Use of Patterns in Formal Development: Systematic Transition From Problems to Architectural Designs

Christine Choppy[1] and Maritta Heisel[2]

[1] LIPN, Institut Galilée - Université Paris XIII, France, email:
Christine.Choppy@lipn.univ-paris13.fr
[2] Otto-von-Guericke-Universität Magdeburg, Fakultät für Informatik, Institut für Verteilte Systeme,D-39016 Magdeburg, Germany, email: heisel@cs.uni-magdeburg.de

Abstract. We present a pattern-based software lifecycle and a method that supports the systematic execution of that lifecycle. First, *problem frames* are used to develop a formal specification of the problem to be solved. In a second phase, *architectural styles* are used to construct an architectural specification of the software system to be developed. That specification forms the basis for fine-grained design and implementation.

1 Elaborating the software development process

Experience has shown that problems and bugs in software systems take their source mainly in the early phases of the software development process[1]. Hence, a software development lifecycle that derives the design of the software directly from the requirements and then passes on to the implementation cannot be regarded as satisfactory. The step between requirements and design is too large.

An additional phase should be introduced between the requirements and the design. One idea that has been accepted for some time now is that some kind of *specification* should be set up on the basis of the requirements, so that the requirements are transformed into documents useful for developers. Specifications lead to a deeper understanding of the problems to be solved, and they can be used to support other development activities (e.g. coding, testing, maintenance). However, producing appropriate specifications often turns out to be difficult for practitioners. For instance, finding an appropriate starting point for the formal specification process is a very common problem.

M. Jackson [Jac95,Jac01] proposes the use of *problem frames* for presenting and understanding software development problems. A problem frame is a characterization of a class of problems in terms of their main components and the connections between these components. A set of typical solution methods is associated to each problem frame. The basic idea is that once an appropriate problem frame for a given problem is found, we also have good proposals for constructing a solution to that problem. We think this idea

[1] See for example http://www.standishgroup.com/sample_research

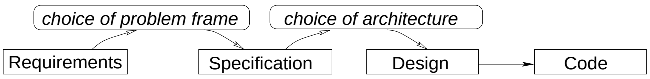


Fig. 1. Lifecycle using problem frames and architectures

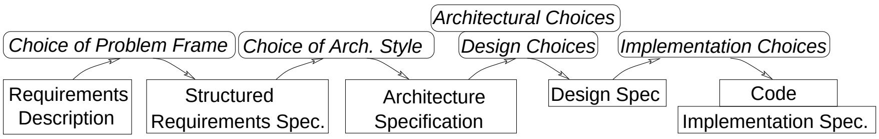


Fig. 2. Complete lifecycle using problem frames and architectural styles

is useful, but it provides only a coarse structure of the problem. Hence, problem frames should be supplemented by means that allow for a finer structuring.

Architectural styles [SG96,BCK98] are a means to structure a software system, i.e. to choose its architecture. Since architectural styles are used to construct designs, they should not be used right at the beginning of the development process, but only after the problem has been fully understood and specified. Figure 1 shows how to bridge the gap between the requirements and the design of a software system. It is possible to elaborate the software development lifecyle further, as suggested in Figure 2. Here, several phases are introduced between the requirements and the design of a software system.

Problem frames and architectural styles are both forms of *patterns*. While problem frames are concerned with *problems*, architectural styles are concerned with *solutions*. Hence, with Figures 1 and 2, we propose a pattern-based software lifecycle. Patterns should be used systematically and on different levels of abstraction.

