An Outsiders Evaluation of PAISLey

Thomas Deiß

250/94

Fachbereich Informatik
Universität Kaiserslautern
Erwin-Schrödinger Straße
67663 Kaiserslautern, Germany
deiss@informatik.uni-kl.de

An Outsiders Evaluation of PAISLey

Thomas Deiß
Fachbereich Informatik
Universität Kaiserslautern
deiss@informatik.uni-kl.de

Abstract

The language PAISLey can be used to specify reactive systems. We used PAISLey to carry out a case study in the context of home systems and to evaluate this language thereby.

1 Introduction

A crucial point in developing a new system is to capture the requirements correctly. It is difficult and expensive to correct errors which are made in this early phase of the development. Therefore, the requirements should be stated in a precise and unambiguous way. One approach to achieve this goal is the use of formal description techniques.

But which language or method should be used? In this paper we will try to evaluate the language PAISLey which has been developed during the years 1982 to 1991, [Zav82, Zav91]. The evaluation will be done by performing a case study in the context of home systems. The main parts of the specified system are an alarm system and a system to heat resp. ventilate the rooms and to monitor the heating plant. The informal specification was plain (German) text and we tried to make a requirements specification of the whole system. This goal was only partially achieved. Only the system to ventilate a room (and some other small parts) are specified together with its environment. The other parts have been specified without a specification of their environment. The user interface has not been specified because we did not regard it as part of the control system and because there was no appropriate informal specification.

We made three versions of the specification. This was due to our (at first) limited experience in the specification of reactive systems. Formal specification is nothing new to us, but our experience is mainly in the field of algebraic specifications. Therefore the functional style of PAISLey was very familiar to us. Nevertheless the first and to some amount the second version have been used to get acquainted with this kind of task and with the language itself.

As a short summary of our work we can state on the one hand, that PAISLey could be used to specify this kind of systems. Using communicating processes seems to

be a very natural way to describe reactive systems. On the other hand there are, in our opinion, some severe deficiencies. A PAISLey-specification is mainly a set of functions. These are described textually in a kind of programming language which makes it difficult to talk with non computer scientists directly about the specification. Furthermore, there is no hierarchy in the description of processes. Therefore, all requirements have to be stated on the same level of abstraction. Especially when specifying timing constraints this becomes very difficult. As a last point, some of the available tools should be improved to be more useful.

The organization of this paper is as follows. In the next section we summarize what a requirements specification should be. In the third section we give a short introduction to PAISLey. In addition to PAISLey we used some kind of simple process diagrams to structure the system, these are explained in section 3 too. The informal specification is presented in section 4, followed by the three versions of the specification in section 5. Section 6 is used for the evaluation of our work and of PAISLey. We finish with some general remarks concerning the specification and the development of reactive systems.

Acknowledgments

Before starting with the presentation of this case study, I would especially like to thank Dirk Zeckzer, who wrote large parts of the first two versions. I am thankful to Prof. Madlener who brought my attention to this language and to Prof. Zimmermann who answered a lot of questions about the informal specification. Also I want to thank numerous colleagues at our department for a lot of fruitful discussions. Special thanks go to Pamela Zave who made the system available to us and to Bill Schelter who answered technical questions about details of the implementation of the interpreter.

2 Requirements specification

Usually, the first step in developing a new system is to capture the requirements of the user or customer. This task can be paraphrased as to answer the question: 'What should be done?'. Van Vliet [vV93, pages 10,11] states:

> *The goal of the requirements analysis phase is to get a complete description of the problem to be solved and the requirements posed by and to the environment in which the system is going to function. ...*
>
> *A description of the problem to be solved includes such things as:*
>
> - *the functions of the software to be developed;*
> - *possible future extensions to the system;*
> - *the amount, and kind, of documentation required;*
> - *response time and other performance requirements of the system.*

This information should be presented in a readable and understandable way, see [vV93, page 144]. Furthermore, a requirements specification should be unambiguous, complete, verifiable, consistent, modifiable, traceable, and usable.

A problem of requirements analysis is that different groups of people have to work together. These groups may have very different background, e.g. when developing home systems there might be designers (of reactive systems), architects and inhabitants of the house. Therefore it is necessary to state precisely the concepts and objects used in this domain and, using these, to describe the problems to be solved. It is not sufficient to have an intuitive understanding, though this might be necessary. In this case study we experienced that even in this well-known domain our intuitions differed.

An additional problem in the context of reactive systems is, that they have timing constraints. Furthermore, not only the functionality of the system itself has to be specified, but also the interaction of the system with its environment.

One approach to specify this interaction is to build a model of the system and its environment. If this model is described by an executable specification language, then simulations can be used to validate, that the user's needs are captured correctly in the requirements specification. Some examples of this approach are Petri Nets [Rei85], statecharts, resp. the corresponding tool STATEMATE [HLN+90], and, as it will be used in this case study, PAISLey [Zav82, Zav91].

Notice, that executable specifications must not be mixed up with prototypes. Specifications are used to express the users requirements as precise and complete as possible. On the contrary, prototypes can be used to explore and clarify vague or unknown requirements.

Following Zave [Zav82], a

> *requirements specification is an executable model of the proposed system interacting with its environment.*

According to her, it should be possible to do four things with requirements specifications [Zav82, page 250]

1. *use them as vehicles for communication*
2. *change them*
3. *use them to constraint target systems*
4. *use them to accept or reject final products*

In section 6 we will try to evaluate whether PAISLey fulfills these goals and whether a PAISLey specification is readable, complete, consistent etc.

Let us remark here, that PAISLey is intended as a language to write requirements specifications, but not as a complete method to perform the requirements analysis.

3 Introduction to the languages

We give a short introduction to the concepts, syntax and semantics of PAISLey. For a full definition see [Zav88]. Furthermore, we introduce a simple graphical notation we used to keep an overview over a set of communicating processes.

3.1 PAISLey

The name PAISLey is an acronym for process oriented, applicative, interpretable specification language and is thereby denoting important concepts of the language.

A specification in PAISLey is a set of descriptions of communicating *processes*. This set has a static structure, it is not possible to express creation and destruction of processes. A process is described by its set of possible states, a nullary function computing an initial state and an unary function, taking one state and computing the next one. The evaluation of functional expressions within a process is done in parallel. Therefore, besides the concurrent behavior of different processes, there is also a kind of intraprocess parallelism. As a first example see figure 1. `RAIN-STATE` is the set of states, `rain-init` is the initialization function and `rain-cycle` is the transition function. The definition of some auxiliary functions (`rain-state-signal`, `rain-get-sensor`) is omitted in this example. For each function there is a declaration of the types of the parameters and of the returned values and the definition itself. Besides the computation of the next state, transition functions usually have side effects. They can exchange messages with other processes.

Communication between different processes is embedded in this functional setting via so called *exchange functions*. If there are two matching exchange functions then they exchange their arguments and return these as their corresponding values.

```
RAIN-STATE = {Rain,No-Rain};

rain-init : --> RAIN-STATE;
rain-init = No-Rain;

rain-cycle : RAIN-STATE --> RAIN-STATE;
"If the state is Rain, and a signal No-Rain arrives, then unlock all windows,
 If the state is No-Rain, and a signal Rain arrives, then lock all windows."
rain-cycle[state] =
    rain-state-signal[(
        state,
        rain-get-sensor[Null]
        )];

...
```

Figure 1: Example of a PAISLey-specification: a part of a rain sensor.

There are two conditions, which must be fulfilled by two exchange functions to match. At first, they must access the same channel. The name of the channel is encoded in the name of the functions, e.g. `x-`*channelname*.The second condition concerns the type of synchronization of the communication. Exchange functions of type `x` model synchronous communication, whereas exchange functions of type `xr` model asynchronous one. Therefore, if an exchange function of type `x` is called, it is not evaluated until there is another exchange function accessing the same channel. Then they exchange their argument and return these as their corresponding values. If an exchange function of type `xr` is called, its evaluation is stopped for an undetermined amount of time, and then it is checked whether there is an exchange function accessing the same channel. If there is one, then the communication takes place, otherwise the exchange function returns a default value (`Null`) to indicate that no communication took place. Notice, that it is not possible, that two exchange functions of type `xr` match. If there are several functions accessing the same channel, then possible conflicts are resolved in a FIFO manner to prevent that some functions are waiting forever.

The flow of information is bidirectional, but in this case study we mainly[1] use them with dedicated roles as senders and receivers of information. But e.g. two senders accessing the same channel must not match. This can be expressed using a variant – `xm` – of exchange functions of type `x`. Exchange functions of this type can only match functions of type `x` or of type `xr`, but not other functions of type `xm`. There is no direct possibility to express broadcast communication. This has to be simulated by the evaluation of the appropriate number of exchange functions sending the information. The number of sending functions must be equal to the number of receiving functions.

The set of functions constituting a specification can be executed by an interpreter. This interpreter is able to handle *incomplete specifications*. A specification is incomplete if not all functions called are also defined. There are four possibilities to get a value for one of these undefined functions: by using a default value, a random value, by asking the user or by calling a C-function.

Besides the properties of the language mentioned in the acronym PAISLey, there are further important properties. *Timing properties* can be expressed as upper or lower bounds on the evaluation time of single functions, as e.g. the next-state function of a process. Furthermore, all datatypes used, except the set of all possible values (`ANY`), are *finite*. The datatypes `STRING`, `REAL`, and `INTEGER` are assumed to be restricted by the machine on which the specification is interpreted. Although recursion is allowed, it is seldom necessary to use it due to this finite datatypes. Hence it is possible, at least in principle, to do a lot of consistency checks in this static setting.

