A Case Study on the Use of SDL

Thomas Deiß and Thomas Hillenbrand

`{deiss,hillen}@informatik.uni-kl.de`

03/1997

Sonderforschungsbereich 501

Universität Kaiserslautern
Erwin-Schrödinger-Straße
67663 Kaiserslautern

A Case Study on the Use of SDL[1]

Thomas Deiß and Thomas Hillenbrand
Universität Kaiserslautern
Erwin-Schrödinger-Straße
67663 Kaiserslautern
`{deiss,hillen}@informatik.uni-kl.de`

Abstract: This paper presents the experience the authors gained in applying formal methods — mainly MSC and SDL — when specifying a reactive system. The experience not only deals with the descriptions of the system, but also with the methodology used to develop the descriptions.

1 Introduction

Formal methods have been proposed as an aid in developing information processing systems. They are considered beneficial because it is possible to produce more precise descriptions in the early phases of software development. This should assist in comprehending the system and it should — at least partially — make automated analysis of its descriptions possible.

A large variety of formal methods has been developed up to now, but only for few of them there is a corresponding methodology. In order to make formal methods more useful, it is necessary to gain deeper insight into and to develop further the existing methodologies.

The methodology presented in [BH93] was used to specify the kernel of an event-driven simulator for heat flow within buildings. The goal of this study[2] is to evaluate the appropriateness of the methodology and the languages used for specifying the simulator. The results should be interesting both for researchers — to gain deeper understanding of the use of formal methods in general — as well as for practitioners — how to use specific languages.

After introducing the context of this study, a short summary of the methodology presented in [BH93] is given. Sections 4 and 5 present the simulator problem and parts of the documents produced to support evidence on the lessons learned. The development process is described in section 6. The lessons learned by the authors are given in section 7. The paper finishes with hints on related and possible future work.

2 Context and history of this study

The study was performed within the Sonderforschungsbereich (SFB) 501 of the Deutsche Forschungsgemeinschaft. The large and ambitious goal of the SFB 501 is to develop *generic methods for the development of large systems*. As a first application domain, building-automation has been chosen, because a large number of variants and different scales of systems are possible here. At the beginning of SFB 501 a team of researchers — called `team1`— constituted to evaluate whether and if so which formal methods are applicable in this domain. There was at least one member from each of the seven initial subprojects of the SFB 501 in `team1`. This resulted in a lot of different knowledge within the team. The initial goal of `team1` was also quite

1. This work was supported by the Deutsche Forschungsgemeinschaft, Sonderforschungsbereich 501, “Entwicklung großer Systeme mit generischen Methoden”.
2. See [Bas96] for an overview of the use of experiments in software engineering.

ambitious because all its members could be regarded more or less as novices in the application of formal methods. In order to get acquainted with their use, a simulator for heat flow within buildings was specified as a first step. Due to an important design constraint — the simulator had to be event-driven[3] — it fit nicely into the paradigm of processes which communicate by messages. This paradigm is supported by the languages MSC (Message Sequence Charts, [C93b]) and SDL (System Description Language, [C93a]). The methodology presented by Bræk and Haugen [BH93] is based on these languages and suggests how to use them. A first version was produced to gain a common understanding of specifications in general, the language SDL, and the methodology above. After finishing this version, `team1` proceeded by specifying building automation systems using formal methods.

A lot has been learned by the members of `team1` while specifying the simulator, but the resulting documents were not very mature and contained some weak points. It was decided to produce a 'better' specification for this problem to grasp the experience gained on a more solid basis. A revision of the problem statement produced by `team1` was used as a starting point. All descriptions produced by `team1` have been available and have been used to understand the problem domain. This allowed to use a process model closely related to the waterfall model. In the new descriptions several aspects of the simulator problem have been stressed and others have been simplified. Therefore, all descriptions were developed from scratch. The more concrete ones such as e.g. the functional design resemble the version of `team1` only in their use of common concepts of the problem domain. This study presents essentially the production of this improved version.

The study was performed by the authors themselves. Most of the actual work in producing the descriptions was done by the second author, who had knowledge about SDL on the level as presented in [BH93], but few practical experience in applying the language. He also has a strong background in theoretical computer science, especially in logic and term rewriting. The first author supervised and guided the work. He has been a member of `team1`, his background on SDL and its application is based mainly on the experience gained in producing the first version. It is complemented with general background and research interests on formal methods for reactive systems and experience gained from case studies using other languages, see e.g. [Dei94].

Since the authors did not consider themselves as experts on the methodology and the notations used it was anticipated that errors requiring rework would be made, and in fact, they have been made. As a consequence no detailed data on the time used to produce the descriptions have been collected. To produce the descriptions approximately 550 hours of work of the second authors were necessary. The work was carried out within one year. In addition, time has been spent by both authors to discuss open problems and to perform reviews of the descriptions. Besides a lot of work done with paper and pencil, two tools have been used: Framemaker to edit text and to draw diagrams (including MSCs) and SDT (version 3.02) to produce and analyse the functional design written in SDL. As the main source of information about the languages and the methodology the textbook [BH93] has been used, in addition [OF+94] has been taken as a reference on SDL.[4]

3 Overview of the development methodology

This section summarizes the methodology used in this study as it is presented in [BH93]. There [page 10, *emphasis* by the authors of this study] a *methodology* is defined as a set of *meth-*

3. See [Eva88] or [Meh94] on the topic of event-driven simulation.
4. This book also contains a methodology similar to [BH93], but it is presented there only shortly. Hence, for methodological issues only [BH93] has been used by the authors. While performing this study, a summary [Ree96] of the new version of the recommendation on the use of SDL and related languages was published. This recommendation is briefly discussed in section 8.1.

ods where each method is a systematic way of producing some result. Each method in turn should provide *guidelines* for structuring and using *descriptions* in given *notations*. This emphasis on descriptions stems from a broad *systems engineering* perspective. There, clear and precise descriptions of the system are considered essential to understanding, analysing, and communicating about the system.

The main descriptions proposed in [BH93] are requirements specification, design, and implementation. In the *requirements specification* the needs of the owner and of the users are expressed in a way understood by the owner, the users, and the developers of a system. It is not intended that a requirements specification is complete in all details. These details are filled in by the *functional design*. The functionality of the system is given there as clearly and completely as possible. Based on the functional design and the non-functional requirements, the architecture of the realization can be given, which is called here the *implementation design*. Both designs together form the basis of the actual *implementation*. In the study the authors concentrated on the requirements specification and the functional design.

These descriptions and some of their relationships are shown in figure 1. The arrows show the flow of information between different descriptions, they do not prescribe the use of a certain process model, e.g. the waterfall model. Note that the non-functional requirements do not contribute to the functional design, but to the implementation design.

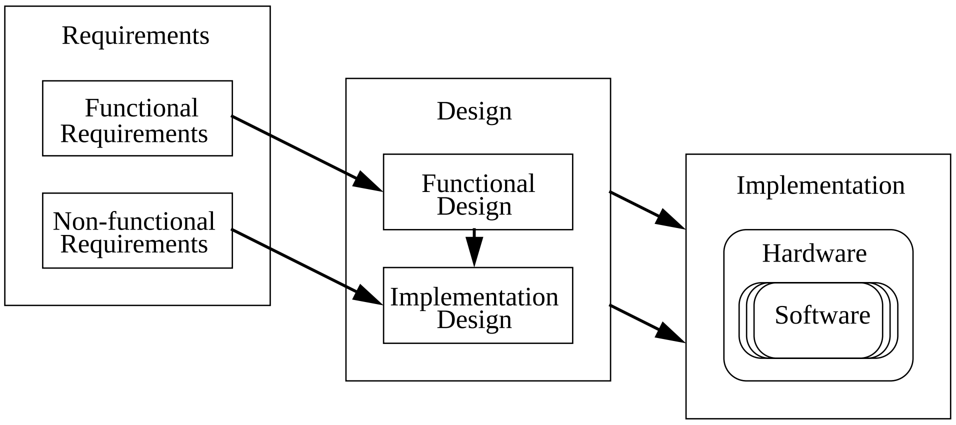


Figure 1: Main descriptions ([BH93], page 13)

In this methodology, the design is the central part. In [BH93] it is stated (page xiv) that using the methodology to its full potential is to work *design-oriented*. This means that systems are understood, validated, and maintained primarily on the design level. Implementations are derived more or less automatically. The methodology aims at reducing unnecessary redundancy between different descriptions ([BH93], page 16). One specific piece of information should be expressed only once. The descriptions shall not replace, but complement each other.

3.1 Requirements specification

To express the needs of owners and users, the requirements specification focuses on the purpose and the role of the system as seen from its environment. The following parts of this document are proposed:

- a *problem statement* to answer the question *why* the system shall be build,
- a *domain analysis* for the problem domain; this includes a *concept model*, a *dictionary* of terms, and a separation of the system from its environment as a *context model*,
- a description of the *interface behaviour*, including projections of it to certain roles,
- a list of the owner (*non-functional*) *requirements*,
- a *sketch of the system structure* to capture implications of the requirements on the design.

In the *concept model* a static conceptual description of the problem domain is given. Diagrams show relationships between instances of the concepts. For each concept, its attributes and relationships are additionally described in more detail in a *type diagram*. If necessary, the concepts are classified in a *specialization hierarchy*.

In the *context diagram*, the system is identified within its environment. Also described are its communication interfaces and other relationships it handles. As a part of the communication interface, a *static interface description* is given, i.e. the information which may flow through this interface.

In the *dynamic interface description* the sequential ordering of interactions is described. This is restricted to typical interaction sequences and is therefore intentionally not a complete description. A *role* is associated with the needs of one user, who expects a system to play a certain role. On the other hand, a user must also play a certain role, otherwise, the cooperation between user and systems will not work. For each type introduced in the concept model, roles are defined and related to entities. The roles are used to connect several interaction sequences. Therefore, one concentrates on one role at a time. The interface behaviours of each role can be used as a basis of synthesis and validation of the complete system behaviour. In this study only the synthesis aspect is relevant, validation of the implementation against the requirements is not considered.

The languages used in these descriptions are natural language, an object-oriented notation (SOON, [BH93]) for the concept model, message sequence charts (MSC, [C93b]) for instances of interface behaviours, and transition charts (TC, [BH93]) for patterns of interface behaviour.

3.2 Functional design

The complete functionality of the system at its external interfaces is described in the *functional design*. This answers in detail the question *what* the system has to do. It shall not be biased towards a specific realization, expressing *how* the system satisfies the requirements. A realization is described by an implementation design and an implementation, see figure 1.

It is considered preferable to use a formal language with a well-defined semantics for the functional design. This allows understanding and analysis of the system without referring to its implementation. It is clear that the semantics of the language should be well-suited to the application domain.

In [BH93], the formal specification language SDL [C93a] is suggested to express the functional design. The system is seen as a set of concurrent processes communicating via asynchronous message passing. The processes are essentially finite state machines extended by data and communication. Processes are aggregated into so-called blocks, which may be aggregated into blocks again. The system’s top-level structure consists of blocks only. Hence, the behaviour is described in an operational way, including internal structure of the system. The methodology contains a lot of guidelines on the use of SDL related to the structure of the descriptions, e.g. when to partition a process, as well as to notational or layout issues, e.g. how to present one process on several pages. For developing the functional design, it is recommended to proceed step by step, see figure 2. This procedure was applied throughout this case study.

