used to develop conditionals: the *conditional* strategy reflects the rule for conditionals of the Hoare calculus, the *disjunctive conditional* applies if the goal is of disjunctive form.

Since the *while* strategy defined in Section 6.2.4 is not yet implemented, loops must be developed using the *loop* strategy of Section 6.2.3 in combination with the *strengthening* and *protection* strategies.

Higher-level strategies like strategies for the development of divide-and-conquer algorithms or re-usable procedures have been defined, but are not yet implemented. A complete description of all defined and implemented strategies can be found in (Heisel, 1994; Heisel, 1992).

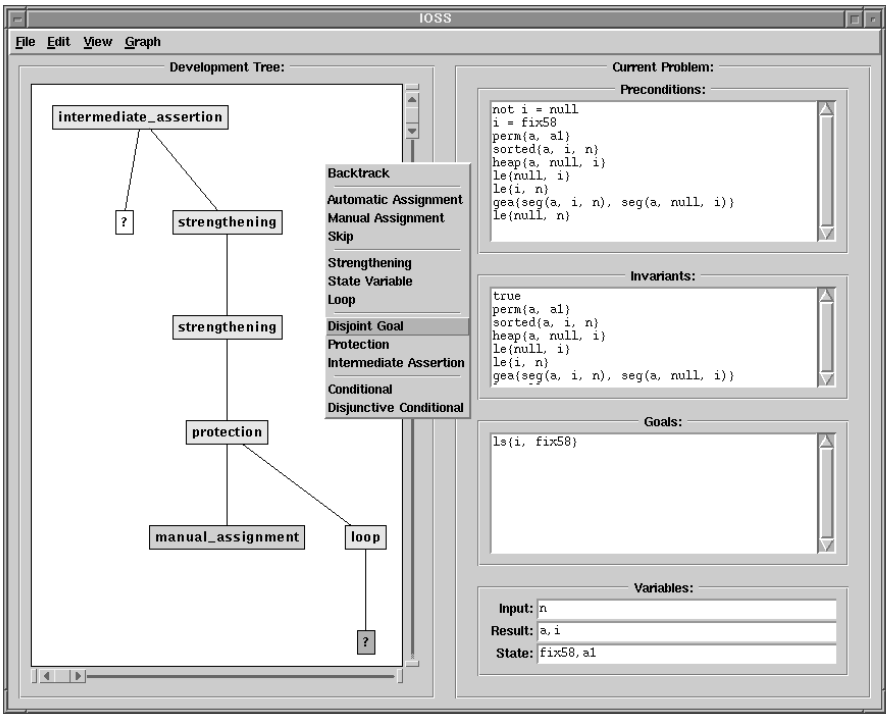
6.3.2 The Interface

Figure 6.1 shows the graphical user interface of IOSS. The main window displays the current development task, represented by the development tree on the left-hand side of the window, and the current programming problem, which appears on the right-hand side of the window. The tree graphically represents the process and the state of the program development. Each node is labeled with the name of the strategy, which has been applied to it. The state of the node is color coded, showing at a glance whether it is reducible, or solved, etc.

A node is selected by simply clicking it with the mouse. If that node is reducible, it becomes the current node, and its problem specification is shown on the right-hand side of the window. Any node can be selected for the purpose of inspecting it, but only reducible nodes can become the current node. A node can be inspected via the View menu. A separate window pops up for each node, and several nodes can be inspected at the same time.

The explanation (i.e., proof) for the synthesized solution – as far as it has been constructed – is accessible via the View menu, too. IOSS combines the explanations for each strategy application to form a coherent proof that – once the development process is completed – verifies the developed program.

The strategy base of IOSS is accessible via the Edit menu. A strategy is applied to the current node by invoking the respective menu entry. In Figure 6.1 the menu is shown in the center of the window. It is possible for any menu to be kept on the screen at an arbitrary position. This allows developers to quickly access frequently used features such as the strategy base. Whenever a strategy requires user input, the user is prompted for it in a window. IOSS also provides features to manipulate the graph. The user can, for example, re-scale the tree or hide subgraphs.



Preconditions:

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

Invariants:

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

Goals:

```
ls{i, fix58}
```

Variables:

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

Figure 6.1: The IOSS interface

6.3.3 Experience with Re-Use and Integration

Technically speaking, IOSS is not one single program. It makes use of a number of software packages to realize what the user perceives as IOSS. The implementation of IOSS has been carried out in two steps. First, the kernel system has been implemented as an instance of the architecture described in Section 5.4. Second, a graphical user interface has been designed and implemented on top of the kernel system.

The Kernel System

The *Karlsruhe Interactive Verifier* (KIV) (Heisel et al., 1988) is a shell for the implementation of proof methods for imperative programs. It provides a functional *Proof Programming Language* (PPL) with higher-order features and a backtrack mechanism. Assertions about programs can be formulated in dynamic logic (Heisel et al., 1989). The language for programs itself is a Pascal-like language with while-loops and recursive procedures. KIV serves for the implementation of the IOSS kernel, its data structures, and strategy modules.

Strategies are implemented as collections of PPL functions in separate modules, and, as a result, new strategies can be incorporated into IOSS in a routine way. A template file for new strategies currently supports incorporating of new strategies; for the future, we envision tool support relieving the implementor of everything other than handling the characteristics of the newly implemented strategy.

A severe restriction of KIV is its command-line interface. There is no reasonable way to bring into effect the potential to inspect the state of development and to take advantage of the freedom of choice provided by the architecture. A more sophisticated was needed to fully exploit the benefits of the strategy framework and the associated system architecture.

The Graphical Interface

Since we had very limited resources in terms of person-power to realize the interface, we decided to rely as much as possible on existing software packages and toolkits. The interface was to be built with minimal changes to the kernel system. We needed a means to transfer data from KIV to whichever interface system we would use. For the visualization of the state of development, we needed a graph layout system. Moreover, we wanted to avoid programming on a level such as the X Window Toolkit, since this is a tedious, time-consuming task.

With the following packages we found just what we needed:

`Tcl` – A simple, extensible scripting language providing generic programming facilities. Each application can implement new features as `Tcl` commands (Ousterhout, 1994).

`Tk` – An extension to `Tcl` providing a toolkit for the X Window System. `Tk` extends the core `Tcl` facilities by commands for building user interfaces. It hides much detail C programmers must address when constructing a user interface (Ousterhout, 1994).

`expect` – An extension to `Tcl/Tk` designed to control interactive programs using standard terminal I/O. For the controlled programs, `expect` takes over the part of the user “typing” commands and interpreting output (Libes, 1991).

`TkSteal` – An extension to `Tk` to integrate stand-alone X applications in a `Tk`-built interface (Delmas, 1994).

`daVinci` – A generic visualization system for directed graphs (Fröhlich and Werner, 1995).

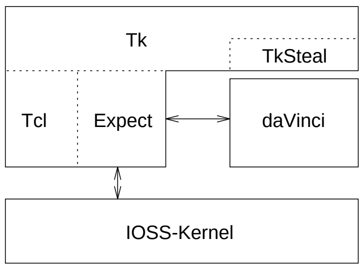


Figure 6.2: System integration for the IOSS interface

Figure 6.2 illustrates the integration of these packages to construct the graphical interface for IOSS. `expect` controls the command-line interface of KIV and the application interface of `daVinci`. `TkSteal` provides “interface sugar” for the graphical interface. It integrates `daVinci` with the other parts of the IOSS interface.

We think it is remarkable how little effort was required to build the interface. It took only one person-month to build it in its current shape. Only 800 lines of code needed to be written in `Tcl`, 116 lines of code of additional code had to be written in `PPL`.

For a more complete description of IOSS, which also contains example developments, the reader is referred to (Heisel et al., 1995a).

6.4 Related Work

The strategy framework in general, and IOSS in particular, make it possible to integrate a variety of methods for program synthesis, provided they can be expressed in its basic formalism. The synthesis systems CIP (CIP System Group, 1987), PROSPECTRA (Hoffmann and Krieg-Brückner, 1993) and LOPS (Bibel and Hörnig, 1984), in contrast, are all designed to support specific methods. Their authors did not intend to integrate these methods with other ones, nor are these systems customizable. Moreover, the support of activities other than program synthesis was not a design goal for any of these systems.

The approach underlying KIDS (Smith, 1990) is to fill in algorithm schemas by constructive proofs of properties of the schematic parts. This is achieved using highly specialized code, called *design tactics*, of which at least one is defined for each schema. There is no general concept of a design tactic, and no notion of how to incorporate a new one into the system. Information about the development process is maintained implicitly, so that, working with KIDS, it is hard to keep track of where one is in a development. There is a logging facility and a replay facility, but these provide no possibility of browsing the state of development. Since design tactics are linearly programmed, there is no way to change the order of independent design steps or to "interleave" the applications of tactics.

6.5 Summary

In this chapter, we have presented a first instantiation of the strategy framework that supports the synthesis of totally correct imperative programs. This instantiation demonstrates that the strategy framework can be profitably employed in program synthesis. Moreover, we have described a prototype system that implements the instantiation for program synthesis. This prototype shows that carrying out development tasks with strategies is feasible.

IOSS in its current version is useful for teaching students the systematic development of provably correct programs. However, the currently available strategies are relatively low-level. This makes program synthesis with IOSS a time-consuming and highly interactive task. Consequently, IOSS is not yet applicable for the development of industrial-scale programs.

IOSS was built to serve as a proof of concept, rather than as a full-fledged development tool. In particular, the following facts became apparent:

- Because of the uniform modular representation of strategies, an integration of different methods for program synthesis becomes possible. In IOSS, the methods of Gries and Dershowitz can be combined freely.

- The strategy approach leads to open systems that can be improved gradually. Such an improvement can be the routine incorporation of new strategies, the replacement of an interactive heuristic function by a semi- or fully automatic one, or the combination of existing strategies into more powerful ones, using strategicals.

- Implemented systems that support an instance of the strategy framework can be built with relatively little effort, using freely available software packages.

6.6 Further Research

IOSS has the potential to be developed further into a tool that can be used to tackle more realistic program development tasks than this is currently the case. The following improvements would make IOSS considerably more powerful:

Selection of strategies. More support could be provided for the users of IOSS in selecting the strategy to be applied to the current problem. Whereas positively proposing candidate strategies seems a very ambitious aim, it is easier to exclude those strategies that cannot be applied to the current problem, e.g., because they require a problem of a specific syntactic form that does not match the form of the current problem.

Automation. Although the strategy framework provides a high potential for automation, most of the strategies available in IOSS are interactive. To better exploit the potential for automation, we need to build up libraries that contain theories for frequently used data structures. KIDS (Smith, 1990), for example, heavily relies on such libraries.

More strategies. More powerful strategies should be incorporated into IOSS. Candidates are strategies that are equivalent to the design tactics available in KIDS. Strategies for the synthesis of divide-and-conquer algorithms are already defined.

Replay mechanisms. It happens that during the synthesis of a program, different parts of the program are developed in almost the same manner. We need mechanisms that support the adaptation and reuse of sequences of development steps.

Programming language. The programming language supported by IOSS should be made more expressive, e.g., by incorporating powerful procedural constructs, or introducing dynamic data structures, such as linked lists.

Proof support. The theorem prover of KIV, which is used to prove the verification conditions generated by IOSS, is not very sophisticated. For instance, there is no built-in theory of ordering relations. The prover should be parameterized with theories, and rewriting techniques should be incorporated.

Safety invariants. When we implement software for safety-critical systems as specified in Chapter 3, then safety cannot be guaranteed in the intermediate states that occur when a program that implements a system operation is executed. The specification only states that the state before and the state after execution of the system operation are safe. This situation can be improved under the condition that sequences of assignments are considered to be sufficiently fast. In this case, we can require a "safety invariant" to hold before and after each sequence of assignments. Then the system can be in an unsafe state only for the time that is needed to execute the longest assignment sequence occurring in the implementation. With little effort, IOSS can be extended to deal with such safety invariants.

Chapter 7

Strategy-Based Specification Acquisition

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

In Chapter 2, we have presented an agenda for specification acquisition, which integrates all activities that must be carried out to develop a specification smoothly into traditional software processes. Some of these activities (Steps 1, 2, and 4) are not carried out with formal techniques; others (Steps 4 and 5) can only be performed after the specification has been developed. Therefore, this chapter concentrates on Step 3 of the agenda of Chapter 2, namely the transformation of the requirements into a formal specification.

The pragmatic relaxations of specification discipline proposed there are reflected in the definitions of problems, solutions, and acceptability of the instantiation: Chapter 8 presents an instantiation of the strategy framework for the combination of Z and real-time CSP defined in Chapter 3. If restrictions of the specification language are to be ignored, we can, for example, define acceptability of Z specifications as type correctness because it can be checked by tools. Then, the specification of Section 2.4.3 is acceptable. Leaving out details, on the other hand, does not show up in the definition of an instantiation of the framework, but only in individual specification developments.

In this chapter, we first introduce the concept of a specification style in Section 7.1. Specification styles are different manners in which specifications can be developed. We then present the definitions of problems, solutions, and acceptability we use to instantiate the strategy framework for specification acquisition in Section 7.2. In Section 7.3, we define strategies associated with the styles introduced in Section 7.1. These are used in an example, where we present the development of a specification of the Unix file system in Section 7.4. Finally, in Section 7.5, we show how Z specifications developed with the instance of the strategy framework of this chapter can be transformed into IOSS programming problems. As usual, we close with a discussion of related work, a summary of the achieved results, and directions for further research. The instantiation of the framework and some of the strategies presented here are taken from (Heisel, 1996c). The concept of style is explained in more detail in (Souquières and Heisel, 1996). The example and some reflections on specification languages can be found in (Heisel, 1995b).

7.1 The Concept of a Specification Style

Different specification languages are distinguished by the language constructs they offer to their users. For each specification language, we can find requirements on or aspects of systems we want to specify that are supported very well and others that can be expressed only in a clumsy way or not at all. For example, algebraic languages do not support the specification of state-based systems very well; on the other hand, the generic constructs offered by the language Z leave much to be desired. In this way, specification languages encourage the use of some constructs and discourage the use of others. As a result, they implicitly represent certain *specification styles*, because specifiers proceed differently, according to the specification language they use.

In contrast to this situation, we strongly advocate orienting the development of a specification on the *problem*, not on the specification language that is used. When developing a formal specification, we should ask ourselves the following questions:

- Does the system to be built have a global state that is changed by some operations?
- Is it suitable to abstractly describe properties of (parts of) the system to be specified?
- Is it possible to combine and adjust existing specifications to obtain a specification for (parts of) the new system?

Depending on the answers to these questions, a specifier will follow different paths to develop a specification. We propose to define these different approaches to developing specifications as specification styles and support them in a systematic way, i.e., by agendas and strategies. Such definitions make styles *explicit*, instead of representing them implicitly by specification languages. The previous questions correspond to the *state-based*, *algebraic*, and *reuse* styles.

A specification style describes a certain "spirit" in which a specification is set up. For example, the aim of the reuse style is to reuse specifications contained in a library whenever possible. Of course, we usually cannot expect to set up a new specification exclusively using existing specifications of a library. Library items will have to be modified, and certain parts will have to be developed from scratch. Hence, a style is nothing strict, but it will have to be combined with other styles. Styles are used locally, i.e., even within one development, one switches between different styles. It is therefore not reasonable to identify styles with specification languages.

Specification styles can be described as sets of strategies. Strategies that belong to a particular style are to a large extent independent of the specification language to be used. That is, the strategies associated with a particular style will define the same number of subproblems with the same dependencies for the same purpose independent of the choice of the specification language; only the specification expressions that are generated as solutions will look different. In (Souquières and Heisel, 1996), we show that, in performing the same steps, we can obtain "equivalent" specifications in different languages. This shows that, to a large extent, the development process can be driven exclusively by the problem.

In the following, we define the notions of problems, solutions, and acceptability for specification acquisition, and then present strategies that are associated with different styles.

7.2 Problems, Solutions, and Acceptability

The instantiation of the strategy framework which we now present can be used in developing specifications in Z. This instance integrates well with the instance for IOSS, because Z supports the explicit modeling of states. Z specifications are usually implemented in imperative languages, such as the one used in IOSS, and Z operation schemas are easily transformed into programming problems for IOSS (see Section 7.5).

In contrast to program synthesis, where problems and solutions are purely formal objects, specification acquisition transforms informal requirements into formal specifications. A problem to be solved will therefore contain a natural language description of the purpose of the specification to be developed.

In addition, the successive development of a specification requires knowledge about the parts of the specification that have already been developed. Since problems should contain all information needed to solve them, problems must contain expressions of the chosen specification language – in our case, Z.

Finally, a problem contains a *schematic* Z expression that can be instantiated with an appropriate concrete Z expression. The schematic Z expression specifies the syntactic class of the specification fragment to be developed, as well as how the fragment is embedded in its context (see e.g. Section 7.3.2 below). These considerations lead us to the basic types

$$[SynZ, Text, SchematicZ]$$

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

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

We will use the following syntactic classes whose names are self-explanatory. The class *specification* is the most general one.

$$\begin{array}{|l}
specification, free\_type, ax\_def, \\
schema, schema\_list, \\
declaration\_list, predicate, ident : SyntacticClass
\end{array}$$

Each schematic Z expression is associated with the syntactic class of Z expressions with which it can be instantiated. The function `NL` concatenates two Z expressions. As in the Z reference manual, it means "new line". Since concatenating two arbitrary Z expressions does not always yield a syntactically correct Z expression, the function `NL` is necessarily partial. The empty specification ϵ is a neutral element with respect to `NL`.

$$\begin{array}{l}
syn\_class : SchematicZ \longrightarrow SyntacticClass \\
instantiate : SchematicZ \times SynZ \rightarrowtail\!\!\!\!\!+ SynZ \\
\_\mathtt{NL}\_ : SynZ \times SynZ \rightarrowtail\!\!\!\!\!+ SynZ \\
\hline
\forall schem\_expr : SchematicZ \bullet \forall v : syn\_class\ schem\_expr \bullet \\
\quad (schem\_expr, v) \in \operatorname{dom} instantiate \\
\forall spec : SynZ \bullet \\
\quad (spec, \epsilon) \in \operatorname{dom} \mathtt{NL} \Rightarrow spec\ \mathtt{NL}\ \epsilon = spec\ \land \\
\quad (\epsilon, spec) \in \operatorname{dom} \mathtt{NL} \Rightarrow \epsilon\ \mathtt{NL}\ spec = spec
\end{array}$$

A specification problem consists of the parts mentioned before, i.e., a requirement, expressed in natural language, the parts of the specification already developed, and a schematic Z expression. As an integrity condition, we require that each Z expression belonging to the syntactic class associated with the schematic Z expression can be combined with the specification already developed.

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

Solutions are Z expressions:

$SpecSolution == SynZ$

A solution *sol* 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 specification.

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

In practice, it is useful to define *SemZ* to be the set of Z expressions which are accepted by available tools, such as the Fuzz type checker (Spivey, 1992a).

7.3 Strategies for Specification Acquisition

Apart from some general-purpose strategies, we present strategies for the state-based, algebraic, and reuse styles. As usual, we use a semi-formal Z-like notation to describe these strategies, neither formalizing the syntax and semantics of Z, nor giving definitions for all functions and predicates we use. The type *Value* denotes here the disjoint union of the schema types *SpecProblem* and *SpecSolution*, and its members are denoted by bindings, as in Chapter 6.

7.3.1 General-Purpose Strategies

These strategies are independent of a particular specification style. They are needed in almost every specification development, because they solve a specification problem directly.

The *terminate* Strategy

This strategy, which does not generate any subproblems, allows the user to type in some specification text.

$$
\begin{array}{l}
\mathit{terminate} = \{\mathit{term\_sol}\}, \text{ where} \\
\mathit{IA}\ \mathit{term\_sol} = \{\mathit{P\_init}\} \\
\mathit{OA}\ \mathit{term\_sol} = \{\mathit{S\_final}\} \\
\mathit{term\_sol} = \{\ t : \mathit{scheme}\ \mathit{term\_sol} \longrightarrow \mathit{Value}\ | \\
\qquad\qquad \exists\, \mathit{sol} : \mathit{SpecSolution} \mid \mathit{sol}\ \mathit{spec\_acceptable\_for}\ t(\mathit{P\_init}) \bullet t\ \mathit{S\_final} = \mathit{sol}\}
\end{array}
$$

As in Chapter 6, the existential quantifier indicates that external information is needed to determine the value $t\ \mathit{S\_final}$. In an implementation, the user would be asked to type in a specification fragment. The condition $\mathit{sol}\ \mathit{spec\_acceptable\_for}\ t(\mathit{P\_init})$ ensures that only specification fragments that are admissible in the context specified by the input problem are accepted by the strategy.

The *empty* Strategy

This strategy generates the empty specification ϵ. It will be used to terminate strategies that are defined with the REPEAT or LIFT strategicals, and to skip optional parts of specifications.

$$
\begin{array}{l}
\mathit{empty} = \{\mathit{empty\_sol}\}, \text{ where} \\
\mathit{IA}\ \mathit{empty\_sol} = \{\mathit{P\_init}\} \\
\mathit{OA}\ \mathit{empty\_sol} = \{\mathit{S\_final}\} \\
\mathit{empty\_sol} = \{\ t : \mathit{scheme}\ \mathit{empty\_sol} \longrightarrow \mathit{Value}\ | \\
\qquad\qquad t\ \mathit{S\_final} = \epsilon \wedge \epsilon\ \mathit{spec\_acceptable\_for}\ t(\mathit{P\_init})\}
\end{array}
$$

7.3.2 Strategies for the State-Based Style

The state-based style of specification should be applied if a state-based system has to be specified. Here, we must specify the legal states of the system and the operations that define how the system state may evolve.

We present three strategies associated with the state-based style: the *state_based* strategy, which defines a top-level method for specifying state-based systems, the *develop_schema* strategy, which is used to develop a single schema, and a strategy for developing lists of schemas, which is defined in terms of the strategicals LIFT and REPEAT.

The manner of specifying a system that is captured in the *state_based* strategy is independent of the specification language that is used. In (Souquières and Heisel, 1996; Heisel, 1995b) it is shown how the same manner of specification can be applied using an algebraic specification language. The two other strategies, however, refer to the schema construct of Z. They would have no counterpart in an instantiation of the strategy framework for specification acquisition that uses a specification language other than Z.