The *formal semantics* of a specification written in PAISLey is defined as, (see [Zav91, page 217])

> *... a set of traces, where each trace is a sequence of timestamped events. An event is either the initiation of a function evaluation (with attributes of the function name, argument, and process in which the evaluation is taking*

[1] An example where a bidirectional flow is used, is a process, where parts of its state are offered in exchange with new parameters entered by a potential user.

place) or the termination of a function evaluation (with attributes of the function name, value, and process). Adjacent events can have the same timestamp, if the difference between their actual times is too small to show up in the current time unit. Because nothing ever happens during execution of a PAISLey specification except function evaluations, these events can supply complete information about a particular execution.

There are four *tools* available. A *parser* can be used to check the syntactic correctness of the specification. There is a *cross-referencer* which helps to locate in which files functions and datatypes are defined resp. used. A *consistency checker* can be used to do type-checking of the functions in a specification. Thereby structural equivalence between types is checked. Inconsistencies between timing constraints can be partially detected by the *interpreter*, all violated constraints are reported to the user during the simulations. The interpreter can also be used to execute the specification by interpreting the functions while respecting the timing constraints. Using the results of a simulation it is possible to do some performance evaluation.

3.2 Process diagrams

In this section we introduce a graphical notation we used to get a view on the structure of a specification. This notation was created ad-hoc and is not intended as a new graphical language to describe the architecture of systems of communicating processes. The diagrams contain only information, which is also present in the textual description of the processes. But a lot of structural knowledge is presented in a condensed form. We experienced, that it was not possible to talk about specifications without using such diagrams.

The diagrams consist of two kinds of graphical symbols: arrows and boxes. Arrows correspond to communication channels and indicate the flow of messages between processes. They are marked with the name of the channel and the type of the exchange functions accessing it. Boxes represent processes and are marked with an optional type of the process, a short name of the process, the file in which it is defined and bounds on the cycle time of the process. The optional types of the process are common types such as menus, sensors and ports. If the lower and upper bound of the cycle time of a process coincide, then only this single number is written. Information concerning the language, in which a process is described, is represented in the type of the boundary line of the box. A box with a solid line is described by PAISLey-functions, one with a dashed line by C-functions. Some of the processes exist only once in the specification, while others are replicated for each room in the house. To show this difference the boxes are filled with a grey pattern resp. are not filled.

As an example of these kind of diagrams see figure 2. This diagram is part of a larger diagram, the dotted lines are connected to processes in this larger diagram. You can see five processes in this example:

`clock`: A system clock, which sends the current time to two other processes. This process accesses the cannel `clk-clock` via an exchange function of type `xr`. That is, it offers the current time without waiting whether another process wants to read it. These processes use exchange functions of type `xm`. It is necessary to use exchange functions of type `xm` here instead of ones of type `x` to prevent two processes simultaneously accessing this channel to match each other instead of matching the exchange function of the process `clock`.

The cycle time is not indicated here, because it can be changed in the specification via a macro definition. But this cycle time should be small to prevent unnecessary delays of other processes.

`protocol`: This process can be seen as part of a server to write a system log file. It gathers informations from many other processes, thereby accessing its channel `protocol` in the same way as the process `clock`.

`protocol.c` This is a process which is directly specified (programmed) in the language C. It is used to write the informations, which are gathered by the process `protocol`, onto a file.

`control`: This process controls and monitors the movement of a single window. Therefore there exists an instance of this process for each window. Processes of this

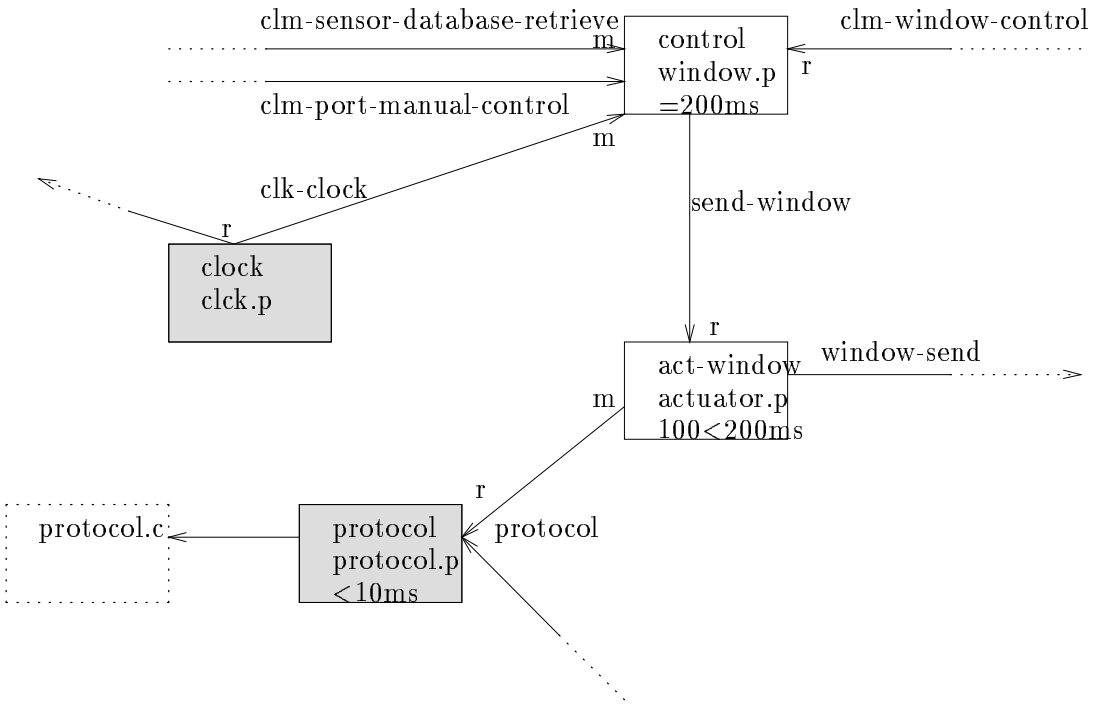

Figure 2: Example of a process diagram

kind receive signals from several other processes and send signals to the actuator of the corresponding window.

`act-window`: This process, too, is replicated for each window. Processes of this kind model the actuator of a window, they send signals to the process, which represents the corresponding window. The actions taken by these processes are protocolled on the log file via the process protocol.

Some information is not contained in this kind of diagrams. E.g. the process `clock` is able to serve more than one request for the current time in one cycle, whereas the process `protocol` is able to serve at most one. Furthermore it is not indicated how many instances of the replicated processes eventually exist and how these instances could be distinguished. Also, the diagram does not contain information whether there is one channel for the communication between all processes of kind `control` and `act-window` or whether there is one channel for each corresponding pair of these processes (as it is eventually specified).

4 The informal specification

This case study was performed in the context of house systems: Various aspects of an one-family house have to be monitored and controlled. The original problem description or informal requirements specification was plain German text, assuming a lot of common knowledge about one-family houses. In the following we will present a revised version of this informal specification. We restricted this presentation to a level of detail which makes it possible to understand the specifications in PAISLey. We begin with the 'physical structure' and 'equipment' used in this case study, continue with the intended behavior of the system and conclude with some remarks about this informal specification.

4.1 Physical structure

The house under consideration is an one-family house consisting of ten normal rooms and one extra room, where the heating plant is located. This room will not be considered in the following specification.

Each room has one window and one radiator. The hall is the only room with a door to the outside. The windows can be moved by the system and 'manually' by the inhabitants (by pressing certain buttons besides a window). In case of an emergency it is also possible to open a window really manually, but then this window cannot be moved again by the system or by pressing the buttons. For each window sensors provide the following information:

- is the window open or closed?
- is it moved 'manually'?

- can it be moved again?
- is its movement blocked?

It is possible to measure the temperature in each room and of each radiator. The radiators can be switched on and off only by the system, but not by the inhabitants. Various parameters concerning heat engineering aspects of the house and the rooms are stated.

Furthermore, it can be recognized, whether a room is empty or whether a person is in it. Doors between the rooms are neither monitored nor controlled by the system, they are non-existent in the informal specification. The outside door is monitored and is equipped with a panel. This can be used to activate resp. deactivate an alarm system by entering a key code.

The temperature of the environment of the house can be measured, too. It can be recognized, whether it is raining or not. In each room is a sensor to detect fire.

The heating plant is equipped with sensors, as e.g. gas pressure or heating water temperature, to prevent harmful behavior of the plant. This part will not be elaborated further. We will only use the possibility to measure the temperature of the heating water and to switch the gas-burner on resp. off.

As last part of the equipment there is a user console to set various parameters and to present information to the user. But besides its pure existence nothing is said about it in the informal specification and we did not elaborate on it too.

4.2 Specification of the behavior

The *alarm system* has to monitor the windows and the door to detect entrance of non-inhabitants. There are two typical situations which should be detected:

> If a room is empty and the window in this room is opening, then assume that a burglar is entering.
>
> If somebody enters the house through the door and within a specified amount of time no correct keycode has been typed in at the panel, then assume that a burglar is entering.

It should be possible that the alarm system can be activated or deactivated for each single room. Furthermore it should be possible to change the time interval between a supposed house breaking and 'ringing the bell' individually for each window and the door. A prealarm should be started when a housebreaking is assumed. If it is not cleared during the appropriate interval, then the main alarm has to be triggered.

There should be three different modes of the alarm system:

away: Trigger of an alarm, which can be recognized from the outside of the house.

home: Trigger of an alarm, which can be recognized only from inside the house.

day: Signals of the sensors and assumed housebreakings are protocolled only, but no alarm is started.