4 Requirements specification

In this and the next section typical excerpts of the documents are shown to give an impression of the problem and its specification. Readers interested in the complete documents are invited to contact the authors or to visit the web pages of SFB 501, subproject C1, at
`http://wwwsfb501.informatik.uni-kl.de:8080/`

1. *Describe the environment.* Decompose the environment into blocks and processes. Identify the behaviour roles these blocks impose on the system and the knowledge they require in the system.
2. *Mirror the environment behaviour.* Mirror the behaviour roles imposed by the environment, by actor roles and actor processes within the system. (...)
3. *Mirror the environment knowledge.* Assign the knowledge about the environment to processes in the system. (...)
4. *Analyse the behaviour.* Can the behaviour be clearly and concisely described in state-oriented form? If not, analyse the reasons and modify the structure to solve the problem either by splitting processes, adding processes or combining processes.
5. *Analyse the block structure.* Check that the blocks fulfil the block purposes. (...)
6. *Look for similarities.* Identify types and type hierarchies. Organise the types into libraries. (...)
7. *Analyse the variability.* What variability is needed in the product? Check that it can be accommodated by the types developed.
8. *Iterate until satisfaction.*

Note that the steps above, in principle, apply to each block in the same way as to the system. To derive a good system and product structure is an iteration process. We therefore recommend going through the steps several times, gradually adding detail to the description.

Figure 2: Step-wise procedure for the functional design ([BH93], page 214)

4.1 Problem description of the customer

This study is based upon the problem to develop a simulator for heat flow within buildings. The simulator had to be usable in quite a broad range of application contexts. Two important design constraints were given: The numerical part of the simulation had to be encapsulated within so-called simulation objects. It was neither part of the problem to specify the numerical methods needed to simulate the physical quantities, nor to develop the simulation objects. Also, the simulation had to be an event-driven one, see e.g. [Eva88] or [Meh94]. Hence, the main task of the simulator was to collect information from the input sources and to distribute it to appropriate simulation objects in a timely fashion. Vice versa, information from the simulation objects had to be distributed to the output destinations, again in a timely fashion. The simulator could be seen as keeping a large collection of events which had to be distributed at the correct point of time. It thereby had to ensure that all simulation objects kept synchronized with a global simulation time. Furthermore, it had to be possible to use the simulator together with a control system for the building being simulated. Therefore, the simulator had to be able to maintain a close binding between real and simulation time.

The problem description was a slightly modified version of that used already by `team1`. It consisted of:

- a short and informal description of the simulator problem and of the requirements,
- an introduction to the physics to be simulated, e.g. the geometry of a building and the installation contained in it,
- a list of usage scenarios, differing mainly with respect to possible types of input sources and output destinations of the simulator, such as e.g. files, human users, or a control system.

This document did not contain a bias towards formal methods, neither in general nor to specific ones. Except the part related to the physics, it was written in natural language.

4.2 Problem statement

The information given by the customer was looked through, and important information contained in it was collected in the problem statement. This was similar in content to the introductory part of the problem description of the customer. The purpose of the system and its observable interface was made clearer in this document.

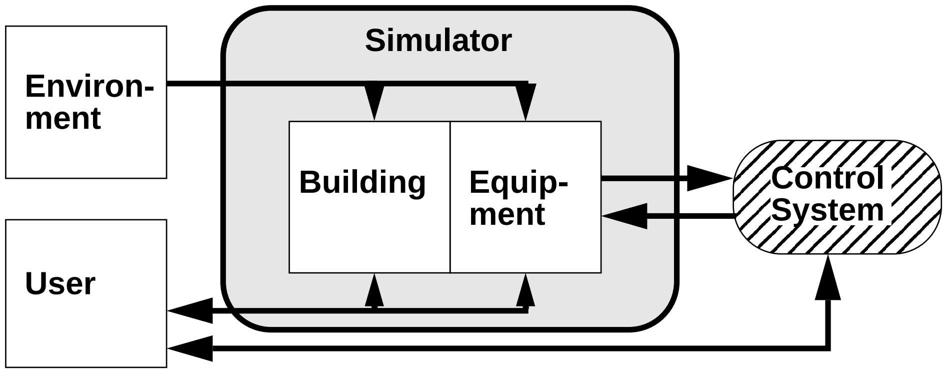
Due to the required use of simulation objects it was possible to consider the numerical part of the simulation as being encapsulated in them, and it appeared no longer necessary to talk about the physics. The different types of input sources and output destinations were considered as subtypes of a more abstract type. As a consequence, the difference between the usage scenarios diminished, and they were not included in this document. Due to these abstractions, the simulator, as it was developed further, became a very generic one: It can be used for different kinds of buildings, the simulation of other physical properties of buildings, and even for simulation in other domains, such as e.g. railroad transportation and signalling systems.

The document was still written in natural language. It is reprinted completely below to give an overview of the problem to be solved.

Introduction

The main purpose of the system to be developed is the following: Given an adequate model of some arbitrary building and its equipment, the system simulates some of the physical properties of parts of the building during a certain period of time and with an appropriate accuracy.

Some of the situations the system shall be useful in:
- Properties of a future building and its installation can be tested before construction; this is useful in the context of architecture.
- The owner of a building may want to test new equipment, new automatization soft- and hardware or new settings for existing ones.
- Developers of automatization software can validate their products in an environment where faults and errors are less unpleasant.



The arrows in the above illustration depict dependencies and interactions in the physical world, and the shadowed simulator box demonstrates that the system has to play the role of the building and its equipment, including all of the modelled arrow relations.

Figure 3a: Problem statement

Behaviour at the user interface

Before starting, it is possible to set:

- installation parameters, such as the standard heat output of the radiators;[1]
- initial values of physical quantities and equipment state, the so-called visible variables; an example is the air temperature of a room,
- input and output configuration,
- halting-points,
- simulation-specific parameters, such as the mode for simulation time advancing or the slack.

For everything not explicitly set, default values are assumed. It is possible to dump the initial values. The simulation can be run until a specified point in time or “forever”.

When running, it is possible to stop and resume the simulation. When stopped, it is possible to change the values of all visible variables, but not of installation parameters. Furthermore, the simulator accepts inputs during a run without being stopped.

Inputs are setting of or request for the values of visible variables.
Requests for values are responded to the requester immediately and without increase of simulation time.

Concurrent sources of input are: the user via keyboard while the simulator is running or halting; files; sensors; external programs controlling a part of the installation.

Input collisions – two different attempts to set a visible variable at the same time – are detected and reported.
The format of input is a triple *(timestamp, object, message)*. In case of periodic events, a fourth element *period* may be added. Furthermore, there is a special timestamp meaning “as soon as possible”.

Outputs are sent to several destinations and consist of values of visible variables. The format of output is similar to that of input. Outputs can be written periodically, on request or after change of value. This may differ for every pair of source – some visible variable – and destination.

The simulator starts with a specified internal **simulation time** and may advance in two different manners: as fast as possible or a given **timing-factor** faster than real time, the time going by when the simulator is executed. In the latter case, there is a close binding between real time and simulation time: If the factor is e.g. *24*, it should take exactly an hour to simulate the behaviour of the building during a complete day. Simulation time is allowed to be a maximum amount of time behind; this deviation will be called slack. Violation of this condition must be reported; the simulation run will stop in this case if this is chosen before starting.

1. In contrast, the topology of the building is fixed: the size of the building, the number of rooms, etc.

Figure 3b: Problem statement

Design constraints

The **simulation** is to be done as an **event-driven** one with discretely advancing simulation time. The physical properties of the building and its installation shall be encapsulated in "simulation objects" which have to be invoked periodically to perform their simulation. To ensure numerical stability, there is a maximum invocation period for each object. This may change during a simulation run. The whole process of modelling shall be done in an **object-oriented** manner, thereby giving rise to re-use and forcing a generic approach. To avoid the situation that at time t an object o_1 does its simulation using part of the state information of object o_2 which has just changed at time t, every object has to buffer its previous state. This technique will be called **double-buffering**.

Errors occurring during a simulation run are reported. The program continues as long as the simulation results are not affected. Incoming data has to be checked for **incorrectness**, which must be stated. As soon as the simulation objects themselves realize that their internal state is **inconsistent**, the simulation run is aborted.
The system is able to write **dumps** containing all the visible state information of the simulated objects and **checkpoints** which serve to re-start an identical simulation run.

On a typical workstation, it shall be possible to simulate the example given in the problem description such that simulation time is cut down by a **timing-factor** of *500*.

Figure 3c: Problem statement

4.3 Concept model

The notion of simulation based on simulation objects was described in an abstract way in the concept model. This document remained on an abstract level and contained no more concrete notions of the problem domain (such as e.g. event collection or simulation time).

The document itself consisted of one entity-relationship-like diagram and explaining text. A part of it is shown in figure 4. The notation SOON (SISU object-oriented notation [BH93]) was used for this diagram.

4.4 Data dictionary

The concepts of the simulation domain for the version produced throughout this study are the same as for that of `team1`. Therefore it was possible to reuse the data dictionary of `team1`.