The *state_based* strategy

One of the factors that contribute to the relatively widespread acceptance of Z in industry is the existence of a method (Potter et al., 1991) that gives guidance for its use. This method recommends that the following process – which we express in terms of the agenda shown in Table 7.1 – is followed when developing Z specifications. However, as already noted, the method is also useful when a different language is used.

No.	Step	Validation Conditions
1	Develop the global definitions.	
2	Develop the global state and the initial state.	There must exist an initial state.
3	Develop the the system operations.	No operation has precondition *false*.

Table 7.1: Agenda for developing state-based systems

To perform Step 3 of this agenda, i.e., to develop the system operations, another agenda, shown in Table 7.2, can be given. If Step 3 is performed with the agenda of Table 7.2, its associated validation condition is automatically satisfied.

No.	Step	Validation Conditions
1	Develop the operations for the normal case.	
2	Develop the operations for error cases.	
3	Define total operations, combining the operations for the normal and the error cases.	Each so defined operation has precondition *true*.

Table 7.2: Agenda for developing operations

The *state_based* strategy, which we now define, captures the top-level agenda of Table 7.1 for developing Z specifications. To be more general, we allow a fourth step that can be used to complete the specification. Since the *state_based* strategy is a top-level strategy, its input problem must permit the development of expressions of the syntactic class *specification*. Hence, we have

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

where *global_defs* is defined by

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

Using the concatenation function for text, denoted ";", a natural-language text that describes the purpose of the new subproblem *P_global* is added to the informal requirements component *req*. The schematic expression *to_develop* is denoted by sp : *specification*. This notation means that a Z expression belonging to the syntactic class *specification* must be developed, and the instantiation function is the identity. The constituting relation *system_state* is defined by

$$\begin{aligned}
&IA\ system\_state = \{P\_init, S\_global\}\\
&OA\ system\_state = \{P\_state, S\_state\}
\end{aligned}$$

$$\begin{aligned}
&system\_state = \{\, t : scheme\ system\_state \longrightarrow Value \mid\\
&\quad t(P\_state) = \langle\, req \Rrightarrow t(P\_init).req\,; \text{“specify global system state”},\\
&\qquad\qquad context \Rrightarrow t(P\_init).context\, \mathtt{NL}\, t(S\_global),\\
&\qquad\qquad to\_develop \Rrightarrow \mathsf{state\_def} : schema\_list\rangle \wedge\\
&\quad t(S\_state)\ spec\_acceptable\_for\ t(P\_state) \wedge\\
&\quad t(S\_state) \neq \epsilon\}
\end{aligned}$$

To define *P_state*, the global definitions *S_global* are added to the *context* component of *P_init*. The system state must be defined as a non-empty list of schemas. The constituting relation *system_ops* is defined by

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

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

$$\begin{aligned}
&IA\ other\_defs = \{P\_init, P\_ops, S\_ops\}\\
&OA\ other\_defs = \{P\_other, S\_other\}\\
&other\_defs = \{\, t : scheme\ other\_defs \longrightarrow Value \mid\\
&\quad t(P\_other) = \langle\, req \Rrightarrow t(P\_init).req\,; \text{“other definitions”},\\
&\qquad\qquad context \Rrightarrow t(P\_ops).context\, \mathtt{NL}\, t(S\_ops)\\
&\qquad\qquad to\_develop \Rrightarrow \mathsf{others} : specification\rangle \wedge\\
&\quad t(S\_other)\ spec\_acceptable\_for\ t(P\_other)\}
\end{aligned}$$

No assumptions can be made on the other definitions necessary to complete the specification. Hence the specification fragment that can be developed to solve *P_other* may have the syntactic class *specification*, and no additional acceptability conditions for *S_other* besides *spec_acceptable_for* can be stated.

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

$$\begin{array}{l}
\mathit{IA}\ \mathit{state\_based\_sol} = \{\mathit{S\_global}, \mathit{S\_state}, \mathit{S\_ops}, \mathit{S\_other}\} \\
\mathit{OA}\ \mathit{state\_based\_sol} = \{\mathit{S\_final}\} \\
\mathit{state\_based\_sol} = \{\ t : \mathit{scheme}\ \mathit{state\_based\_sol} \longrightarrow \mathit{Value} \mid \\
\quad t(\mathit{S\_final}) = t(\mathit{S\_global})\ \mathtt{NL}\ t(\mathit{S\_state})\ \mathtt{NL}\ t(\mathit{S\_ops})\ \mathtt{NL}\ t(\mathit{S\_other}) \wedge \\
\quad t(\mathit{S\_global}) \text{ does not contain state or operation schemas} \wedge \\
\quad t(\mathit{S\_state}) \text{ contains a state schema } S \text{ that is not imported by any} \\
\qquad \text{other schema in } t(\mathit{S\_state}) \text{ and an initial state schema for } S \wedge \\
\qquad \text{the set of initial states is non-empty } \wedge \\
\quad t(\mathit{S\_ops}) \text{ contains at least one operation schema} \wedge \\
\qquad \text{none of the operations defined in } t(\mathit{S\_ops}) \text{ have precondition } \mathit{false}\}
\end{array}$$

A schema S is a *state schema* if it has neither inputs nor outputs if and there are other schemas which import it. There must not be variable declarations of the kind $x : S$, i.e. declaration of variables that have the schema type S. Note that this condition can be checked only in the context of the other parts of the specification. A schema is an operation schema if it imports a state schema with the $'$, Δ, or Ξ notation.

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

The *define_schema* Strategy

This is a simple strategy, which can be used to define a schema in two steps: first the declaration part and then the predicate part are defined. The *define_schema* strategy requires that solutions of the syntactic class *schema* are permitted. It is given by

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

where *define_decls* is defined by

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

Here, we have used the function *make_schema* instead of the graphical schema notation. The schematic expression $t(\mathit{P\_decls}).\mathit{to\_develop}$ captures the information that a Z expression belonging to the syntactic class *declaration_list* must be developed, and that the instantiation function, which embeds the developed solution *S_decls* into a a schema, is *make_schema*. This function is applied to the name n of the schema, which must be provided as external information, the developed expression *S_decls* and the predicate *true*. The trivial predicate *true* must be used as long as the predicate part of the schema is not developed. The constituting relation *define_pred* is defined by

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

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

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

An Iterative Strategy

The subproblems *P_state* and *P_ops* generated by the *state_based* strategy can be solved by repeated application of the strategy *define_schema*. To relieve strategy users of the task of selecting the same strategy several times in a row, we define a new strategy that generates lists of schemas instead of just one schema. According to the definitions of Section 5.2, we can define

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

where *p_rep* is a problem attribute newly introduced by Lift and *empty* is the terminating strategy that generates the empty specification ϵ, see Section 7.3.1. The other arguments of Lift are defined as follows:

$$
\begin{array}{l}
p\_down == (\lambda\, pr : SpecProblem \mid syn\_class(pr.to\_develop \supseteq schema\_list) \bullet \\
\qquad\qquad \langle req \Rrightarrow pr.req, context \Rrightarrow pr.context, to\_develop \Rrightarrow \mathsf{sch} : schema\rangle)
\end{array}
$$

$$
\begin{array}{l}
p\_combine == (\lambda\, pr : SpecProblem; sol : SpecSolution \mid \\
\qquad\qquad syn\_class(pr.to\_develop) \supseteq schema\_list \land sol \in schema \bullet \\
\qquad\qquad \langle req \Rrightarrow pr.req\, ; \text{“define more schemas”}, \\
\qquad\qquad context \Rrightarrow pr.context\, \mathtt{NL}\, sol, \\
\qquad\qquad to\_develop \Rrightarrow pr.to\_develop\rangle)
\end{array}
$$

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

where *p_down* converts the problem of defining a list of schemas into the problem of defining a single schema, the function *p_combine* incorporates a developed schema into the *context* part of a problem, and the function *s_combine* concatenates two specifications, thereby allowing the concatenation of a given schema with an existing list of schemas.

The requirements for the arguments of Lift (see page 5.2.3) are fulfilled: First, the function *p_combine* is injective. Secondly, if we first develop a schema S, then this schema is added to the *context* component of the original problem *P_init* by the function *p_combine*. When we then develop a schema list *sl* that is acceptable for the combined problem *p_combine*(*P_init*, S), the definition of the predicate *spec_acceptable_for* gives us

$$(\mathit{p\_combine}(\mathit{P\_init}, S)).\mathit{context}\,\mathtt{NL}\,\mathit{sl} \in \mathit{SemZ}$$

Since $(\mathit{p\_combine}(\mathit{P\_init}, S)).\mathit{context} = \mathit{P\_init}.\mathit{context}\,\mathtt{NL}\,S$, we have

$$\mathit{P\_init}.\mathit{context}\,\mathtt{NL}\,S\,\mathtt{NL}\,\mathit{sl} \in \mathit{SemZ}$$

and hence (because NL is associative) *s_combine*(S, *sl*) *spec_acceptable_for P_init*.

Defining the strategy *define_schema_list* with strategicals has two advantages over defining it from scratch, namely that the existing strategy *define_schema* is reused, and that the user need not manually select the same strategy over and over again. The only possible development steps left after application of *define_schema_list* are to develop one more schema or to terminate the iteration.

7.3.3 Strategies for the Algebraic Style

The algebraic style of specification should be applied if a system or aspects of a system are to be defined in an abstract way, by stating properties. The algebraic style supports the definition of data types and functions on these types. Functions are defined by giving their signatures and axiomatizing their properties.

In Z, this style of specification is associated with the syntactic constructs of axiomatic and generic boxes, and free types. All of the definitions of Chapter 5 belong to this "algebraic sublanguage" of Z.

In the following, we present the *adt* strategy, which can be used to define an abstract data type consisting of constructor functions and other functions. Other strategies associated with the algebraic style, which we do not define here, but which are used in the example of Section 7.4, include the following.

- The strategy *define_generic_construct* produces the subproblems to specify the declaration part and the predicate part of a generic box, the syntactic construct of Z that is used to define generic constructs. It is defined similarly to the strategy *define_schema*. The list of generic parameters is obtained as external information.

- The strategy *define_global_function* is defined similarly to the *define_schema* and *define_generic_construct* strategies. It generates an axiomatic box, and the subproblems consist of defining the declaration and the predicate parts of the box.

The *adt* strategy captures a language-independent approach to the development of abstract data types. An instantiation of the strategy framework that supports another specification language than Z would need a similar strategy. The counterparts of the *define_generic_construct* and *define_global_function* strategies, however, would probably look different in an alternative instantiation of the strategy framework. This is because other languages may follow other principles than Z, where we often define constructs consisting of a declaration part and a predicate part.

The *adt* Strategy

This strategy captures the definition of an abstract data type in the following manner: first, we must define how the members of the data type are constructed, i.e., we must define the constructor functions of the abstract data type. For this purpose, the free type construct of Z is suitable. The definition of the abstract data type is continued by defining more functions that take members of the type as their arguments or yield members of the type as their results. Finally, some more definitions may be made to complete the type specification. The *adt* strategy is defined by

$$adt = \{constructor\_defs, function\_defs, other\_axdefs, adt\_sol\}$$

where *constructor_defs* is defined by

$$\begin{array}{l}
IA\ constructor\_defs = \{P\_init\} \\
OA\ constructor\_defs = \{P\_constr, S\_constr\} \\
constructor\_defs = \{\ t : scheme\ constructor\_defs \longrightarrow Value \mid \\
\quad t(P\_constr) = \langle\ req \Rrightarrow t(P\_init).req\ ; \text{“specify ADT constructors as free type”}, \\
\qquad\qquad context \Rrightarrow t(P\_init).context, \\
\qquad\qquad to\_develop \Rrightarrow \mathsf{adt} : free\_type \rangle \wedge \\
\quad t(S\_constr)\ spec\_acceptable\_for\ t(P\_constr)\}
\end{array}$$

The constructor functions of the abstract data type must take the form of a free type definition. The constituting relation *function_defs* is defined by

$$\begin{array}{l}
IA\ function\_defs = \{P\_init, S\_constr\} \\
OA\ function\_defs = \{P\_fct, S\_fct\} \\
function\_defs = \{\ t : scheme\ function\_defs \longrightarrow Value \mid \\
\quad t(P\_fct) = \langle\ req \Rrightarrow t(P\_init).req\ ; \text{“specify functions on ADT”}, \\
\qquad\qquad context \Rrightarrow t(P\_init).context\ \mathtt{NL}\ t(S\_constr), \\
\qquad\qquad to\_develop \Rrightarrow \mathsf{fcts} : ax\_def \rangle \wedge \\
\quad t(S\_fct)\ spec\_acceptable\_for\ t(P\_fct) \wedge \\
\quad t(S\_fct)\ \text{must refer to}\ S\_constr\}
\end{array}$$

Non-constructor functions that take members of the defined type as their arguments or yield members of the defined type as their result must be defined axiomatically. The syntactic class *ax_def* is defined in such a way that the empty specification ϵ is a member of it, and that state and operation schemas (as described in the definition of the *state_based* strategy in Section 7.3.2 above), are not allowed. The constituting relation *other_axdefs* is defined by

$IA\ other\_axdefs = \{P\_init, P\_fct, S\_fct\}$
$OA\ other\_axdefs = \{P\_other, S\_other\}$
$other\_axdefs = \{\ t : scheme\ other\_defs \longrightarrow Value \mid$
$\quad t(P\_other) = \langle\ req \Rrightarrow t(P\_init).req\ ;$ "other definitions",
$\qquad context \Rrightarrow t(P\_fct).context$ `NL` $t(S\_fct)$
$\qquad to\_develop \Rrightarrow$ others : $specification\rangle \wedge$
$\quad t(S\_other)\ spec\_acceptable\_for\ t(P\_other) \wedge$
$\quad t(S\_other)$ must not contain state or operation schemas }

The constituting relation *adt_sol* assembles the final solution by concatenating the solutions to the subproblems.

$IA\ adt\_sol = \{S\_constr, S\_fct, S\_other\}$
$OA\ adt\_sol = \{S\_final\}$
$adt\_sol = \{\ t : scheme\ adt\_sol \longrightarrow Value \mid$
$\quad t(S\_final) = t(S\_constr)$ `NL` $t(S\_fct)$ `NL` $t(S\_other)\}$

7.3.4 Strategies for the Reuse Style

Reusing specifications is a non-trivial task. It is not realistic to assume that a specification can be reused just by including it without modification into a new specification or simply instantiating its generic parameters. Instead, we must expect to change the specification we intend to reuse.

A strategy that allows for the modification of a generic specification for the purpose of reuse is the *use_generic_spec* strategy, which we define below. Other strategies associated with the reuse style, which we do not define here, but which are used in the example of Section 7.4, include the following.

- The *combine* strategy generates two subproblems. The first is to specify an arbitrary number of component specifications. These may either be reused or developed from scratch. The second subproblem is to specify how the components are combined. The *combine* strategy is associated with the reuse style because it is normally used to combine specifications that already exist.
- The *directly_reuse* strategy allows its users to reuse a generic specification by instantiating its generic parameters.

The principles underlying all of these strategies are language independent. Hence, similar strategies would also be needed to support specification acquisition in languages other than Z.

The *use_generic_spec* Strategy

The *use_generic_spec* strategy takes into account that, in order to successfully reuse an existing generic specification, it may be necessary, first, to change the generic specification, and, second, to change the instantiated (previously changed) specification. The subproblems it generates consist of selecting an existing generic specification, adjusting it, defining the actual parameters to instantiate it, and define the actual instantiated specification, which also may involve further changes. The strategy is defined by

$$\begin{aligned}&\mathit{use\_generic\_spec} = \{\mathit{select\_spec}, \mathit{adjust\_spec}, \mathit{actual\_params}, \mathit{instantiate\_spec},\\ &\qquad\qquad\qquad\qquad \mathit{generic\_spec\_sol}\}\end{aligned}$$

where *select_spec* is defined by

$$\begin{aligned}
&\mathit{IA}\ \mathit{select\_spec} = \{\mathit{P\_init}\}\\
&\mathit{OA}\ \mathit{select\_spec} = \{\mathit{P\_generic}, \mathit{S\_generic}\}\\
&\mathit{select\_spec} = \{\ t : \mathit{scheme}\ \mathit{select\_spec} \longrightarrow \mathit{Value}\ |\\
&\quad \mathit{syn\_class}(t(\mathit{P\_init}).\mathit{to\_develop}) = \mathit{specification} \wedge\\
&\quad t(\mathit{P\_generic}) = \langle\ \mathit{req} \Rrightarrow t(\mathit{P\_init}).\mathit{req}\,;\ \text{“select a generic specification”},\\
&\qquad\qquad\qquad\quad \mathit{context} \Rrightarrow t(\mathit{P\_init}).\mathit{context},\\
&\qquad\qquad\qquad\quad \mathit{to\_develop} \Rrightarrow \textsf{sp} : \mathit{specification}\rangle \wedge\\
&\quad t(\mathit{S\_generic})\ \mathit{spec\_acceptable\_for}\ t(\mathit{P\_generic}) \wedge\\
&\quad t(\mathit{S\_generic})\ \text{is generic}\}
\end{aligned}$$

The solution *S_generic* of the problem *P_generic* may have the syntactic class *specification*, and it must be generic. The constituting relation *adjust_spec* is defined by

$$\begin{aligned}
&\mathit{IA}\ \mathit{adjust\_spec} = \{\mathit{P\_init}, \mathit{S\_generic}\}\\
&\mathit{OA}\ \mathit{adjust\_spec} = \{\mathit{P\_add}, \mathit{S\_add}\}\\
&\mathit{adjust\_spec} = \{\ t : \mathit{scheme}\ \mathit{adjust\_spec} \longrightarrow \mathit{Value}\ |\\
&\quad t(\mathit{P\_add}) = \langle\ \mathit{req} \Rrightarrow t(\mathit{P\_init}).\mathit{req}\,;\ \text{“adjust generic specification”},\\
&\qquad\qquad\qquad\quad \mathit{context} \Rrightarrow t(\mathit{P\_init}).\mathit{context}\,\texttt{NL}\,t(\mathit{S\_generic}),\\
&\qquad\qquad\qquad\quad \mathit{to\_develop} \Rrightarrow \textsf{add\_spec} : \mathit{specification}\rangle \wedge\\
&\quad t(\mathit{S\_add})\ \mathit{spec\_acceptable\_for}\ t(\mathit{P\_add})\}
\end{aligned}$$

To define *P_add*, the identified generic specification *S_generic* is added to the context. If the generic specification needs no adjustments, the empty specification can be developed for *S_add*. The constituting relation *actual_params* is defined by

$$\begin{aligned}
&\mathit{IA}\ \mathit{actual\_params} = \{\mathit{P\_init}, \mathit{P\_add}, \mathit{S\_add}\}\\
&\mathit{OA}\ \mathit{actual\_params} = \{\mathit{P\_params}, \mathit{S\_params}\}\\
&\mathit{actual\_params} = \{\ t : \mathit{scheme}\ \mathit{actual\_params} \longrightarrow \mathit{Value}\ |\\
&\quad t(\mathit{P\_params}) = \langle\ \mathit{req} \Rrightarrow t(\mathit{P\_init}).\mathit{req}\,;\ \text{“specify actual parameters”},\\
&\qquad\qquad\qquad\quad \mathit{context} \Rrightarrow t(\mathit{P\_add}).\mathit{context}\,\texttt{NL}\,t(\mathit{S\_add})\\
&\qquad\qquad\qquad\quad \mathit{to\_develop} \Rrightarrow \textsf{params} : \mathit{specification}\rangle \wedge\\
&\quad t(\mathit{S\_params})\ \mathit{spec\_acceptable\_for}\ t(\mathit{P\_params}) \wedge\\
&\quad t(\mathit{S\_params}) \neq \epsilon\}
\end{aligned}$$

The actual parameters that will be used to instantiate the adjusted generic specification must not be empty. Since different syntactic forms of the parameter specifications are possible, the syntactic class of the solution *S_params* is *specification*. The constituting relation *instantiate_spec* is defined by

$$
\begin{array}{l}
\mathit{IA}\ \mathit{instantiate\_spec} = \{\mathit{P\_init}, \mathit{P\_params}, \mathit{S\_params}\} \\
\mathit{OA}\ \mathit{instantiate\_spec} = \{\mathit{P\_inst}, \mathit{S\_inst}\} \\
\mathit{instantiate\_spec} = \{\ t : \mathit{scheme}\ \mathit{instantiate\_spec} \longrightarrow \mathit{Value}\ | \\
\quad t(\mathit{P\_inst}) = \langle\ \mathit{req} \Rrightarrow t(\mathit{P\_init}).\mathit{req}\,; \text{“instantiate adjusted generic specification with actual parameters”}, \\
\qquad \mathit{context} \Rrightarrow t(\mathit{P\_params}).\mathit{context}\,\mathtt{NL}\,t(\mathit{S\_params}) \\
\qquad \mathit{to\_develop} \Rrightarrow \mathsf{inst} : \mathit{specification}\rangle \wedge \\
\quad t(\mathit{S\_inst})\ \mathit{spec\_acceptable\_for}\ t(\mathit{P\_inst}) \wedge \\
\quad t(\mathit{S\_inst}) \neq \epsilon \wedge \\
\quad t(\mathit{S\_inst}) \text{ is concrete}\}
\end{array}
$$

The instantiated specification $\mathit{S\_inst}$ must not be empty or contain generic parameters. The constituting relation *generic_spec_sol* assembles the final solution by concatenating the developed parts. There are no additional acceptability conditions.

$$
\begin{array}{l}
\mathit{IA}\ \mathit{generic\_spec\_sol} = \{\mathit{S\_generic}, \mathit{S\_add}, \mathit{S\_params}, \mathit{S\_inst}\} \\
\mathit{OA}\ \mathit{generic\_spec\_sol} = \{\mathit{S\_final}\} \\
\mathit{generic\_spec\_sol} = \{\ t : \mathit{scheme}\ \mathit{generic\_spec\_sol} \longrightarrow \mathit{Value}\ | \\
\quad t(\mathit{S\_final}) = t(\mathit{S\_generic})\,\mathtt{NL}\,t(\mathit{S\_add})\,\mathtt{NL}\,t(\mathit{S\_params})\,\mathtt{NL}\,t(\mathit{S\_inst})\}
\end{array}
$$

7.4 Specification of the Unix File System

We now use the strategies presented in the previous section to develop a specification of the user's view of the Unix file system. The system to be specified is a *tree* of files and directories, where the root and the inner nodes of the tree are directories, and the leaves are either files or empty directories. The user can navigate in this tree, add and remove directories and files, and access information stored in the system. Each user has a home and a working directory.