Further requirements are that the alarm system should be automatically activated if nobody is in the house for a specified amount of time and that is should be possible to indicate the present state of the system on the panel.

The *heating system* has the task to heat and air the rooms. These are essentially two different tasks. The more simple one is to ventilate a room and is described by the following scenario:

> If somebody is in a room for a certain amount of time, then open the window for another amount of time, then close it again.

The window should also be closed after this amount of time if somebody opened the window 'manually' or if the temperature is below a certain value. The time intervals, where a person is in a room should be accumulated. It should be possible, that the durations could be changed by the inhabitants.

The more difficult task is to heat a room. For each room there is a minimal temperature (specified by the inhabitants). Furthermore, there should be a control of the temperature which is triggered by the presence of a person in a room:

> If somebody is in a room for at least x minutes, then a temperature of y degrees has to be reached within z minutes.
>
> If a person left a room x' minutes ago, then it is sufficient to hold the minimal temperature.

Again, it should be possible to change these parameters individually for each room.

A third 'control program' is required to guarantee that during specified time intervals certain rooms have a certain temperature.

The second and third kind of control are mutually exclusive with priority of the second one. In all cases, the minimal temperature must be reached.

When entering or changing the diverse parameters the system should check, whether these could be achieved in reality. Therefore it has to take into account only 'invariant' parameters (such as size of the room) and cautious assumptions concerning the environment and the heating plant (e.g. low heating water temperature). The state and the heating requests of other rooms need not to be considered.

When heating a room, the control is able to influence the radiators and the temperature of the heating water. If the water is hot, it is possible to heat the rooms fast, otherwise this may be slow. It should be tried to meet the timing requirements with a temperature of the heating water, which is as low as possible, to minimize waste of energy.

Furthermore, it is required to protocol energy consumption, to check the internal model against reality, to learn habits of the inhabitants and so on. We considered this not as part of the control system and did not elaborate on these requirements further.

The *system to monitor and control the heating plant* has to ensure, that the plant functions correctly, e.g. in case of low gas or water pressure the plant must be stopped in a convenient way. This system has to maintain a certain temperature of the heating water, as it is requested by the heating system.

The *rain* and the *fire* systems have the task to close all windows if it begins to rain or when a fire is detected. As long as it is raining or burning, the windows should be kept closed.

As can be seen, almost all subsystems access the windows. Possible conflicts have to be resolved in favor of safety. E.g. if a room is monitored by the alarm system, the window must not be opened to ventilate the room.

4.3 Comments on the requirements

These requirements can be seen as a typical example of informal requirements written in a natural language:

- They are imprecise.
- A lot of questions are left open.
- A lot of common sense knowledge is used.

The 'specified' kind of system is a reactive system, but here only functional aspects are stated. Timing requirements are omitted due to the implicit knowledge, that the system will be fast enough compared to the physical processes in the house. Other nonfunctional requirements as e.g. reliability or implementation constraints are missing totally. Some information is given about the sensors, but nothing is said about the user interface: What are acceptable values for the parameters the inhabitants may change and how are these values entered into the system?

5 The PAISLey-specifications

5.1 Version 1

The structure of the first version can be seen in figure 3. For each of the sensors and actuators there is one small process. The panel is represented by two processes, one to type in the keycode (`panel`) and one to display the state of the alarm system for each single room (`panel-display`). The values of the sensors are read by the process `collect`, which combines them to one large structure containing all sensor values. This structure is then send to each of the five processes in the middle of the diagram. These processes get the parameters which can be changed by the inhabitants via corresponding processes (`terminal-buffer`) from the process `terminal`. The sensor data and the parameters are used to check whether it is necessary to react in the specified way. If

this is the case, then a corresponding signal is send to the process `actuator-control`. This process gets the signals of the central processes, resolves conflicts and distributes the signals to the appropriate actuators.

Here we used only one process to model the functionality of each of the subsystems. Therefore the statesets and the transition functions of these processes became very large and difficult to understand. A lot of auxiliary functions were needed, which made it also difficult to understand the flow of information. Furthermore, there was a lot of unnecessary synchronization between processes because the processes `collect` and `control-actuators` gather a lot of information before redistributing it. Also, the flow of information is disarranged by these processes. Therefore it is difficult to see which subsystems monitors which sensors and controls which actuators.

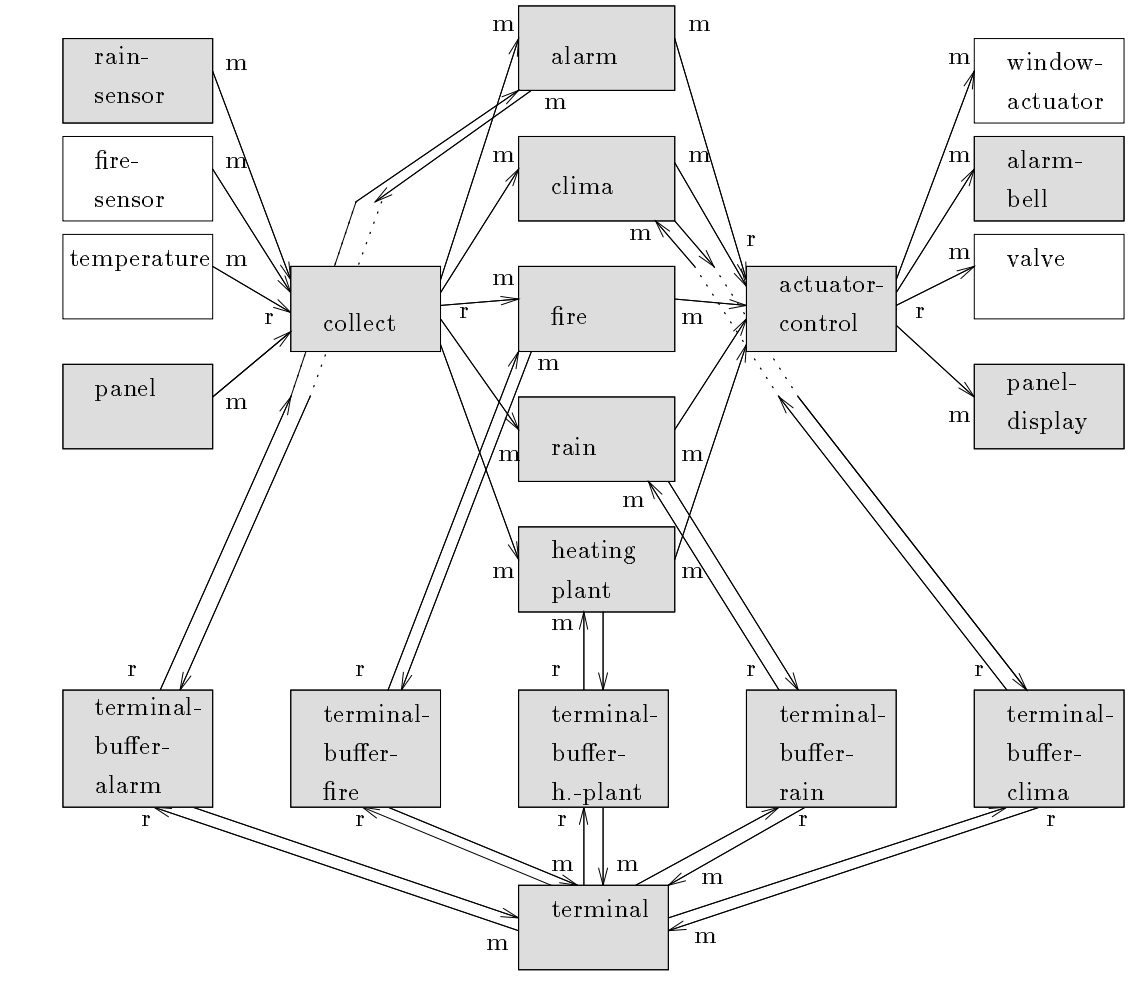

Figure 3: Structure of Version 1

5.2 Version 2

In the second version we split these large processes into several smaller ones, following one of the very few hints to structure a specification, [Zav88, Part II, page 46]:

> *A specification should be decomposed into processes so that each process corresponds to a recognizable function.*

One might argue, that the problem to structure the specification is only shifted towards another question: How to recognize these functions? But here models of and knowledge about the considered domain can be used. Detection of situations which need a reaction of the system and specifying these reactions has been distributed onto several processes. Detecting and reacting are different functionalities which are mixed up in the informal specification. We experienced, that it was very helpful to make this difference explicit.

The first version is also cluttered with functions to retrieve sensor values and parameters and to send signals to the actuators. In the second version we concentrated on the five central processes using functions to read and send these signals, but without defining them. As examples we will present the decomposition of the subsystems `fire` and `alarm` of the first version.