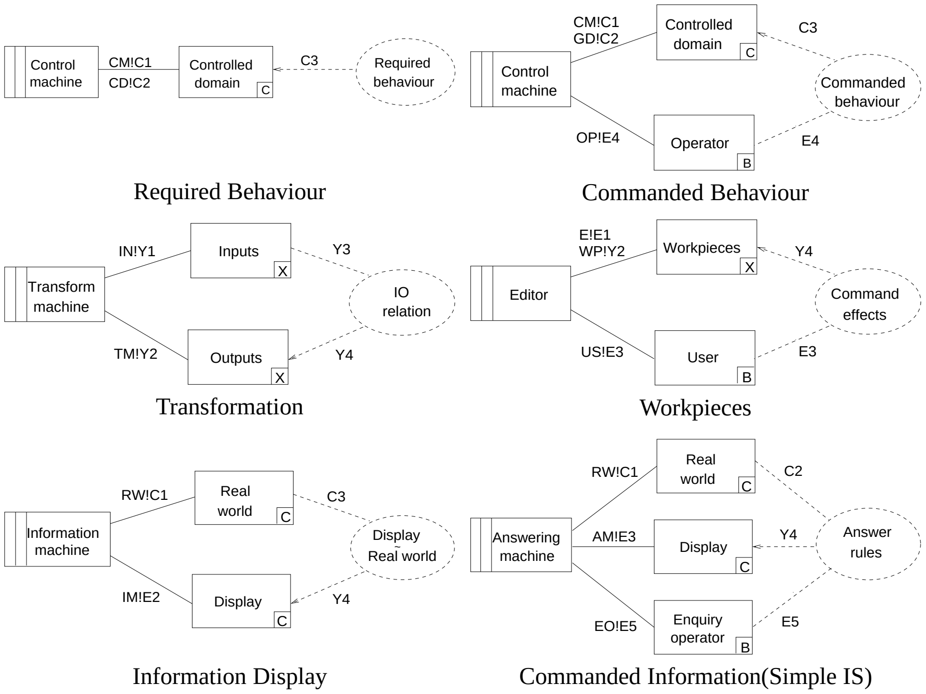
In the following, we show how the steps from an informal requirements description to an architectural specification shown in Figure 2 can be carried out in a systematic way. This work further elaborates the approach by Choppy and Reggio [CR00], where problem frames are used to structure formal specifications.

We first discuss how patterns can be used on different abstraction levels and in different phases of the software development process in Section 2. Section 3 presents a method to carry out pattern based formal development in a systematic way. The application of that method is illustrated by the case study of a robot simulation in Section 4. In Section 5, we summarize our work and also discuss related work that aims at methodological support for developing formal specifications.

2 Patterns for different software development activities

Patterns are a means to reuse software development knowledge on different levels of abstraction. Patterns classify sets of software development problems or solutions that share the same structure.

Patterns have been introduced on the level of detailed object oriented design [GHJV95]. Today, patterns are defined for different activities. *Problem Frames* [Jac01] are patterns



Required Behaviour

Commanded Behaviour

Transformation

Workpieces

Information Display

Commanded Information(Simple IS)

Fig. 3. Frame diagrams

that classify software development *problems*. *Architectural styles* are patterns that characterize software architectures [SG96,BCK98]. They are sometimes called "architectural patterns". *Design Patterns* are referred to as "micro-architectures", while *frameworks* are considered as less abstract, more specialized. Finally, *idioms* are low-level patterns related to specific programming languages [BMR+96], and are sometimes called "code patterns".

Using patterns, we can hope to construct software in a systematic way, making use of a body of accumulated knowledge, instead of starting from scratch each time. In the following, we briefly introduce problem frames and architectural styles, which will be used in our method.

2.1 Problem Frames

Jackson [Jac01] describes problem frames as follows:

> A problem frame is a kind of pattern. It defines an intuitively identifiable problem class in terms of its context and the characteristics of its domains, interfaces and requirement.

For each problem frame, a *frame diagram* is set up (cf. Figure 3), which contains the different parts involved. Plain rectangles denote *application domains*. The characteristics of these domains play an important role in the application of a problem frame to a problem. A problem frame features also a *machine domain* denoted by a rectangle with a double vertical stripe, and a *requirement* denoted by a dashed oval. The connecting lines represent interfaces that consist of so-called “shared phenomena”.

Jackson distinguishes *causal domains* that may control some shared phenomena (e.g. events) at the interface with another domain, *biddable domains* (people), and *lexical domains* that are physical representation of data. *Causal phenomena* (e.g. events) are caused or controlled by some domain, and can cause in turn other phenomena. *Symbolic phenomena* (e.g. values) can be changed, but cannot change themselves or cause changes elsewhere.