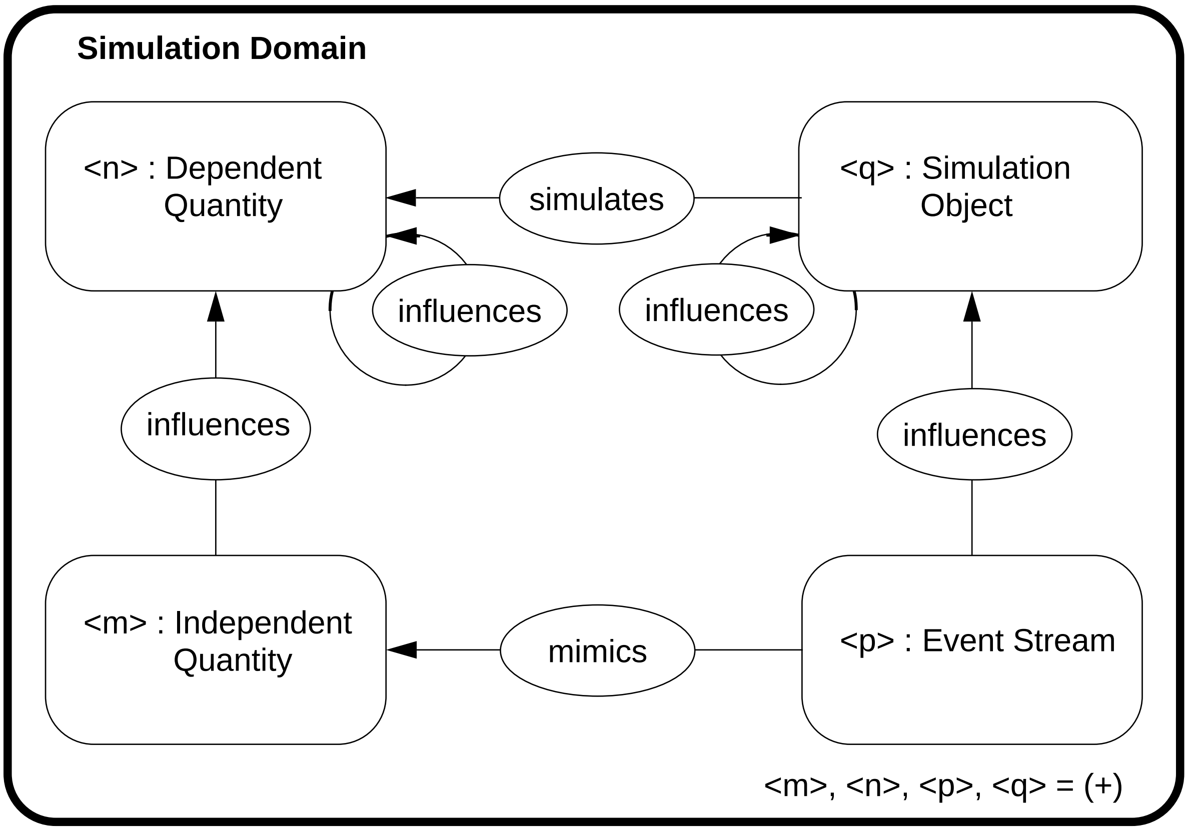
4.5 Context diagram

In the context diagram the relationships and objects of the concept model which had to be realized by the simulator[5] were described. Also, the environment of the simulator was characterized, i.e. which objects had an interface to the simulator. These objects were the simulation objects and the input sources and output destinations, which were abstracted to the concept of event streams. One special event stream, called the control stream, connected the simulator to a user controlling the simulator itself.

The context diagram was on the same level of abstractness as the concept model. As one can see from the part of the context diagram shown in figure 5, the relationship influences between event streams and simulation objects had to be realized by the simulator. Therefore it was di-

5. Names of objects, relations, etc., used in the documents are written using a sans serif typeface to distinguish them from surrounding text.

First of all, the basic notion of simulation shall be revisited, with the design constraint in mind that the simulation is to be done in an object-oriented fashion. We thereby get the following general picture of the simulation process to take place:



In the left half of the diagram we find representations of objects in the real world. These are looked at from the physical point of view. In our context, **independent quantities** are such as the dimensioning of the rooms, the size of the windows or the standard heat output of radiators. These have an **influence** on the **dependent quantities**, for example on the indoor air temperature or the state of a radiator valve that might be opened by an inhabitant or even by an automatic control system. The outdoor climate could be subject to simulation as well. However, we are aiming at automatization of buildings and interested in observing the indoor climate. Therefore, the outdoor climate will be regarded as independent quantity, which itself is subject to change in dependence of time.

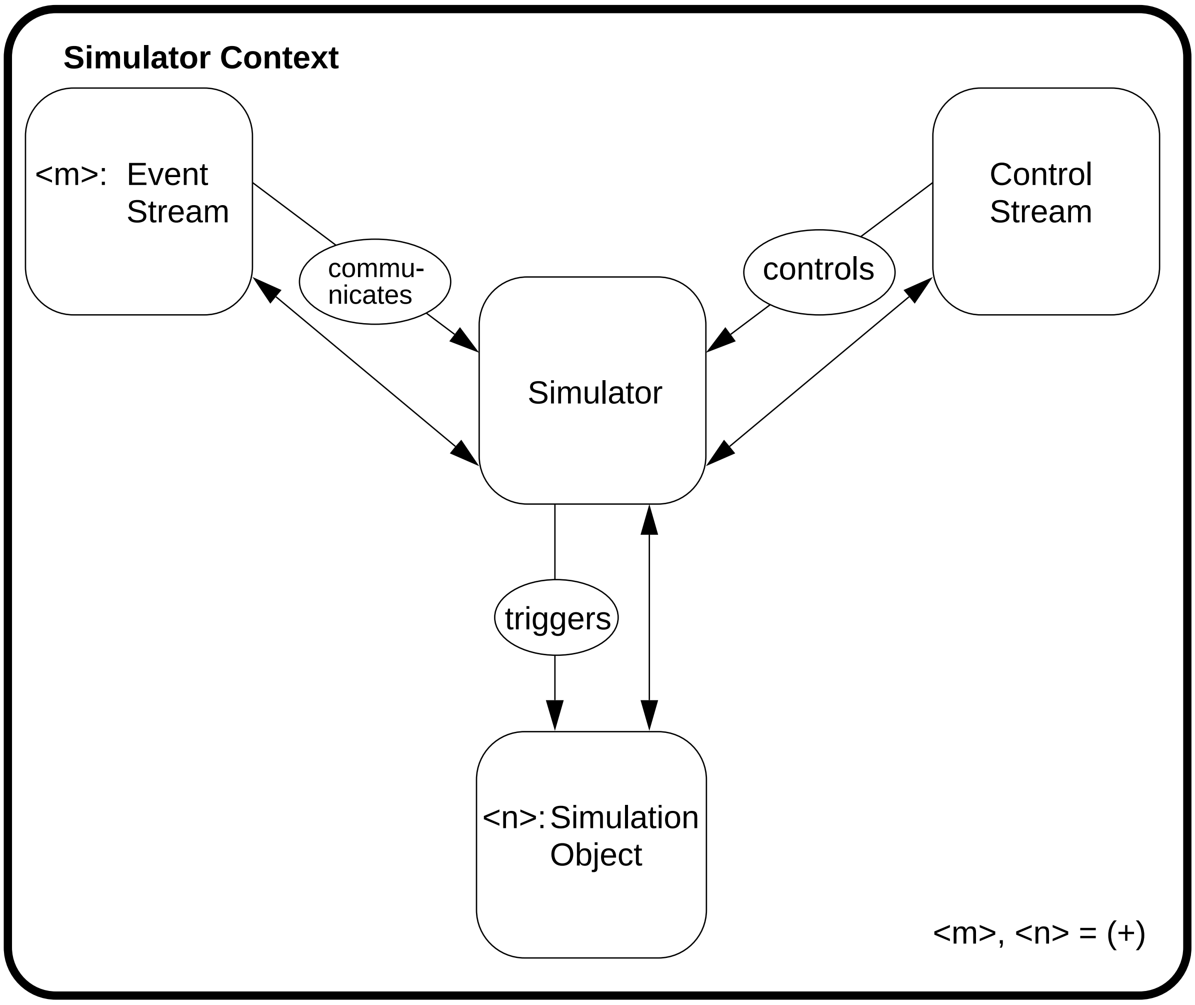
Figure 4: Part of concept model

vided into the relationship communicates between event streams and the simulator and the relationship triggers between simulation objects and the simulator.

4.6 Static interface description

The objects forming the environment of the simulator and their interfaces to it were specified in this description. For each of the three types of objects, a type diagram showed the relationships between an instance of this type and other objects. The attributes of objects of this type were given as well. Signals which could be sent between objects were declared, and their intended use explained textually, as well as the data types the signal carried. Thus the document was structured such that all information concerning one interface was presented together. The explanations of elements of the different categories *object types*, *signals*, and *data types* were arranged according to their usage. These categories were not used to structure the descriptions of each of the interfaces.

The main task of the system is to realize the **influence**-relation between the different event streams and the various simulation objects. The **simulate**-relation, that is the actual modelling of the physical, dynamic behaviour of the dependent quantities, is encapsulated in the simulation objects (design constraint). The **mimic**-relation demands correspondence of independent quantities and event streams, which have a format the system is able to process, and is subject to modelling as well. Hence, only a few entities and relations of interest remain:



From the simulation domain, only the event streams and the simulation objects are left. The **influence**-relation is decomposed into the entity **Simulator**, which on the one hand **triggers** the **Simulation Objects**. On the other hand, the **Event Streams** are **communicating** with it. There is a special **Control Stream** which is used to supervise the behaviour of the simulator itself.

Figure 5: Part of context diagram

Following the paradigm of object-oriented design, inside each category inheritance was applied as structuring principle when possible. For example, the type control stream was defined as a specialization of the type event stream.

The document was supplemented with a list of all data types used in the signal declarations. Since in the very abstract concept model no concepts or objects within the simulator were identified, no type diagrams for such objects were produced.

As an example the type diagram of the simulation objects is depicted in figure 6. Visible_variables[6] is the list of variables to be simulated by an object, whereas Internal_variables are auxiliary ones which can be used in the simulation, but are not visible out-