An overview of the development is given in Figure 7.1. The numbers shown in the nodes of the development tree correspond to the order in which the problems are solved. In the upper parts of the nodes, the problems to be solved are indicated, and the reducing strategies are shown in the lower parts of the node.

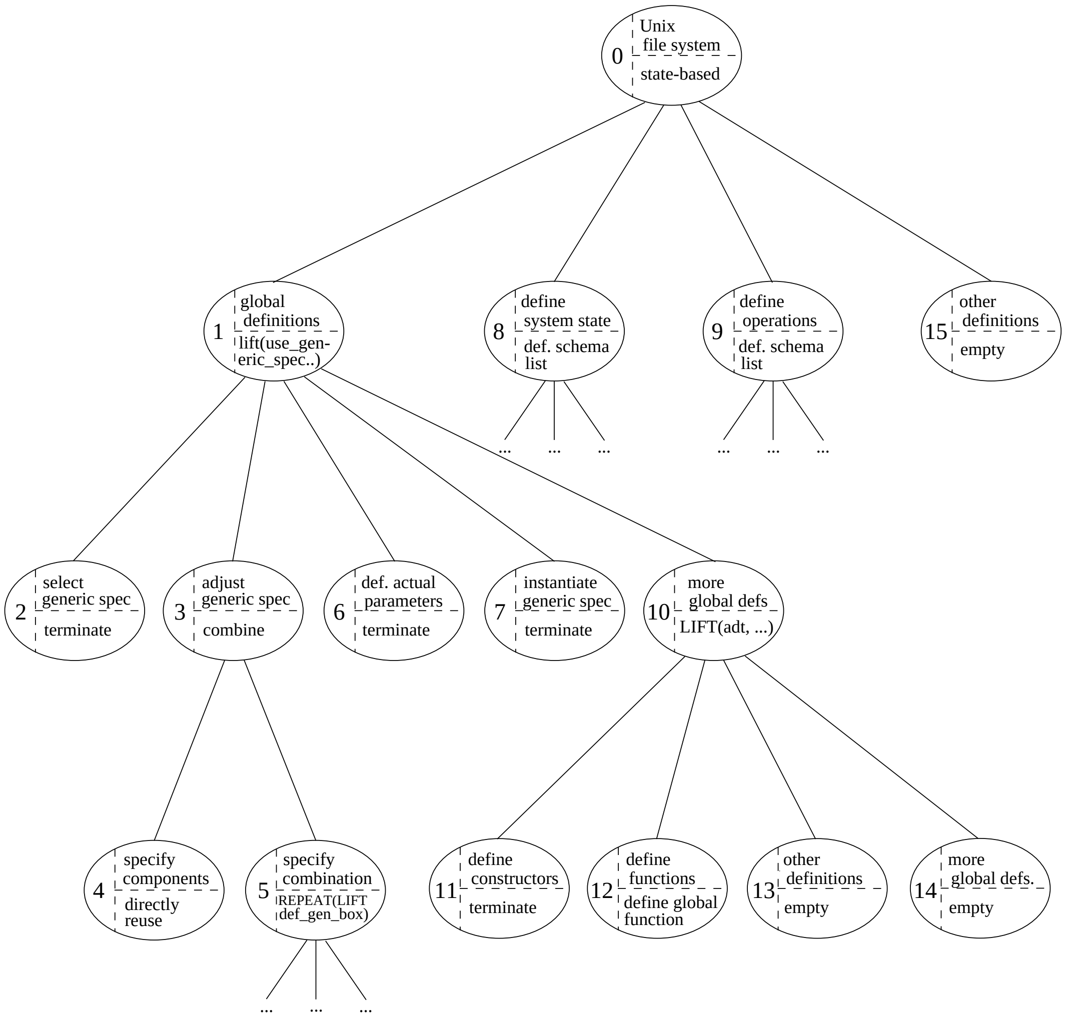
The initial problem P_0 for the specification task has the form

$$
\begin{array}{lll}
P_0 = \langle \mathit{req} & \Rrightarrow & \text{“Specification of the Unix file system”}, \\
\quad\ \ \mathit{context} & \Rrightarrow & \epsilon, \\
\quad\ \ \mathit{to\_develop} & \Rrightarrow & \mathsf{sp} : \mathit{specification}\rangle
\end{array}
$$

The numbering of the problems in the text coincides with the numbering of the nodes in Figure 7.1.

Since the tree of files and directories together with the home and working directories form a system state, we start with the state-based style to develop the top-level specification and apply the *state_based* strategy. The first subproblem generated by the *state_based* strategy is to develop the global definitions:

$$
\begin{array}{lll}
P_1 = \langle \mathit{req} & \Rrightarrow & \text{“Specification of the Unix file system; specify global definitions”}, \\
\quad\ \ \mathit{context} & \Rrightarrow & \epsilon, \\
\quad\ \ \mathit{to\_develop} & \Rrightarrow & \mathsf{sp} : \mathit{specification}\rangle
\end{array}
$$



0 Unix file system | state-based
1 global definitions | lift(use_generic_spec..)
8 define system state | def. schema list
9 define operations | def. schema list
15 other definitions | empty
2 select generic spec | terminate
3 adjust generic spec | combine
6 def. actual parameters | terminate
7 instantiate generic spec | terminate
10 more global defs | LIFT(adt, ...)
4 specify components | directly reuse
5 specify combination | REPEAT(LIFT def_gen_box)
11 define constructors | terminate
12 define functions | define global function
13 other definitions | empty
14 more global defs. | empty

Figure 7.1: Development tree for specification of Unix file system

We first must define the tree structures that will be part of the system state. These trees are characterized by the fact that each node has a name, a content, and an arbitrary number of successors. We assume that a specification of such trees, where the content of the nodes (as opposed to their names) is a generic parameter, is available for reuse in our library. Hence, the *use_generic_spec* strategy will be applicable. However, we should not reduce problem P_1 with the *use_generic_spec* strategy, because we certainly will have to make other global definitions. Therefore, we reduce problem P_1 with with a lifted version of *use_generic_spec*. This leaves us the freedom to add more global definitions afterwards. The strategy for reducing P_1 is

$$\textsc{Lift}(use\_generic\_spec, p\_down, p\_combine, s\_combine)$$

where

$$p\_down = (\lambda\, pr : SpecProblem \mid syn\_class(pr.to\_develop) = specification \bullet pr)$$

$$\begin{array}{l}
p\_combine = (\lambda\, pr : SpecProblem;\ sol : SpecSolution \mid \\
\qquad syn\_class(pr.to\_develop) = specification \bullet \\
\qquad \langle req \Rrightarrow pr.req\ ;\ \text{“more global definitions”}, \\
\qquad context \Rrightarrow pr.context\ \mathtt{NL}\ sol, \\
\qquad to\_develop \Rrightarrow pr.to\_develop\rangle) \\
s\_combine = \_\mathtt{NL}\_
\end{array}$$

The function *p_down* is the identity on those specification problems that admit the development of a solution that belongs to the syntactic class *specification*, because P_1 requires the development of an item of class *specification* and the *use_generic_spec* strategy yields an item of this class. The syntactic class *specification* is the most general one, and it is closed under the function _NL_. The function *p_combine* incorporates a developed specification fragment into the *context* part of a problem, and the function *s_combine* is the concatenation function on specifications.

That the functions *p_down*, *p_combine* and *s_combine* fulfill the requirements for the arguments of the LIFT strategical follows by a similar argument to the one for the strategy *define_schema_list* on page 164.

When we apply the above strategy to P_1, we get the problem P_2 as the value of the attribute *P_generic*:

$$\begin{array}{lll}
P_2 = \langle req & \Rrightarrow & \text{“Specification of the Unix file system; specify global definitions;} \\
 & & \text{select a generic specification”}, \\
\quad context & \Rrightarrow & \epsilon, \\
\quad to\_develop & \Rrightarrow & \mathsf{sp} : specification\rangle
\end{array}$$

This problem can be solved by a strategy similar to the *terminate* strategy, differing only in the implementation of the corresponding heuristic function. Instead of being asked to type in a specification, the user may select a specification from a library. We select the generic specification

$$\begin{array}{l}
NAMED\_TREE[X] == \\
\quad \{f : \operatorname{seq} \mathbb{N}_1 \mathrel{+\!\!\!\!\!\rightarrow\!\!\!\!\!+} NAME \times X \mid \\
\qquad \langle\rangle \in \operatorname{dom} f \\
\qquad \land\ (\forall\, path : \operatorname{seq}_1 \mathbb{N}_1 \mid path \in \operatorname{dom} f \bullet \\
\qquad\qquad front\ path \in \operatorname{dom} f \\
\qquad\qquad \land\ (last\ path \neq 1 \Rightarrow front\ path \frown \langle last\ path - 1\rangle \in \operatorname{dom} f))\}
\end{array}$$

which we have already explained in Section 2.4.3 on page 21. For *NAMED_TREE*[*X*], several functions, e.g., to select a child or the leaves of the tree, are defined (the function *child_named* can be found in Section 2.4.3); for a more detailed presentation, see (Heisel, 1995b). The above generic definition, together with the predefined functions on named trees, form the solution S_2 of problem P_2.

The next problem to work on is the second subproblem, *P_add*, generated by the *use_generic_spec* strategy. It consists of adjusting named trees to our purposes.

$$\begin{array}{lll}
P_3 = \langle req & \Rrightarrow & \text{“Specification of the Unix file system; specify global definitions;} \\
 & & \text{adjust generic specification”}, \\
\quad context & \Rrightarrow & S_2, \\
\quad to\_develop & \Rrightarrow & \mathsf{add\_spec} : specification\rangle
\end{array}$$

We now have to combine named trees with *paths*, which allow us to use names to navigate in named trees. To this end, we reduce P_3 with the strategy *combine*, which was outlined in Section 7.3.4, and which, like *use_generic_spec*, is associated with the reuse style. To develop the solution S_3 for P_3, the subproblem P_4 to specify the component specifications must be solved. We use the strategy *directly_reuse*, which has been briefly described in Section 7.3.4, to instantiate non-empty sequences with the actual parameter $NAME$:

$$PATH == \mathrm{seq}_1\, NAME$$

To solve the subproblem P_5 of combining named trees with paths, we have to define several functions and predicates that take named trees as well as paths as their arguments. To do so, we apply the strategy

$$\textsc{Repeat}(\textsc{Lift}(\mathit{define\_generic\_construct}, \mathit{p\_down}, \mathit{p\_combine}, \mathit{s\_combine}), \mathit{p\_rep}, \mathit{empty})$$

which is defined similarly to the strategy *define_schema_list* of Section 7.3.2. The strategy *define_generic_construct* was briefly sketched in Section 7.3.3. One of the definitions we develop in this way is the predicate *is_existing_path_of* that decides if a given path exists in a given tree.

$$\begin{array}{l}
[X] \\
\hline
\_\mathit{is\_existing\_path\_of}\_ : PATH \leftrightarrow NAMED\_TREE[X] \\
\hline
\forall\, t : NAMED\_TREE[X];\ p : PATH \bullet \\
\quad p\ \mathit{is\_existing\_path\_of}\ t \Leftrightarrow \\
\qquad (\mathit{head}\ p = \mathit{name\_of\_tree}\ t \wedge \\
\qquad (\mathit{tail}\ p \neq \langle\rangle \Rightarrow (\exists\, t_1 : \mathit{children}\ t \bullet \mathit{tail}\ p\ \mathit{is\_existing\_path\_of}\ t_1)))
\end{array}$$

Let S_3 denote the concatenation of the definition of paths and the definitions of the functions and predicates that establish the combination of named trees with paths. The next problem to solve is the definition of the actual parameters to instantiate the extended definition of named trees.

$$\begin{array}{lll}
P_6 = \langle \mathit{req} & \Rrightarrow & \text{“Specification of the Unix file system; specify global definitions;} \\
 & & \text{specify actual parameters”}, \\
\quad \mathit{context} & \Rrightarrow & S_2\ \mathtt{NL}\ S_3, \\
\quad \mathit{to\_develop} & \Rrightarrow & \mathsf{params} : \mathit{specification}\rangle
\end{array}$$

With the *terminate* strategy, we define the solution S_6 of P_6:

$$UNIX\_NODE ::= \mathit{dir} \mid \mathit{file}\langle\!\langle FILE\rangle\!\rangle$$

where we do not present the development of the type $FILE$, which denotes files. This type can either be defined as a given type or a free type, distinguishing e.g. text files and binary files. The last subproblem P_7 of P_1 generated by the *use_generic_spec* strategy is to instantiate the adjusted generic definition with an actual parameter. Again using *terminate* we solve P_7 by typing in the following definition of directories:

$$\begin{array}{l}
DIRECTORY : \mathbb{P}\, NAMED\_TREE[UNIX\_NODE] \\
\hline
\forall\, d : DIRECTORY;\ p : PATH \mid p\ \mathit{is\_existing\_path\_of}\ d \\
\quad \bullet \forall\, \mathit{adr} : \mathrm{dom}\, d \bullet \\
\qquad (\mathit{second}(d\ \mathit{adr}) \in \mathrm{ran}\, \mathit{file} \Rightarrow \mathit{adr} \in \mathit{leaves}\, d) \wedge \\
\qquad \mathit{namelist}(\mathit{subtrees}\,(\mathit{object\_at\_in}(p, d))) \in \mathrm{iseq}\, NAME
\end{array}$$

The set of directories, $DIRECTORY$, is a subset of $NAMED\_TREE[UNIX\_NODE]$. In a directory, files may only occur as leaf nodes, and all children of a given node must have different names.

The concatenation of the solutions S_2, S_3, S_6 and the definition of directories forms a *preliminary* solution S_1 of problem P_1. There still is one open subproblem of P_1, namely the subproblem P_{10} to define further global definitions. It was generated because we used a lifted version of *use_generic_spec*. As already explained in Section 5.7, it is unrealistic to assume that we can foresee all necessary global definitions in advance. Since the specification developed so far suffices to define the system state, we leave P_{10} open and start working on the second subproblem generated by the *state_based* strategy, P_8. We will come back to P_{10} when developing the system operations.

$$\begin{array}{lll} P_8 = \langle req & \Rrightarrow & \text{“Specification of the Unix file system; specify global system state”}, \\ \quad context & \Rrightarrow & S_1, \\ \quad to\_develop & \Rrightarrow & \mathsf{state\_def} : schema\_list \rangle \end{array}$$

The system state consists of a directory tree, a path leading to the home directory, and a path leading to the working directory. Both paths must exist in the root directory, and they must lead to directories, not to files. For an initial state, we require a directory tree and a home directory as inputs. The working directory is set to the home directory by default.

Using the strategy *define_schema_list* of Section 7.3.2, we develop the following two schemas, which form the solution S_8 of problem S_8.

$$\begin{array}{|l} \multicolumn{1}{l}{OneUserView} \\ \hline root : DIRECTORY \\ home\_dir : PATH \\ working\_dir : PATH \\ \hline home\_dir\ is\_existing\_path\_of\ root \\ second(object\_at\_in(home\_dir, root)(\langle\rangle)) = dir \\ working\_dir\ is\_existing\_path\_of\ root \\ second(object\_at\_in(working\_dir, root)(\langle\rangle)) = dir \\ \hline \end{array}$$

$$\begin{array}{|l} \multicolumn{1}{l}{InitOneUserView} \\ \hline OneUserView' \\ newdir? : DIRECTORY \\ newhd? : PATH \\ \hline root' = newdir? \\ home\_dir' = newhd? \\ working\_dir' = newhd? \\ \hline \end{array}$$

The next problem P_9 is to define the system operations:

$$\begin{array}{lll} P_9 = \langle req & \Rrightarrow & \text{“Specification of the Unix file system; specify system operations”}, \\ \quad context & \Rrightarrow & S_1\ \mathtt{NL}\ S_8, \\ \quad to\_develop & \Rrightarrow & \mathsf{ops\_def} : schema\_list \rangle \end{array}$$

Like P_8, we reduce this problem with the strategy *define_schema_list*. Of the many Unix commands, we only define the command *cd* that changes the working directory. The command *cd* can be called with varying numbers and types of parameters. The strong typing of Z forces us to define several schemas to cope with the different parameter constellations of *cd*. If no argument is supplied to *cd*, the working directory is set to the home directory by default. If an absolute path is supplied to *cd*, the working directory is set to this path, provided it is a legal one. Legal means that the path exists in the directory and that it leads to a directory, not to a file. Hence, with two repetitions of the strategy *define_schema*, we obtain the two operations

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{cd\_def}} \\
\hline
\Delta \mathit{OneUserView} \\
\hline
\mathit{root}' = \mathit{root} \\
\mathit{home\_dir}' = \mathit{home\_dir} \\
\mathit{working\_dir}' = \mathit{home\_dir} \\
\hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{cd\_abs}} \\
\hline
\Delta \mathit{OneUserView} \\
p? : \mathit{PATH} \\
\hline
p?\ \mathit{is\_existing\_path\_of}\ \mathit{root} \\
\mathit{second}(\mathit{object\_at\_in}(p?, \mathit{root})(\langle\rangle)) = \mathit{dir} \\
\mathit{root}' = \mathit{root} \\
\mathit{home\_dir}' = \mathit{home\_dir} \\
\mathit{working\_dir}' = p? \\
\hline
\end{array}$$

There is a third version of *cd*, which takes a *relative path*, i.e. a path starting at the current working directory, as its argument. Since this version of *cd* also allows upward movement in the directory tree (using the notation ../), we must define a data type called *DISPLACEMENT* that can be combined with paths to yield paths. Hence, we have to go back to the last open subproblem P_{10} of P_1, which has been generated by a lifted version of the *use_generic_spec* strategy.

$$\begin{aligned}
P_{10} = \langle \mathit{req} \quad &\Rrightarrow \text{“Specification of the Unix file system; more global definitions”}, \\
\mathit{context} \quad &\Rrightarrow S_1, \\
\mathit{to\_develop} &\Rrightarrow \mathsf{sp} : \mathit{specification}\rangle
\end{aligned}$$

Since *DISPLACEMENT* is an abstract data type, we now switch to the algebraic style and apply the *adt* strategy of Section 7.3.3 to define displacements. But because we do not know if this will be the last global definition, we use a lifted version of *adt* that defines one additional subproblem:

$$\textsc{Lift}(\mathit{adt}, \mathit{p\_down}, \mathit{p\_combine}, \mathit{s\_combine})$$

where *p_down*, *p_combine* and *s_combine* are defined as in the strategy that we used to reduce P_1, see page 169.

The first subproblem P_{11} that is generated by this reduction is to define the constructors of the abstract data type with a free type. With the *terminate* strategy, we give the definition

$$\begin{aligned} \mathit{DISPLACEMENT} ::= {} & \mathit{empty\_d} \\ \mid {} & d\langle\!\langle \mathit{PATH} \rangle\!\rangle \\ \mid {} & (\_/\_)\langle\!\langle \mathit{DISPLACEMENT} \times \mathit{NAME} \rangle\!\rangle \\ \mid {} & ../\langle\!\langle \mathit{DISPLACEMENT} \rangle\!\rangle \end{aligned}$$

The empty displacement *empty_d* is a displacement. Paths are embedded into displacements via the function d. Combining a displacement with a name using the function / yields a new displacement. Given a displacement dp, $../dp$ yields a new displacement. All displacements can be obtained by application of these functions.

The next subproblem P_{11} to be solved is to define non-constructor functions on the abstract data type. We reduce this problem by the strategy *define_global_function* described in Section 7.3.3, which is also associated with the algebraic style. The function we need combines paths and displacements.

$$\begin{array}{|l} \_ \mid\mid \_ : \mathit{PATH} \times \mathit{DISPLACEMENT} \rightarrowtail\!\!\!\!\!\!+\; \mathit{PATH} \\ \hline \forall\, p : \mathit{PATH};\ n : \mathit{NAME};\ dp : \mathit{DISPLACEMENT} \bullet \\ \quad p \mid\mid \mathit{empty\_d} = p \land \\ \quad p \mid\mid d\,\langle n \rangle = p \frown \langle n \rangle \land \\ \quad p \mid\mid (dp/n) = (p \mid\mid dp) \frown \langle n \rangle \land \\ \quad \langle n \rangle \mid\mid (../dp) = \langle n \rangle \mid\mid dp \land \\ \quad (p \frown \langle n \rangle) \mid\mid ../dp = p \mid\mid dp \end{array}$$

No other definitions are necessary to define displacements. Hence, we solve the third subproblem P_{13} by the *empty* strategy. The concatenation of the free type definition and the definition of the function || forms a preliminary solution S_{10} to the problem P_{10} (there is still one open subproblem P_{14}, generated by LIFT), and the concatenation $S_1\,\mathtt{NL}\,S_{10}$ forms a new preliminary solution to the problem P_1. This new preliminary solution must be propagated into the problem P_9, which yields:

$$\begin{aligned} P_9' = \langle \mathit{req} & \Rrightarrow \text{“Specification of the Unix file system; specify system operations”}, \\ \mathit{context} & \Rrightarrow S_1\,\mathtt{NL}\,S_{10}\,\mathtt{NL}\,S_8, \\ \mathit{to\_develop} & \Rrightarrow \mathsf{ops\_def} : \mathit{schema\_list} \rangle \end{aligned}$$

The new solution $S_1\,\mathtt{NL}\,S_{10}$ is also propagated into all subproblems of P_9. Now that the definition of displacements has become visible in node 9 of the development tree and all its successors, we can define the third version of the *cd* command:

$$\begin{array}{|l} \multicolumn{1}{l}{\mathit{cd\_rel}} \\ \hline \Delta \mathit{OneUserView} \\ dp? : \mathit{DISPLACEMENT} \\ \hline (\mathit{working\_dir} \mid\mid dp?)\ \mathit{is\_existing\_path\_of}\ \mathit{root} \\ \mathit{second}(\mathit{object\_at\_in}(\mathit{working\_dir} \mid\mid dp?, \mathit{root})(\langle\rangle)) = \mathit{dir} \\ \mathit{root}' = \mathit{root} \\ \mathit{home\_dir}' = \mathit{home\_dir} \\ \mathit{working\_dir}' = \mathit{working\_dir} \mid\mid dp? \\ \hline \end{array}$$

If a displacement is supplied to *cd*, the new working directory is computed as the absolute path yielded by combining the old working directory with the given displacement.