Jackson [Jac01] defines five basic frames (that are variants of those given in [Jac95]). These (and a sixth derived problem frame) are briefly presented below. For each problem frame, we quote the description given by Jackson [Jac01] (see also Fig. 3).

Required Behaviour “There is some part of the physical world whose behaviour is to be controlled so that it satisfies certain conditions. The problem is to build a machine that will impose that control.” The “C” in the frame diagram indicates that the domain *Controlled domain* must be causal. The machine is always a causal domain (so an explicit “C” is not needed). The notation “CM!C1” means that the causal phenomena *C1* are controlled by the Control machine *CM*. The dashed line represents a requirements reference, and the arrow shows that it is a *constraining* reference.

Commanded Behaviour “There is some part of the physical world whose behaviour is to be controlled in accordance with commands issued by an operator. The problem is to build a machine that will accept the operator’s commands and impose the control accordingly.” The “B” indicates that the domain $Operator$ is a biddable domain, and the phenomena *E4* are the operator commands.

Transformation “There are some computer-readable input files whose data must be transformed to give certain required output files. The output data must be in a particular format, and it must be derived from the input data according to certain rules. The problem is to build a machine that will produce the required outputs from the inputs.” The “X” indicates that $Inputs$ and $Outputs$ are lexical (inert) domains.

Workpieces “A tool is needed to allow a user to create and edit a certain class of computer processable text or graphic objects, or similar structures, so that they can be subsequently copied, printed, analysed or used in other ways. The problem is to build a machine that can act as this tool.”

Information Display “There is some part of the physical world whose states and behaviour is continually needed. The problem is to build a machine that will obtain this information from the world and present it at the required place in the required form.” Here, the purpose of the machine is to display things that happen in the real world. Both domains are causal. *Y4* are symbolic requirement phenomena.

Commanded Information is derived from the Simple IS frame [Jac95]. There is some part of the physical world whose states and behavior are needed upon requests from an operator. The problem is to build a machine that will obtain this information from the world and present it at the required place in the required form.

Let us note that these problem frames do not cover every conceivable problem class. Some more problem frames have been identified by Souquières and Heisel [SH00].

2.2 Architectural Styles

According to Bass, Clements, and Kazman [BCK98],

> the software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them.

Architectural styles are patterns for software architectures. A style is characterized by [BCK98]:

- a set of component types (e.g., data repository, process, procedure) that perform some function at runtime,
- a topological layout of these components indicating their runtime interrelationships,
- a set of semantic constraints (for example, a data repository is not allowed to change the values stored in it),
- a set of connectors (e.g., subroutine call, remote procedure call, data streams, sockets) that mediate communication, coordination, or cooperation among components.

Important architectural styles are the following:

- **Data-Centered** with substyles *Repository* and *Blackbord*
- **Data Flow** with substyles *Batch Sequential* and *Pipe-and-Filter*
- **Virtual Machine** with substyles *Interpreter* and *Rule-Based Systems*
- **Call-and-Return** with substyles *Main Program and Subroutine, Layered, Object-Oriented* or *Abstract Data Types*
- **Independent Components** with substyles *Communicating Processes* and *Event Systems* (implicit/explicit invocation)

When choosing an architecture for a system, usually several architectural styles are possible, which means that all of them could be used to implement the functional requirements. Which architectural style is the most appropriate must then be decided using *non-functional* criteria such as efficiency, scalability, or modifiability. How such a choice is made is illustrated in Section 4.

2.3 Design Patterns

Design patterns [GHJV95] are used on a lower level of abstraction than problem frames or architectural styles. They provide concrete means to combine objects, or classes, respectively. In our overall software lifecyle, they would be used after an architectural style has been chosen. This step is beyond the scope of this paper.