The decomposition of the subsystem `fire` can be seen in figure 4. For each room there is a process `fire`. Each of these processes (see figure 5) reads the value of the corresponding physical sensor (`fire-sensor-room`) and sends signals to the process `fire-collect` as long as the sensor recognizes a fire (`fire-report`). The process `fire-collect` (see figure 6) is in one of the states `Inactive` (system switched off), `No-Fire` (all is ok), `Fire-Confirmed` (An inhabitant confirmed, that he recognized the fire) or a state, which is indicating by a number the room in which it is burning. In each cycle this process tests whether there is a signal from one of the processes `fire` (`fire-rooms`)and offers its state to the terminal (`fire-terminal`). This last communication is a bidirectional one: new user commands influencing the state are

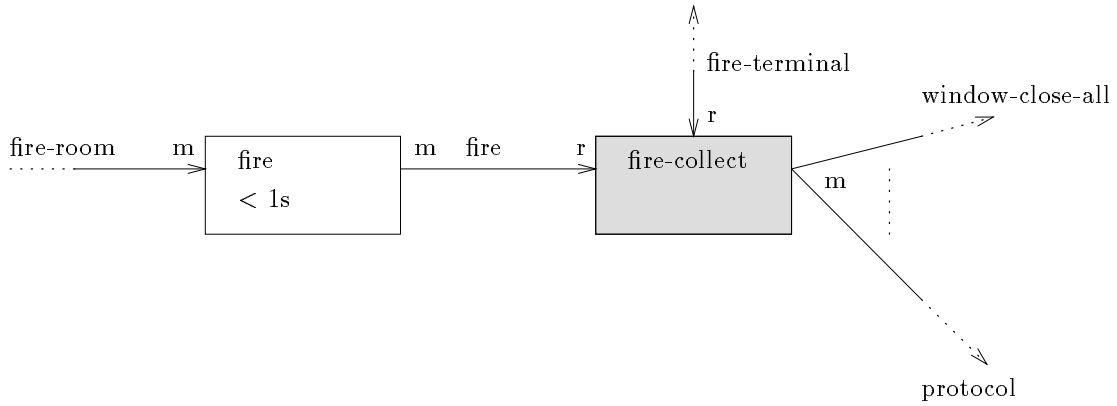

Figure 4: Processes to detect fire

read in exchange of the current state. Then a new state is computed using a possible signal from `fire` and the input from the terminal (`fire-state-and-signal`). If a fire is newly detected, then signals are send to various other processes (`fire-actions`). These communications are achieved by using interface functions of other modules, e.g. `window-close-all` which initiates the closing of all windows.

These processes are able to recognize and display all rooms, in which a fire is detected by the physical sensors until an inhabitant confirms that he recognized the fire too. Afterwards the system waits until it is reset or switched off. An important property of the specified processes is, that it is not possible to switch off the system while it is burning and there is no confirmation of an inhabitant. But it is difficult to extract this information from the specification. It would be more appropriate to state that the system must have this property instead of encoding it in the description of the process.

In our specification it is not possible to detect breakdown of a physical sensor. To recognize this situation one has to know, how the physical sensors are connected

---

```
#define Number_Of_Rooms 10

FIRE = { room#1..  Number_Of_Rooms < , room > };
                "The set of all roomnumbers"

room#1.. Number_Of_Rooms < ;

"No Initialization necessary"
"Next state function.  Propagate Fire signals of one sensor"
fire-cycle-room : FILLER --> FILLER;
fire-cycle-room : ! ub 1.0 s;     "At least one cycle per second"
fire-cycle-room[null] =
        proj[(1,Null,
                fire-state-signal-room[fire-sensor-room[Null]])
                )];

fire-sensor-room : FILLER --> {Fire,No-Fire} | FILLER;
fire-sensor-room[null] = xm-fire-room[null];

fire-state-signal-room : {Fire,No-Fire} | FILLER --> FILLER;
fire-state-signal-room[signal] =
        /
        equal[(signal,Fire)] :    "sensor recognizes fire"
                fire-report[room],
        True :  Null              "sensor recognizes no fire"
        /
>;

fire-report : FIRE --> FILLER;
fire-report[signal] = proj[(1,(Null,xm-fire[signal]))];
```

Figure 5: The specification of the process `fire`

```
FIRE-STATE = {Inactive,No-Fire,Fire-Confirmed} | FIRE; "Stateset"
fire-collect-init : --> FIRE-STATE;                    "Initialization"
fire-collect-init = Inactive;
fire-collect-cycle : FIRE-STATE --> FIRE-STATE;        "Next-state function"
fire-collect-cycle[state] =
        fire-state-and-signal[(
                state,
                fire-rooms[Null],
                fire-terminal[state]
                )];
fire-rooms : FILLER --> FIRE | FILLER;
fire-rooms[null] = xr-fire[null];
...
fire-state-and-signal :
        FIRE-STATE *
        (FIRE | FILLER) *
        ({Inactive,No-Fire,Fire-Confirmed} | FILLER)
        --> FIRE-STATE;
fire-state-and-signal[(old-state,fire,new-state)] =
        /
        equal[(old-state,Inactive)] :  ...
        equal[(old-state,Fire-Confirmed)] :  ...
        equal[(old-state,No-Fire)] :  ...
        True :  /               "Fire detected"
                equal[(new-state,Fire-Confirmed)] :    "Inhabitant confirmed fire"
                        Fire-Confirmed,
                True :  proj[(1,(old-state,
                                fire-actions[
                                        /
                                        is-null[fire] : old-state,
                                        True:  fire
                                        / ]))]
                /
        /;
fire-actions : FIRE --> FILLER;
fire-actions[fire] =
        proj[(1,(Null,
                window-close-all[Fire],
                ...
                ))];
```

Figure 6: Part of the specification of the process `fire-collect`

to the controlling system, but this information was not contained in the informal requirements. One possibility to detect breakdown would be to require that a value is available from the sensor if one is requested. And the system must periodically request a value. If there is no value available, then appropriate reactions have to be taken.

This decomposition can be seen as a trivial one since the only task of a process `fire` (as specified) is to filter the signals of the corresponding sensor. But already this simple decomposition made it easier to specify the process `fire-collect`, because it only receives values from the sensors in those rooms, where a fire has been detected. Since a process `fire` has no internal state it is possible to replace such a process by a function. But we preferred to follow the decomposition into processes detecting resp. processing critical situations. Also this decomposition would make it easier to incorporate e.g. the recognition of the breakdown of a sensor. Then each of the processes `fire` could be used to periodically request a signal from its corresponding sensor and to report if there is no signal available.

The same kind of decomposition can be found in the alarm system (see figure 7): detection of a housebreaking (`room`) and reacting in an appropriate way (`alarm-room`) are specified by different processes. Similar ones exist for the door of the hall. The alarm system has further processes to recognize whether the house is

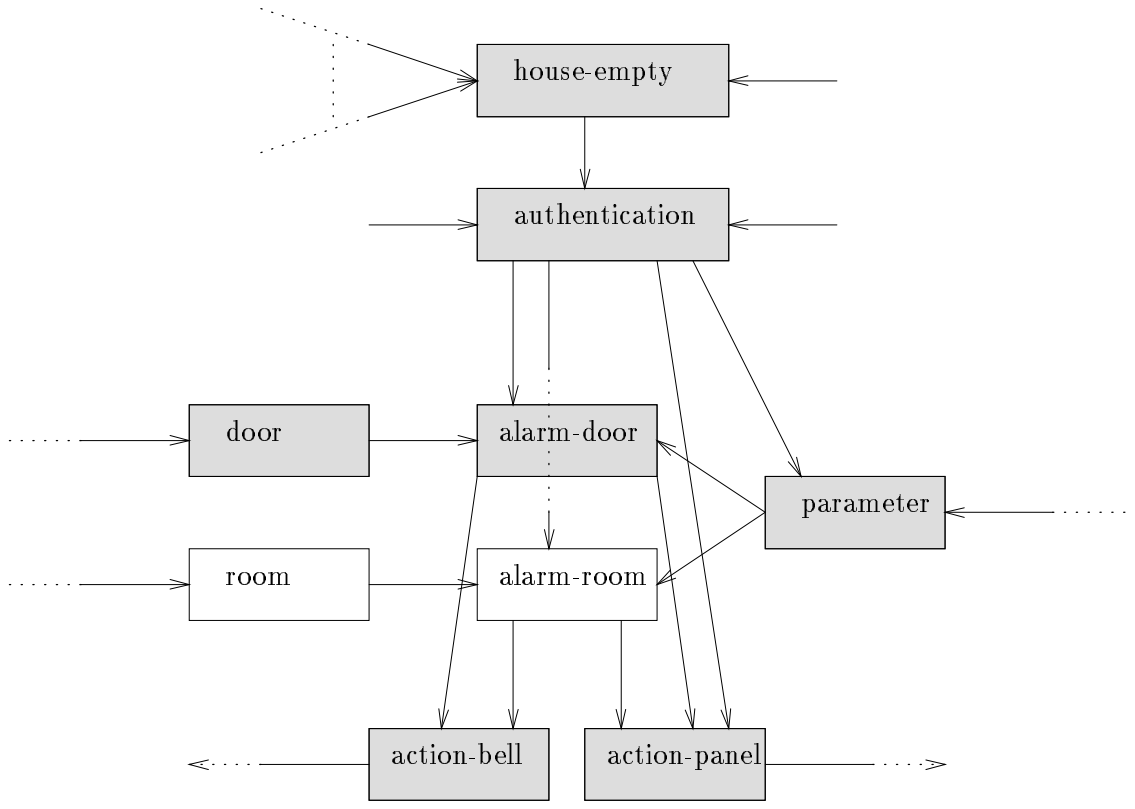

Figure 7: Decomposition of the alarm system (simplified diagram)

empty (`house-empty`), to handle authentication of inhabitants via the panel in the hall (`authenticate`), to store several parameters (`parameter`), and to specify the actions of the 'bell' (`action-bell`) and the display of the panel (`action-panel`) in detail.