This definition concludes our specification (at least for the purposes of this chapter). All other open problems (the open subproblems of P_{10} and P_9, as well as the fourth subproblem P_{15} generated with the *state_based* strategy) can be solved with the *empty* strategy.

7.5 Connecting Instantiations

We now take a first step toward the combination of different instances of the strategy framework. We show how specifications developed with the instance of this chapter can be transformed into programming problems, as they are defined in Chapter 6. Directly proceeding from the specification to the implementation phase of software development, however, is only possible in cases where the specification contains data types that can easily be mapped onto the data types available in conventional programming languages. An instance of the strategy framework that supports the refinement of Z specifications, i.e., the refinement of the abstract data types to data types available in programming languages, is not yet defined.

The combination of Z and IOSS can be achieved easily: since both formalisms allow for states and have concepts to deal with changing values of variables, Z specifications can mechanically be translated into IOSS programming problems.

Four kinds of variables occurring in a Z schema have to be considered (not to be confused with the variable classification of IOSS programming problems, see page 141): *Input variables* are the ones decorated with "?". *Output variables* are the variables decorated with "!". *State variables* are the variables of the global state schema. All other variables are *auxiliary variables*. With this classification, the translation of a Z schema into an IOSS programming problem proceeds as follows.

- Each input variable of the Z schema becomes an input variable of the corresponding problem.
- Each output variable of the Z schema becomes a result variable of the problem.
- Each variable x of the Z state schema becomes an input variable if the schema predicate entails $x = x'$.
- Otherwise x becomes a result variable, and a new state variable x_0 is generated for x if x occurs in the schema predicate.
- Each auxiliary variable becomes a result variable.
- The precondition of the IOSS problem is the precondition of the Z schema plus an equation $x = x_0$ for each state variable x_0 generated as described above.
- The invariant of the IOSS problem is the invariant of the Z schema defining the system state.
- The goal of the IOSS problem consists of those conjuncts of the schema predicate that depend on result variables of the IOSS problem, where primed state variables of the schema have to be replaced by plain variables and plain state variables of the schema have to be replaced by the corresponding state variables of the IOSS problem. Auxiliary variables remain unchanged.

As an example, we consider the transformation of the operation schema *OpReleaseSucceed* of Section 3.3.2. The programming language of IOSS allows for enumeration types, so that a data refinement is not necessary. Recall the definition of this schema:

$$
\begin{array}{|l}
\hline
\textit{OpReleaseSucceed} \\
\Delta \textit{InertGasSystem} \\
\textit{Sensors}; \textit{Actuators} \\
\hline
\textit{mode} = \textit{RELEASE\_SUCCEED} \\
(\textit{consistency} = \textit{NO} \Rightarrow \textit{mode}' = \textit{INCONSISTENCY}) \\
(\textit{consistency} = \textit{YES} \Rightarrow \\
\quad (\textit{reset\_button}? = \textit{PRESSED} \Rightarrow \textit{mode}' = \textit{NORMAL}) \wedge \\
\quad (\textit{reset\_button}? = \textit{NOT\_PRESSED} \Rightarrow \textit{mode}' = \textit{RELEASE\_SUCCEED})) \\
\hline
\end{array}
$$

where

$$
\begin{array}{|l}
\hline
\textit{InertGasSystem} \\
\textit{mode} : \textit{MODE} \\
\textit{warning\_timer} : 0 \mathrel{..} \textit{WARNING\_DURATION} \\
\textit{release\_check\_timer} : 0 \mathrel{..} \textit{CHECK\_DURATION} \\
\textit{release\_bank\_A}, \textit{release\_bank\_B} : \textit{OPEN\_CLOSED} \\
\textit{warning\_light} : \textit{LIGHT\_STATUS} \\
\textit{warning\_beeper} : \textit{BEEP\_STATUS} \\
\hline
\textit{mode} \neq \textit{RELEASE\_INITIATED} \Rightarrow \\
\quad \textit{release\_bank\_A} = \textit{release\_bank\_B} = \textit{CLOSED} \\
\textit{mode} = \textit{WARNING} \Leftrightarrow \textit{warning\_timer} > 0 \\
\textit{mode} = \textit{RELEASE\_INITIATED} \Leftrightarrow \textit{release\_check\_timer} > 0 \Leftrightarrow \textit{warning\_light} = \textit{ON} \\
\textit{mode} \notin \{\textit{WARNING}, \textit{RELEASE\_INITIATED}, \textit{INCONSISTENCY}\} \\
\quad \Leftrightarrow \textit{warning\_light} = \textit{OFF} \\
\textit{mode} = \textit{NORMAL} \Leftrightarrow \textit{warning\_beeper} = \textit{NOT\_BEEPING} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{Sensors} \\
\textit{InertGasSystem} \\
\textit{bank\_selector}? : \textit{BANK\_SELECTOR\_STATUS} \\
\textit{request\_button}? : \textit{BUTTON\_STATUS} \\
\textit{reset\_button}? : \textit{BUTTON\_STATUS} \\
\textit{inhibit\_button}? : \textit{BUTTON\_STATUS} \\
\textit{fire\_detector}1?, \textit{fire\_detector}2? : \textit{DETECTION\_STATUS} \\
\textit{gas\_detector}? : \textit{DETECTION\_STATUS} \\
\textit{fire\_detector} : \textit{DETECTION\_STATUS} \\
\textit{consistency} : \textit{YES\_NO} \\
\hline
\textit{fire\_detector} = \textit{DETECTION} \Leftrightarrow \\
\quad \textit{fire\_detector}1? = \textit{fire\_detector}2? = \textit{DETECTION} \\
\textit{consistency} = \textit{NO} \Leftrightarrow \\
\quad \textit{mode} \neq \textit{RELEASE\_INITIATED} \wedge \textit{gas\_detector}? = \textit{DETECTION} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Actuators}} \\
\hline
\mathit{InertGasSystem}' \\
\mathit{release\_bank\_A}!, \mathit{release\_bank\_B}! : \mathit{OPEN\_CLOSED} \\
\mathit{warning\_light}! : \mathit{LIGHT\_STATUS} \\
\mathit{warning\_beeper}! : \mathit{BEEP\_STATUS} \\
\mathit{mode}! : \mathit{MODE} \\
\hline
\mathit{release\_bank\_A}! = \mathit{release\_bank\_A}' \\
\mathit{release\_bank\_B}! = \mathit{release\_bank\_B}' \\
\mathit{warning\_light}! = \mathit{warning\_light}' \\
\mathit{warning\_beeper}! = \mathit{warning\_beeper}' \\
\mathit{mode}! = \mathit{mode}' \\
\hline
\end{array}
$$

From the invariant of the schema *InertGasSystem* it follows that in the mode *RELEASE-_SUCCEED*, both timers are zero, the warning light is off, but the beeper is on. The only possible successor modes are *RELEASE_SUCCEED*, *NORMAL* and *INCONSISTENCY*. Hence, according to the definition of the system operations corresponding to these successor states (see pages 48 and 51), only the variables *mode*, *warning_light* and *warning_beeper* of the state schema *InertGasSystem* can change their values. Following the preceeding translation rules, we obtain the following programming problem:

input variables:	$\mathit{bank\_selector}?, \ldots, \mathit{gas\_detector}?,$ $\mathit{warning\_timer}, \mathit{release\_check\_timer}, \mathit{release\_bank\_A}, \mathit{release\_bank\_B}$
result variables:	$\mathit{release\_bank\_A}!, \ldots, \mathit{mode}!,$ $\mathit{mode}, \mathit{warning\_light}, \mathit{warning\_beeper},$ $\mathit{fire\_detector}, \mathit{consistency}$
state variables:	$\mathit{mode}_0, \mathit{warning\_light}_0, \mathit{warning\_beeper}_0$
precondition:	$\mathit{mode} = \mathit{mode}_0 \land \mathit{warning\_light} = \mathit{warning\_light}_0$ $\land\ \mathit{warning\_beeper} = \mathit{warning\_beeper}_0 \land \mathit{mode} = \mathit{RELEASE\_SUCCEED}$
invariant:	predicate of *InertGasSystem*
goal:	$(\mathit{fire\_detector} = \mathit{DETECTION} \Leftrightarrow$ $(\mathit{fire\_detector}1? = \mathit{DETECTION} \land \mathit{fire\_detector}2? = \mathit{DETECTION}))$ $\land\ (\mathit{consistency} = \mathit{NO} \Leftrightarrow$ $(\mathit{mode}_0 \neq \mathit{RELEASE\_INITIATED} \land \mathit{gas\_detector}? = \mathit{DETECTION}))$ $\land\ (\mathit{consistency} = \mathit{NO} \Rightarrow \mathit{mode} = \mathit{INCONSISTENCY})$ $\land\ (\mathit{consistency} = \mathit{YES} \Rightarrow$ $((\mathit{reset\_button}? = \mathit{PRESSED} \Rightarrow \mathit{mode} = \mathit{NORMAL})$ $\land\ (\mathit{reset\_button}? = \mathit{NOT\_PRESSED} \Rightarrow \mathit{mode} = \mathit{RELEASE\_SUCCEED}))$ $\land \ldots$ see *ACTUATORS*

The synthesis of a program for *OpReleaseSucceed* can be found in (Heisel and Sühl, 1996a), and another example of a specification transformation and the synthesis of the program that implements the transformed specifictation is presented in (Heisel, 1996a).

7.6 Related Work

Souquières and Lévy (Souquières, 1993; Souquières and Lévy, 1993) have developed an approach to specification acquisition whose underlying concepts have much in common with the ones presented here. Specification acquisition is performed by solving *tasks*. The agenda of tasks is called a *workplan* and resembles our development tree. Tasks can be reduced by *development operators* similar to strategies. Development operators, however, do not guarantee semantic properties of the product. Therefore, incomplete reductions and a variable number of subtasks for the same operator can be are possible. In (Souquières and Heisel, 1996), language-independent development operators for the various styles are presented.

Johnson and Feather (Johnson and Feather, 1991) take a transformational approach to supporting the specification process. Starting out from a simple initial specification, *evolution transformations* are applied. These may change the semantics of the specification and add more detail to it. Compared to these, specification styles and strategies are concepts of a higher level of abstraction and closer to human reasoning.

In (Woodcock and Larsen, 1993), several support systems for formal specification techniques – mostly for VDM and Z – are presented. Apart from some theorem provers, these can be divided into two classes: the first class performs type-checking or other analyses of a given specification. These systems cannot be used to set up a specification. The second class provides editing facilities for the language they support. Editors do not provide a process model and cannot support design decisions.

7.7 Summary

In this chapter, we have introduced an approach to machine-supported specification acquisition, which provides methodological support for specifiers and validation mechanisms for the developed specifications. In particular:

- We have introduced the concept of a specification style. Specification styles capture different approaches to the development of a specification. In one specification, several styles may be needed to specify different components. This clearly shows that it is not satisfactory to identify styles with specification languages. Instead, specification styles are to a large extent language-independent.

- We have shown that specification styles can be represented as sets of strategies, and we have given examples of strategies that are associated with the state-based, algebraic, and reuse specification styles.

- By way of an example, we have shown that strategy-based specification acquisition is feasible. The strategies keep track of the activities to be performed and guarantee that the developed solutions satisfy the general acceptability predicate as well as additional strategy-specific acceptability conditions that capture context-dependent integrity contraints for specifications.

- We have demonstrated that the Lift and Repeat strategicals can be profitably employed in specification acquisition. Using the Lift strategical, we can develop an indefinite number of different specification fragments. Combining the Repeat strategical with the Lift strategical is useful when the same strategy has to be applied several times.

- Finally, we have shown how different instances of the strategy framework can be combined. The solutions to specification problems developed with the instance for specification acquistion can be transformed into programming problems of the instance for program synthesis.

7.8 Further Research

The work presented in this chapter merits further research on the following subjects:

More specification styles. The specification styles we considered here do not cover all useful specification styles that should be supported. For example, an object oriented specification style is also of advantageous. More of these specification styles should be identified and expressed as sets of strategies.

More instances for specification acquisition. To gain more knowledge about the extent to which specification styles are language-independent, other instances of the strategy framework should be defined that support languages other than Z. It could then be investigated how strategies for Z can be transformed in a systematic way into strategies that support other languages.

Instance for refinement. To bridge the gap between the instance of the strategy framework of this chapter and the one used for IOSS, an instance that supports the refinement of Z specifications should be defined. When such an instance exists, an integration of three instances, ranging from specification over refinement to program synthesis, is promising.

More strategies. It is not yet clear to what extent the specification strategies defined so far are complete. More case studies should be conducted to find additional strategies that are widely applicable.

Implementation. The instance of the strategy framework presented in this chapter is not yet implemented. Hence, the benefits of strategy-based specification acquisition cannot yet be fully exploited. An implemented system could also help to convince other researchers and practitioners to start working with strategies.

Chapter 8

Strategy-Based Specification of Safety-Critical Software

In this chapter, we present a third instance of the strategy framework. As for the instance of Chapter 7, its purpose is to support the development of formal specifications. However, we now consider the task to specify software for safety-critical applications as described in Chapter 3.

The agendas of Chapter 3 will be transformed into strategies. Such a transformation follows general principles, which we describe in Section 8.1. In Section 8.2, we instantiate the generic parameters of the strategy framework, using the combination of Z and real-time CSP introduced in Chapter 3. In Sections 8.3 and 8.4, we present strategies formalizing the agendas of Sections 3.3 and 3.4, respectively. A summary and directions to further research conclude the chapter.

8.1 From Agendas to Strategies

The transformation of agendas into strategies can be expressed as a "meta-agenda" because – regardless of the particular agenda to be formalized – the same decisions have to be taken. The meta-agenda is summarized in Table 8.1.

Step 1 *Decide which steps of the agenda become subproblems of the strategy.*

To decide on the subproblems generated by the strategy, we consider the steps of the agenda to be formalized one by one. The result of each step is a specification fragment. Each step can be captured in a strategy in three ways. First, if it cannot be forseen how the specification fragment being the result of the step looks like or how it is obtained, then the step is transformed into a subproblem.

Secondly, if the specification fragment always has the same shape, but user interaction will be necessary to set it up, then the step will not correspond to a subproblem, but there will be an existentially quantified expression in the definition of a constituting relation that needs the result of the step as its input. An example is the decision on the operational modes of a system.

No.	Step	Validation Conditions
1	Decide which steps of the agenda become subproblems of the strategy.	
2	Decide on the schemes of the constituting relations.	The dependency relation induced by the schemes of the constituting relations must be consistent with the dependencies of steps of the agenda. Only attributes corresponding to steps of the agenda that become subproblems occur in the schemes of the constituting relations.
3	Define the constituting relations.	All steps of the agenda must be treated.

Table 8.1: Agenda for transforming agendas into strategies

Thirdly, if the information needed to generate the specification fragment being the result of the step is already contained in the specification developed so far and need only be collected, then the step will not correspond to a subproblem, but constituting relations needing the result of the step as an input will contain a **let** expression defining the specification fragment in terms of the specification fragments developed previously. An example is step 5 of the passive sensors architecture, see page 42, where the Z control operation is routinely defined as a case distinction on the operational modes.

Step 2 *Decide on the schemes of the constituting relations.*

For each step of the agenda that is transformed into a subproblem of the strategy, we use two attribute names, one for the problem, and one for the corresponding solution. Independent subproblems should be gathered in a single constituting relation. To keep the strategy definition as simple as possible, the "clusters" of independent subproblems defined by each constituting relation should comprise as many subproblems as possible.

Validation Condition 2.1 *The dependency relation induced by the schemes of the constituting relations must be consistent with the dependencies of steps of the agendas.*

If steps i and j of the agenda are both transformed into subproblems, then an arrow from node i to node j in the dependency diagram of the agenda means that the constituting relation cr_j that contains the attributes defined for step j as its output attributes has at least one of the attributes defined for step i as an input attribute. If step i does not correspond to a subproblem, but step j does, then cr_j will contain an existential proposition or a **let** expression asserting the existence of the result of step i.

Validation Condition 2.2 *Only attributes corresponding to steps of the agenda that become subproblems occur in the schemes of the constituting relations.*

This condition requires that only for those steps of the agenda that become subproblems attribute names are chosen. The steps that do not become subproblems do not occur in any scheme of any constituting relation.

Step 3 *Define the constituting relations.*

To define a constituting relation, we must provide constraints on all its output attributes. Problem attributes must be given concrete values, which may refer to the values of the input attributes and to external information if necessary. Since solutions are generated by strategy applications, they are usually not defined (except for strategies that generate no subproblems), but constrained by local acceptability conditions. To define the local acceptability conditions, we must check if the corresponding step of the agenda has validation conditions associated with it that can be expressed formally. These validation conditions become the local acceptability conditions for the solution attribute.

Validation Condition 3.1 *All steps of the agenda must be treated.*

This condition is necessary for the strategy to be a faithful formalization of the agenda.

8.2 Problems, Solutions, and Acceptability

The definition of problems, solutions, and acceptability resembles the definition of Chapter 7. We do not develop Z expressions, however, but expressions of the combined language defined in Chapter 3, which consists of Z and real-time CSP. In the following definitions, the suffix "Z-CSP" refers to the combined language, the suffix "Z" refers to the Z part of a combined specification, and the suffix "CSP" refers to the real-time CSP part of a specification.

As in Chapter 7, we introduce basic types for the syntactically correct expressions of the combined language, for natural-language text, and for schematic expressions of the combined language.

$$[\mathit{SynZ\text{-}CSP}, \mathit{Text}, \mathit{SchematicZ\text{-}CSP}]$$

Semantically valid combined specifications are a subset of the syntactically correct combined specifications. As in Chapter 7, expressions of the combined language are associated with syntactic classes that are sets of expressions of the combined language. The empty string ϵ is a syntactically correct Z-CSP expression.

$$\begin{array}{|l}
\mathit{SemZ\text{-}CSP} : \mathbb{P}\,\mathit{SynZ\text{-}CSP} \\
\mathit{SyntacticClass} : \mathbb{P}(\mathbb{P}\,\mathit{SynZ\text{-}CSP}) \\
\epsilon : \mathit{SynZ\text{-}CSP}
\end{array}$$

We will use the following syntactic classes whose names are self-explanatory. The class *Z-CSP-specification* is the most general one.

$$\begin{array}{|l}
\mathit{Z\text{-}CSP\text{-}specification}, \mathit{ident} : \mathit{SyntacticClass} \\[1ex]
\mathit{Z\text{-}specification}, \mathit{Z\text{-}ax\_def}, \mathit{Z\text{-}enum\_type\_def} : \mathit{SyntacticClass} \\[1ex]
\mathit{CSP\text{-}specification}, \mathit{CSP\text{-}process\_expr}, \mathit{CSP\text{-}pred\_def}, \\
\qquad\qquad\qquad \mathit{CSP\text{-}alphabet\_def} : \mathit{SyntacticClass}
\end{array}$$

Each schematic Z-CSP expression is associated with the syntactic class of Z-CSP expressions with which it can be instantiated. The partial function `NL` concatenates two Z-CSP expressions. As in the previous chapter, the empty specification ϵ is a neutral element with respect to NL.

$$
\begin{array}{|l}
syn\_class : SchematicZ\text{-}CSP \longrightarrow SyntacticClass \\
instantiate : SchematicZ\text{-}CSP \times SynZ\text{-}CSP \pfun SynZ\text{-}CSP \\
\_\mathtt{NL}\_ : SynZ\text{-}CSP \times SynZ\text{-}CSP \pfun SynZ\text{-}CSP \\
\hline
\forall schem\_expr : SchematicZ\text{-}CSP \bullet \forall v : syn\_class\ schem\_expr \bullet \\
\quad (schem\_expr, v) \in \operatorname{dom} instantiate \\
\forall spec : SynZ\text{-}CSP \bullet \\
\quad (spec, \epsilon) \in \operatorname{dom} \mathtt{NL} \Rightarrow spec\,\mathtt{NL}\,\epsilon = spec \land \\
\quad (\epsilon, spec) \in \operatorname{dom} \mathtt{NL} \Rightarrow \epsilon\,\mathtt{NL}\,spec = spec
\end{array}
$$

A specification problem again consists of a requirement, expressed in natural language, the parts of the specification already developed, and a schematic Z-CSP expression. Each Z-CSP expression belonging to the syntactic class associated with the schematic Z-CSP expression must be combinable with the specification already developed.

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

Solutions are Z-CSP expressions:

$$SafSolution == SynZ\text{-}CSP$$

A solution *sol* 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.

$$
\begin{array}{|l}
\_saf\_acceptable\_for\_ : SafSolution \longleftrightarrow SafProblem \\
\hline
\forall sol : SafSolution; pr : SafProblem \bullet \\
\quad sol\ saf\_acceptable\_for\ pr \\
\quad \Leftrightarrow \\
\quad sol \in syn\_class(pr.to\_develop) \land \\
\quad pr.context\,\mathtt{NL}\,instantiate(pr.to\_develop, sol) \in SemZ\text{-}CSP
\end{array}
$$

8.3 A Strategy for the Passive Sensors Architecture

With this instantiation of the strategy framework, we can define a strategy that formalizes the agenda for the passive sensors architecture given in Section 8.3. Recall the definition of the agenda, which is repeated in Table 8.2. We follow the agenda of Table 8.1 to transform this agenda into a strategy.

No.	Step	Validation Conditions
1	Model the sensor values and actuator commands as members of Z types.	
2	Decide on the operational modes of the system.	
3	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. 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.	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.
5	Define the Z control operation.	
6	Specify the control process in real-time CSP.	
7	Specify further requirements if necessary.	