3 An agenda for pattern-based specification and design

We now present our method for carrying out a pattern-based software lifecycle as shown in Figures 1 and 2. As a means of presentation, we use the *agenda* concept [Hei98]. An agenda is a list of steps or phases to be performed when carrying out some task in the

Table 1. Agenda for pattern-based specification

No.	Description	Result	Validation
1.	Fit the problem into an appropriate problem frame.	Instantiated frame diagram	All important issues of the problem must be treated adequately, see also [Jac01].
2.	Set up a formal specification for each domain of the instantiated frame diagram (including the machine domain) and the requirements.	Set of formal specifications	– The specification must be coherent with the instantiated problem frame diagram. – The shared phenomena must belong to the interfaces of all domains where they are visible. – Control of phenomena must be taken into account. – The specification S of the machine domain (in combination with the domain knowledge D) must suffice to satisfy the requirements R, i.e., $S \wedge D \rightarrow R$ must hold.
3.	Choose an appropriate architectural style for structuring the machine domain and instantiate it.	Architectural diagram and informal text	The chosen architecture must be able to satisfy the machine specification.
4.	Set up a formal specification of all components obtained in Step 3 and of the overall system (i.e., specify how the components cooperate).	Set of formal specifications	– The formal specification must correspond to the architectural diagram. – The overall specification must be a *refinement* of the machine specification developed in Step 2. – The constraints imposed by the chosen architectural style must be satisfied.

context of software engineering. The result of the task will be a document expressed in some language. Agendas contain informal descriptions of the steps, which may depend on each other. Agendas are not only a means to guide software development activities. They also support quality assurance, because the steps may have validation conditions associated with them. These validation conditions state necessary semantic conditions that the developed artifact must fulfill in order to serve its purpose properly.

Table 1 shows an agenda that precisely describes how to carry out and validate the first steps of the lifecycle proposed in Figure 2. A precondition for the applicability of the agenda is that the problem is sufficiently small that it may be fitted into one problem frame. Complex problems have to be decomposed first, for example by projection, as described by [Jac01].

Step 1 of the agenda is performed in principle as described by Jackson [Jac01]. To find the right problem frame, the structure of the frame diagram and the domain characteristics as described in Section 2.1 must be taken into account. However, this is not as straightforward as it might seem, because we first need to choose between possibly different viewpoints on the problem. For instance, the choice of taking into

Table 2. Problem frames and related architectural styles

Problem Frame	Architectural Style
Required Behaviour Commanded Behaviour	Communicating Processes, Event/Action, Process Control
Transformation Workpieces	Repository, Batch Sequential, Pipe and Filter, Virtual Machine, Layered, ADT/OO, Event Systems
Information Display Commanded Information	Repository, Blackboard

account a user/operator influences the choice of problem frame, and it also changes the characteristics of the domains and phenomena. We think it is worthwhile to examine for each problem frame whether we find a meaningful instantiation of it or else a clear reason why not.

Once the choice of a problem frame is made, we rely on the structure provided by the problem frame to proceed and establish a corresponding formal specification [CR00].

Step 2 uses the instantiated frame diagram from Step 1 that determines the structure of the formal specification to be set up. For each box in the instantiated frame diagram, a specification must be given. The validation condition "coherence of the instantiated frame diagram and specification" means that the phenomena at the interfaces of the requirements box must be used in expressing the requirements. Moreover, the shared phenomena that are given in the instantiated problem frame must belong to the interfaces of the respective domain specifications. A domain which is in control of a shared phenomenon must be able to produce that phenomenon as an output, and a domain which is able to observe a phenomenon of which it is not in control must be able to take the phenomenon as an input. The domain knowledge D mentioned in the last validation condition of Step 2 refers to the specification of the application domains, i.e. the domains of the instantiated problem frame other than the machine domain.

Step 3 uses the specification of the machine domain developed in Step 2. This specification describes the machine to be developed, whose structure will be determined by the architectural style. Several possible architectural styles should be explored and assessed according to those non-functional criteria that are regarded to be important for the given problem.

Table 2 gives heuristics for performing Step 3. It has been developed from the general characteristics of the involved problem frames and architectural styles as well as by conducting several case studies. It shows rules of thumb giving hints which architectural styles to consider first.

As can be seen, there are several architectural styles associated to each problem frame. Which one is finally chosen depends on *non-functional* requirements. It remains to make these explicit in order to really guide the transition from a problem frame to an architectural style.