Let us have a closer look at the processes `room` and `alarm-room`. The first one is an example of a faulty specification. The specification of the the process `room` (see figure 8) defines a housebreaking as the situation where a window is open and nobody is in the corresponding room. But in the informal requirements it is not considered a housebreaking, if somebody enters a room, opens the window, and leaves the room again. In our specification, the detection of a housebreaking is defined in terms of the state of a window. It is assumed, that if a window is open, it must have been opened some time ago. The informal specification relates to this transition. This difference would be no problem, if this transition would be the only one to enter the state 'no person present' and 'window is open'. As one can see, it is not easy to find this error

---

```
#define Number_Of_Rooms 10

BRK-ROOM-STATUS = FILLER;          "No internal state"
...
j#1..  Number_Of_Rooms <;          "Transition function"
brk-room-j : BRK-ROOM-STATUS --> BRK-ROOM-STATUS;
brk-room-j[status] =
        brk-room-next-cycle-j[(
                status,
                brk-read-sensor-person-j[Null],
                brk-read-sensor-window-j[Null]
                )];

brk-room-next-cycle-j :
        BRK-ROOM-STATUS *
        ( { Present,Absent } | FILLER ) *
        ( { Open,Closed } | FILLER )
        --> BRK-ROOM-STATUS;
brk-room-next-cycle-j[(status,person,window)] =
        /
        equal[(person,Present)] :
                /
                equal[(window,Open)] :
                        proj[(1,(status,brk-send-breaking-j[Breaking]))],
                True :  status
                /
        True :  status
        /;
>;
...
```

Figure 8: Specification of the process `room`

in the textual presentation of PAISLey.

Now let us assume, that the process `room` detects a housebreaking correctly. Then the processes `alarm-room` have the task to trigger a prealarm and after a specified amount of time a main alarm. These processes also handle the activating, deactivating, and resetting of the alarm system for individual rooms. Here we decided to not incorporate the duration of the prealarm-interval into the state of the processes. Every time these durations are needed, they are requested from the process `parameter`. The influence of this choice is twofold. On the one hand we have a simpler state set and transition function. On the other hand we have an extra process with additional channels in the static structure and much more communications when executing this specification.

Specifying a system using such small processes was easier than using large processes as in the first version. Especially understanding a specification is less difficult, though it is not easy to detect errors. We will continue the discussion of PAISLey in section 6.

Now we will motivate the existence of version 3. There are two reasons, which are related. First, in version 2 we concentrated on some processes and did not specify the sensors, the processes distributing the sensor values etc. But when executing the specification using the PAISLey-interpreter, we now had to enter a lot of sensor values manually. Further, Zave suggests to model the system and its environment, e.g. there should be a process representing a window. The third version specifies a part of the system including its environment and allows to perform long[2] executions with easy user interaction.

5.3 Version 3

In the second version we concentrated on the description of the essential subsystems omitting the description of physical components and the user interface. Here, we modeled the subsystem to ventilate a room and to control the movements of the window including its environment. To investigate the cooperation of several subsystem we modeled the system to detect rain, too. As in the second version we used small process to specify the subsystems. We separated the essential processes from the environment and the user interfaces by introducing databases to store sensor values and parameters. Each time a process needs one of these informations he calls an appropriate function and receives the corresponding value. The advantage is, that the processes can be specified with the assumption that they receive a value in all cases. To enter sensor values in an easy way we attached C-function to the PAISLey-part of the specification. These C-function build up and access simple menus. Furthermore we tried to use macros to describe common types of processes as e.g. buffers. Together with the structure of this version we will present some observations we made.

The part of the system modeling the environment, the sensors, ports and the databases can be seen in figure 9. Each window is modeled by one process (`window`), which gets signals from the corresponding actuator and from a menu. This menu repre-

[2]'long' means about half an hour of simulated time. See also the discussion of the tools in 6.3.

sents the switches to open the window 'manually' and to indicate whether its movement is blocked etc. The physical state is then offered to processes representing some sensors (`blockade`, `manual` resp. `window`[3]). The values of the sensors `temperature` and `present` can be set using menus too. A physical sensor can be seen as a process with continuous[4] output. This cannot be modeled in PAISLey since a discrete model of time is used. Therefore we approximated physical sensors with PAISLey-processes with a very small cycle time.

The sensors offer their values to ports. These are processes which read the sensor values and store them into a database or a buffer. Notice, that the ports are part of the system, whereas the sensors are part of the environment. It is not possible to express this distinction in the specification. The processes to specify the ports have different

[3]This is another process as the process modeling the window.

[4]With 'continuous' we mean, that at every point in time a value is available

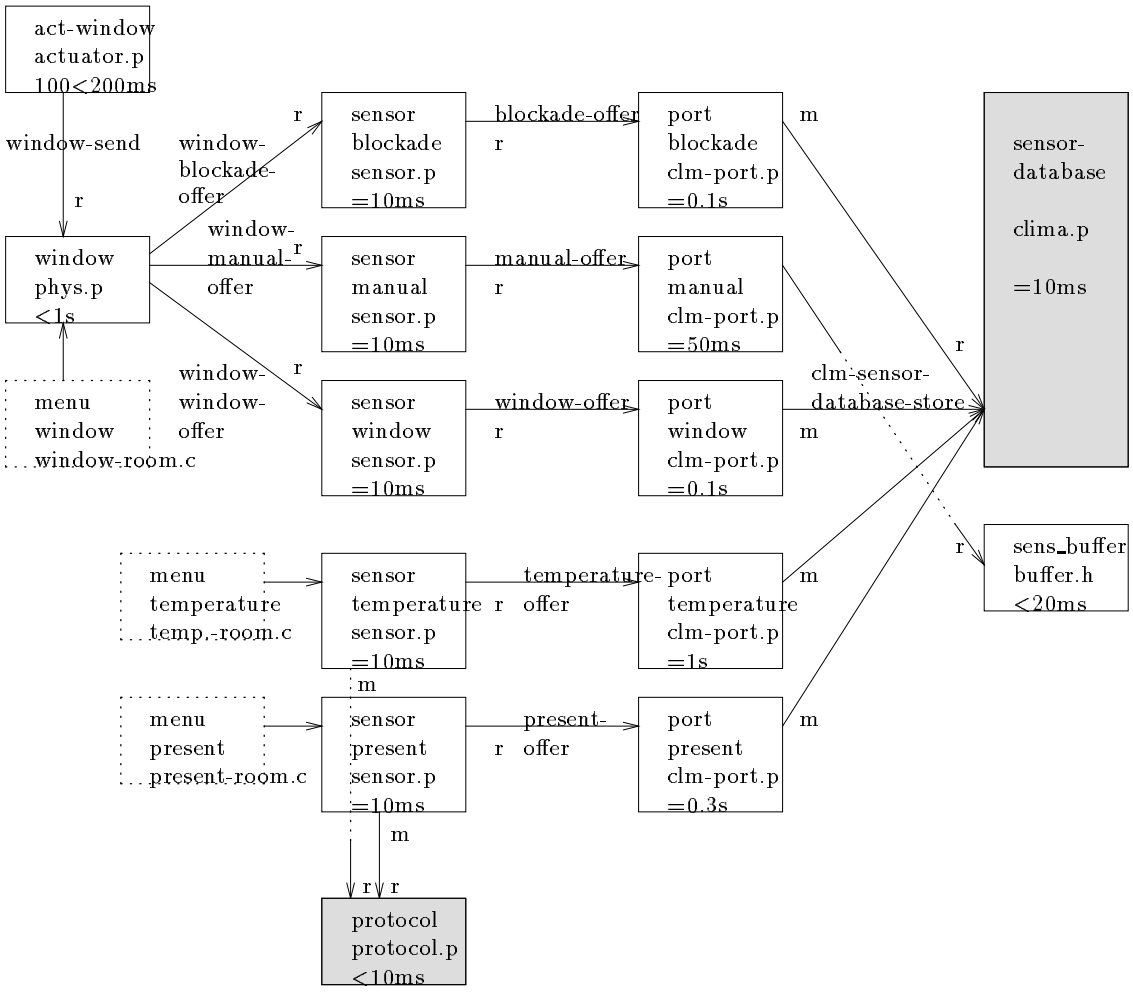

Figure 9: Structure of Version 3: sensors and ports

cycle times, indicating that some of the sensor values are used frequently, whereas others are used seldom. Since we restricted ourselves to a small part of the whole system, the ports distribute their values to only one database. If we had modeled e.g. the alarm system too, then the values read by the ports `window` and `present` would be distributed to this subsystem too. As in the second version it is not possible to detect breakdown of sensors.

The process `protocol` is not part of the system, but is used to support the execution of the specification by collecting important events. This process uses C-functions to write short messages together with a time stamp onto a file.

Since the processes which model the control are not connected directly to the ports it is possible that not all signals from the ports are used and some might be used more than once. For most of the sensors this is no problem. As an example, the control uses the most recent value of the temperature, discarding all older values. It may also use the most recent value several times. But the signals of the port `manual` must not be used twice. The most recent one of the signals `Open` resp. `Close` has to be used. Therefore the most recent signal is stored in a special buffer, which invalidates a signal if it has been read. This kind of buffer is able to store one element. Its content may be overwritten with one of the above mentioned signals, but not with other signals.

Alternatively it might have been better to incorporate this buffer in the database to get a more clear structure of the system. Then the database has to ensure, that these signals are used at most once. But in the structure chosen it is made explicit, that this sensor value is treated differently. Nevertheless it is difficult to see this difference because in the specification the behavior is described, but not the reasons why the system should behave in this way.

Since this buffer is a very simple process we specified it as a macro with parameters using the language of the macroprocessor `m4`. The concrete specification of this process is then just one line consisting of the macro call with the appropriate actual parameters. Other similar simple processes are the sensors and ports. But we did not replace their definitions out of two reasons. On the one hand, it already proved to be useful to use macros in the case of this buffer. On the other hand, the definition of the macros are difficult to read and we did not want to mix up the specification with definitions in the macro language.