Table 8.2: Agenda for the passive sensors architecture

Step 1: Decide which steps of the agenda become subproblems of the strategy

Step 1 of Table 8.2 should become a subproblem because several types will have to be defined. This could be done, e.g., with an iterated strategy. For Step 2 no separate subproblem is generated, because the user only needs to decide on the operational modes of the system, which are then collected in a Z enumeration type. Steps 3 and 4 are larger development tasks that certainly should become subproblems. Steps 5 and 6, however, are performed in a routine way, merely instantiating schematic expressions of Z or CSP. Hence, they do not become separate subproblems. We need to introduce a subproblem for Step 7, because we know nothing about the further requirements or how they are developed.

Step 2: Decide on the schemes of the constituting relations

Using the information of Step 1 and the dependency diagram of the passive sensors agenda, we build the largest possible clusters of steps to be treated in the constituting relations. Figure 8.1 displays the results of Steps 1 and 2. The steps marked with an asterisk are the ones for which no separate subproblems are set up. The steps that are combined in the same constituting relation are indicated by shaded ovals. The names of the attributes defined for the various steps are given with the definition of the constituting relations.

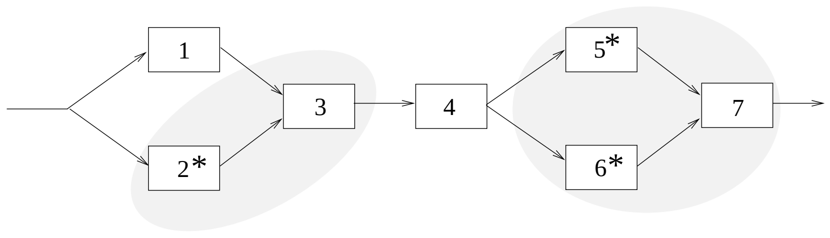


Figure 8.1: Strategy for passive sensors architecture

The clusters corresponding to the constituting relations respect the dependencies of the steps of the agenda. Hence Validation Condition 2.1 is fulfilled.

Step 3: Define the constituting relations

In the strategy *passive_sensors* we define now, the names of the constituting relations refer to the step numbers of the agenda.

As usual, we use a semi-formal Z-like notation to describe strategies, neither formalizing the syntax and semantics of Z-CSP, nor giving definitions for all functions and predicates we use. The type *Value* denotes the disjoint union of the schema types *SafProblem* and *SafSolution*, and its members are denoted by bindings, as in Chapter 6.

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

where *step_1* is defined by

$$
\begin{array}{l}
IA\ step\_1 = \{P\_init\} \\
OA\ step\_1 = \{P\_sens/act, S\_sens/act\} \\
step\_1 = \{\, t : scheme\ step\_1 \longrightarrow Value \mid \\
\quad syn\_class(t(P\_init).to\_develop) = Z\text{-}CSP\text{-}specification \wedge \\
\quad t(P\_sens/act) = \langle\, req \Rrightarrow t(P\_init).req\,;\text{“Model the sensor values and actuator commands as members of Z types.”}, \\
\qquad\qquad context \Rrightarrow t(P\_init).context, \\
\qquad\qquad to\_develop \Rrightarrow \mathsf{type\_defs} : Z\text{-}ax\_def \rangle \wedge \\
\quad t(S\_sens/act)\ saf\_acceptable\_for\ t(P\_sens/act)\}
\end{array}
$$

A natural-language text taken from the agenda describes the purpose of the new subproblem *P_sens/act*; it is added to the informal requirements component *req* using the concatenation function “;” for text. The schematic expression *to_develop* is denoted by type_defs : *Z-ax_def*. This notation means that type definitions must be developed that are expressed as axiomatic definitions in Z. The constituting relation *steps_2/3* is defined by

```
$IA\ steps\_2/3 = \{P\_init, S\_sens/act\}$
$OA\ steps\_2/3 = \{P\_state, S\_state\}$
$steps\_2/3 = \{\ t : scheme\ steps\_2/3 \longrightarrow Value\ |$
    $(\exists\ modes : Z\text{-}enum\_type\_def\ \bullet$
        $t(P\_state) = \langle\ req \Rrightarrow t(P\_init).req$ ; "Define the internal system states and
                                                                the initial states.",
                      $context \Rrightarrow (t(P\_init).context)$ NL $t(S\_sens/act)$ NL $modes$,
                      $to\_develop \Rrightarrow$ state_def : $Z\text{-}specification\rangle) \wedge$
    $t(S\_state)\ saf\_acceptable\_for\ t(P\_state) \wedge$
    $t(S\_state) \neq \epsilon \wedge$
    $t(S\_state)$ contains a state schema $S$ that
        has a component of the type defined by $modes$
        and is not imported by any other schema in $t(S\_state)$
        and an initial state schema for $S \wedge$
        the set of initial states is non-empty $\}$
```

The existential quantifier indicates that a heuristic function is used to obtain the enumeration type defining the operational modes. To define *P_state*, the types defining the sensor values and actuator commands contained in the solution *S_sens/act* and the type defined by *modes* are added to the *context* component of *P_init*. The solution *S_state* is constrained only to the syntactic class *Z-specification* because introducing new global definitions may be necessary for defining the system state schema, see Section 3.3.2. The local acceptability conditions for *S_state* capture those validation conditions associated with Step 3 that can be shown formally. The constituting relation *step_4* is defined by

```
$IA\ step\_4 = \{P\_init, P\_state, S\_state, S\_sens/act\}$
$OA\ step\_4 = \{P\_ops, S\_ops\}$
$step\_4 = \{\ t : scheme\ step\_4 \longrightarrow Value\ |$
    $t(P\_ops) = \langle\ req \Rrightarrow t(P\_init).req$ ; "Specify an internal Z operation for
                                                      each operational mode.",
                 $context \Rrightarrow t(P\_state).context$ NL $t(S\_state)$
                 $to\_develop \Rrightarrow$ ops_def : $Z\text{-}specification\rangle \wedge$
    $t(S\_ops)\ saf\_acceptable\_for\ t(P\_ops) \wedge$
    (let $Modes == (\mu\ enum\_type : Z\text{-}enum\_type\_def\ |$
            $t(P\_state).context = t(P\_init).context$ NL $t(S\_sens/act)$ NL $enum\_type)$;
        $mode == (\mu\ id : ident\ |$ the state schema defined in $t(S\_state)$ contains
            $id : Modes$ as a component) $\bullet$
        $t(S\_ops)$ contains an operation schema named $mOp$ for each mode $m$
                defined by $Modes \wedge$
            for each operation $mOp$: $mode = m \Rightarrow$ pre $mOp \wedge$
            for each operational mode and each combination of sensor values
                there is exactly one successor mode $\wedge$
            each operational mode is reachable from an initial state $\wedge$
            there are no redundant modes)$\}$
```

The system operations are defined in Z. As for the defining the system state, global definitions may be necessary. The local acceptability conditions for *S_ops* capture those validation conditions associated with Step 4 that can be shown formally. The type defined by *Modes*, which is needed to express the local acceptability conditions for S_ops, is the last specification fragment in $t(P\_state).context$. The constituting relation $steps\_5/6/7$ is defined by

$$\begin{array}{l}
IA\ steps\_5/6/7 = \{P\_init, P\_ops, S\_ops\} \\
OA\ steps\_5/6/7 = \{P\_other, S\_other\} \\
steps\_5/6/7 = \{\ t : scheme\ steps\_5/6/7 \longrightarrow Value \mid \\
\quad \exists\, INTERVAL : \mathbb{N} \bullet \\
\quad (\mathbf{let}\ z\_control\_op == (\mu\, s : schema \mid \\
\qquad\qquad s \text{ conforms to the schematic expression given in Step 5, page 42}); \\
\qquad csp\_control\_proc == (\mu\, proc : CSP\text{-}specification \mid \\
\qquad\qquad proc \text{ conforms to the schematic expression given in Step 6, page 43,} \\
\qquad\qquad \text{with wait interval } INTERVAL); \\
\qquad env\_ass == (\mu\, ass : CSP\text{-}pred\_def \mid \\
\qquad\qquad ass \text{ conforms to the schematic expression given in Step 6, page 43}); \\
\qquad sens/act\_ass == (\mu\, ass : CSP\text{-}pred\_def \mid \\
\qquad\qquad ass \text{ conforms to the schematic expression given in Step 6, page 43}) \bullet \\
\qquad t(P\_other) = \langle\ req \Rrightarrow t(P\_init).req\, ; \text{“Specify further requirements if necessary.”}, \\
\qquad\qquad\qquad context \Rrightarrow (t(P\_ops).context)\ \mathtt{NL}\ t(S\_ops)\ \mathtt{NL}\ z\_control\_op\ \mathtt{NL} \\
\qquad\qquad\qquad\qquad csp\_control\_proc\ \mathtt{NL}\ env\_ass\ \mathtt{NL}\ sens/act\_ass \\
\qquad\qquad\qquad to\_develop \Rrightarrow \mathsf{others} : Z\text{-}CSP\text{-}specification\rangle) \wedge \\
\quad t(S\_other)\ saf\_acceptable\_for\ t(P\_other)\}
\end{array}$$

The μ operator yields an item satisfying the given condition. The expressions $z\_control\_op$, $csp\_control\_proc$, env_ass and $sens/act\_ass$ can be automatically generated using the information contained in $t(P\_ops).context$ and $t(S\_ops)$, and the constant $INTERVAL$ that has to be supplied by the user. No assumptions can be made on the other definitions necessary to complete the specification. Hence the specification fragment that can be developed to solve P_other may have the syntactic class $Z\text{-}CSP\text{-}specification$, and no additional acceptability conditions for S_other besides $saf\_acceptable\_for$ can be stated. The constituting relation $pass\_sol$ assembles the final solution, using the context component of P_other, where all developed specification fragments are collected, and the solution S_other.

$$\begin{array}{l}
IA\ pass\_sol = \{P\_init, P\_other, S\_other\} \\
OA\ pass\_sol = \{S\_final\} \\
pass\_sol = \{\ t : scheme\ pass\_sol \longrightarrow Value \mid \\
\quad (\mathbf{let}\ sol == (\mu\, s : SafSolution \mid t(P\_other).context = t(P\_init).context\ \mathtt{NL}\ s) \bullet \\
\qquad t(S\_final) = sol\ \mathtt{NL}\ S\_other)\}
\end{array}$$

All steps of the agenda are taken into account in one of the constituting relations. Hence, validation condition 3.1 of the meta-agenda is fulfilled.

8.4 A Strategy for the Active Sensors Architecture

Recall the agenda for the active sensors architecture, which we repeat in Table 8.3. Again, we proceed according to the meta-agenda of Table 8.1 in transforming the agenda into a strategy.

No.	Step	Validation Condition
1	Model the sensors as CSP events or members of 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 successor mode for a mode-event pair must be justified. All modes must be reachable from an initial mode, and there must be no redundant modes.
5	Model the actuator commands as members of Z types or CSP events.	
6	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.
7	Specify a Z operation for each event that can cause a mode transition.	These operations must be consistent with the mode transition relation.
8	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.
9	Specify priorities on events if necessary.	The priorities must not be cyclic.
10	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 interface control process must be deterministic. The preconditions of the invoked Z operations must be satisfied.
11	Define the overall control process.	The auxiliary processes must communicate with the interface control process.
12	Define further requirements or environmental assumptions if necessary.	

Table 8.3: Agenda for the active sensors architecture

Step 1: Decide which steps of the agenda become subproblems of the strategy

There are four steps for which subproblems are not necessary. Step 2 yields just a set of internal events that must be given by the user. Similarly to the passive sensors strategy, to perform Step 3, the user must give the operational modes of the system and point out the initial modes. This information is then transformed into a Z enumeration type and a subset of this type. Third, the process defining the priorities on events, which is the result of Step 9, can be generated automatically from the information which event has priority over which other events. This information will again be obtained from the user. Finally, the overall control process being the result of Step 11 can be set up automatically according to the schematic expressions given in the agenda on page 61.

Step 2: Decide on the schemes of the constituting relations

As for the active sensors strategy, we use the information of Step 1 and the dependency diagram of the active sensors agenda to determine the largest possible clusters of steps to be treated in the constituting relations. Figure 8.2 displays the results of Steps 1 and 2. Again, the names of the attributes defined for the various steps are given with the definition of the constituting relations.

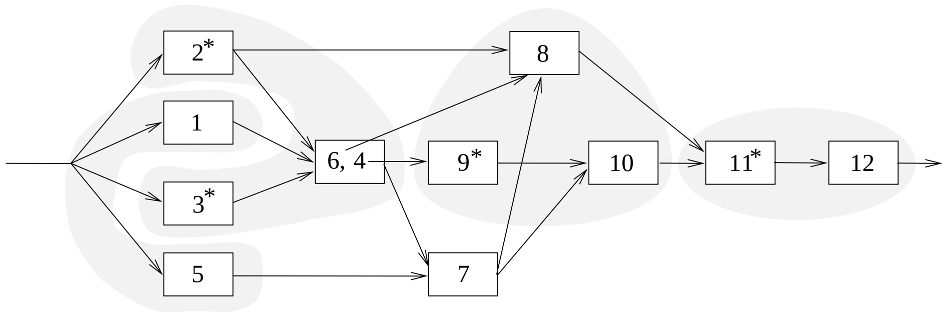


Figure 8.2: Strategy for active sensors architecture

Step 3: Define the constituting relations

As for the passive sensors strategy, the names of the constituting relations refer to the step numbers of the agenda.

$$active\_sensors = \{steps\_1/5, steps\_2/3/4/6, step\_7, steps\_8/9/10, steps\_11/12, act\_sol\}$$

where $step\_1/5$ is defined by

$IA\ steps\_1/5 = \{P\_init\}$
$OA\ steps\_1/5 = \{P\_sens, S\_sens, P\_act, S\_act\}$
$steps\_1/5 = \{\ t : scheme\ steps\_1/5 \longrightarrow Value\ |$
$\quad syn\_class(t(P\_init).to\_develop) = Z\text{-}CSP\text{-}specification\ \wedge$

$t(P\_sens) = \langle\, req \Rrightarrow t(P\_init).req$; "Model the sensors as CSP events or members of Z types.",
  $context \Rrightarrow t(P\_init).context,$
  $to\_develop \Rrightarrow \mathsf{sensor\_defs} : Z\text{-}CSP\text{-}specification \rangle \wedge$
$t(P\_act) = \langle\, req \Rrightarrow t(P\_init).req$; "Model the actuator commands as members of Z types or CSP events.",
  $context \Rrightarrow t(P\_init).context,$
  $to\_develop \Rrightarrow \mathsf{actuator\_defs} : Z\text{-}CSP\text{-}specification \rangle \wedge$
$t(S\_sens)\ saf\_acceptable\_for\ t(P\_sens) \wedge$
  $(\exists\, ext\_events : CSP\text{-}alphabet\_def;\ sol : Z\text{-}CSP\text{-}specification \bullet$
    $t(S\_sens) = sol\, \mathtt{NL}\, ext\_events)$
$t(S\_act)\ saf\_acceptable\_for\ t(P\_act) \}$

The modeling of the sensors and actuators may use Z as well as real-time CSP. Therefore, $t(P\_sens).to\_develop$ and $t(P\_act).to\_develop$ have syntactic class *Z-CSP-specification*. The solution $t(S\_sens)$ must contain the definition of a set of events *External_Events*, as described in Section 3.4.1, page 55. Because we will refer to this set in one of the other constituting relations, we require that the definition of *External_Events* is the last specification fragment contained in $t(S\_sens)$. The constituting relation *steps_2/3/4/6* is defined by

$IA\ steps\_2/3/4/6 = \{P\_init, S\_sens\}$
$OA\ steps\_2/3/4/6 = \{P\_mode\_trans, S\_mode\_trans, P\_state, S\_state\}$
$steps\_2/3/4/6 = \{\, t : scheme\ steps\_2/3/4/6 \longrightarrow Value \mid$
  $(\exists\, int\_events : CSP\text{-}alphabet\_def;\ modes : Z\text{-}enum\_type\_def;$
    $init\_modes : Z\text{-}ax\_def \mid$
      $init\_modes$ defines a subset of the type defined by $modes \bullet$
    $t(P\_mode\_trans) =$
      $\langle\, req \Rrightarrow t(P\_init).req$; "Set up a mode transition relation, specifying which events relate which modes.",
      $context \Rrightarrow t(P\_init).context\, \mathtt{NL}\, t(S\_sens)\, \mathtt{NL}\, modes\, \mathtt{NL}\, init\_modes$
        $\mathtt{NL}\, int\_events,$
      $to\_develop \Rrightarrow \mathsf{trans\_rel} : Z\text{-}specification \rangle \wedge$
    $t(P\_state) =$
      $\langle\, req \Rrightarrow t(P\_init).req$; "Define the internal system states and the initial states.",
      $context \Rrightarrow t(P\_init).context\, \mathtt{NL}\, t(S\_sens)\, \mathtt{NL}\, modes\, \mathtt{NL}\, init\_modes$
        $\mathtt{NL}\, int\_events,$
      $to\_develop \Rrightarrow \mathsf{state\_def} : Z\text{-}specification \rangle$
    $\wedge$
    $t(S\_mode\_trans)\ saf\_acceptable\_for\ t(P\_mode\_trans) \wedge$
    $(\mathbf{let}\ ext\_events == (\mu\, alph : CSP\text{-}alphabet\_def \mid$
        $\exists\, sol : Z\text{-}CSP\text{-}specification \bullet t(S\_sens) = sol\, \mathtt{NL}\, alph) \bullet$
      all events contained in the alphabet defined by ext_events
        and all modes contained in the type defined by $modes$ occur in
        $t(S\_mode\_trans)) \wedge$
    in the mode transition relation defined by $t(S\_mode\_trans)$ each mode
      must be reachable from an initial mode defined by the set $init\_modes$
      and there must be no redundant modes $\wedge$

$t(S\_state)$ *saf_acceptable_for* $t(P\_state) \wedge$
$t(S\_state) \neq \epsilon \wedge$
$t(S\_state)$ contains a state schema S that is not imported by any
  other schema in $t(S\_state)$ and an initial state schema for $S \wedge$
  the set of initial states is non-empty)}

To define the problems corresponding to Steps 4 and 6 of the agenda, we need the result of Steps 1, 2, and 3. The solution *S_sens* for Step 1 is an input attribute of the constituting relation *steps_2/3/4/6*, the results of Steps 2 and 3 are obtained by heuristic functions. The internal events *int_events* are last added to the context, because we will need them in a later constituting relation. The formalizable validation conditions of Steps 4 and 6 have become local acceptability conditions for the solutions $t(S\_mode\_trans)$ and $t(S\_state)$. The acceptability conditions for $t(S\_mode\_trans)$ refer to the external events defined earlier, whose definition is the last part of $t(S\_sens)$. The constituting relation *step_7* is defined by

$IA\ step\_7 = \{P\_init, P\_state, S\_state, S\_mode\_trans, S\_act\}$
$OA\ step\_7 = \{P\_ops, S\_ops\}$
$step\_7 = \{\ t : scheme\ step\_7 \longrightarrow Value \mid$
  $t(P\_ops) = \langle\ req \Rrightarrow t(P\_init).req\,;$ "Specify a Z operation for each event that can cause a mode transition.",
    $context \Rrightarrow t(P\_state).context\ \mathtt{NL}\ t(S\_mode\_trans)\ \mathtt{NL}\ t(S\_state)\ \mathtt{NL}\ t(S\_act)$
    $to\_develop \Rrightarrow \mathsf{ops\_def} : Z\text{-}specification\rangle \wedge$
  $t(S\_ops)$ *saf_acceptable_for* $t(P\_ops) \wedge$
  the operations defined by $t(S\_ops)$ must be consistent with the state transition
    relation defined by $t(S\_ops)\}$

As can be seen in Figure 8.2, Step 7 needs the results of Steps 5, 4, and 6 as an input. The corresponding solution attributes are input attributes for the constituting relation *step_7*. The operations must be defined in Z, and the validation condition associated with Step 7 is expressed as a local acceptability condition for $t(S\_ops)$. The constituting relation *steps_8/9/10* is defined by

$IA\ steps\_8/9/10 = \{P\_init, P\_ops, S\_ops\}$
$OA\ steps\_8/9/10 = \{P\_aux, S\_aux, P\_int\_ctrl, S\_int\_ctrl\}$
$steps\_8/9/10 = \{\ t : scheme\ steps\_8/9/10 \longrightarrow Value \mid$
  $t(P\_aux) = \langle\ req \Rrightarrow t(P\_init).req\,;$ "Define the auxiliary processes identified in Step 2.",
    $context \Rrightarrow t(P\_ops).context\ \mathtt{NL}\ t(S\_ops)$
    $to\_develop \Rrightarrow \mathsf{aux\_proc} : CSP\text{-}specification\rangle \wedge$
  $(\exists\, priority : CSP\text{-}pred\_def \mid priority$ defines non-cyclic priorities on events $\bullet$
    $t(P\_int\_ctrl) =$
      $\langle\ req \Rrightarrow t(P\_init).req\,;$ "Specify the interface control process.",
      $context \Rrightarrow t(P\_ops).context\ \mathtt{NL}\ t(S\_ops)\ \mathtt{NL}\ priority$
      $to\_develop \Rrightarrow \mathsf{interface\_control} : CSP\text{-}process\_expr\rangle \wedge$

```
    $t(S\_aux)$ $saf\_acceptable\_for$ $t(P\_aux) \wedge$
    $t(S\_int\_ctrl)$ $saf\_acceptable\_for$ $t(P\_int\_ctrl) \wedge$
    (**let** $int\_events == (\mu\, alph : CSP\text{-}alphabet\_def \mid$
            $\exists\, sol : Z\text{-}CSP\text{-}specification \bullet t(P\_state).context = sol \,\mathtt{NL}\, alph) \bullet$
        the union of the alphabets of all processes defined in $t(S\_aux)$
            is a subset of the events defined in $int\_events \wedge$
        all priorized external events and all events defined in $int\_events$ must occur
            as initial events of the branches of the interface control process defined by
            $t(S\_int\_ctrl) \wedge$
        the process defined by $t(S\_int\_ctrl)$ must be deterministic $\wedge$
        the preconditions of the invoked Z operations must be satisfied))}
```

To define the problem P_aux corresponding to Step 8, the values of the attributes P_ops and S_ops suffice. To define the problem P_int_ctrl, on the other hand, the process defining the priorities on events must be obtained using a heuristic function. The local acceptability conditions of both $t(S\_aux)$ and $t(S\_int\_ctrl)$ refer to the internal events defined earlier, whose definition is the last specification fragment contained in $t(P\_state).context$. The constituting relation $steps\_11/12$ is defined as in the passive sensors architecture by

```
$IA\ steps\_11/12 = \{P\_init, P\_int\_ctrl, S\_int\_ctrl, S\_aux\}$
$OA\ steps\_11/12 = \{P\_other, S\_other\}$
$steps\_11/12 = \{\, t : scheme\ steps\_11/12 \longrightarrow Value \mid$
    (**let** $ctrl\_proc == (\mu\, proc : CSP\text{-}specification \mid$
        $proc$ conforms to the schematic expression given in Step 11, page 61) $\bullet$
        $t(P\_other) =$
            $\langle\, req \Rrightarrow t(P\_init).req$ ; "Specify further requirements if necessary.",
            $context \Rrightarrow t(P\_int\_ctrl).context \,\mathtt{NL}\, t(S\_aux) \,\mathtt{NL}\, t(S\_int\_ctrl) \,\mathtt{NL}\, ctrl\_proc$
            $to\_develop \Rrightarrow \mathsf{others} : Z\text{-}CSP\text{-}specification\rangle) \wedge$
    $t(S\_other)\ saf\_acceptable\_for\ t(P\_other)\}$
```

The constituting relation act_sol assembles the final solution, using the context component of P_other, where all developed specification fragments are collected, and the solution S_other.

```
$IA\ pass\_sol = \{P\_init, P\_other, S\_other\}$
$OA\ pass\_sol = \{S\_final\}$
$pass\_sol = \{\, t : scheme\ pass\_sol \longrightarrow Value \mid$
    (**let** $sol == (\mu\, s : SafSolution \mid t(P\_other).context = t(P\_init).context \,\mathtt{NL}\, s) \bullet$
        $t(S\_final) = sol \,\mathtt{NL}\, S\_other)\}$
```

8.5 Summary

The results of this chapter are the following:

- A meta-agenda shows how agendas can be transformed into strategies in a fairly routine way.
- We have presented a third instance of the strategy framework.
- We have used this instance and the meta-agenda to define strategies that capture the agendas developed in Chapter 3, thus making them amenable to machine support.