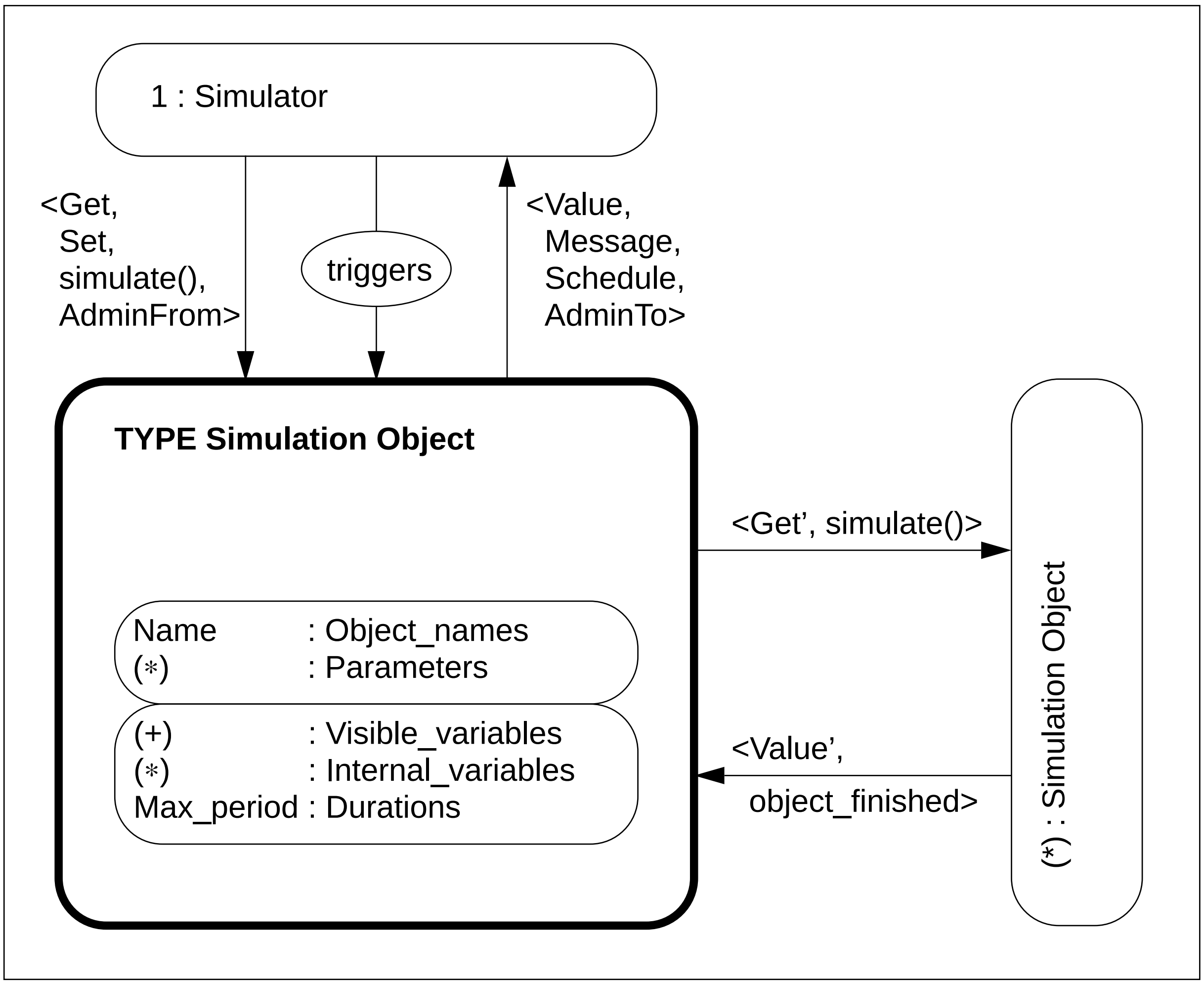


Figure 6: Type diagram of simulation objects

side of the simulation objects. All these variables are subject to double-buffering as introduced in the problem statement. There are two specializations of Visible_variables according to the distinction between dependent and independent quantities, see figure 4. A visible variable corresponding to a dependent quantity can be updated by simulation only, whereas one corresponding to an independent quantity can be updated only by setting it explicitly. Thus there can be no conflicts between these two kinds of updates for one variable, and it is actually sufficient to consider input collisions as stated in the problem statement, see figure 3. Max_period denotes the maximal interval between two points in time for the simulation object to update its variables by numerical simulation. The value of Max_period must be chosen such that numerical stability is guaranteed. Note that the simulation objects are responsible to guarantee this, not the simulator. To achieve this, they may change the value of Max_period during one simulation run by notifying the simulator.

The signals to be exchanged were grouped into signallists according to the kind of information they carried, e.g. administrative or simulation relevant. The signallists were given in the document together with explaining text. As an example, the document part related to the Get signallist of the interface between a simulation_object and the simulator is shown in figure 7.

The signallists were accompanied with a lot of explanations in natural language which diminishes precision of the document. Looking more closely at the natural language parts, it can

6. This explanation of attributes was adapted from a similar one in the static interface description document to be more comprehensible in this paper.

The **signals between the simulator and the simulation objects** are arranged in several groups. The Get group provides read access to the simulation relevant of the object's attributes. The signals dump_values and dump_anything stimulate the object to return all the values of parameters and visible variables and in the latter case even of internal variables. As mentioned above, dump_anything asks for the values not at the current, but at the very next simulation time.

```
<Get> = [get_visible_variable(Visible_variable_IDs, Timestamps),
         dump_values(Timestamps),
         dump_anything(Timestamps)]
```

Figure 7: Part of static interface: Get signallist

The signal simulate triggers the object to perform a single simulation step. According to the course of its calculations, the object returns an answer from the Message group. A signal object_finished is sent exactly in the successful case. In contrast to warnings, errors disable the continuation of the simulation run.

```
simulate(Timestamps)

<Message> = [object_finished,
             object_warning(Strings),
             object_error(Strings)]
```

Figure 8: Part of static interface: Message signallist

The data types in the lower attribute box have to meet yet another requirement: They are subject to double-buffering. An advisable general solution is to refine the corresponding get and set operations. They are extended with an additional parameter, namely the current simulation time. The latter is an element of Timestamps, the set of all possible simulation times. The double-buffered variable now keeps an elder and a younger value together with a limit time, the last simulation time at which the elder one is valid. We may assume that the query timestamps are monotonously increasing.
For get access, if the query timestamp is less than or equal to the limit timestamp, the elder value is returned. As soon as the query time exceeds that timestamp, the younger value is delivered.
For set access, writing with an earlier query timestamp is left undefined. If the query is equal to the limit time, a write collision occurs: two attempts at the same time to alter the same variable. A warning should be issued, and the second attempt be ignored. Finally, if the query is later, the formerly younger value becomes the elder one, the query time is the new limit time, and the argument of set gives the younger value.

Figure 9: Part of static interface: explanation of double-buffering

be seen that they belong to three categories:

- simple explanations, e.g. what was a signal used for or which information did it carry, see again figure 7,
- information about the time when signals were sent, see figure 8,
- further explanation of concepts, see figure 9.

The parts belonging to the second and third category make the separation between this document and the behaviour description and the concept model, respectively, unclear. Information is spread over various document parts, also information is given in a redundant way. More comments on this problem are given in sections 7.2 and 7.3.

4.7 Dynamic interface description: typical interactions

The dynamic system behaviour at the interfaces was described using message sequence charts. Each chart depicts two or more entities exchanging information by means of signal communication. Some surrounding text explains what the situation is typical for and how it is to be generalized. Figure 10 shows a first example: The simulator sends to a simulation object a request to dump the values of all its visible variables and parameters. These are returned successively. The signal object_finished indicates that all requested values have been sent.

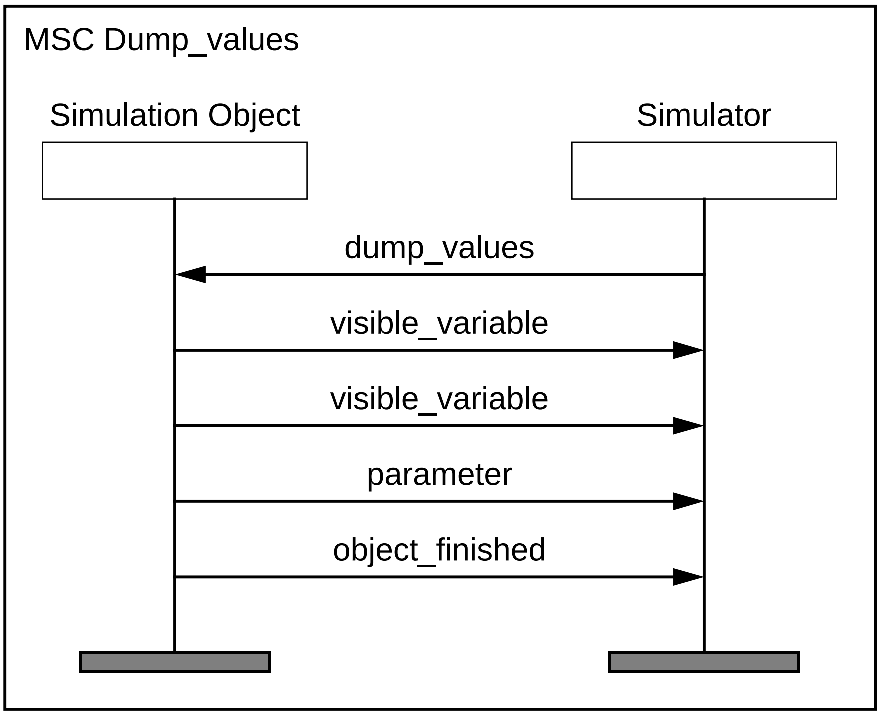


Figure 10: Request–data–acknowledge MSC

This is an example of a non-trivial interaction: A single request is answered with a sequence of data signals, terminated by a positive acknowledge signal. There are three other types of communication:

- One-way communication. An example is given in figure 11: Via the control stream, the simulator receives a request to dump the system state. This dump is written onto the filing system of the simulator and can be considered as an internal action of the simulator. Hence no acknowledge needs to be returned.

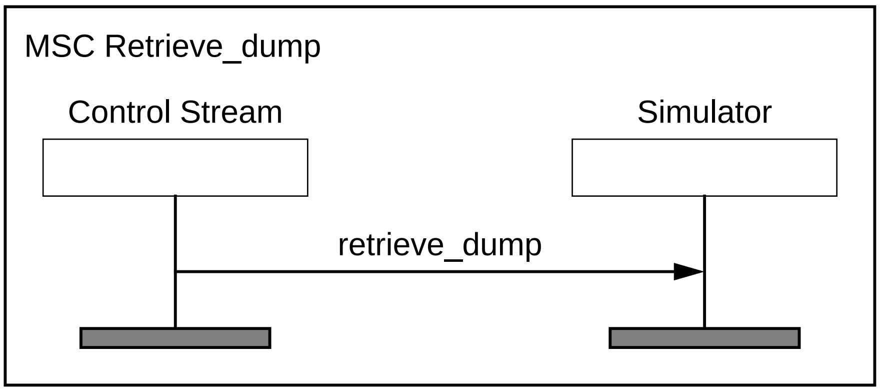


Figure 11: One-way MSC

- Short remote procedure call (RPC) like interactions such as request–acknowledge, request–negative acknowledge, or request–data as shown in figure 12. There an event stream requests the actual simulation time from the simulator, which is returned.

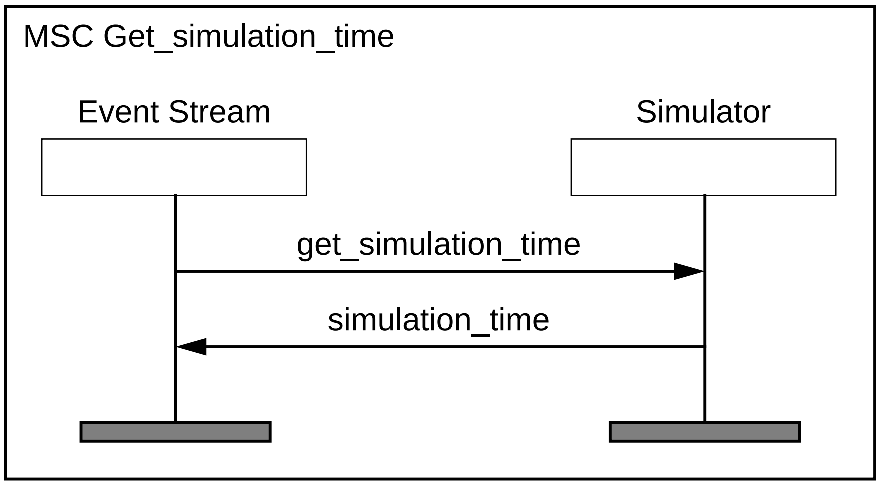


Figure 12: RPC-like MSC

- A notion of discrete time was introduced for MSCs in order to state that signals are sent and delivered at some point in simulation time. E.g. in figure 13, at simulation time t_1, an event stream sends an input or output request for the current simulation time t_1. In this interaction the simulator does not accept this, so at the subsequent simulation time $t_1+\delta$ it returns an io_error signal to the event stream and sends a warning to the control stream as well. The resolution of the discrete simulation time is denoted by δ.