The control processes[5] read the values and parameters they need from the two databases. The process `cold` compares the actual temperature in a room with a parameter set by the user and sends a signal (`Close`) to the process `gather` if the temperature is to low. The process `air` checks whether somebody was in the room for a specified amount of time and then sends the signal `Open` to the process `gather`. If the window was opened long enough then the signal `Close` is sent. The process `air` gets the current time from a system clock (`clock`) since it has to use timers. In the module defining `clock` we introduced an abstract datatype `TIMER` with access functions to reset, stop, restart, and read a single timer. Hiding access to channels using interface-functions is also used in the process `lock`, see e.g. the function `clm-window-new-lock` in figure 10.

[5]See figure 10 for the structure of these processes.

The process `lock` has the task to condense information from other subsystems concerning the windows. It was specified e.g. that the windows should be closed if it begins to rain. Then the corresponding subsystem[6], see figure 11, sends a signal to this process, which stores these requests. If it stops to rain, then the subsystem sends another signal to the process `lock` which resets this lock. The process `lock` manages these requests from various subsystems and sends the signal `Lock` to the process `gather` if a new request arrives. If all requests are released, then the signal `Unlock` is sent (the windows may be opened again). This asymmetric behavior is necessary because the windows are not totally under control of the system, the inhabitants may open and close them too.

[6]The structure of this subsystem is similar to the fire-system of version 2 (figure 4) and should be clear without further explanation.

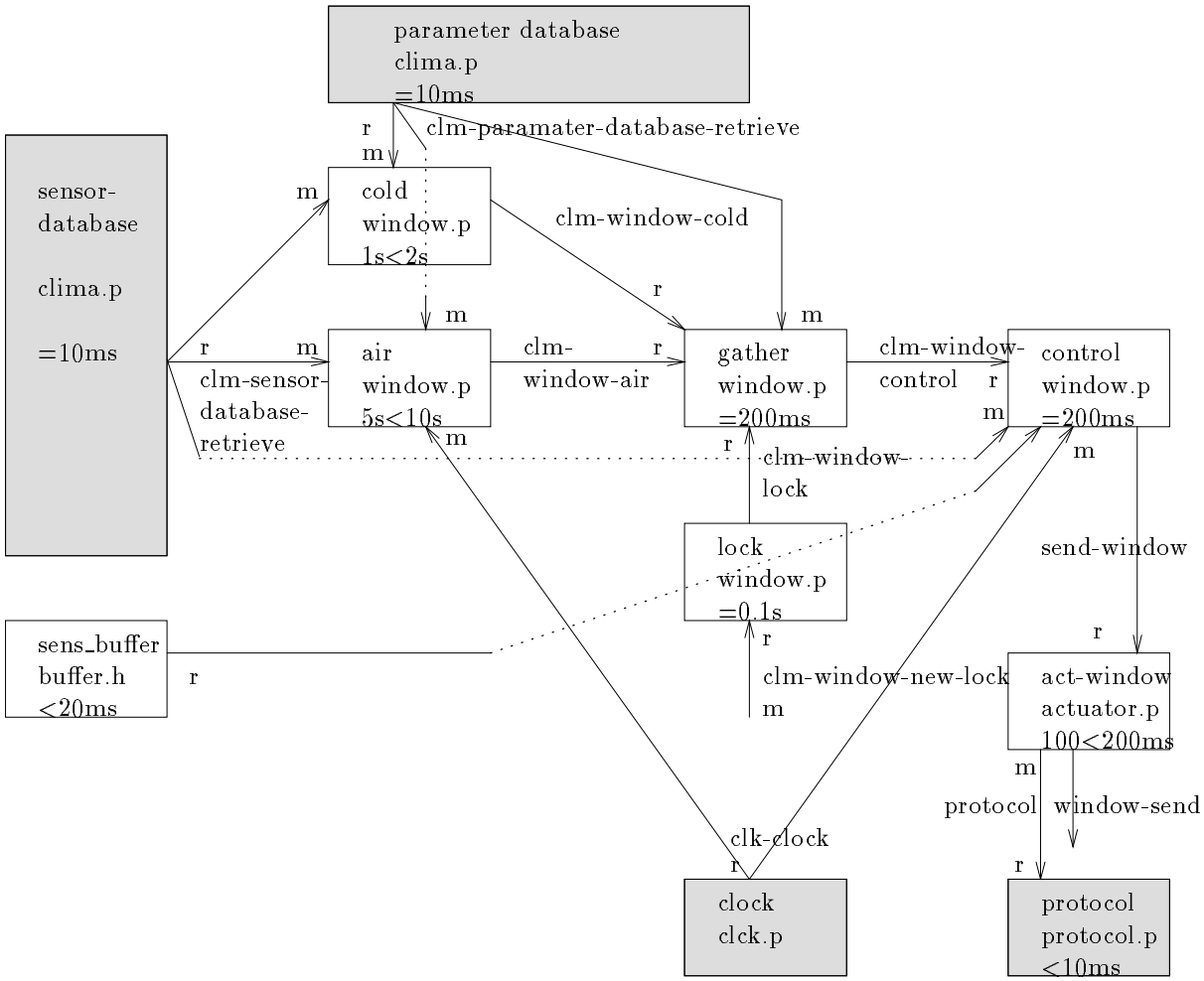

Figure 10: Structure of Version 3: The control processes

The process `gather` collects the various requests to open, close or lock a window of the corresponding processes `cold`, `air`, and `lock`. It further manages activation and deactivation of the ventilation by discarding signals from the process `air` if the subsystem is deactivated, the automatic closing of windows cannot be deactivated. If a window should be moved (and is not locked), then the signal `Open` is sent to the process `control`, which distributes the signal to the corresponding actuator. `Control` further monitors the movement and resends the signal once after a specified amount of time, if the movement is blocked.

Using this complete model of the subsystem it was possible to validate the specification against the informal requirements. But we found that the interpreter is slow which makes it a tedious task to execute the specification. Although the specification of this subsystem was larger than we expected[7], it was not difficult to use PAISLey. We expect that it is possible without serious problems[8] to specify the other subsystems and parts of the user interface too.

6 Evaluation

Based on our experience with the three versions of the specification we will present now the evaluation of PAISLey. Notice, that our comments reflect mostly the viewpoint of a specifier. The specifications have neither been reviewed by potential users nor has the system be implemented using our specification. Let us restate, that at the beginning

[7]The specification contains about 2500 lines of PAISLey (including comments). About 1000 lines are needed to specify the 5 most important processes `air`, `cold`, `lock`, `gather`, and `control`.

[8]It will be necessary to use a strict discipline when writing the specification to avoid e.g. naming conflicts.

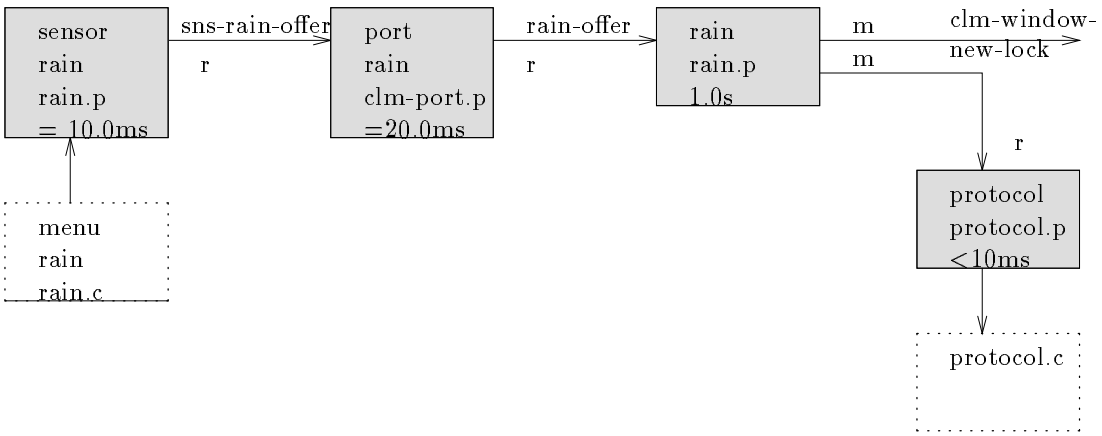

Figure 11: Structure of Version 3: The subsystem `rain`

of the case study the specification of reactive systems as well as the language PAISLey have been something new to us. We will begin with the specifications itself and we will continue with the evaluation of the language and the tools. We will finish by evaluating the goals stated in section 2.

6.1 Evaluation of the Specification

Comparing the first against the second and third version we learned that it is preferable to use small processes. Each of these small processes should describe only one functionality of the system. Then, subsystems can be specified by composing such small parts. Thereby the flow of information can be stated more explicitly. Furthermore, it is easier to change the specification. Assume e.g. that instead of recognizing whether somebody was in a room if a window has been opened there is a sensor to detect whether a window-glass is broken. Then we only have to change the process which detects a housebreaking, but no other ones.[9] These are the same advantages as they are already known for the development of programs. But the use of small processes – and thereby a large number of them – has some disadvantages, too. At first, the complexity of the structure and the number of interprocess connections increases. This makes it more difficult to get an overview of the system. Furthermore it is more difficult to check whether the system is free of deadlocks or whether no important signals are discarded. At second, it is necessary to structure this set of processes. But thereby we create a design of the specification. It might be difficult to relate it to the design of the implementation because they are created for different purposes: The first one helps to understand the task, the second one helps to obtain good solutions.