8.6 Further Research

Further work on strategy-based specification of safety-critical software concerns the following points:

Implementation. The instantiation of the strategy framework presented in this chapter and the strategies for the two architectures should be implemented.

Case studies. This implementation should be used to perform case studies with the goal to compare working with the agendas on paper on the one hand and using the implemented strategies on the other hand. An important question is how restrictive a strategy should be to obtain the best balance between user guidance and flexibility.

Machine support for results of Section 3.8. Most of the research problems stated in Section 3.8 have an implementable counterpart. When concepts for the solutions of the problems stated there are developed, they deserve implementation in the strategy framework.

Chapter 9

Strategy-Based Development of Software Architectures

We now transform the agendas defined in Chapter 4 into strategies, after having defined an appropriate instance of the strategy framework. As in Chapter 8, the transformation follows the meta-agenda of Section 8.1. Before we close the chapter with a summary and directions to future research, we compare the four different instantiations of the strategy framework presented in Chapters 6–9.

9.1 Problems, Solutions, and Acceptability

The definitions of problem, solutions, and acceptability we present now can also be used for the development of LOTOS specifications for purposes other than architectural design.

As in Chapters 7 and 8, we introduce basic types for syntactically correct LOTOS expressions, for natural-language text, and for schematic LOTOS expressions.

$$[\mathit{SynLOTOS}, \mathit{Text}, \mathit{SchematicLOTOS}]$$

Semantically valid LOTOS specifications are a subset of the syntactically correct LOTOS specifications. To be able to state meaningful acceptability conditions, which capture the role of a specification fragment in its context, LOTOS expressions are associated with syntactic classes. These syntactic classes are sets of LOTOS expressions. The empty string ϵ is a syntactically correct LOTOS expression.

$$\begin{array}{|l}
\mathit{SemLOTOS} : \mathbb{P}\, \mathit{SynLOTOS} \\
\mathit{SyntacticClass} : \mathbb{P}(\mathbb{P}\, \mathit{SynLOTOS}) \\
\epsilon : \mathit{SynLOTOS}
\end{array}$$

Before we introduce the syntactic classes we will use in the strategy definitions, we summarize the syntax of the LOTOS constructs we generate as architectural descriptions using our strategies, see (Bolognesi and Brinksma, 1987). A top-level specification in LOTOS has the form

```
specification name [gate_list](parameter_list): functionality
  type_definition_list
behaviour
  behavior_expression
where
  type_definition_list
  process_definition_list
endspec
```

The syntactic classes occurring in this expression are set in *`italics teletype`*. The keyword `specification` only occurs on the top-level of a specification. To guarantee hierarchical compositionality of our architectural descriptions, we will only develop LOTOS expressions of the form

```
  behavior_expression
where
  type_definition_list
  process_definition_list
```

We call the corresponding syntactic class *`spec_body`*. A process definition has the form

```
process name [gate_list](parameter_list): functionality :=
  behavior_expression
where
  type_definition_list
  process_definition_list
endproc
```

We will use the syntactic classes introduced above, and the class *predicate* that comprises the predicates that guard behavioral expressions:

$$\begin{array}{|l}
\mathit{spec\_body},\\
\mathit{behavior\_expression},\\
\mathit{process\_definition\_list}, \mathit{process\_definition},\\
\mathit{type\_definition\_list}, \mathit{gate\_list}, \mathit{parameter\_list}\\
\mathit{predicate}, \mathit{name}, \mathit{functionality} : \mathit{SyntacticClass}
\end{array}$$

Each schematic LOTOS expression is associated with the syntactic class of LOTOS expressions with which it can be instantiated. The partial function `NL` concatenates two LOTOS expressions. The empty specification ϵ is a neutral element with respect to `NL`.

$$\begin{array}{|l}
\mathit{syn\_class} : \mathit{SchematicLOTOS} \longrightarrow \mathit{SyntacticClass}\\
\mathit{instantiate} : \mathit{SchematicLOTOS} \times \mathit{SynLOTOS} \rightarrowtail\!\!\!\!\!+ \mathit{SynLOTOS}\\
\_\mathtt{NL}\_ : \mathit{SynLOTOS} \times \mathit{SynLOTOS} \rightarrowtail\!\!\!\!\!+ \mathit{SynLOTOS}\\
\hline
\forall\, \mathit{schem\_expr} : \mathit{SchematicLOTOS} \bullet \forall\, v : \mathit{syn\_class}\ \mathit{schem\_expr} \bullet\\
\qquad (\mathit{schem\_expr}, v) \in \mathrm{dom}\, \mathit{instantiate}\\
\forall\, \mathit{spec} : \mathit{SynLOTOS} \bullet\\
\qquad (\mathit{spec}, \epsilon) \in \mathrm{dom}\, \mathtt{NL} \Rightarrow \mathit{spec}\, \mathtt{NL}\, \epsilon = \mathit{spec} \land\\
\qquad (\epsilon, \mathit{spec}) \in \mathrm{dom}\, \mathtt{NL} \Rightarrow \epsilon\, \mathtt{NL}\, \mathit{spec} = \mathit{spec}
\end{array}$$

A specification problem again consists of a requirement, expressed in natural language, the parts of the specification already developed, and a schematic LOTOS expression. Each LOTOS expression belonging to the syntactic class associated with the schematic LOTOS expression must be combinable with the specification already developed.

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

Solutions are LOTOS expressions:

$$\textit{ArchSolution} == \textit{SynLOTOS}$$

A solution *sol* 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 LOTOS specification.

$$
\begin{array}{|l}
\_\textit{arch\_acceptable\_for}\_ : \textit{ArchSolution} \leftrightarrow \textit{ArchProblem} \\
\hline
\forall\, \textit{sol} : \textit{ArchSolution};\ \textit{pr} : \textit{ArchProblem} \bullet \\
\quad \textit{sol}\ \textit{arch\_acceptable\_for}\ \textit{pr} \\
\quad \Leftrightarrow \\
\quad \textit{sol} \in \textit{syn\_class}(\textit{pr.to\_develop}) \wedge \\
\quad \textit{pr.context}\ \mathtt{NL}\ \textit{instantiate}(\textit{pr.to\_develop}, \textit{sol}) \in \textit{SemLOTOS}
\end{array}
$$

9.2 Strategies for the Repository Style

As usual, we use a semi-formal Z-like notation to describe strategies, neither formalizing the syntax and semantics of LOTOS, nor giving definitions for all functions and predicates we use. The type *Value* denotes the disjoint union of the schema types *ArchProblem* and *ArchSolution*, and its members are denoted by bindings.

Table 9.1 repeats the the agenda for the repository style. Recall that the steps have to be performed in the given order. To transform this agenda into a strategy, we carry out the steps given in Table 8.1.

Step 1: Decide which steps of the agenda become subproblems of the strategy

Step 3 of the agenda of Table 9.1 need not become a subproblem, because the overall architectural description can be assembled using the results of the first two steps.

Step 2: Decide on the schemes of the constituting relations

We will have one constituting relation for Step 1, one constituting relation for Step 2, and one constituting relation to assemble the final solution.

No.	Step	Validation Conditions
1	Define the types ***shared_memory***, ***id***, ***index*** and ***value***.	The type ***shared_memory*** must be defined according to the schema given in Section 4.2.1. The type ***id*** must contain a constant ***for_nobody***.
2	Define the component processes.	Each component process must be either a read, a write, or a read/write process.
3	Assemble the overall architectural description according to the communication pattern of the repository style.	The processes must communicate with the shared memory according to their being a read, write or read/write process, as described in the communication pattern.

Table 9.1: Agenda for the repository architectural style

Step 3: Define the constituting relations

$$rep\_arch = \{define\_types, define\_components, rep\_sol\}$$

where *define_types* is given by

$IA\ define\_types = \{P\_init\}$
$OA\ define\_types = \{P\_types, S\_types\}$
$define\_types = \{\ t : scheme\ define\_types \longrightarrow Value \mid$
  $syn\_class(t(P\_init).to\_develop) \supseteq spec\_body\ \wedge$
  $t(P\_types) = \langle\ req \Rrightarrow t(P\_init).req$; "Define the types ***shared_memory***, ***id***, ***index*** and ***value***.",
    $context \Rrightarrow t(P\_init).context,$
    $to\_develop \Rrightarrow$ type_defs : $type\_definition\_list\rangle\ \wedge$
  $t(S\_types)\ arch\_acceptable\_for\ t(P\_types)\ \wedge$
  $t(S\_types)$ contains the definition of a type ***shared_memory*** conforming to the schema given in Section 4.2.1, page 77 $\wedge$
  $t(S\_types)$ contains the definition of a type ***id*** with a constant ***for_nobody***$\}$

In contrast to the usual LOTOS semantics, we not only regard specification bodies, but also lists of type definitions and lists of process definitions as valid LOTOS specifications. Therefore, it is not necessary to embed the type definition list type_defs into a specification body with a trivial behavioral expression. The constituting relation *define_components* is given by

$IA\ define\_components = \{P\_init, S\_types\}$
$OA\ define\_components = \{P\_comps, S\_comps\}$
$define\_components = \{\ t : scheme\ define\_components \longrightarrow Value \mid$
  $t(P\_com) = \langle\ req \Rrightarrow t(P\_decls).req$; "Define the component processes.",
    $context \Rrightarrow t(P\_init).context\ \mathtt{NL}\ S\_types,$
    $to\_develop \Rrightarrow$ comp_procs : $process\_definition\_list\rangle\ \wedge$
  $t(S\_comps)\ arch\_acceptable\_for\ t(P\_comps)\ \wedge$
  $t(S\_comps) \neq \epsilon\ \wedge$
  each member of the list $t(S\_comps)$ defines a read, a write, or a read/write process$\}$

The constituting relation *rep_sol* is defined by

$IA\ rep\_sol = \{P\_init, S\_types, S\_comps\}$
$OA\ rep\_sol = \{S\_final\}$
$rep\_sol = \{\ t : scheme\ rep\_sol \longrightarrow Value \mid$
$\quad$ (**let** $shared\_mem\_def == (\mu\ proc : process\_definition \mid$
$\qquad\qquad$ *proc* conforms to the schematic process definition given on page 76);
$\qquad behav == (\mu\ bexp : behavior\_expression \mid$
$\qquad\qquad$ *bexp* conforms to the communication pattern given on page 79) $\bullet$
$\qquad t(S\_final) = behav$ `where` S_types `NL` $shared\_mem\_def$ `NL` $S\_comps \wedge$
$\qquad t(S\_final)\ arch\_acceptable\_for\ t(P\_init))\}$

The μ operator yields an item satisfying the given condition. In an implementation, we would have functions computing *shared_mem_def* and *behav*.

To perform Step 2 of the agenda of Table 9.1, we had defined another agenda in Chapter 4, which we repeat in Table 9.2. The steps of Table 9.2 must be performed in the given order. We transform this agenda into a strategy, again following the steps of the meta-agenda.

No.	Step	Validation Conditions
1	Decide if the component is a read, write, or read/write process.	
2	Define the component as a process.	The process definition must contain the patterns for the chosen kind of component.

Table 9.2: Agenda to develop components for a repository architecture

Step 1: Decide which steps of the agenda become subproblems of the strategy

Step 1 of the agenda of Table 9.2 needs only user interaction. No subproblem is necessary.

Step 2: Decide on the schemes of the constituting relations

We will have one constituting relation for Step 2, and one constituting relation to assemble the solution generated by the strategy.

Step 3: Define the constituting relations

$rep\_comp = \{define\_component, rep\_comp\_sol\}$

where *define_component* is defined by

$IA\ define\_component = \{P\_init\}$
$OA\ define\_component = \{P\_comp, S\_comp\}$

$define\_component = \{\, t : scheme\ define\_component \longrightarrow Value \mid$
$\quad syn\_class(t(P\_init).to\_develop) \supseteq process\_definition \wedge$
$\quad \exists\, ind : \{read, write, read\_write\} \bullet$
$\qquad t(P\_comp) = \langle\, req \Rrightarrow t(P\_init).req$; "Define the component as a *ind* process.",
$\qquad\qquad context \Rrightarrow t(P\_init).context,$
$\qquad\qquad to\_develop \Rrightarrow \textsf{comp\_def} : process\_definition\rangle \wedge$
$\qquad t(S\_comp)\ arch\_acceptable\_for\ t(P\_comp) \wedge$
$\qquad t(S\_comp)$ conforms to the pattern for the process of the kind indicated by
$\qquad\quad$ *ind*, see page 78$\}$

The constituting relation *rep_comp_sol* is defined by

$IA\ rep\_comp\_sol = \{S\_comp\}$
$OA\ rep\_comp\_sol = \{S\_final\}$
$rep\_comp\_sol = \{\, t : scheme\ rep\_comp\_sol \longrightarrow Value \mid t(S\_final) = t(S\_comp)\}$

The solution of problem *P_comp* can be either a subsystem that is an instance of some architectural style, or a simpler process. In the first case, problem *P_comp* would be reduced by a strategy *subarch* that generates the subproblem to develop a solution of syntactic class *spec_body* and embeds the solution into a process definition. The subproblem generated by the *subarch* strategy would then be reduced with a top-level strategy associated with the chosen style. For routine development of sub-architectures, combinations of the *subarch* strategy and the top-level strategies associated with the various architectural styles can be defined using the THEN strategical.

For the second case, a strategy *develop_process_definition* would be useful that generates the subproblems to define the behavior part of the process definition and its local definition part. This strategy would be defined similarly to the *define_schema* strategy of Section 7.3.2.

To solve the subproblem *P_comps* generated by the *rep_arch* strategy, we can iterate *rep_comp* using the strategy

$\textsc{Repeat}(\textsc{Lift}(rep\_comp, p\_down, p\_combine, s\_combine), p\_rep, empty)$

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

$p\_down == (\lambda\, pr : ArchProblem \mid syn\_class(pr.to\_develop) \supseteq process\_definition\_list \bullet$
$\qquad \langle req \Rrightarrow pr.req,$
$\qquad context \Rrightarrow pr.context,$
$\qquad to\_develop \Rrightarrow \textsf{comp\_def} : process\_definition\rangle)$

$p\_combine ==$
$\quad (\lambda\, pr : ArchProblem;\ sol : ArchSolution \mid$
$\qquad syn\_class(pr.to\_develop) \supseteq process\_definition\_list \wedge sol \in process\_definition \bullet$
$\qquad \langle req \Rrightarrow pr.req$; "define more component processes",
$\qquad context \Rrightarrow pr.context\ \mathtt{NL}\ sol,$
$\qquad to\_develop \Rrightarrow pr.to\_develop\rangle)$

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

The function *p_down* converts the problem of defining a list of processes into the problem of defining a single process, the function *p_combine* incorporates a developed process definition into the *context* part of a problem, and the function *s_combine* concatenates two specifications, thereby allowing the concatenation of a given process definition with an existing list of process definitions. Proving that the requirements for the application of LIFT are fulfilled proceeds as in Section 7.3.2, page 164.

9.3 Strategies for the Pipe/Filter Style

To develop architectures conforming to the pipe/filter style, we have the agenda repeated in Table 9.3, whose steps must be performed in the given order. In transforming the agenda into a strategy we proceed as usual.

No.	Step	Validation Conditions
1	Define the filters one by one.	Each filter must fulfill the conditions stated in the component characteristics part of the style characterization.
2	Assemble the filters according to the pattern given in the communication pattern part of the style characterization.	The architectural description must fulfill the constraints stated in the constraints part of the style characterization.

Table 9.3: Agenda for the pipe/filter architectural style

Step 1: Decide which steps of the agenda become subproblems of the strategy

Assembling the developed filters in the way prescribed by the communication pattern of the pipe/filter architectural style characterization is a routine task. Hence, there will be no subproblem corresponding to Step 2 of the agenda shown in Table 9.3.

Step 2: Decide on the schemes of the constituting relations

Consequently, we will define one constituting relation for Step 1, and one to assemble the final solution.

Step 3: Define the constituting relations

$$p/f\_arch = \{define\_filters, p/f\_sol\}$$

where *define_filters* is defined by

$$IA\ define\_filters = \{P\_init\}$$
$$OA\ define\_filters = \{P\_filt, S\_filt\}$$

$define\_filters = \{\, t : scheme\ define\_filters \longrightarrow Value \mid$
$\quad syn\_class(t(P\_init).to\_develop) \supseteq spec\_body \wedge$
$\quad t(P\_filt) = \langle\, req \Rrightarrow t(P\_init).req$; "Define the filters one by one.",
$\qquad context \Rrightarrow t(P\_init).context,$
$\qquad to\_develop \Rrightarrow \mathsf{filt\_defs} : process\_definition\_list\rangle \wedge$
$\quad t(S\_filt)\ arch\_acceptable\_for\ t(P\_filt) \wedge$
$\qquad$ each member of the list $t(S\_filt)$ fulfills the conditions stated in the component characteristics part of the style characterization, see page 81}

The constituting relation p/f_sol is defined by

$IA\ p/f\_sol = \{S\_filt\}$
$OA\ p/f\_sol = \{S\_final\}$
$p/f\_sol = \{\, t : scheme\ p/f\_sol \longrightarrow Value \mid$
$\quad$(**let** $behav == (\mu\ bexp : behavior\_expression \mid$
$\qquad bexp$ conforms to the schematic behavioral definition given on page 82) •
$\qquad t(S\_final) = behav$ `where` $t(S\_filt) \wedge$
$\qquad t(S\_final)$ fulfills the constraints stated in the style characterization on page 82}

To perform Step 1 of the agenda of Table 9.3, we had defined another agenda, which we repeat in Table 9.4. The steps 1 and 2 of Table 9.4 are independent of one another. We transform this agenda into a strategy, again following the steps of the meta-agenda.

No.	Step	Validation Conditions
1	Decide on the pipes that connects the filter with other filters.	
2	Decide on the gates of the filter with the environment.	
3	Define the filter as a process.	The process must fulfill the conditions stated in the characteristics part.

Table 9.4: Agenda to develop components for a pipe/filter architecture

Step 1: Decide which steps of the agenda become subproblems of the strategy

Steps 1 and 2 of the agenda of Table 9.4 will not correspond to subproblems because their results are simple gate lists that must be given by the user.

Step 2: Decide on the schemes of the constituting relations

Consequently, we will define one constituting relation corresponding to Step 3, and one to assemble the final solution.

Step 3: Define the constituting relations

$p/f\_comp = \{define\_component, p/f\_comp\_sol\}$

where *define_component* is defined by

$IA\ define\_component = \{P\_init\}$
$OA\ define\_component = \{P\_comp, S\_comp\}$
$define\_component = \{\ t : scheme\ define\_component \longrightarrow Value\ |$
$\quad syn\_class(t(P\_init).to\_develop) \supseteq process\_definition\ \wedge$
$\quad \exists\ pipe\_list, ext\_gate\_list : gate\_list\ \bullet$
$\quad\quad t(P\_comp) = \langle\ req \Rrightarrow t(P\_init).req\,;$ "Define the filter as a process.",
$\quad\quad\quad context \Rrightarrow t(P\_init).context,$
$\quad\quad\quad to\_develop \Rrightarrow \mathsf{comp\_def} : process\_definition\rangle\ \wedge$
$\quad\quad t(S\_comp)\ arch\_acceptable\_for\ t(P\_comp)\ \wedge$
$\quad\quad gatelist(t(S\_comp)) = pipe\_list \cup ext\_gate\_list\ \wedge$
$\quad\quad t(S\_comp)$ fulfills the conditions stated in the component characteristics part
$\quad\quad\quad$ of the style characterization, see page 81$\}$

The function *gatelist* yields the list of gates of a process definition. The constituting relation *p/f_comp_sol* is defined by

$IA\ p/f\_comp\_sol = \{S\_comp\}$
$OA\ p/f\_comp\_sol = \{S\_final\}$
$p/f\_comp\_sol = \{\ t : scheme\ p/f\_comp\_sol \longrightarrow Value\ |\ t(S\_final) = t(S\_comp)\}$

To solve the subproblem *P_filt* generated by the *p/f_arch* strategy, we can iterate *p/f_comp* using the strategy

$\textsc{Repeat}(\textsc{Lift}(p/f\_comp, p\_down, p\_combine, s\_combine), p\_rep, empty)$

where all arguments of LIFT except *p/f_comp* are defined as in the previous section.