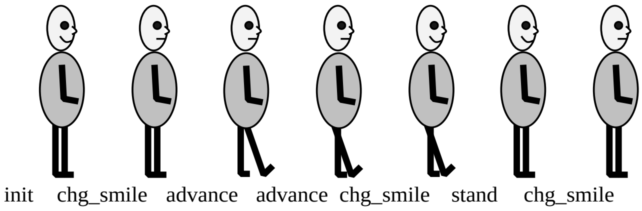


Fig. 4. The movements of the robot

For the problem frames Transformation and Workpieces, we have quite a number of architectural styles to consider. This is due to the fact that these problem frames cover most of the "classical" software development problems and that they are less constraining than the other frames. For Required Behaviour and Commanded Behaviour, we should consider architectural styles that are well suited for reactive systems, and for Information Display and Commanded Information, it seems natural to choose data-centered architectures.

Step 4 uses the architectural style instantiated in Step 3 to develop a specification that formally describes the chosen architecture. That instantiation determines a set of components that structure the system to be developed. It also shows the cooperations between these components. For each component a formal specification must be given. Furthermore, it must be specified how the components cooperate. Such an architectural specification is the basis for detailed design and implementation. The most important validation condition associated with Step 4 of the agenda is to show that the chosen architecture indeed correctly implements the machine specified in Step 2, i.e., that the architectural specification refines the machine specification. Because we use formal specifications, this validation condition can be demonstrated in a rigorous or formal way.

In the following section, we demonstrate the application of the agenda by means of a concrete example.

4 Case study: robot simulation

This case study is taken from [HL97], where it was used to illustrate different architectural styles. Here, we demonstrate how the most suitable architectural style can be found in a systematic manner, performing the steps of the agenda presented in Table 1.

The task is to build a system simulating a simple robot. This robot can make the movements shown in Figure 4: it can advance by moving its right or its left leg; it can stand still; and it can smile or not. The robot can be modeled as an automaton with three states: `standing`, `left_up` and `right_up` as shown in Figure 5. To each state a boolean value is associated indicating whether the robot is smiling or not. The initial state is standing and smiling.

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,

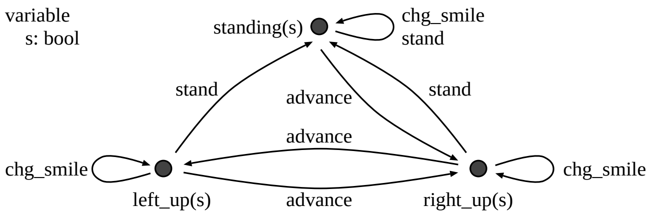


Fig. 5. The robot automaton

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.

The input for the system to be built is a list of commands to be executed by the robot, i.e., a list consisting of the elements `stand`, `advance` and `chg_smile`. The output is a list of pairs, where the first component of each pair is the current state of the robot, and the second component of each pair is the list of commands not yet executed. Each command must be executed, and the intermediate states entered during execution of the command list must be given as an output.

Step 1: Choice and Instantiation of a Problem Frame We consider the problem frames (cf. Figure 3) one by one and give reasons for each problem frame why it is rejected or accepted.

Required Behaviour The “C” says that the $Controlled\ Domain$ must be causal. Since the robot is defined by an abstract data type, the domain corresponding to the robot is not causal but lexical. Moreover, the problem frame Required Behaviour does not let us distinguish between the input domain (being a list of commands) and the output domain (being a list of pairs). Hence, we reject this problem frame.

Commanded Behaviour This problem frame must be rejected for the same reasons as before. Moreover, we cannot find a domain corresponding to the $Operator$ domain.

Information Display Here, the purpose of the machine is to display things that happen in the real world. Both domains are causal, which does not fit well with the robot problem.

Commanded Information This frame must be rejected, because we cannot find an *Enquiry operator* and because the domains involved in the robot problem are not causal.

Workpieces This problem frame is more promising than the ones considered before, because we have a lexical domain here. The workpieces are the robot’s state, together with the current command list. However, we cannot find an instantiation for the *User* domain, because command lists are not biddable. Hence, we finally reject the Workpieces frame.