A second observation is, that it is not sufficient to choose meaningful names for processes, functions etc. The specification must be annotated with comments[10] to express e.g. the purpose of parts of the specification or the connection to the corresponding part of the problem description. Comments help a lot to get an intuitive understanding of the specification.

As already mentioned in section 3.2, process diagrams helped us to capture the structure of a specification and the flow of information between processes. We experienced these diagrams as extremely helpful and regard notations to express the structure of a specification as an important part of each specification method. This view is supported by similar kinds of diagrams, which are used heavily, as e.g. Dataflowdiagrams, activity-charts in STATEMATE [HLN+90] or graphical notations in SDL [BH93].

Using macros or preprocessing steps as in the third version we tried to specify similar processes by instantiating macro definitions. On the one hand this proved to be useful. But on the other hand this version is now a mixture of the languages PAISLey (most of the processes), C (menu interface), and m4 (macros) and of some shell commands to do the preprocessing. These, or similar languages, should be integrated

[9] The processes specifying the sensors, ports and databases might also be changed, but these are very regular structures and changing is therefore an easy task.

[10] This is not surprising as PAISLey can also be seen as a programming language.

better in PAISLey, because the mixture does not enhance the readability and it is error prone.

One might ask, whether the structures of our specification have been optimal or at least good ones. A part of the decomposition is analogue to the functionalities specified and therefore seems to be convenient. The introduction of the databases might be a more debatable decision. But such questions concern essentially design problems and one could ask, whether our specifications are problem descriptions or problem solutions. As this seems to be a general problem of executable specifications we will take up this question in the last section.

6.2 Evaluation of the language PAISLey

We will now present some deficiencies of the language PAISLey. We begin with a very detailed level of the syntax and progress towards more general remarks.

It is annoying, that if a function has several parameters, these must be put in an extra pair of parentheses. Furthermore, in PAISLey tuples with only one component are identified with the component itself. This is difficult concerning the semantics of the datatypes and is a source of errors which are difficult to locate.

There is no possibility to use local variables within functions to avoid multiple evaluation of the same expression. This can only be achieved by using auxiliary functions. Their formal parameters can be used to simulate local variables. But this lowers the readability of the specification.

A PAISLey-specification is a set of descriptions of functions and sets. Syntactical constructs to define e.g. modules or subsystems could help to express the structure of the specification and to avoid naming conflicts. We used different prefixes to name functions and sets of different subsystems. Some support by the language and by the tools would have been helpful here.

In PAISLey similarity between objects can only be defined by a syntactical construct – replication[11] – which inserts indices into names of functions, sets etc. But this is very simple and it is restricted to the part of a specification which is written in PAISLey. Replication cannot be used in the definition of C-functions. The use of macros to describe common types of processes is a further attempt to express these similarities. But, in our opinion, one needs a much more expressive language to describe similarities. It would be appropriate to have types of processes as e.g. in SDL. If these are ordered in a hierarchy of types as it is common in object-oriented approaches, then most of the simple and repeatedly used processes could be described as instances of predefined classes.

Besides processes it should be possible to have objects of type '`channel`' because information about them may be scattered across a specification. It is recommended in the user manual to rename each exchange function to have useful names and to support execution of partial specifications. But it might be more appropriate to define

[11] See the line `room#1.. Number_Of_Rooms` and the word `room` in the function names in figure 4.

channels together with a set of possible signals. Then, to define a function accessing a channel it would be sufficient to state the name of this function, the name of the channel accessed and the type of the exchange function to access the channel.

These syntactic deficiencies are to some part responsible for the large size of the specifications. Even very simple processes are described using several functions. With the use of additional and user defined types and by avoiding unnecessary auxiliary functions the size of the specifications could be a more reasonable one.

Often a process is described by some auxiliary functions receiving or sending signals and preparing them and by one (very) large[12] function computing the next state and the output. Due to the flexibility of the functional language one is not restricted to the description of Mealy- or Moore-automata. But on the other hand, these functions tend to be large and nested case statements which are difficult to read. It might be more convenient to describe them using graphical notations as e.g. statecharts or tabular notations as e.g. NRL, see [CP93].

We found, that PAISLey is not expressive enough to state all desirable timing properties. In our opinion, it is not sufficient to constrain the evaluation time of single functions. Furthermore it is necessary to state constraints on paths connecting several processes. As an example it might be necessary to express the following statement: 'if event a is recognized by a sensor and action b is the corresponding action of an actuator, then there is an upper bound t on the time between the occurrences of a and b'. But in PAISLey such properties must be split into several constraints on the transition functions of the corresponding processes.

Since we found some deficiencies and stated here a lot of critical remarks this section might have an overly negative touch. Nevertheless the combination of the process-oriented and the functional approach is very elegant. A similar approach can be found in the work of Barbacci and Wing, [BW86]. There, tasks and functions accessing ports of the tasks can be used to describe systems and some of their timing constraints. And although there are syntactical inconveniences, it should be possible to specify medium-sized systems, as e.g. the complete home system of this case study, without serious problems.

6.3 Evaluation of the tools

There are four tools available. A *parser* can be used to check a specification for syntactic correctness. A *cross-referencer* can be used to locate, where in a specification a set or a function is defined or used. These two tools provide simple, but nevertheless valuable information.

A third tool – *checker* – can be used to detect inconsistencies in the specification. Among others it is checked whether functions are called with arguments of the correct type. Thereby structural equality on sets is used[13]. Furthermore some inconsistencies

[12] See e.g. the function `fire-state-and-signal` in figure 6.

[13] This is possible, since sets in PAISLey are finite.

between timing constraints can be detected. Unfortunately, this tool reports a lot of situations as errors where eventually no error occurs. A typical situation can be seen in figure 12. The first part defines a function with result `True` if its argument is not `Null`. The second part is a part of a case distinction. To enter this case the variable `new-command` must be not `Null`. But the tool reports an error, which says, that the function `clm-window-control-state-send-window-2` is called in a situation where the first argument is a member of the union of the sets `CLM-WINDOW-CONTROL-SIGNALS` and `FILLER` [14]. There are a lot of such wrongly detected errors which make it difficult to detect the real errors. Further, since the exchange functions are provided by the system, they are declared to be functions which take arguments of the type `ANY`[15] and produce results of the same type. Therefore it cannot be checked whether two functions accessing one channel coincide in the set of objects they want to transmit.

The fourth tool is an *interpreter* to execute PAISLey-specifications or parts of it. Again our criticism is twofold. On the one hand we could use the interpreter successfully, that is, we found errors in our specification. On the other hand the interpreter is difficult to use. The main reasons therefore are that there are only simple commands and that there is no language to control simulations. Attaching C-functions which realized menus helped us to input sensor signals in an easier way. But due to interprocess communication the interpreter became very slow. The interpreter needed more than 4 hours user time to simulate about 1 minute. In this specification we have, besides processes with very short cycle times (e.g. the sensors), processes, which control

[14] `FILLER` is the singleton set consisting of the element `Null`.

[15] `ANY` is the set consisting of all possible objects.

```
value-received-p : ANY --> BOOLEAN;
"to check whether a value is Null"
value-received-p[value] =
        not[is-null[value]];

...

value-received-p[new-command] :
        "Command to move the window was given by the other processes"
        clm-window-control-state-send-window-room[(
                new-command,Not-Blocked,clk-timer-init
                )],

...

"window.psl", line 791:  Inconsistency between application of
mapping "clm-window-control-state-send-window-2" and declaration on line 876.
Argument set given to "clm-window-control-state-send-window-2" is:
        ((CLM-WINDOW-CONTROL-SIGNALS|FILLER)*Not-Blocked*(0*0))
```

Figure 12: Wrongly detected error

slow physical processes (Opening and Closing the windows to ventilate a room). As an example of the use of the interpreter see figure 13, which shows the menus to enter the sensor values, the output of the interpreter and parts of the contents of the file which is written by the process `protocol`. The output of the interpreter shows, that not all timing constraints have been met.