9.4 Strategies for the Event-Action Style

Table 4.5 repeats the agenda defined in Chapter 4, whose steps must be performed in the given order.

No.	Step	Validation Conditions
1	Define the type `event`.	
2	Define pairs, consisting of a predicate on the type `event` and a process defining the corresponding action.	Each action process must communicate with the event manager and define the reaction to the events that fulfill the defined predicate.
3	Define the process `Event_Manager` and assemble the overall architectural description according to the communication pattern.	The definition of the event manager must conform to the pattern given in the component characteristics, and it must be consistent with Step 2.

Table 9.5: Agenda for the event-action architectural style

Step 1: Decide which steps of the agenda become subproblems of the strategy

The definition of the process `Event_Manager` can be assembled from the results of the first two steps. Therefore, the third step of the agenda of Table 9.5 does not require the definition of separate subproblem.

Step 2: Decide on the schemes of the constituting relations

Two of the constituting relations correspond to the steps 1 and 2, and the third defines the final solution.

Step 3: Define the constituting relations

$$\mathit{e\text{-}a\_arch} = \{\mathit{define\_events}, \mathit{define\_components}, \mathit{e\text{-}a\_sol}\}$$

where *define_events* is defined by

```
$\mathit{IA\ define\_events} = \{\mathit{P\_init}\}$
$\mathit{OA\ define\_events} = \{\mathit{P\_events}, \mathit{S\_events}\}$
$\mathit{define\_events} = \{\ t : \mathit{scheme\ define\_events} \longrightarrow \mathit{Value}\ |$
    $\mathit{syn\_class}(t(\mathit{P\_init}).\mathit{to\_develop}) \supseteq \mathit{spec\_body}$
    $t(\mathit{P\_events}) = \langle\ \mathit{req} \Rrightarrow t(\mathit{P\_init}).\mathit{req}$ ; "Define the type event.",
                    $\mathit{context} \Rrightarrow t(\mathit{P\_init}).\mathit{context}$,
                    $\mathit{to\_develop} \Rrightarrow \mathsf{event\_def} : \mathit{type\_definition\_list}\rangle \wedge$
    $t(\mathit{S\_events})\ \mathit{arch\_acceptable\_for}\ t(\mathit{P\_events})\}$
```

The constituting relation *define_components* is defined by

```
$\mathit{IA\ define\_components} = \{\mathit{P\_init}, \mathit{S\_events}\}$
$\mathit{OA\ define\_components} = \{\mathit{P\_comps}, \mathit{S\_comps}\}$
$\mathit{define\_components} = \{\ t : \mathit{scheme\ define\_components} \longrightarrow \mathit{Value}\ |$
    $t(\mathit{P\_com}) = \langle\ \mathit{req} \Rrightarrow t(\mathit{P\_decls}).\mathit{req}$ ; "Define the component processes.",
                    $\mathit{context} \Rrightarrow t(\mathit{P\_init}).\mathit{context}\ \mathtt{NL}\ \mathit{S\_events}$,
                    $\mathit{to\_develop} \Rrightarrow \mathsf{comp\_procs} : \mathit{process\_definition\_list}\rangle \wedge$
    $t(\mathit{S\_comps})\ \mathit{arch\_acceptable\_for}\ t(\mathit{P\_comps}) \wedge$
    $t(\mathit{S\_comps}) \neq \epsilon \wedge$
    each member of the list $t(\mathit{S\_comps})$ conforms to the component characteristics
        of the style characterization, see page 85}
```

The constituting relation *e-a_sol* is defined by

```
$\mathit{IA\ rep\_sol} = \{\mathit{P\_init}, \mathit{S\_events}, \mathit{S\_comps}\}$
$\mathit{OA\ rep\_sol} = \{\mathit{S\_final}\}$
$\mathit{rep\_sol} = \{\ t : \mathit{scheme\ rep\_sol} \longrightarrow \mathit{Value}\ |$
    $(\exists\ \mathit{event\_manager\_def} : \mathit{process\_definition}\ |$
        $\mathit{event\_manager\_def}$ conforms to the pattern given in the component
            characteristics, see page 84, and is consistent with$\mathit{S\_comps}$ $\bullet$
        (let $\mathit{behav} == (\mu\ \mathit{bexp} : \mathit{behavior\_expression}\ |$
            $\mathit{bexp}$ conforms to the communication pattern given on page 85) $\bullet$
        $t(\mathit{S\_final}) = \mathit{behav}$ where $\mathit{S\_events}\ \mathtt{NL}\ \mathit{event\_manager\_def}\ \mathtt{NL}\ \mathit{S\_comps} \wedge$
        $t(\mathit{S\_final})\ \mathit{arch\_acceptable\_for}\ t(\mathit{P\_init}))\}$
```

We have used an existential quantifier to indicate that user interaction may be necessary to set up the definition of the `Event_Manager` process. The other parts of the solution can be assembled automatically, as indicated by the use of the **let** construct.

To perform Step 2 of the agenda of Table 9.5, we had defined another agenda, which we repeat in Table 9.6. The steps of Table 9.6 must be performed in the given order. We transform this agenda into a strategy, again following the steps of the meta-agenda.

No.	Step	Validation Conditions
1	Decide on the events to be treated.	
2	Define the action to be taken as a process.	The process definition conforms to the component characteristics given in the style characterization.

Table 9.6: Agenda to develop components for an event-action architecture

Step 1: Decide which steps of the agenda become subproblems of the strategy

The predicate that defines to which events the component process reacts will be given by the user. Hence, there will only be a subproblem corresponding to Step 2 of the agenda.

Step 2: Decide on the schemes of the constituting relations

Consequently, we will define one constituting relation corresponding to Step 2, and one to assemble the final solution.

Step 3: Define the constituting relations

$$\mathit{e\text{-}a\_comp} = \{\mathit{define\_component}, \mathit{e\text{-}a\_comp\_sol}\}$$

where *define_component* is defined by

```
$\mathit{IA\ define\_component} = \{\mathit{P\_init}\}$
$\mathit{OA\ define\_component} = \{\mathit{P\_comp}, \mathit{S\_comp}\}$
$\mathit{define\_component} = \{\ t : \mathit{scheme\ define\_component} \longrightarrow \mathit{Value}\ |$
    $\mathit{syn\_class}(t(\mathit{P\_init}).\mathit{to\_develop}) \supseteq \mathit{process\_definition} \land$
    $\exists\, \mathit{event\_type\_def} : \mathit{type\_definition};\ \mathit{sol} : \mathit{ArchSolution}\ |$
        $t(\mathit{P\_init}).\mathit{context} = \mathit{sol}\ \mathtt{NL}\ \mathit{event\_type\_def}\ \bullet$
        $\exists\, \mathit{pred} : \mathit{predicate}\ |$
            $\mathit{pred}$ is a predicate on the type defined by $\mathit{event\_type\_def}\ \bullet$
        (let $\mathit{compname\_aux} : \mathit{name}$ be a new name $\bullet$
        $t(\mathit{P\_comp}) =$
            $\langle\ \mathit{req} \Rrightarrow t(\mathit{P\_init}).\mathit{req}$ ; "Define the action to be taken as a process.",
            $\mathit{context} \Rrightarrow t(\mathit{P\_init}).\mathit{context}$,
        $to\_develop \Rrightarrow$
            process $compname\_aux$ : $functionality$(comp_def) :=
                [ $pred$ ] -> $process\_instantiation$(comp_def)
                where
                comp_def : $process\_definition$
                endproc$\rangle \wedge$
    $t(S\_comp)$ $arch\_acceptable\_for$ $t(P\_comp) \wedge$
    $t(S\_comp)$ fulfills the conditions stated in the component characteristics part
        of the style characterization, see page 85)}
```

The *e-a_comp* strategy is applicable only if the context part of the initial problem contains the definition of an event type. The predicate obtained by a heuristic function must refer to this type. The new name *compname_aux* can be generated automatically. The solution $t(S\_comp)$ of the problem $t(P\_comp)$ is an expression comp_def that must belong to the syntactic class *process_definition*. This process definition is embedded in a larger process definition that contains the predicate indicating when the action must be taken. The embedding process *compname_aux* has the same functionality as the process defined by comp_def. Its behavior part only consists of a guarded expression: in case the predicate holds, the process defining the action, which is contained in the local definition list of the embedding process definition, is executed. The constituting relation *e-a_comp_sol* is defined by

$$
\begin{aligned}
&IA\ e\text{-}a\_comp\_sol = \{P\_comp, S\_comp\}\\
&OA\ e\text{-}a\_comp\_sol = \{S\_final\}\\
&e\text{-}a\_comp\_sol = \{\ t : scheme\ e\text{-}a\_comp\_sol \longrightarrow Value \mid\\
&\qquad t(S\_final) = instantiate(t(P\_comp).to\_develop, t(S\_comp))\}
\end{aligned}
$$

The final solution is just the instantiation of the schematic expression $t(P\_comp).to\_develop$ with the process definition $t(S\_comp)$, using the function *instantiate* introduced in Section 9.1.

To solve the subproblem *P_comps* generated by the *e-a_arch* strategy, we can iterate *e-a-_comp* using the strategy

$$\text{REPEAT}(\text{LIFT}(e\text{-}a\_comp, p\_down, p\_combine, s\_combine), p\_rep, empty)$$

where all arguments of LIFT except *embox−a_comp* are defined as in the previous section.

9.5 Comparing Instantiations of the Strategy Framework

The instantiations of the strategy framework presented in Chapters 6–9 differ in several important respects. The differences between program synthesis on the one hand and specification acquisition and software design on the other hand are reflected in their respective instantiations, and are visible in the following specific phenomena[1]:

[1]Since we use a specification language to express software designs, we only speak of "specification acquisition" in the following.

Instantiation of generic parameters

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

In specification acquisition, such a formal relation is impossible because the requirements for a software system are described informally. Indeed, specification acquisition actually leads from informal to formal artifacts in the software engineering process. The general definition of acceptability can therefore refer only to the formal specification, and not to the requirements. By contrast with program synthesis, where all partial solutions are statements, the partial solutions in specification acquisition belong to different syntactic classes, and so the *general* notion of acceptability is static type correctness, although for *individual* strategies, stronger acceptability conditions can be stated. These conditions reflect the purpose of the different parts of the specification in the context of a given strategy, requiring, e.g., that – when specifying a state-based system – the global definition part of a specification should not define the system state, or that system operations should have satisfiable preconditions. Consistency and completeness criteria can also be stated in the context of particular strategies.

Independent subproblems

In program synthesis, the subproblems generated by a strategy are often independent of each other. When developing a conditional, for example, the two branches can be developed in any order or in parallel.

Specification acquisition, on the other hand, proceeds much more incrementally, and so later parts of a specification usually refer to its earlier parts. To define the operations of a system, for instance, its state must be known. None of the strategies defined for specification acquisition in Chapter 7, or software design in Chapter 9 accommodates solution of independent subproblems. Only the strategy for the active sensors architecture presented in Section 8.4 generates independent subproblems.

Incomplete solutions

The fact that subproblems in specification acquisition can depend strongly on one another has an influence on how work with strategies proceeds. Experience has shown that it is unrealistic to assume that, if problem P_2 depends on the solution S_1 of problem P_1, then it will necessarily be possible first to solve P_1 completely and then start working to solve P_2. In the *state_based* strategy (see page 160), the definition of the state and the operations will usually make use of the global definitions, but we cannot assume that a specifier foresees all necessary global definitions in advance. The process that implements problem solving with strategies must therefore allow specifiers to work on a problem even if the solution it depends on is not yet completely known. Technically, we can achieve this by propagating incomplete solutions.

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

Use of repetition

Frequently, in specification acquisition, several items of the same kind must be developed to solve a problem, as is the case with in *P_state* and *P_ops* in the *state_based* strategy. Such development can be supported by the strategicals REPEAT and LIFT, as described in Section 7.3. If several items of different syntactic classes have to be developed, as with the global definitions *P_global* of the *state_based* strategy, then this can be achieved using the LIFT strategical without combining it with REPEAT.

For program synthesis, repeating a strategy is not as useful as it is in specification acquisition. In developing a program, it is not necessarily helpful to consider it as a concatenation of items from the syntactic class *statement*. This is due to the fact that programming problems provide much more detailed and semantic information than do specification problems, simply because the former are formal, which the latter are not. In addition, the syntactic forms of programming problems may already suggest strategies to apply to them, such that strategy selection can rely more on the specific problem in program synthesis than it can in specification acquisition.

These considerations show that program synthesis on the one hand and specification acquisition and software design on the other hand are fairly different activities. Strategies are, however, sufficiently general to support them all.

9.6 Summary

The results of this chapter are the following:

- We have presented a fourth instance of the strategy framework.
- We have used this fourth instance and the meta-agenda of Section 8.1 to define strategies that capture the agendas developed in Chapter 4, thus making them amenable to machine support.
- We have compared the different instantiations of the strategy framework, showing that this framework is sufficiently powerful to support a variety of software engineering activities.

9.7 Further Research

Further work on strategy-based development of software architecture concerns the following points:

Implementation. The instantiation of the strategy framework presented in this chapter and the strategies for the three architectural styles should be implemented.

More strategies. General-purpose strategies as the ones defined in Chapter 7 should be defined for the LOTOS instantiation of the strategy framework.

Integration with existing software. When conducting our case study of Chapter 4, we first developed the robot designs and then analyzed and compared the designs using an existing tool. It should be investigated how existing LOTOS tools can be used *during* the development of a design. First, such tools could be useful to check some of the acceptability conditions of the strategies. Secondly, they can support an explorative process of software design.

Comparative Studies. The LOTOS instantiation of the strategy framework could also be used to specify safety-critical software or for general specification tasks. It should be investigated how strategies defined for one instantiation of the strategy framework, e.g., the instantiation of Chapter 8, can be transformed into strategies for a different instantiation, e.g., the instantiation of this chapter, and to what extent the development steps are independent of the used specification language.

Machine support for results of Section 4.8. Most of the research problems stated in Section 4.8, particularly architecture refinement, have an implementable counterpart. Therefore, when concepts for the solutions of the problems stated there are developed, more implementation tasks will arise.

Chapter 10

Conclusions

In the previous chapters, we have introduced the concepts of an agenda and of a strategy. We have shown how these concepts can be profitably employed in various phases of the software life cycle. In particular, we have investigated the use of these concepts in the areas of software specification, design, and implementation. We have presented a general pragmatic approach to the usage of formal specification techniques, and a specialized approach to the specification of software for safety-critical applications. Furthermore, we have shown how software design according to architectural styles can be supported with formal methods in a semantically sound manner. Program synthesis techniques were also considered. In summary, this work contributes to the following areas of research:

Methodological support for the application of formal techniques in software engineering

With the concept of an agenda, we have introduced a means for organizing work that has to be carried out in a particular context. Agendas are obtained in a knowledge engineering process from domain experts and – if formal techniques are to be applied – from experts in formal techniques. Their purpose is to capture the knowledge used by the domain experts when carrying out their tasks. Agendas are specific to the task to be performed, not to the formalism to be used. Therefore, the use of agendas can be smoothly introduced into an organization. Developers essentially proceed as before, only that the steps to be taken in performing the task are made explicit. Hence, introducing agendas does not de-skill developers.

The use of agendas to guide the application of formal techniques in software engineering results in a precise description of the artifacts to be developed and a rigorous means for validating them. In this way, the application of formal methods in software engineering contributes to a better quality of the artifacts produced during the software development process. Informal techniques are not made superfluous, but there is a synergetic effect between formal and informal techniques.

Agendas have much in common with approaches to software process modeling (Huff, 1996). The difference is that software process modeling techniques concentrate more on management activities, e.g., roles of developers, than on technical issues. In contrast, with agendas we always develop a document, thus concentrating on technical activities in software engineering. Furthermore, software process modeling does not place so much emphasis on validation issues as agendas do.

In this work, we have shown that agendas are useful for the specification, design and implementation phases of the software lifecycle, when formal techniques are used. But also software development methods that are not based on formal techniques can be supported with agendas: First, we have set up a preliminary agenda for requirements elicitation, which shows that also requirements engineering can be supported with the agenda approach. Second, we have defined an agenda for the object-oriented Fusion method (Coleman et al., 1994), which provides valuable consistency checks for the various models of the analysis and design phases that have to be set up during the Fusion process. It seems reasonable to assume that systematic testing, transformation of specifications and programs, and re-engineering can be supported with agendas as well. The definition of concrete agendas for these tasks, however, remains a task for the future. Potentially, agendas may even be used to guide performing tasks in other fields than software engineering.

Currently, more agendas for the specification of safety-critical embedded software are developed in the German project ESPRESS (ESPRESS), which is a joint project with partners from industry, research institutions, and universities. In this project, a combination of Z and statecharts (Harel, 1987) is used instead of the combination of Z and CSP described in Chapter 3. First results have shown that – if the languages used are comparable in their expressive power, as is the case for CSP and statecharts – the activities constituting the agendas are the same for different languages; only the schematic expressions and some of the validation conditions are different.

Because agendas can be employed whenever there is a systematic way of performing a software development task, it is clear that agendas can also be defined when specification languages other than Z, CSP, or LOTOS are used. However, further research is necessary to find out to which degree agendas can be oriented solely on the task to be performed, rather than on the language that is used. For a language-independent agenda, changing the language in which a document is expressed would effect only the schematic expressions and language-specific validation conditions. The steps of the agenda would remain the same.

Application of formal specification techniques

In addition to using agendas, the barriers that currently prevent the application of formal techniques in software engineering practice can further be lowered by adopting a pragmatic attitude toward formal techniques. We have shown how formal specification techniques can be smoothly integrated into traditional processes, and how formal specification discipline can be relaxed to avoid some of the difficulties that make the application of formal techniques sometimes tedious. We have shown that legacy systems can benefit from a formal specification, too.

Moreover, we have identified different specification styles that make the process of specification acquisition language-independent to a large extent. These styles can be formalized as sets of strategies.

Safety

To support the formal specification of software for safety-critical applications, we have investigated the expressional power a formal language suitable for this purpose must possess. We have defined such a language by combining two existing, well-established languages. We have defined a software model for the use of the combined language and have further refined this model by reference architectures capturing frequently used designs of safety-critical systems. For each such reference architecture, we have defined an agenda that gives detailed guidance for developing specifications of software components suitable for the architecture and for the component validation. Besides the application-independent validation mechanisms provided by the agendas, we have shown how the developed specifications can be further validated, taking properties of the particular application into account. The strategies that were defined corresponding to the agendas make the implementation of a support system for strategy-based specification of safety-critical software a routine task.

All in all, our approach to supporting the specification of software for safety-critical applications enhances the safety of the entire technical system, because the embedded software is specified in a systematic way, the specification has an unambiguous semantics, and the specification is validated more rigorously than this would be possible with purely informal specifications.

Software Architectures

The definition of architectural styles is a means for making software design knowledge explicit and supporting its reuse by using styles as guidelines for the development of concrete software designs. We have provided a semantic foundation of architectural styles by characterizing such styles using patterns over the formal description language LOTOS. A number of agendas support designers in the development of software architectures conforming to the characterized styles. The agendas were formalized as strategies, providing a basis for machine-supported software design.

Our approach to software design leads to standardized, comprehensible, and comparable designs that have a precise semantics and thus can be analyzed and validated in a rigorous manner. But not only the *result* of the design process is improved in comparison to using informal style descriptions or no styles at all, but also the *process* of developing the architectural designs is now standardized and comprehensible.

Automated Software Engineering

Agendas are designed to be applied by humans. As already pointed out in Chapter 1, the benefits of formal methods can be even better appreciated when they are supported by machine.

Strategies support the representation and application of software development knowledge by machine. As they are formally defined, they introduce even more preciseness and rigor into software engineering processes than agendas. Strategies are implementation-oriented. Detailed implementation guidance is provided in the form of an architecture for support systems for strategy-based development activities. Strategies also form the basis for automating portions of software development activities.

The strategy framework is generic. The notions of problems, solutions, and acceptability can be freely defined. In the previous chapters, we have presented four different instances of the specification framework, covering the specification, design, and implementation phases of the software life cycle. The existence of these instances shows that the strategy framework is truly generic, and that strategies are powerful enough to support a wide variety of software engineering activities. Moreover, we have shown that the formalization of appropriately engineered knowledge is possible in a routine manner.

Different support systems implementing different instantiations of the strategy framework have a strong potential for successful combination. Such combinations can provide integrated tool support for several consecutive phases of a software life cycle.

Agendas, resulting from our approach to knowledge *engineering*, and strategies, resulting from our approach to knowledge *representation*, lead to a uniform and flexible approach for supporting the application of formal techniques in software engineering.

Standardization of Products and Processes in Software Engineering

This work supports recent trends in software engineering that attempt to detect patterns of use and represent connections between software engineering artifacts that go beyond syntax. Achieving these goals is important for mastering the increasing complexity of software engineering artifacts. The emerging field of software architectures (Shaw and Garlan, 1996) and the development of design patterns (Gamma et al., 1995) are prominent examples of this kind of research.

In the field of software architecture, architectural styles (which have been formally characterized in Chapter 4) capture frequently used design principles for software systems. Design patterns 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. In the same way as architectural styles, we have formally characterized design patterns for which communication between objects is important[1]. Whereas concrete agendas are very much oriented on the activity they support, the general concept of an agenda is not specialized to an activity such as software design or a programming paradigm such as object-orientedness, as is the case for architectural styles and design patterns. Apart from the fact that these concepts are more specialized in their application than agendas, the main difference is that they do not describe *processes* but *products*.

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 and strategies support the trends described above because development tasks are performed in a standardized way. This not only supports developers but also other persons that must understand the development process and its results, for example, because they must change or further develop the artifact in question. This standardization also makes certification procedures much more realistic and meaningful.

In engineering disciplines such as mechanical or civil engineering, it is taken for granted that the design of a new machine or a new building uses standardized parts and that standardized processes are followed. Agendas and strategies can surely help us achieve such a situation.

[1]Examples are the *Facade*, *Chain of Responsibility*, *Mediator*, *Observer*, and *Strategy* design patterns.

Bibliography

Abowd, G., Allen, R., and Garlan, D. (1993). Using style to understand descriptions of software architecture. In Notkin, D., editor, *Proceedings of the first ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 9–20. ACM Press.