Transformation It is this frame that we finally choose for our problem. A lexical input list is transformed into a lexical output list. The relation between the two lists is given by the robot automaton. Figure 6 shows the instantiated frame diagram.

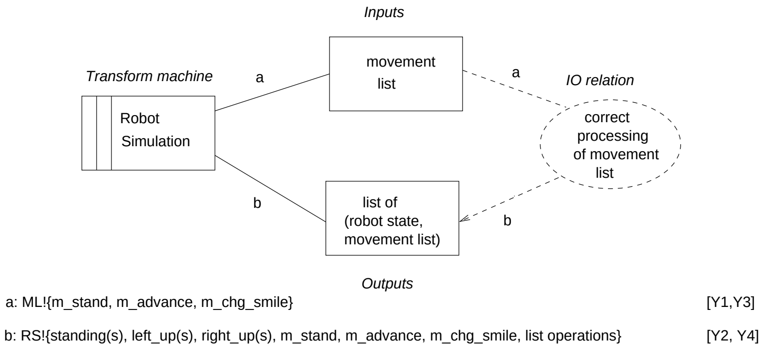

Inputs
Transform machine
a
movement list
a
IO relation
Robot Simulation
correct processing of movement list
b
list of (robot state, movement list)
b
Outputs
a: ML!{m_stand, m_advance, m_chg_smile}
[Y1,Y3]
b: RS!{standing(s), left_up(s), right_up(s), m_stand, m_advance, m_chg_smile, list operations}
[Y2, Y4]

Fig. 6. The robot problem fitted into the Transformation problem frame

Step 2: Structured Requirements Specification Having chosen a problem frame for the robot problem, we must give a specification of all the domains involved and of the requirements. As a specification language, we use LOTOS [BB87], because LOTOS is one of the specification languages allowing us to define software architectures, and especially the interaction of different components, in a suitable way.

Specification of the $Inputs$ Domain. As shown in Figure 6, the input domain is a list of movement commands.

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

The robot will be asked to execute several movements collected in a list. This list is defined by an abstract data type `M_LIST` whose definition is straightforward.

Specification of the $Outputs$ Domain. The output consists of a list of pairs, whose first element is the current state of the robot and whose second element is the list of movements yet to be performed.

The definition of the abstract data type `ROBOT` reflects exactly the automaton given in Figure 5.

To define the $Outputs$ domain `O_LIST`, a data type `VALUE` must be defined as the Cartesian product (with constructor `make`) of the two types `ROBOT` and `M_LIST`. The type `O_LIST` of lists of elements of type `VALUE` is then defined in much the same way as the type `M_LIST`.

Specification of the *IO relation*. The IO relation says that, given a list of commands, the robot simulation must execute that list of commands one by one and output the current state of the robot after execution of each command, together with the commands yet to be executed.

For example, if the input command list has the form $(m_1, m_2, m_3, \ldots m_k)$ then the output list has the form

$$((m_1(init\ of\ robot), (m_2, m_3, \ldots m_k)), (m_2(m_1(init\ of\ robot)), (m_3, \ldots m_k)),$$
$$\ldots, (m_k(\ldots(m_3(m_2(m_1(init\ of\ robot))))\ldots), empty))$$

where $m(r)$ denotes the robot state that is reached from state r by executing movement

m. This requirement is defined by a predicate `is_correct` which takes a movement list and and output list as its arguments. This predicate is defined in a type `IO_REL`.

Specification of the Machine Domain *Robot Simulation*. For each input list, the robot simulation must produce an output list in such a way that the two lists are in the relation `is_correct`.

```
type ROBOT_SIMULATION
is   IO_REL
     opns  robsim : m_list  -> o_list
     eqns
           forall ml : m_list
           ofsort bool
             is_correct(ml, robsim(ml)) = true
endtype
```

Steps 3 and 4: Architectural design of the robot We will explore several possibilities to structure the machine domain specified in Step 2. The non-functional criteria for assessing the different architectures will be efficiency and simplicity. Moreover, we give a specification of the top-level behavior for each considered architecture. For reasons of space, we cannot give the specifications of the different components.