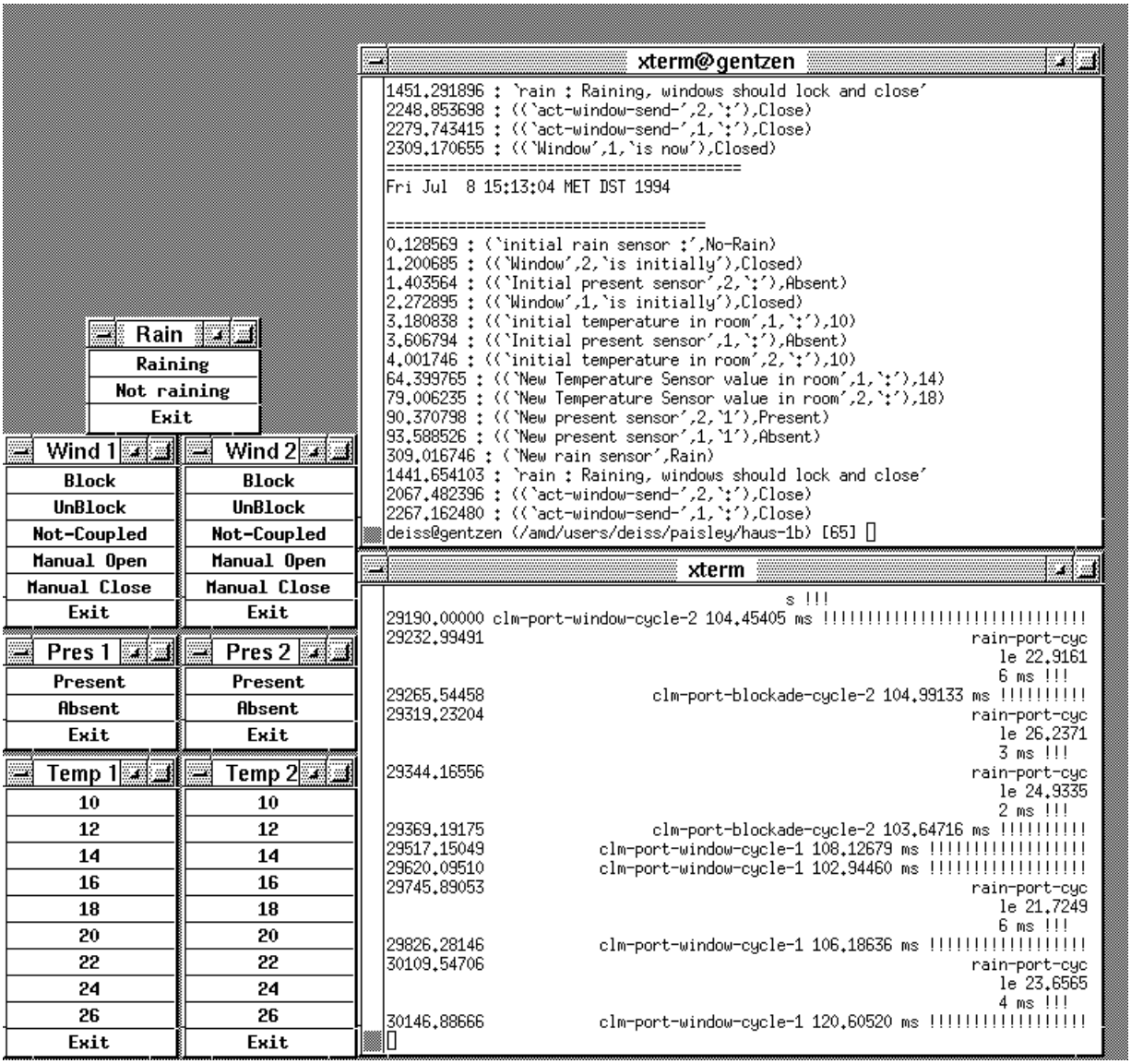

Figure 13: The PAISLey interpreter

6.4 Achievement of the goals

In the previous sections we made more specific remarks on PAISLey. Now let us come back to the goals concerning requirements specifications stated in section 2. We will begin with the four criteria of Zave:

Use a specification as a vehicle for communication: We assume, that the plain textual representation in the style of a programming language could be understood only by computer scientists. To be usable by other persons, the specification must be translated into a convenient language. In the literature it is proposed to execute a specification and to prepare the results in a readable way. But every translation bears the danger of new misunderstandings, therefore it remains difficult to communicate with users.

Change a specification: When using small processes it is relatively easy to change a specification. Local changes in the problem description will probably lead to local changes in the specification. But modularization is not supported and must be replaced by a very strict discipline when writing a specification.

Use a specification to constrain target systems: See below.

Use a specification to accept or reject final products: This case and the previous one can be answered together. In a PAISLey-specification one builds a model of the system planned. The functional approach, especially the use of exchange functions, hinders a direct translation of each process in this model into a corresponding process in the implementation. Successively replacing each PAISLey-function with corresponding functions in the implementation language is not possible, because at last the communication – the exchange functions – and the scheduling strategies of the PAISLey-interpreter have to be be translated into the implementation. Hence the structure of the specification and that of the implementation will be different. This makes it difficult to compare them. In our opinion one should avoid such large steps. Instead, there should be a number of small steps to derive the implementation from the specification as it is suggested e.g. in [BH93]. Then it is possible to achieve traceability of requirements down to the code.

This large step prevents the use of analytical methods when comparing a specification and a final product. Conformity must be checked by extensive testing of the final system against the model. Therefore we find it difficult to use a PAISLey-specification to accept or reject a final product.

The criteria of van Vliet [vV93, page 144] are originally stated for the document 'requirement specification'. Here we will try to answer the question whether it is possible to create documents using PAISLey which fulfill his criteria.

readable: As already mentioned,the plain textual representation should be extended with graphical or tabular notations.

understandable: There should be the possibility to express more structural information.

unambiguous: As long as all functions used are also defined, a PAISLey-specification is unambiguous and complete.

complete: See the case above.[16]

verifiable: It is difficult to verify the final system against the specification.

consistent: A lot of inconsistencies can be detected by the checker and by the interpreter. (But remember, that there are also wrongly detected inconsistencies.)

modifiable: It can be achieved, that the effects of changes in the specification are restricted to a small part of it.

traceable: Since requirements may be modeled by the behavior of several processes, traceability is low. One has to use comments to link the description of the processes with the corresponding requirements.

useable: We have no experience on the usefulness of the specification during operation of the final system. But we expect that it is low due to the large gap we suppose between the specification and the implementation.

Furthermore it should be noticed, that it is only possible to specify the functionality (including some timing constraints). It is not possible to express e.g. constraints on the implementation language or non-functional properties as robustness, fault-tolerance, user-friendliness.

Altogether we found that PAISLey is a good approach to the specification of reactive systems. Unfortunately its usefulness is diminished by deficiencies ranging from syntactical inconveniences to a lack of expressiveness concerning temporal properties.

7 Concluding Remarks

At the end of section 6.1 we posed the question whether our specifications are problem descriptions or problem solutions. But we think that this question has to be asked for all specifications written in languages which emphasize executability. As PAISLey, these languages follow the approach to have an *executable model of the proposed system*[17]. Then a specification contains not only a description of the problem (**What** should be done?), but also a solution of it (**How** can it be done?). This situation violates the commonly accepted definition of 'requirements': the problem only should be stated as precise as possible. Solutions should be developed during design and implementation.

[16] Completeness can be checked only on a syntactic level. It is not possible to check whether the whole problem has been captured.

[17] [Zav82, page 250]

In the following we will try to shed some light on this dilemma without being able to solve it. There exist a lot of models how software should be developed. Most of them have a clear separation between the documents describing the requirements and the design. It is accepted, that in general the creation of these documents can not be done in phases which follow strictly one after the other: At first make a requirements specification, then make a design. It is seen that these tasks are coupled tightly and are performed in an interleaving way in practice. At least in the brains of the specifiers there are rough models of the specified system. Furthermore, the specification itself is structured. There are structures according to the domain under consideration and there are structures in the document itself which help in understanding it. These structures are very close to a design. The thoughts above are true for all kinds of systems and all specification languages. Because reactive systems often have additionally a clear physical structure, these thoughts are even more demanding for this kind of systems and for executable specifications. Therefore it seems appropriate to weaken this strict separation between specification and design, without denying the value of a consistent and complete requirements specification. It might be better to see specification and design as different viewpoints of the same model. The viewpoints might be separated by the level of abstractness.

Another point is that different kinds of requirements have to be specified. Two typical examples are: 'If x happens, then do y' and 'x and y must not happen together'. It seems to be very natural to express the first one by building a corresponding – possibly executable – model. But the second one is more difficult to handle. If it is modeled in an executable specification there are two problems: Does the model really have this property and it is recognizable, that it should have it? It might be better to declare that the system must have this property without encoding it into the model.

Therefore it might be worth to think about a way to build reactive systems by successively refining an abstract model of the proposed systems[18]. This model might have a rough design being a basis for the design of the system itself. It might consist of descriptions of pattern of behavior and descriptions of properties. During design and implementation these descriptions and declarations are broken down to the description of single processes and probably declarations which must be fulfilled by the scheduler of the underlying operating system.

Finishing our excurs we think that it is essential to know which kind of information is needed to give a correct answer to the question 'What should be done?' and how these information should be presented. Up to now different languages or methodologies provide different answers concerning the kind of information as well as its presentation.

References

[BH93] Rolv Bræk and Øystein Haugen. *Engineering Real-Time Systems, An object-oriented methodology using SDL.* Prentice Hall, 1993.

[18] Our ideas are influenced by the method described in [BH93]

[BW86] M.R. Barbacci and Jeanette M. Wing. Specifying functional and timing behaviour for real-time applications. Technical Report CMU/SEI-86-TR-4, Software Engineering Institute, CMU, 1986.

[CP93] P.-J. Courtois and David Lorge Parnas. Documentation for safety critical software. In *Proc. of the 15th International Conference on Software Engineering*, pages 315–323, 1993.

[HLN+90] David Harel, Hagi Lachover, Amnon Naamad, Amir Pnueli, Michal Politi, Rivi Sherman, Aharon Shtull-Trauring, and Mark Trakhtenbrot. STATEMATE: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16(4):403–414, April 1990.

[Rei85] W. Reisig. *Petri Nets.* Number 4 in EATCS Monographs on Theoretical Computer Science. Springer, 1985.

[vV93] Hans van Vliet. *Software Engineering, Principles and Practice.* John Wiley and Sons, 1993.

[Zav82] Pamela Zave. An operational approach to requirements specification for embedded systems. *IEEE Transactions on Software Engineering*, 8(2):250–269, May 1982.

[Zav88] Pamela Zave. *PAISLey User Documentation.* Computing Systems Research Laboratory, AT&T Bell Laboratories, 1988.

[Zav91] Pamela Zave. An insider's evaluation of PAISLey. *IEEE Transactions on Software Engineering*, 17(3):212–225, March 1991.