Allan, R. and Garlan, D. (1994). Formalizing architectural connection. In *Proceedings 16th Int. Conf. on Software Engineering.* ACM Press.

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

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

Boehm, B. W. (1988). A spiral model of software development and enhancement. *IEEE Computer*, 21(5):61–72.

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

Brooks, F. P. (1987). No silver bullet – essence and accidents of software engineering. *Computer*, pages 10–19.

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

CIP System Group (1987). *The Munich Project CIP. Volume II: The Program Transformation System CIP-S.* LNCS 292. Springer-Verlag.

Clements, P. (1996). A survey of architecture description languages. In *Proceedings of the 8th International Workshop on Software Specification and Design*, pages 16–25, Schloss Velen, Germany. IEEE Computer Society Press.

Coleman, D., Arnold, P., Bodoff, S., Dollin, C., Gilchrist, H., Hayes, F., and Jeremaes, P. (1994). *Object-Oriented Development: The Fusion Method.* Prentice-Hall.

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

Davies, J. (1993). *Specification and Proof in Real-Time CSP.* Cambridge University Press.

Davies, J. and Schneider, S. (1995). Real-time CSP. In Rus, T. and Rattray, C., editors, *Theories and Experiences for Real-Time System Development.* World Scientific Publishing Company.

Delmas, S. (1994). Kidnapping X Applications. Unpublished Paper, TU Berlin.

Dershowitz, N. (1983). *The Evolution of Programs.* Birkhäuser, Boston.

Dold, A. (1995). Representing, verifying and applying software development steps using the PVS system. In Alagar, V. and Nivat, M., editors, *Proc. 4th Int. Conference on Algebraic Methodology and Software Technology*, LNCS 936. Springer-Verlag.

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

Fernandez, J., Garavel, H., Mounier, L., Rasse, A., Rodriguez, C., and Sifakis, J. (1992). A Toolbox for the Verification of LOTOS Programs. In Clarke, L. A., editor, *Proceedings of the 14th International Conference on Software Engineering ICSE'14*, Melbourne, Australia. ACM.

Fernandez, J.-C. (1989). Aldebaran: A tool for verification of communicating processes. Rapport SPECTRE C14, Laboratoire de Génie Informatique — Institut IMAG, Grenoble.

Fiadeiro, J. and Maibaum, T. (1996). A mathematical toolbox for the software architect. In J.Kramer and A.Wolf, editors, *Proc. 8th International Workshop on Software Specification and Design*, pages 46–55, Schloss Velen, Germany. IEEE Computer Society Press.

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

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns – Elements of Reusable Object-Oriented Software.* Addison Wesley, Reading.

Garlan, D., Allen, R., and Ockerbloom, J. (1995). Architectural mismatch: Why reuse is so hard. *IEEE Software*, pages 17 –26.

Garlan, D., Kompanek, A., Melton, R., and Monroe, R. (1996). Architectural Style: An Object-Oriented Approach. In *Submitted for publication.*

Garlan, D. and Shaw, M. (1993). An introduction to software architecture. In Ambriola, V. and Tortora, G., editors, *Advances in Software Engineering and Knowledge Engineering*, volume 1. World Scientific Publishing Company.

Goldblatt, R. (1982). *Axiomatising the Logic of Computer Programming.* LNCS 130. Springer-Verlag.

Gries, D. (1981). *The Science of Programming.* Springer-Verlag.

Halang, W. and Krämer, B. (1994). Safety assurance in process control. *IEEE Software*, 11(1):61–67.

Hansen, K. M. (1994). Modelling railway interlocking systems. Available via ftp from ftp.ifad.dk, directory /pub/vdm/examples.

Harel, D. (1987). Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8:231–274.

Heisel, M. (1992). *Formale Programmentwicklung mit dynamischer Logik*. Deutscher Universitätsverlag, Wiesbaden.

Heisel, M. (1994). A formal notion of strategy for software development. Technical Report 94–28, Technical University of Berlin.

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

Heisel, M. (1995b). Specification of the Unix file system: A comparative case study. In Alagar, V. and Nivat, M., editors, *Proc. 4th Int. Conference on Algebraic Methodology and Software Technology*, LNCS 936, pages 475–488. Springer-Verlag.

Heisel, M. (1996a). An approach to develop provably safe software. *High Integrity Systems*, 1(6):501–512.

Heisel, M. (1996b). A pragmatic approach to formal specification. In Kilov, H. and Harvey, W., editors, *Object-Oriented Behavioral Specifications*, pages 41–62. Kluwer Academic Publishers.

Heisel, M. (1996c). Strategies – a generic knowledge representation mechanism for software development activities. Submitted for publication.

Heisel, M. and Krishnamurthy, B. (1995a). Bi-directional approach to modeling architectures. Technical Report 95-31, Technical University of Berlin.

Heisel, M. and Krishnamurthy, B. (1995b). YEAST – a formal specification case study in Z. Technical Report 95-32, Technical University of Berlin.

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

Heisel, M., Reif, W., and Stephan, W. (1988). Implementing verification strategies in the KIV system. In Lusk, E. and Overbeek, R., editors, *Proceedings 9th International Conference on Automated Deduction*, LNCS 310, pages 131–140. Springer-Verlag.

Heisel, M., Reif, W., and Stephan, W. (1989). A dynamic logic for program verification. In Meyer, A. R. and Taitslin, M. A., editors, *Proceedings Logic at Botik*, number 363 in Lecture Notes in Computer Science, pages 134–145. Springer-Verlag.

Heisel, M., Santen, T., and Zimmermann, D. (1995a). A generic system architecture for strategy-based software development. Technical Report 95-8, Technical University of Berlin.

Heisel, M., Santen, T., and Zimmermann, D. (1995b). Tool support for formal software development: A generic architecture. In Schäfer, W. and Botella, P., editors, *Proceedings 5-th European Software Engineering Conference*, LNCS 989, pages 272–293. Springer-Verlag.

Heisel, M. and Sühl, C. (1996a). Combining Z and real-time CSP for the development of safety-critical systems. Submitted for publication.

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

Hinchey, M. G. and Jarvis, S. (1995). *Concurrent Systems: Formal Development in CSP*. McGraw-Hill.

Hoare, C. (1985). *Communicating Sequential Processes*. Prentice Hall.

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

Hörcher, H.-M. and Peleska, J. (1995). Using formal specifications to support software testing. *Software Quality Journal*, 4(4).

Houston, I. and King, S. (1991). CICS project report. Experiences and results from the use of Z in IBM. In *VDM'91: Formal Software Development Methods. Symposiom of VDM Europe, Noordwijkerhout*, LNCS 551, pages 588–596, Berlin. Springer-Verlag.

Huff, K. (1996). Software process modelling. In Fuggetta, A. and Wolf, A., editors, *Software Process*, number 4 in Trends in Software, chapter 2, pages 1–24. Wiley.

ITSEC (1991). Information technology security evaluation criteria. Commission of the European Union.

Jackson, M. and Zave, P. (1995). Deriving specifications from requirements: an example. In *Proceedings 17th Int. Conf. on Software Engineering, Seattle, USA*, pages 15–24. ACM Press.

Jacky, J. (1995). Specifying a safety-critical control system in Z. *IEEE Transactions on Software Engineering*, 21(2):99–106.

Johnson, W. L. and Feather, M. S. (1991). Using evolution transformations to construct specifications. In Lowry, M. R. and McCartney, R. D., editors, *Automating Software Design*, pages 65 – 92. AAAI Press.

Jones, C. B. (1990). *Systematic Software Development using VDM*. Prentice Hall.

Kanellakis, P. C. (1990). Elements of relational database theory. In van Leeuwen, J., editor, *Handbook of Theoretical Computer Science*, volume B, chapter 17, pages 1073–1156. Elsevier.

Krishnamurthy, B. and Rosenblum, D. S. (1995). Yeast: A general purpose event-action system. *IEEE Transactions on Software Engineering*, 21(10):845–857.

Leveson, N. (1986). Software safety: Why, what, and how. *Computing Surveys*, 18(2):125–163.

Leveson, N. (1991). Software safety in embedded computer systems. *Communications of the ACM*, 34(2):34–46.

Leveson, N. (1995). *Safeware: System Safety and Computers.* Addison-Wesley.

Libes, D. (1991). expect: Scripts for controlling interactive processes. *Computing Systems*, 4(2).

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

Lowry, M. R. and McCartney, R. D., editors (1991). *Automating Software Design.* AAAI Press, Menlo Park.

Luckham, D., Kenney, J., Augustin, L., Vera, J., Bryan, D., and Mann, W. (1995). Specification and analysis of system architecture using Rapide. *IEEE Transactions on Software Engineering*, 21(4):336–355.

McDermid, J. and Pierce, R. (1995). Accessible formal method support for PLC software development. In Rabe, G., editor, *Proceedings of the 14th International Conference on Computer Safety, Reliablity and Security (SAFECOMP), Belgirate, Italy*, pages 113–127, London. Springer-Verlag.

Milner, R. (1972). Logic for computable functions: description of a machine implementation. *SIGPLAN Notices*, 7:1–6.

Milner, R. (1980). *A Calculus of Communicating Systems.* LNCS 92. Springer-Verlag.

Moriconi, M. and Qian, X. (1994). Correctness and composition of software architectures. In Wile, D., editor, *Proceedings of the second ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 164–174. ACM Press.

Moser, L. E. and Melliar-Smith, P. (1990). Formal verification of safety-critical systems. *Software – Practice and Experience*, 20(8):799–821.

Mukherjee, P. and Stavridou, V. (1993). The formal specification of safety requirements for storing explosives. *Formal Aspects of Computing*, 5:299–336.

Osterweil, L. (1987). Software processes are software too. In *9th International Conference on Software Engineering*, pages 2–13. IEEE Computer Society Press.

Ousterhout, J. K. (1994). *Tcl and the Tk Toolkit.* Addison-Wesley.

Paulson, L. C. (1994). *Isabelle.* LNCS 828. Springer-Verlag.

Peleska, J. (1995). *Formal Methods and the Development of Dependable Systems.* University of Kiel, Habilitation thesis.

Potter, B., Sinclair, J., and Till, D. (1991). *An Introduction to Formal Specification and Z.* Prentice Hall.

Potts, C. (1989). A generic model for representing design methods. In *International Conference on Software Engineering*, pages 217–226. IEEE Computer Society Press.

Ravn, A., Rischel, H., and Hansen, K. (1993). Specifying and verifying requirements of real-time systems. *IEEE Transactions on Software Engineering*, 19(1):41–55.

Rich, C. and Waters, R. C. (1988). The programmer's apprentice: A research overview. *IEEE Computer*, pages 10–25.

Shaw, M. and Garlan, D. (1996). *Software Architecture.* IEEE Computer Society Press, Los Alamitos.

Shepard, T., Sibbald, S., and Wortley, C. (1992). A visual software process language. *Communications of the ACM*, 35(4):37–44.

Smith, D. R. (1990). KIDS: A semi-automatic program development system. *IEEE Transactions on Software Engineering*, 16(9):1024–1043.

Souquières, J. (1993). *Aide au Développement de Specifications.* Thèse d'Etat, Université de Nancy I.

Souquières, J. and Heisel, M. (1996). Expression of style in formal specification. In Samson, W. B., editor, *Proceedings Software Quality Conference*, pages 56–65, ISBN 1 899796 02 9. University of Abertay Dundee.

Souquières, J. and Lévy, N. (1993). Description of specification developments. In *Proc. of Requirements Engineering '93*, pages 216–223.

Spivey, J. M. (1992a). The fuzz manual. Computing Science Consultancy, Oxford.

Spivey, J. M. (1992b). *The Z Notation – A Reference Manual.* Prentice Hall, 2nd edition.

Sühl, C. (1996). Eine Methode für die Entwicklung von Softwarekomponenten zur Steuerung und Kontrolle sicherheitsrelevanter Systeme. Master's thesis, Technical University of Berlin.

Weber, M. (1996). Combining Statecharts and Z for the design of safety-critical systems. In Gaudel, M.-C. and Woodcock, J., editors, *FME '96 — Industrial Benefits and Advances in Formal Methods*, LNCS 1051, pages 307–326. Springer-Verlag.

Wile, D. S. (1983). Program developments: Formal explanations of implementations. *Communications of the ACM*, 26(11):902–911.

Williams, L. (1994). Assessment of safety-critical specifications. *IEEE Software*, pages 51–60.

Wirsing, M. (1990). Algebraic specification. In von Leeuwen, J., editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, pages 675–788. Elsevier.

Woodcock, J. and Larsen, P., editors (1993). *FME '93: Industrial-Strength Formal Methods*, number 670 in Lecture Notes in Computer Science. Springer-Verlag.

Zave, P. and Jackson, M. (1993). Conjunction as composition. *ACM Transactions on Software Engineering and Methodology*, 2(4):379–411.

Appendix A

Summary of Z Notation

Sets

Basic types: $[TYPE_1, TYPE_2, \ldots, TYPE_n]$

Variable declaration: $x : TYPE$

Powerset: $\mathbb{P}\, X = \{Y \mid Y \subseteq X\}$

Finite subsets: $\mathbb{F}\, X = \{Y \mid Y \subseteq X \land Y \text{ finite}\}$

Number of members of finite sets: $\#X$

Cartesian product: $X \times Y = \{(x, y) \mid x \in X \land y \in Y\}$

Notation for sets: $\{Decls \mid Pred \bullet Expr\}$
denotes the set of all *Expr* that satisfy *Pred*, based on the variables declared in *Decls*

Predicates

Connectives: $\neg\, , \land, \lor, \Rightarrow, \Leftrightarrow$

Quantifiers:

$$\forall\, Decls \mid Pred_1 \bullet Pred_2 \Leftrightarrow \forall\, Decls \bullet (Pred_1 \Rightarrow Pred_2)$$
$$\exists\, Decls \mid Pred_1 \bullet Pred_2 \Leftrightarrow \exists\, Decls \bullet (Pred_1 \land Pred_2)$$
$\exists_1 \ldots$: there exists exactly one

Relations

Binary relation: $X \leftrightarrow Y = \mathbb{P}(X \times Y)$

Member of a relation: $x \mapsto y = (x, y)$

Let $R : X \leftrightarrow Y, S \subseteq X, T \subseteq Y, Q : Y \leftrightarrow Z$:

Domain of binary relation: $\operatorname{dom} R = \{x : X \mid (\exists\, y : Y \bullet x \mapsto y \in R)\}$

Range of binary relation: $\operatorname{ran} R = \{y : Y \mid (\exists\, x : X \bullet x \mapsto y \in R)\}$

Inverse of binary relation: $R^{\sim} = \{x : X; y : Y \mid x \mapsto y \in R \bullet y \mapsto x\}$

Domain restriction: $S \lhd R = \{x : X; y : Y \mid x \in S \land x \mapsto y \in R\}$

Range restriction: $R \rhd T = \{x : X; y : Y \mid y \in T \land x \mapsto y \in R\}$

Domain subtraction: $S \ndres R = \{x : X; y : Y \mid x \notin S \land x \mapsto y \in R\}$

Range subtraction: $R \nrres T = \{x : X; y : Y \mid y \notin T \land x \mapsto y \in R\}$

Composition: $R \comp Q = \{x : X; y : Y; z : Z \mid x \mapsto y \in R \land y \mapsto z \in Q \bullet x \mapsto z\}$

Functions

Lambda expressions: $\lambda\, \mathit{Decls} \mid \mathit{Pred} \bullet \mathit{Expr}$
denotes a function, mapping each (composed) value of the type contained in the declaration *Decls* that satisfy the predicate *Pred* to the value determined by the expression *Expr*.

Partial functions: $X \pfun Y = \{R : X \leftrightarrow Y \mid \forall\, x : X; y, z : Y \bullet x \mapsto y \in R \land x \mapsto z \in R \Rightarrow y = z\}$

Total functions: $X \fun Y = \{f : X \pfun Y \mid \operatorname{dom} f = X\}$

Injektive partial functions: $X \pinj Y = \{f : X \pfun Y \mid \forall\, x_1, x_2 : \operatorname{dom} f \bullet f x_1 = f x_2 \Rightarrow x_1 = x_2\}$

Injektive total functions: $X \inj Y = (X \pinj) Y \cap (X \fun Y)$

Partial onto functions: $X \psurj Y = \{f : X \pfun Y \mid \operatorname{ran} f = Y\}$

Total onto functions: $X \surj Y = (X \psurj Y) \cap (X \fun Y)$

Bijective total functions: $X \bij Y = (X \inj Y) \cap (X \surj Y)$

Finite partial functions: $X \ffun Y = \{f : X \pfun Y \mid \operatorname{dom} f \in \mathbb{F}\, X\}$

Finite partial injections: $X \finj Y = (X \ffun Y) \cap (X \pinj Y)$

Overriding: Let $f, g : X \pfun Y$. $f \oplus g = ((\operatorname{dom} g) \ndres f) \cup g$

Sequences

Representation: $\operatorname{seq} X = \{f : \mathbb{N} \nrightarrow X \mid \operatorname{dom} f = 1..\#f\}$

Example: $\langle a, b, c \rangle = \{1 \mapsto a, 2 \mapsto b, 3 \mapsto c\}$

Nonempty sequences: $\operatorname{seq}_1 X = \operatorname{seq} X \setminus \{\langle\rangle\}$

Injective sequences: $\operatorname{iseq} X = \operatorname{seq} X \cap (\mathbb{N} \rightarrowtail X)$

Functions on sequences:

- $\mathit{head}\ s$ selects first element of s
- $\mathit{last}\ s$ selects last element of s
- $\mathit{tail}\ s$ contains all elements of s except the first one
- $\mathit{front}\ s$ contains all elements of s except the last one
- $s \frown t$ is the concatenation of s and t
- $\mathit{rev}\ s$ is the reverse of s

Axiomatic Descriptions

They introduce global variables whose values may be restricted. Notation:

$$\begin{array}{|l} \mathit{Decls} \\ \hline \mathit{Preds} \end{array}$$

Example:

$$\begin{array}{|l} \mathit{table\_length} : \mathbb{N} \\ \hline \mathit{table\_length} \leq 1000 \end{array}$$

introduces the global variable $\mathit{table\_length}$ and restricts its value to be less than or equal to 1000. The predicate part of an axiomatic definition is optional.

Free Type Definitions

Free types are algebraically defined data types.

$$T ::= c_1 \mid \ldots \mid c_n \mid d_1\langle\!\langle E_1 \rangle\!\rangle \mid \ldots \mid d_m\langle\!\langle E_m \rangle\!\rangle$$

is an abbreviation for

$$[T]$$

$$
\begin{array}{|l}
c_1, \ldots c_n : T \\
d_1 : E_1 \rightarrowtail T \\
\qquad \ldots \\
d_m : E_m \rightarrowtail T \\
\hline
\langle \{c_1\}, \ldots, \{c_n\}, \operatorname{ran} d_1, \ldots, \operatorname{ran} d_m \rangle \text{ partition } T
\end{array}
$$

The c_i are the constants of the type; the d_j are its constructors, i.e. total injective functions from E_j to T. The constants and the values yielded by the constructor functions are all disjoint, and each member of the type is either one of the constants or in the range of one of the constructor functions.

Global Abbreviations

An abbreviation definition introduces a new global constant: $Name == Expr$. Example: $Even == \{e : \mathbb{N} \mid \exists\, n : \mathbb{N} \bullet 2 * n = e\}$

Local Abbreviations

In the let-expression $\textbf{let } x_1 == E_1; \ \ldots; \ x_n == E_n \bullet E$ the variables $x_1, \ldots, x_n$ are local; their scope includes the expression E, but not the expressions $E_1, \ldots, E_n$ that are the right-hand sides of the local definitions.

Schema Notation

Schemas are the structuring mechanism of Z. They have a name and consist of a declaration and a predicate part. In the declaration part, local names are introduced. The predicate part can be used to restrict the values of the schema components declared in the declaration part. The syntactic form of a schema is:

$$
\begin{array}{|l}
\hline
\textit{SchemaName} \\
\textit{Decls} \\
\hline
\textit{Preds} \\
\hline
\end{array}
$$

or alternatively

$$\textit{SchemaName} \mathrel{\widehat=} [\textit{Decls} \mid \textit{Preds}]$$

Schema Inclusion

If a schema name S_1 occurs in the declaration part of another schema S, then all declarations of S_1 become visible in S, and the predicates of S_1 and S are conjoined. Names occurring in both schemas must have the same type; they are then identified.

Schema Decoration

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 $v_1', v_2', \ldots$. This is done in the declaration as well as in the predicate part.

The Δ notation uses schema decoration and inclusion: for a schema S, ΔS is defined as:

```
__ΔS_____________
| S
| S′
|________________
```

S and S' denote the state before and after execution of an operation, respectively. The postcondition of an operation is usually expressed by equations of the form $v' = \ldots v \ldots$, where v is declared in S.

The schema ΞS says that the values of the declared variables are not changed:

```
__ΞS_____________
| ΔS
|__________
| NoChange
|________________
```

State and Operation Schemas

Schemas are used to define the global state of a system as well as the operations working on that state. A state schema introduces the components of the state and integrity constraints defining the legal system states.

```
__State__________
| x₁ : Type₁
| ...
| xₙ : Typeₙ
|__________
| integrity constraints
|________________
```

An operation schema usually imports the state schema and its decorated version ($\Delta State$). Inputs of an operation are marked with "?", outputs are marked with "!". The predicate part of an operation schema states the precondition of an operation, how the system state evolves, and what conditions the output must fulfill.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Operation}} \\
\hline
\Delta \mathit{State} \\
\mathit{input?} : \mathit{Type}_i \\
\mathit{output!} : \mathit{Type}_o \\
\hline
\mathit{precondition} \\
\mathit{post\_state} \\
\mathit{outputs} \\
\hline
\end{array}$$

Schema Types

Each schema defines a type. If the declaration part of a schema consists of the declarations $x_1 : T_1; \ldots; x_n : T_n$ then the corresponding type is denoted $\lblot x_1 : T_1; \ldots; x_n : T_n \rblot$. The members of the schema type are *bindings* $z = \langle x_1 \Rrightarrow v_1, \ldots, x_n \Rrightarrow v_n \rangle$, with components $z.x_i = v_i$.