All architectures we will consider in the following have the same interface. This interface consists of an input channel `START` and an output channel `OUTPUT`, where `START` corresponds to interface a and `OUTPUT` corresponds to interface b of Figure 6.

The list of movements to be processed is given in one step. The simulation must show the intermediate states of the robot when processing the input list. Hence, instead of producing the output list at once, the machine will produce the elements of the output list one by one. Then, the correctness condition required to be proven in Step 4 of the agenda is that the sequence of events occurring on gate `OUTPUT` is an output list that is in relation `IO_rel` with the input list.

The gate `START` is used to start the simulation, yielding in the following top-level behavior:

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

The different architectures will result in different definitions of *`behav_expr`*.

The Repository Architecture. The basic idea is to use a repository that contains the current state of the robot and the list of commands still to be executed. There are three components, one for each command. These components change the state according to the automaton and discard the first element of the command list.

Figure 7 illustrates the repository architecture, where channel names `R`, `W` and `RW` denote the read, write and read/write access to the repository, respectively. The component $Init\_sm$ serves to write the initial state of the robot and the initial command list into the repository.

The components try to access the shared memory in parallel in order to execute the movement they are responsible for. Each of them first reads the list of movements. If the first movement is the one it is responsible for, the movement is executed, the robot

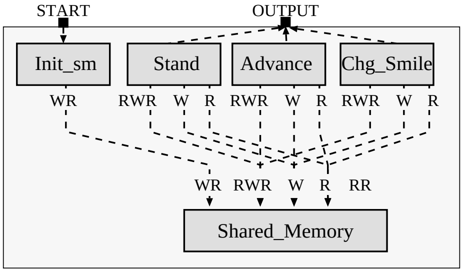


Fig. 7. The repository architecture for the robot

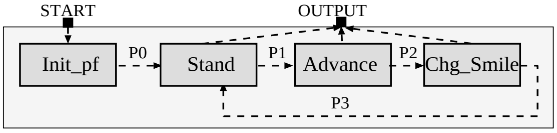


Fig. 8. The pipe/filter architecture of the robot

state changed, and the new state 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. The top-level behavior of this architecture is as follows:

```
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] ))
```

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 (*live-lock*).

The Pipe-and-Filter Architecture. 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 8. The top-level behavior of this architecture is as follows:

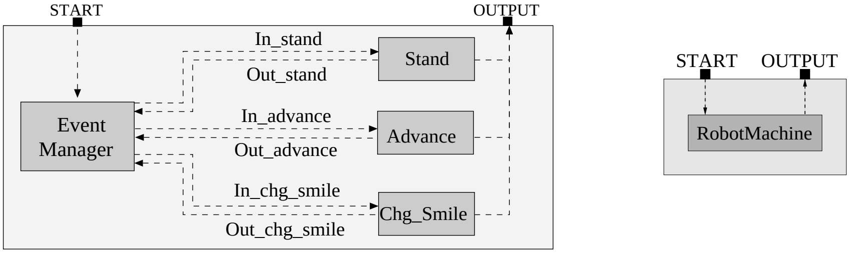


Fig. 9. The event system and the virtual machine architectures for the robot

```
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 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.

The Event System Architecture. The event system style can be used to overcome the disadvantages of the previous two architectures. An event manager inspects the movement list and passes the data only to the component that can process them. The initial state of the robot and the movement list are given to the event manager. No initialization component is required. This architecture is shown on the left-hand side of Figure 9. 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] )
```

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

The Virtual Machine Architecture. The architecture can be improved once more. We should not have three components that can only execute a single command, but a virtual machine that can execute all three commands. This architectural style seems to be the most natural one, because virtual machines are well suited for simulation tasks. This architecture is shown on the right-hand side of Figure 9. It is quite simple:

```
process Robot [START, OUTPUT] : exit :=
      START ? v: value; RobotMachine[OUTPUT](v)
endproc
```

where the process `RobotMachine` just recursively processes the given movement list contained in `v`.