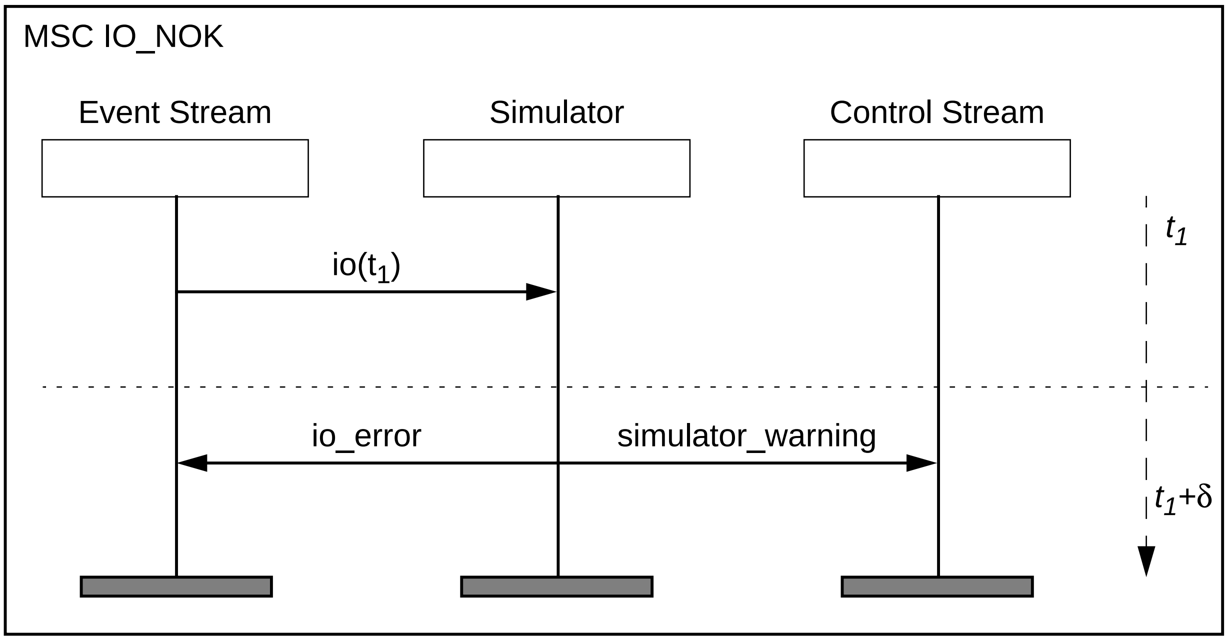


Figure 13: Discretely timed MSC

The dynamic interface description contained 49 MSCs. 23 of these charts were of the one-way type, nine were RPC-like. Therefore, for about two third of the charts, the graphical representation added only few to the understanding of the respective situation; most of the information was contained in the surrounding text.

Similar to the presentation of the static interface behaviour, this document part was composed of the three views on the system as seen from the interacting entities in the system environment. For the development of these views, however, an idea of the behaviour of the complete system was already necessary. This behaviour was explained at the beginning of that document part, see figure 14.

There will be at least five stages in a typical system run:
- First of all, the control stream asks the simulator to initialize.
- Second, the simulator does the same with event streams and simulation objects.
- Third, the user is allowed to modify the initial system state, e.g. change the values of some installation parameters.
- After all the modifications are executed, some of the simulation objects may have got an inconsistent state. Therefore they must be stimulated to re-establish consistency.
- Now, the actual simulation is done, which itself may be interrupted and continued.

Figure 14: Sketch of system behaviour

Due to the decomposition into views, the description of these stages was spread over the corresponding document part.

4.8 Dynamic interface description: behaviour projection

For each of the different views of the system, the typical interaction sequences were composed into a state-oriented description of the expected role behaviour of the system at its respective interface. Transition charts (TCs) consist of optionally named state nodes which are followed by input or output transitions labelled by signals. Outputs are marked with an exclamation mark, inputs with a question mark. Figure 15 shows the interactions between the simulator and the simulation objects in the initialization phase as seen from the simulator.

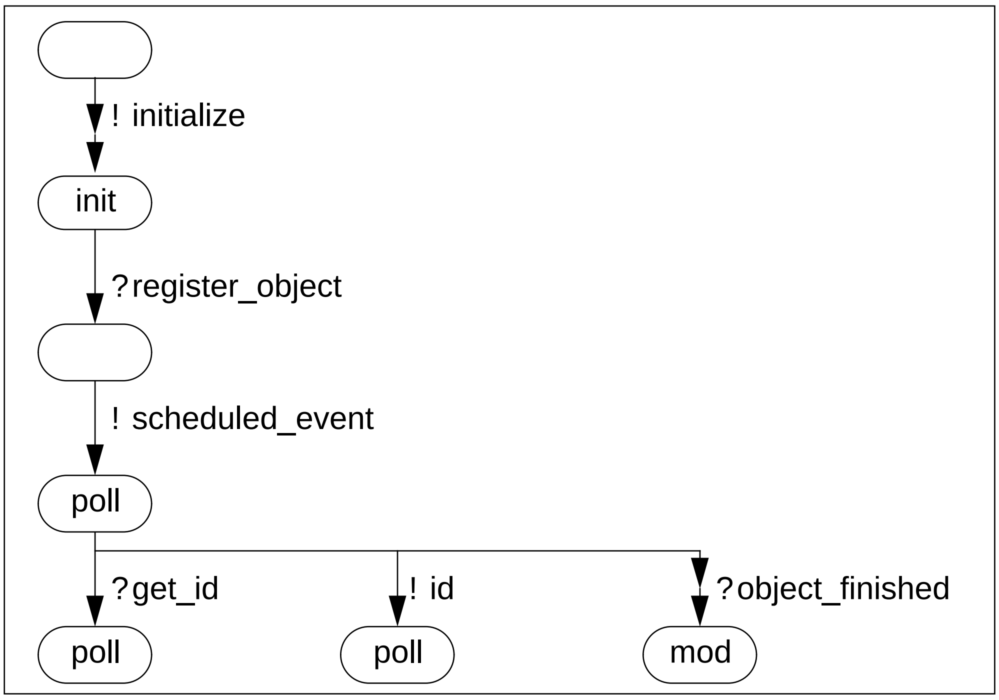


Figure 15: TC for initializing simulation objects

The notion of TCs as described in [BH93] was found to be insufficient to express the synchronization constraints and was extended ad hoc by ‘fork and join’ constructs, see figure 16.

The textual descriptions in that document part repeated only very general information. Again, in each of the three views, some or even all of the five run-time stages described in the previous subsection (figure 14) appeared.

Although no new information was produced, the composition of single interactions into

“To derive a more complete definition of interface behaviour from a set of sequence charts one must focus on one object at a time and the interactions it is involved in.” (Bræk, Haugen: Engineering Real Time Systems, p. 87) This is all what transition charts are able to express and restricts their application to the interactions between the simulator and a single simulation object. However, we wish to distinguish those transitions that will reasonably be done for all objects.
An input-triggered transition will be marked with two subsequent arrowheads such as depicted on the right if the reached state is left not before that specific input signal arrives from every simulation object. An output-triggered transition is marked that way if the simulator sends the output signal to all the objects simultaneously.

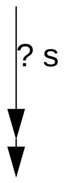


Figure 16: Extension of transition charts

state-oriented descriptions was useful in checking the previous document parts for completeness. In fact a few omissions were found and eliminated. Furthermore, during the functional design, the transition charts served as framework for the processes to be modelled.

4.9 Owner requirements

The owner requirements were rather similar to those addressed to `team1`. Besides the design constraints already mentioned, there were some non-functional requirements, see figure 17. These requirements are important for the implementation design, which has not been carried out in this study.

The non-functional requirements include the following aspects:
- The design has to be object-oriented.
- The accuracy should not be higher than necessary to test control algorithms and provide reasonable responses for the event stream of the building.
- Dumping the system should influence the simulation as less as possible. That is, it should cost as less real time as possible.
- The simulator should run in at least real time.
- Simulation time has to be as close to real time as possible. This is especially important for outputs.

Figure 17: Non-functional owner requirements

4.10 System structure

The last section of the requirements specification document contains a SOON diagram (see figure 18) where the system is depicted in interaction with its environment.[7] Inside of it, there is a corresponding part for each type outside. No additional requirements concerning parts of the system were expressed. Neither was any of the system functionality located in any substructure.

7. Remember that the control stream is a special instance of the type event stream. As such it has at least the same communication relation with the simulator as every other event stream, i.e. the relationship communicates with the interface. In addition to an ordinary event stream, it is related to the controller. Hence the relationship controls shown in figure 5 was refined into the relationships communicates and controls.

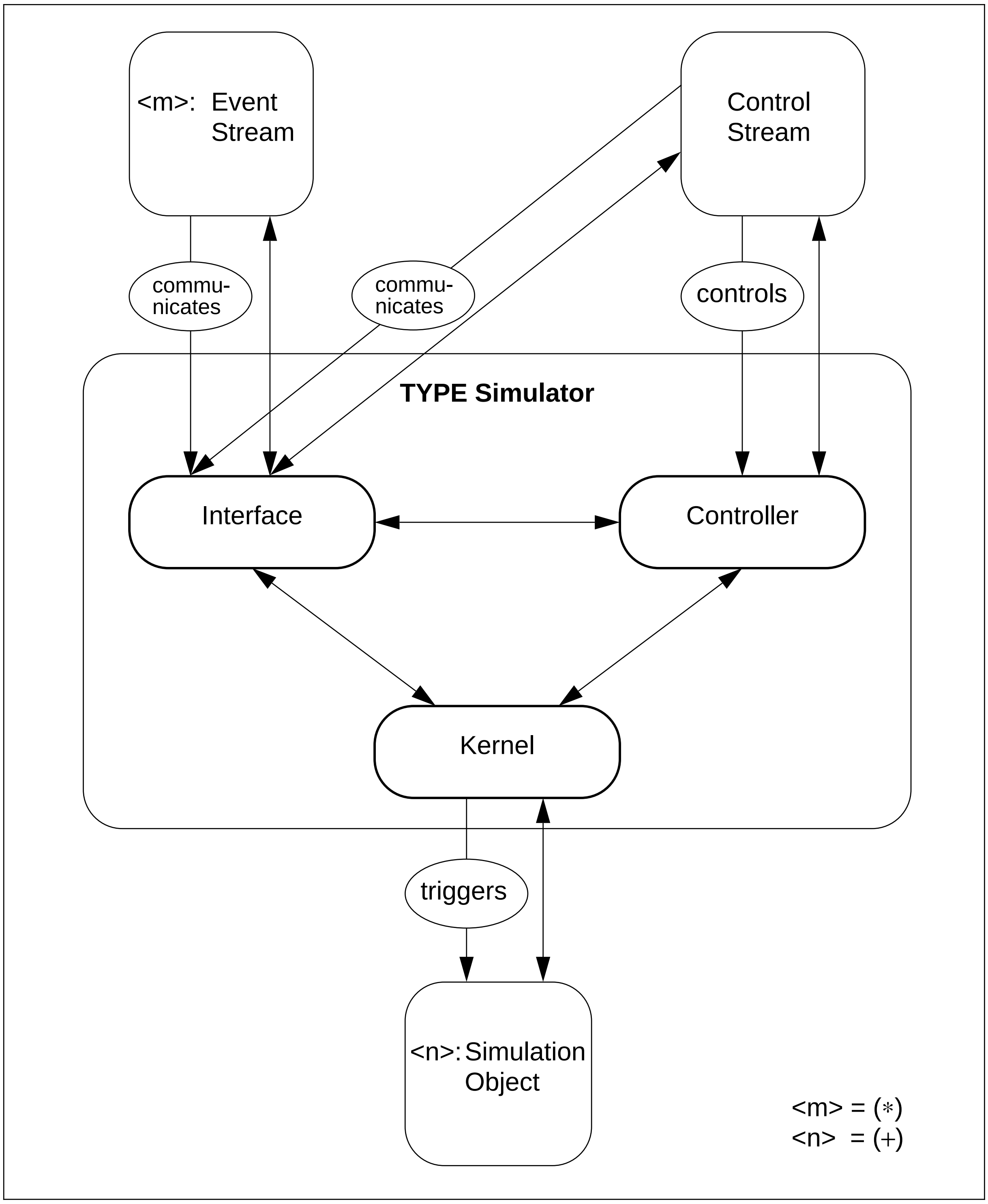


Figure 18: System structure

5 Functional design

The functional design was expressed via the formal specification language SDL [C93a]. A system consists of concurrent processes — extended finite state machines — which are communicating asynchronously. Processes are aggregated into blocks, which may be aggregated into blocks again. The SDT tool of Telelogic AB was used for drawing the diagrams and establishing syntactical correctness (via its internal syntax analyser).