This example shows that Table 2 can only give hints which architectural styles should be considered when developing an architecture for a given problem that was previously fitted into some problem frame. We have demonstrated that several architectures yield correct implementations. However, some of them are better suited than others. The reasons for preferring one architecture over another were efficiency as well as simplicity and elegance. For such a choice, no general rules can be given. However, the architectural styles provide us with an overall structure of the system to be developed. As we have shown, several such structures should be explored in search of the optimal one. The structure finally chosen is the starting point of the subsequent development steps.

For further validation of our approach, we have also carried out other case studies using CASL [CH03].

5 Conclusions

Methodological issues in writing specifications are many, and we would like to point to related work that addresses issues complementary to ours. Roggenbach and Mossakowski [RM02] address the writing of readable specifications in CASL, avoiding semantic pitfalls (these concerns are also addressed in the CASL reference manual [BM02]). Bidoit, Hennicker and Kurz [BHK02] explore the use of observability concepts which are found to be useful and relevant for writing specifications. Blanc [Bla02] proposes guidelines for the iterative and incremental development of specifications.

In this paper, we have introduced a methodology for formal specification that is systematic and that stresses reuse of previously acquired knowledge. Both patterns and agendas are a means to represent knowledge. Patterns are abstractions of the *products* developed during the software lifecycle, and reuse is achieved by instantiating a pattern. Agendas, on the other hand, are explicit representations of process knowledge. Both concepts are orthogonal, and in order to base the software development process as much as possible on previously acquired knowledge, the two concepts should be used in combination. In particular, the contributions of this paper are:

- We have elaborated a software lifecycle where patterns play an important and well-defined role.
- We have developed an agenda that gives guidance how to perform this pattern-based software lifecycle in a systematic way.
- We have shown how to combine problem frames, architectural styles and formal specifications. So far, these three were considered in isolation; no explicit connection between them has previously been established.

In the future we will provide methodological support also for the subsequent development steps of the software lifecycle proposed in Figure 2. In particular, this will involve the application of design patterns. Furthermore, we will investigate problem decomposition and multiframe problems in more detail.

Acknowledgments. We thank Thomas Santen, Carsten von Schwichow and an anonymous referee for their helpful comments on this paper.

References

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

[BCK98] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley, 1998.

[BHK02] M. Bidoit, R. Hennicker, and A. Kurz. On the Integration of Observability and Reachability Concepts. In *Proc. 5th Int. Conf. Foundations of Software Science and Computation Structures (FOSSACS'2002)*, LNCS 2303, pages 21–36. Springer Verlag, 2002.

[Bla02] B. Blanc. *Prise en compte de principes architecturaux lors de la formalisation des besoins - Proposition d'une extension en CASL et d'un guide méthodologique associé*. Thèse de Doctorat, ENS Cachan, 2002.

[BM02] M. Bidoit and P. Mosses. *CASL User Manual*, 2002. `http://www.brics.dk/Projects/CoFI/`.

[BMR$^+$96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley & Sons, 1996.

[CH03] C. Choppy and M. Heisel. Systematic transition from problems to architectural designs. Technical report, Université Paris Nord, 2003. To appear.

[CR00] C. Choppy and G. Reggio. Using CASL to Specify the Requirements and the Design: A Problem Specific Approach. In *Recent Trends in Algebraic Development Techniques*, LNCS 1827, pages 104–123. Springer Verlag, 2000.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, 1995.

[Hei98] M. Heisel. Agendas – a concept to guide software development activites. In R. N. Horspool, editor, *Proc. Systems Implementation 2000*, pages 19–32. Chapman & Hall London, 1998.

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

[Jac95] M. Jackson. *Software Requirements & Specifications: a Lexicon of Practice, Principles and Prejudices*. Addison-Wesley, 1995.

[Jac01] M. Jackson. *Problem Frames. Analyzing and structuring software development problems*. Addison-Wesley, 2001.

[RM02] M. Roggenbach and T. Mossakowski. What is a good CASL specification, 2002. WADT.

[SG96] M. Shaw and D. Garlan. *Software Architecture. Perspectives on an Emerging Discipline*. Prentice-Hall, 1996.

[SH00] J. Souquières and M. Heisel. Structuring the first steps of requirements elicitation. Technical Report A00-R-123, LORIA, Nancy, France, 2000.