As part of their systems engineering methodology, Bræk and Haugen [BH93] recommend to proceed step by step to make the functional design (figure 2). They suggest the design process to advance ‘top-down’, starting from the system level, descending to the block level and ending up with the process level. In the following sections, typical examples for each of these levels will be given. All in all, the SDL document had about 90 pages.

5.1 System level

The system’s top level consists of four blocks (figure 19). The blocks Interface, Controller and Kernel each interact with different entities in the system environment, namely event streams, the control stream, and simulation objects, respectively. The system has to correspond to different behaviour roles as expected by its environment.

The block Interface also encapsulates a dictionary mapping external names of surrounding entities to internal references. The block Kernel maintains the simulation time and performs the activation of the simulation objects. The block Controller transforms commands from the control stream into their internal representation. A fourth block, of type FileSystems, represents the internal file system of the simulator. Its operations are provided as remote procedures. Therefore none of the other blocks needs an explicit channel to FileSystem. Its type is imported from a library, as well as the definitions of signals and data types. In the library GlobalDataTypes, which is seven pages large, the data types which have only been described informally in the requirements specification (section 4.6) are defined axiomatically or derived via predefined constructs.

5.2 Process communication level

The next design step was to split each of the main blocks into communicating processes. (Blocks may be decomposed into blocks as well, but this was not necessary here.) For the blocks Interface and Controller, the result was trivial: Only one process was located in each of them.

The block Kernel was split into four processes and is shown in figure 20. The process Clock encapsulates the simulation time and is responsible for the synchronization the processes of the block Kernel. The process EventCollection maintains the set of scheduled events, whereas for every point in simulation time the process EventHandling fetches the current events and triggers the corresponding simulation objects to perform the simulation. The process PassThru handles data output of the simulation objects, especially when dumps have to be written.

A typical simulation cycle now works as follows: The process Clock sends the new simulation time to the process EventHandling. If there is a fixed binding between real time and simulation time, it also sets an internal timer for the lower bound when the cycle has to be finished and another one for the upper bound. The process EventHandling then requests the current events from the process EventCollection, thereby propagating the current simulation time. After receiving these events the process EventHandling invokes the corresponding simulation objects. As soon as each of them has signalled the end of its calculations, the process EventHandling sends a signal advance_simulation_time to the process Clock. If this signal arrives too late, the second timer has already expired, and the simulation is out of time. Otherwise, the process Clock may initiate the writing of dumps (which the processes EventCollection and PassThru are also

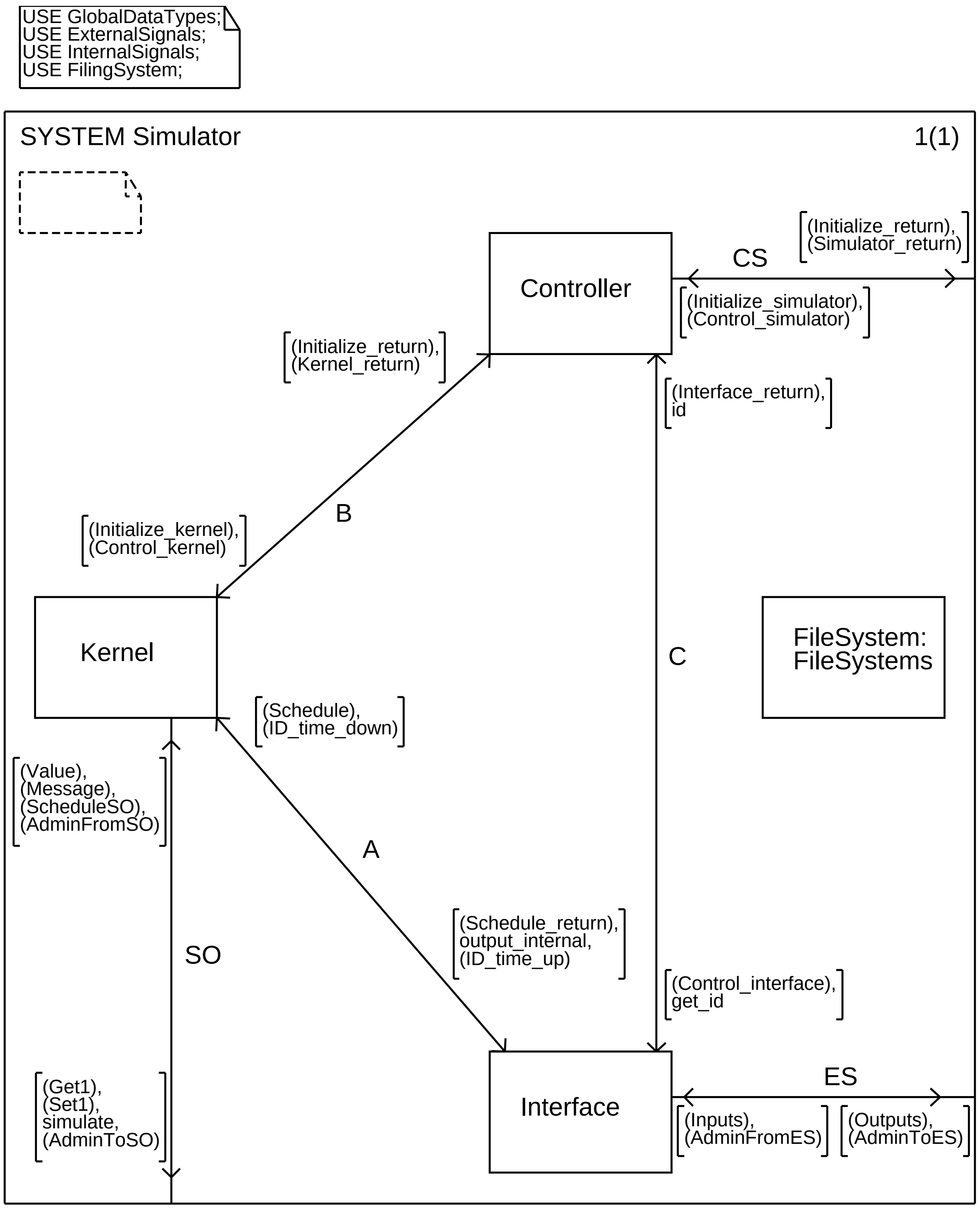

USE GlobalDataTypes;
USE ExternalSignals;
USE InternalSignals;
USE FilingSystem;
SYSTEM Simulator
1(1)
Controller
CS
(Initialize_return), (Simulator_return)
(Initialize_simulator), (Control_simulator)
(Initialize_return), (Kernel_return)
(Interface_return), id
B
(Initialize_kernel), (Control_kernel)
Kernel
C
FileSystem: FileSystems
(Schedule), (ID_time_down)
(Value), (Message), (ScheduleSO), (AdminFromSO)
A
(Schedule_return), output_internal, (ID_time_up)
SO
(Control_interface), get_id
ES
(Get1), (Set1), simulate, (AdminToSO)
Interface
(Inputs), (AdminFromES)
(Outputs), (AdminToES)

Figure 19: SDL system block of Simulator

involved in), or in case the simulation shall be stopped just waits until resuming. Not before the first timer has expired, the process Clock calculates the next simulation time, and then the following cycle starts. During the whole cycle, events may be sent asynchronously to the process EventCollection, where they are only accepted if they have a timestamp in the future with respect to the current simulation time.

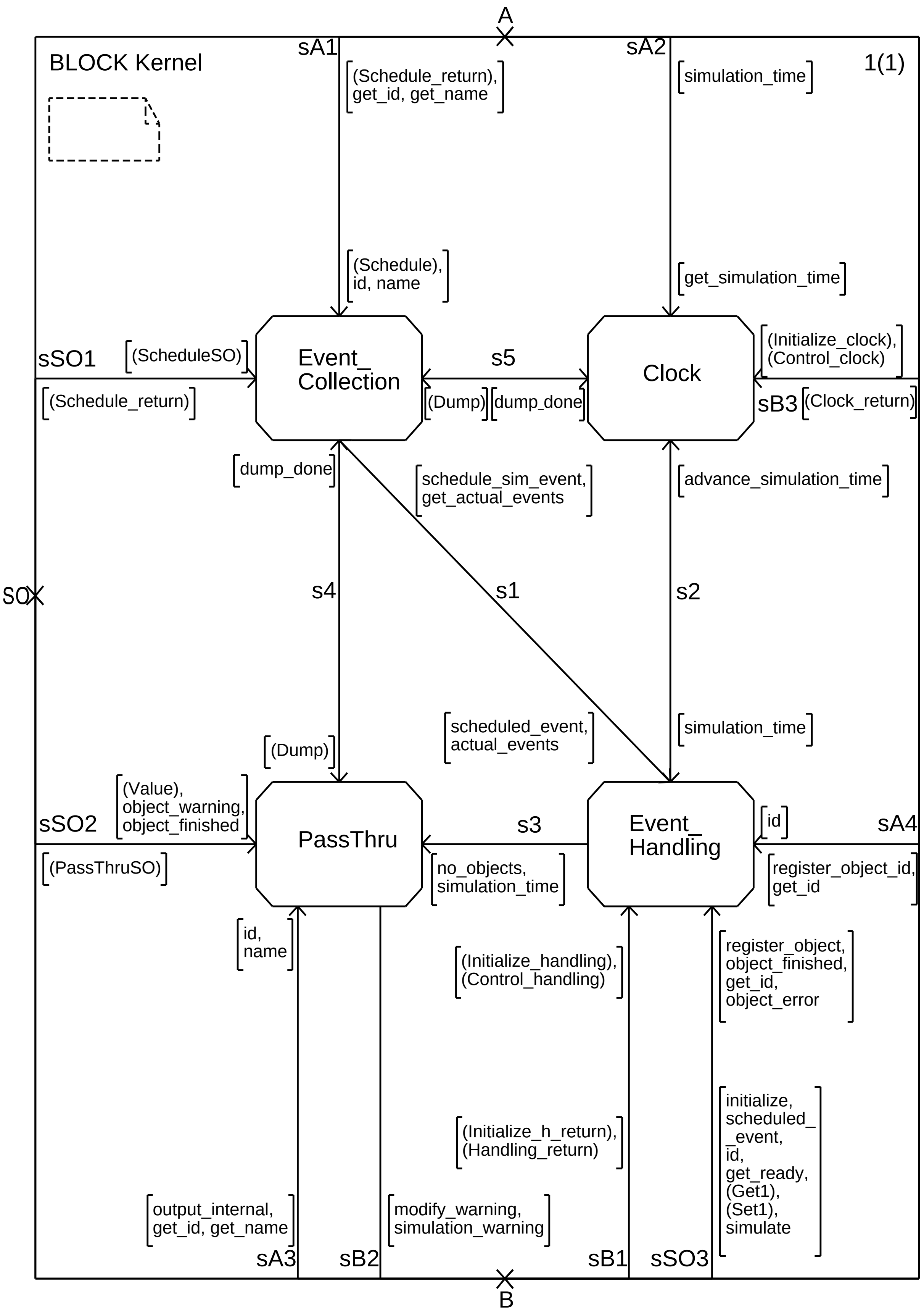

A
BLOCK Kernel
1(1)
sA1
(Schedule_return), get_id, get_name
sA2
simulation_time
(Schedule), id, name
get_simulation_time
sSO1
(ScheduleSO)
(Schedule_return)
Event_ Collection
s5
Clock
(Initialize_clock), (Control_clock)
(Dump)
dump_done
sB3
(Clock_return)
dump_done
schedule_sim_event, get_actual_events
advance_simulation_time
SO
s4
s1
s2
scheduled_event, actual_events
simulation_time
(Dump)
(Value), object_warning, object_finished
sSO2
PassThru
s3
Event_ Handling
id
sA4
(PassThruSO)
no_objects, simulation_time
register_object_id, get_id
id, name
(Initialize_handling), (Control_handling)
register_object, object_finished, get_id, object_error
initialize, scheduled_ _event, id, get_ready, (Get1), (Set1), simulate
(Initialize_h_return), (Handling_return)
output_internal, get_id, get_name
modify_warning, simulation_warning
sA3
sB2
sB1
sSO3
B

Figure 20: Block Kernel

5.3 Process level

Figure 21 shows a part of the process EventHandling, with states depicted as boxes. State transitions are triggered by inputs and invoke outputs. The part shown here describes the invocation of simulation objects during a single simulation cycle as described above: The process EventHandling receives the new simulation time, requests, waits for and receives the current events, sends an appropriate signal to each concerned simulation object, and waits for each of the activated objects to finish its calculations. Unless an error occurs, the process Clock is informed to start the next cycle.

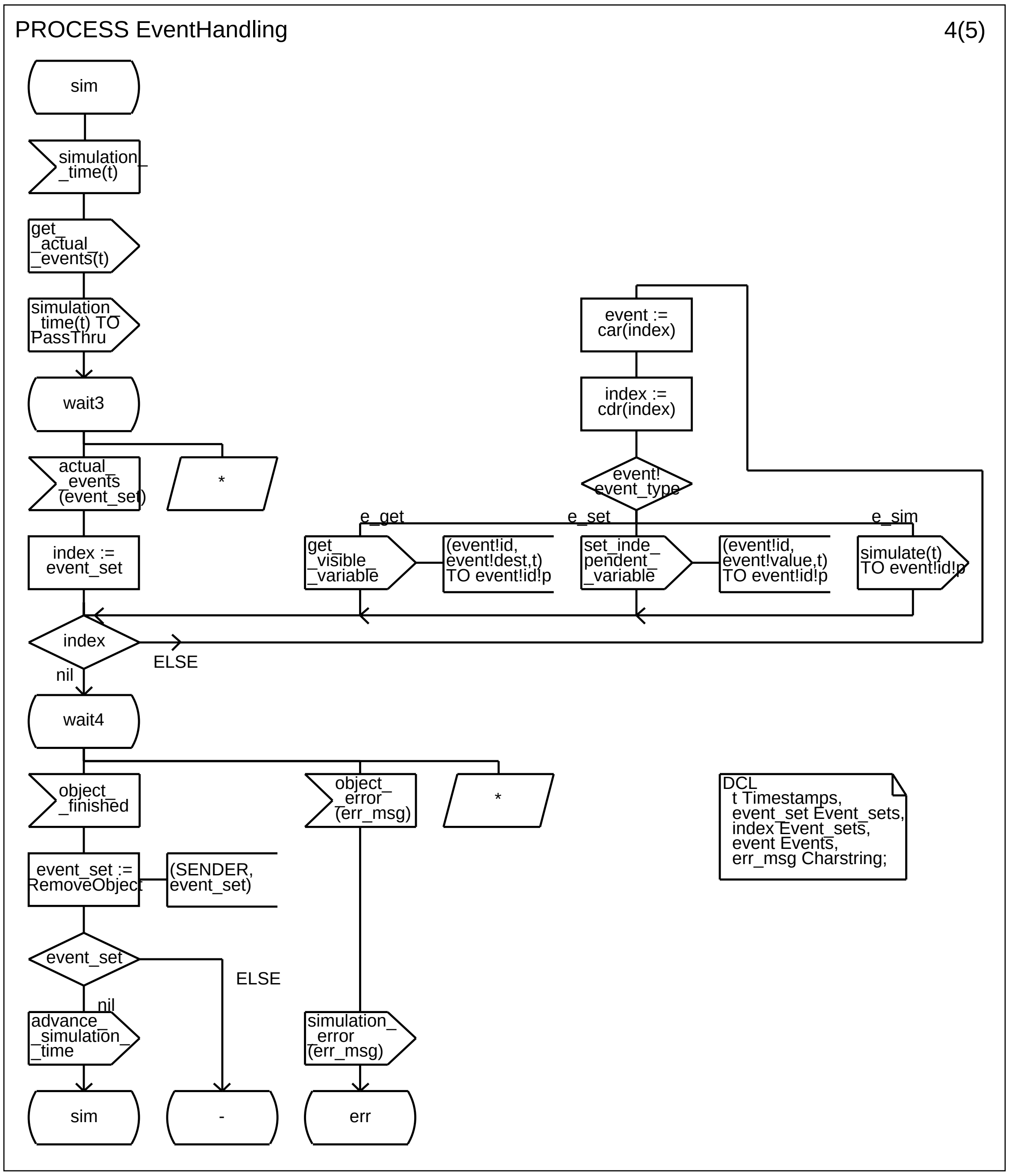


Figure 21: Part of process EventHandling

6 Development process

6.1 Allocation of tasks

The documents have been produced by some kind of role playing by the authors. The second author played the role of an engineer capturing the requirements down to the level of functional design. The first author played several roles, i.e. those of the customer and of the technical experts on the problem domain and on the methodology. In general, allocating peoples to roles was experienced to be helpful to get a more realistic touch on the study. But, whenever possible, one person should not play more than one role to keep the roles distinct. Especially the role of a technical or domain expert should be played by another person than that playing the customer or the developer. Otherwise, wishes of the customer are too easily proposed in the form of already completed solutions.

6.2 Process model

A waterfall model was applied in producing the following documents in subsequent stages:

- problem statement of the developer,
- concept model and context diagram,
- type diagrams and behaviour description,
- projection of behaviour to individual roles,
- functional design.

It was relatively easy to use this process model. This can be contributed to the experience on the problem domain and the availability of the descriptions produced by `team1`. In producing the first version of the simulator by `team1`, a way had been paved, which could be followed without large deviation. Nevertheless all documents except the data dictionary were produced from scratch throughout this case study.

Few information had to be fed back from later to earlier stages, and few rework on this early stages was necessary. The static and dynamic interface description were developed hand in hand because there was a strong feedback from the dynamic to the static one. Therefore these documents have been put in one stage in the list above. A much smaller amount of information was fed back from the behaviour projections to the static and dynamic behaviour descriptions. This information consisted mainly of a few missing signals, which were easily added. Also, developing the functional design was found to be a highly iterative process.

Although it was possible to use the waterfall model in this study, the authors do not expect that this will be possible in general. Note that this model was applied on the top level phases, but not within them.

7 Lessons learned

7.1 Problem statement

The problem statement served well as a first reflection of the developer on the problem posed by the customer. The problem statement can be reviewed by the customer, and misunderstandings concerning the overall goal can be detected and resolved. Therefore, producing this document can be seen as a first step in achieving an agreement on the problem between customer and developer in the sense of [Poh96]. This agreement must be reflected in at least one of these documents such that it can serve as the starting point of the further development.

7.2 Domain analysis

At the beginning of this study it was not clear to which extent concepts of the problem domain have to be included explicitly in the concept model or whether it is sufficient to refer to e.g. textbooks on the topic. In this case study the authors followed the second approach. But as a consequence, it was necessary to add further explanations to other documents when using unusual concepts or when stressing certain aspects of standard ones. These explanations were given in natural language, and as a consequence, the documents became less precise and more ambiguous. Also, these explanations distract from the actual contents of the documents. Therefore the authors argue in favour of presenting even standard concepts in the concept model. Thereby, knowledge about the application domain is made available in a form which is suitable for expressing the requirements of the system to be developed.

As a minor point it should be noted that it seems to useful to follow the suggestion of [Ree96] and to use a more widespread notation such as e.g. OMT [RM+] instead of SOON.

7.3 Requirements specification — interface descriptions

Besides the already mentioned problems related to distracting explanations, another dissatisfying point is that redundant information is kept in these interface descriptions, namely when and for what reason signals had to be sent. This information was included both in the signallists of the static interface description as well as in the dynamic interface description. Finally this information was summarized in the behaviour projections. Clearly, it had to be presented in the dynamic interface description, but it was also presented in the static interface description to motivate the existence of the signals. The information itself was useful in both documents, hence it cannot simply be removed from one of them.

Inspecting the signals related to one object one can see that they are used for different purposes. As the documents are structured now, all signals of one object are described together in the interface descriptions, even if they are used for different purposes, e.g. for controlling the simulation and for performing it. In hindsight it would have been a better choice to use another structure of the documents, namely to describe the static and the dynamic interfaces for each of these different protocols together. This possibility can only be indicated here. The documents have not been adapted to this structure because this was recognized and analysed only after finishing all documents. As a consequence this structure was not evaluated, but it follows the proposal made in [BH93] to describe interfaces separated by layers — if there exist several ones — and should gain therefore from a similar separation of concerns. A possible disadvantage of this document structure could be that information related to one interface would be distributed, which could make it more difficult to relate different protocols at one interface and to detect interferences between them.

As already mentioned, a lot of the message sequence charts describe simple behaviours. For these behaviours the graphical notation was found to be of no help. It could have been a better choice to use the so-called prose representation of message sequence charts for these simple charts. For the other, more complex behaviours, the graphical representation was very helpful.

7.4 Requirements specification — behaviour projection

The behaviour projections expressed by the transition charts have been found to be very useful. Few information had to flow back from the functional design to the specification and analysis stages, which indicates that the behaviour projections are an important intermediate step. In the transition charts, the communications of each object had been put in some useful order, inducing a certain structure of the SDL processes, describing these objects completely. This was even the case for those objects described by several SDL processes as e.g. the block Kernel. The structure indicated by the transition charts was repeated in each of these processes.

The authors found no clear indication whether it would be better to use SDL in an informal way instead of transition charts. On one hand it would be avoided to learn another, although simple language. Also, having a sketch of a process in SDL it should be easy to extend it to a complete one. On the other hand it could be difficult to use SDL in an informal way because it has a formal semantics. One could also be tempted to already put parts belonging to the functional design into these behaviour projections. But then the usefulness of this intermediate step would be diminuished.

Note also that the syntax of the transition charts has been enlarged by fork and join constructs which express some kind of synchronization constraint between objects. It has to be elaborated further whether there are other kinds of such constraints and whether they should be described in this particular document.

7.5 Functional design

For the development of the functional design, the documents of the requirements specification served as a framework which directed the work to be done. Their production had provided sufficient understanding of the problem domain, which paid off well: No additional conceptional problems occurred during that stage. The development of the functional design itself, however, has its own inherent difficulties, which seem to be inherited from the paradigm of asynchronous concurrency. In the following, the observations on the different abstraction levels (system — block — process) will be discussed.

The system structure was already anticipated as the last part of the requirements specification (see section 4.10), but of course without any signal flow. Much of the system's complexity arose from this very signal flow and data types. The information to be exchanged between the simulator and its environment was easily distributed onto the system's main blocks. The signals to be send among the main blocks themselves, however, depended on the needs of the processes deeply inside of them. In order to establish a new signal link between two processes, the signallists in all the diagrams from the sending process via its enclosing block up to the system block and downwards vice-versa to the receiving process had to be extended. This multiple notification was rather inconvenient and the resulting information on the system level often of few interest. For a thorough understanding of the system beyond a first glance, the abstraction level of the system's main blocks has to be broken up.

The block level — especially for the block Kernel — was the most complex part of the SDL document. The realization of the required functionality had to be partitioned into tightly coupled processes. Thereby completeness with respect to the required functionality and correct synchronization of the processes had to be ensured. Naturally, the split had to be done before deep insight into the arising processes had been gained; and some of the functionality had to be shifted from one process to another, also leading to changes elsewhere in the document. The resulting partitioning seems to be quite natural for the given purposes — further partitioning would not have eased the description, whereas reducing the number of processes would have violated the idea of data encapsulation. Together with the lots of administration signals necessary for the different stages in the system run, the difficulties were caused by the need for explicit synchronization and by the requirement of the simulator writing complete system dumps serving to restart an identical run.

Once a required functionality was assigned to a process, the coding of states and transitions was straightforward, often profiting from transition charts in the requirements specification. The transition charts also helped in tracing individual requirements to the SDL processes. Tracing the requirements was not straightforward when splitting a block into several processes. To make this more explicit, traceability should be supported by providing explicit models.

This use of the transition charts is related to step 2 — *Mirror the environment behaviour* — of the procedure outlined in figure 2. Note that not only the roles themselves, but also the sketch of their behaviour was reflected here and used as a basis to describe complete SDL-processes. With respect to step 4 — *Analyse the behaviour* — it should be noted that there are further reasons to restructure blocks, e.g. the necessity of keeping processes in synchrony. Steps 6 and 7, which are related to identifying types and checking their variability, were not carried out. In contrast to the simulation objects, the processes of the kernel are very dissimilar. Step 8 — *Iterate until satisfaction* — was found to be a very important one. As already pointed out, it was not possible to create a satisfactory design in one pass through these steps. It seems to be even an inherent property of this procedure: E.g. when mirroring the environment behaviour, in one pass the roles can be mirrored, and the resulting structure should be analysed. In a second pass the sketch of the processes can be developed from the behaviour projections and should be analysed, too. Thus, information is added to the design in small parts.

At this point it should be noted that even a commercially sold and industrially used tool is not free of bugs, which makes it necessary to work around them. E.g. although the SDT tool provided the SDL macro construct, it did not allow to use it; as a consequence there are some communication patterns that had to be repeated over and over throughout the SDL document. To avoid an excessive number of pages for some of the process descriptions, single document pages had been filled more densely. But then, the graphical editor sometimes needed a notable amount of time to come up with editing changes. Another annoying problem with the tool was that a syntactically correct graphical presentation of the block Kernel was detected by the syntax analyser as erroneous. After working around this bug, it was possible to check the complete functional design for consistency, which was very valuable to detect minor errors, such as e.g. typographical ones.

7.6 Summary

As a qualitative evaluation, the methodology has been found usable for the development of reactive systems. It has been possible to proceed from the problem statement of the customer to a functional design. As can be seen from the discussion above, there have been problems in producing some of the documents which are part of the requirements specification. Possible ways to avoid these problems have been indicated. These proposed improvements can all be done within the methodology as presented in [BH93], there is no need for essential changes of the methodology.

The authors of this study consider it essential to make a detailed concept model instead of referring to literature, even if the problem to be solved is within a well-established field. Otherwise it is not possible to use these concepts properly.

To avoid redundancy it seems to be useful to describe static and dynamic interfaces together for each protocol and each layer of it.

The behaviour projections are considered a useful intermediate step between the requirements given by typical interaction sequences and the functional design. They are used in several of the steps of the procedure shown in figure 2. This procedure itself was found to be useful, too.

7.7 Validity of results

Although thorough reviews have been done, it is possible that problems or omissions in the requirements specification and the functional design are revealed when implementing the simulator. It is planned to extend the case study in this direction, some more information is given in section 8.3 below.

It should also be noted that the authors do not — even after performing this and other case studies in SDL — consider themselves as experts with regard to the languages and the methodology used. The documents have been produced to the best of the knowledge of the authors, but there will always be place for further improvements. Therefore the lessons learned which are presented above will have to be supported by further case studies.

8 Related and future work

8.1 Related work — methodology

Shortly after the requirements specification was finished, the article [Ree96] was published, which therefore could not be considered in this study. The article summarizes the new recommendation on the use of SDL, MSC, and other languages. This new recommendation is based on the use of SDL in several large projects, including those leading to the books [BH93] and [OF+94].

The article is more detailed concerning the early phases of system development as the method used here. As such it would have been helpful in this study. It stresses even more the paradigm of working design-oriented. E.g. it is suggested to decompose the system within the context diagram until it "cannot *be sensibly further divided or it is obvious the block corresponds to a single SDL process*" ([Ree96], page 1695). This means that in comparison with this study, the partitioning of objects into SDL processes is done even earlier.

As a second point it is emphasized that the development processes are done largely in parallel and that they are continuous. All processes are considered to finish usually at about the same time — delivery of the product — but the effort is spent differently between the processes. E.g. most of the *analysis* work will be done before the *draft design*.

The conclusions drawn from this study are in correspondence with [Ree96] which is based on more experience and is more comprehensive. Nevertheless, this study is not subsumed by [Ree96]. Since the methodologies presented in [BH93] and [Ree96] are closely related, the study described here can be used to point out and give a concrete example for this methodology.

8.2 Related Work — simulation

By one of the members of `team1` the idea of using simulation objects and event-driven simulation has been pursued further. Because some of the requirements have changed, the functional design of `team1` could not be used there. A first version of a simulator has been implemented in Smalltalk and is running, see [RSZ96]. Thereby it has been shown that the design constraints can be realized. Due to the similarity between the implemented simulator and the one specified in this study, this indicates that no large obstacles will turn up when transforming the functional design developed in this study into an implementation.

8.3 Future work

It is planned to implement the simulator, such that the running system can be validated against the requirements. In the implementation of a similar simulator already mentioned above the numerical part of the simulation does not need too much runtime. Therefore the simulator fits nicely in the large class of systems composed of small, communicating processes. The language ERLANG[8] is proposed in [AV+96] to be suitable for this kind of systems. Furthermore, it is shown in [Frö93] how ERLANG programs can be derived from SDL descriptions. Therefore the implementation will be based on this language.

Another interesting question to be investigated is whether the methodology used here or the new recommendation on the use of SDL can be adapted to other languages. As an example,

8. ERLANG is a trademark of Ericsson Software Technology AB.

statecharts [Har87] could be used instead of SDL. Both languages are based on communicating, extended finite state machines, but they use different communication paradigms. If this adaption can be done successful, it should be possible to apply the methodology to a larger class than communication systems for which it was developed originally.

Acknowledgements

This study would not have been possible without the support of the Deutsche Forschungsgemeinschaft, Sonderforschungsbereich 501, and without the members of `team1`, whose help is greatly acknowledged: B. Dellen, B. Geppert, E. Kamsties, Prof. Dr. K. Madlener, M. Schütze, Dr. P. Sturm, D. Zeckzer, and Prof. Dr. G. Zimmermann. Helpful comments have been given by M. Kronburg.

References

[AV+96] J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in ERLANG*. 2nd edition, Prentice Hall, 1996.

[Bas96] V. R. Basili, The role of experimentation in software engineering: Past, current, and future. In *Proc. of the 18th Intl. Conf. on Software Engineering*, pp. 442-449. IEEE Computer Society Press, 1996.

[BH93] R. Bræk and Ø. Haugen. *Engineering Real-Time Systems, an object-oriented methodology using SDL*. Prentice Hall, 1993.

[C93a] CCITT. Recommendation Z.100, Specification and Description Language SDL. 1993.

[C93b] CCITT. Recommendation Z.120, Message Sequence Charts. 1993.

[Dei94] T. Deiß. An outsiders evaluation of PAISLey. Internal Report 250/94, Fachbereich Informatik, Universität Kaiserslautern. 1994.

[Eva88] J. B. Evans. *Structures of Discrete Event Simulation, an introduction to the engagement strategy.* Ellis Horwood, 1988.

[Frö93] M. W. Fröberg. Automatic code generation from SDL to a declarative programming language. In A. Sarma and O. Faergemand, editors, *Proc. of the 6th SDL-Forum*, 1993.

[Har87] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231-274, 1987.

[Meh94] H. Mehl. *Methoden verteilter Simulation*. Vieweg, 1994. in german.

[OF+94] A. Olsen, O. Færgemand, B. Moller-Pedersen, R. Reed, and J. R. W. Smith. *Systems Engineering Using SDL-92*. North-Holland, 1994.

[Poh96] K. Pohl. Requirements Engineering: An overview. Aachener Informatik-Berichte 96-05, RWTH Aachen, Fachbereich Informatik, 1996.

[Ree96] R. Reed. Methodology for real time systems. *Computer Networks and ISDN Systems*, 28:1685-1701, 1996.

[RSZ96] J. P. Riegel, M. Schütze, and G. Zimmermann. Objektorientierte Modellierung einer Simulationsumgebung mit Patterns, SFB 501 Bericht 09/1996, Sonderforschungsbereich 501, Fachbereich Informatik, Universität Kaiserslautern, 1996. In german.

[RM+] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, 1991.