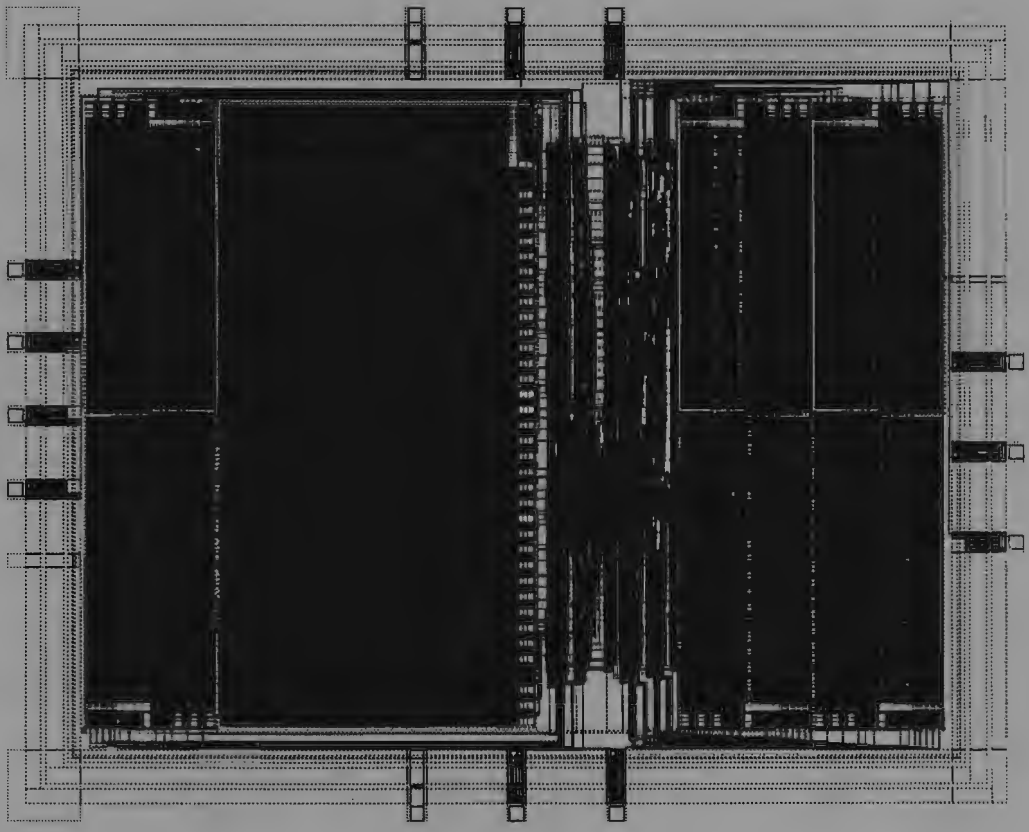
An object-oriented modelling technique for analysis and design of complex (real-time) systems

An Object-Oriented Modelling Technique for Analysis and Design of Complex (Real-Time) Systems



A.C. Verschueren

An Object-Oriented Modelling Technique for Analysis and Design of Complex (Real-Time) Systems

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de Technische Universiteit Eindhoven, op gezag van de Rector Magnificus, prof. dr. J.H. van Lint, voor een commissie aangewezen door het College van Dekanen in het openbaar te verdedigen op

dinsdag 19 mei 1992 om 16.00 uur

door

ADRIANUS CORNELIS VERSCHUEREN

Geboren te Goirle (NB)

Dit proefschrift is goedgekeurd door de promotoren

prof.ir. M.P.J. Stevens

en

prof.dr.ir. C.J. Koomen

CIP-Gegevens Koninklijke Bibliotheek, Den Haag

Verschueren, Adrianus Cornelis

An object-oriented modelling technique for analysis and design of complex (real-time) systems / Adrianus Cornelis Verschueren. - [S.l. : s.n.]. - Fig.
Proefschrift Eindhoven. - Met lit. opg. - Met samenvatting in het Nederlands.
ISBN 90-9005046-9
NUGI 832
Trefw.: object-georiënteerd modelleren / computer-aided systems engineering.

Acknowledgements

This Ph.D thesis is the result of four and a half years hard labor, which could not be performed without the invaluable help of numerous colleagues and students.

Professor Stevens, you thought I was the 'right stuff' and asked me to stay even before I graduated. You had enough trust in me to let me work for more than three months when I said I had come up with a good idea which needed some research. Your numerous advice, support and (sometimes heated) discussions helped shape the model and design path described in this thesis.

Frank Budzelaar, you were there when I needed a 'sounding board' to check my ideas.

Yun Chao Hu, Raymond Hulzebos, Henk van Bezooijen, Robert Huis in 't Veld, Willem Rovers, Ivo van Zandvoort, Erik Baars - thank you all for your contributions in shaping the design path and/or creating the tools.

I thank everyone who used the tools and provided important feed-back.

My thanks go out to everyone in the entire Digital Information Systems group for their interest, support and help (and some football matches).

I thank the staff of Sagantec Europe B.V. (Eindhoven, Netherlands), IBM Zurich research laboratory (Switzerland) and Philips P-ASIC (Hilversum, Netherlands) for their support and help.

Thank you, Gerrie. You helped me get through the last year and kept my feet on the ground.

Table of contents

List of figures

To Gerrie

1. Introduction

Our modern society depends upon the processing, storage and transfer of large amounts of data. Digital data- and signal processing systems have invaded our life.

Designing these systems is a task of ever-increasing complexity. Because of time-to-market restrictions, the design time should be as short as possible. To avoid accidents and accompanying legal problems, the system should not exhibit 'strange' behaviour (just another way to state that it should not contain too many bugs). Customers will only buy the system if initial, operational and maintenance costs are not too high. These often contradicting demands are depicted in figure 1-1.

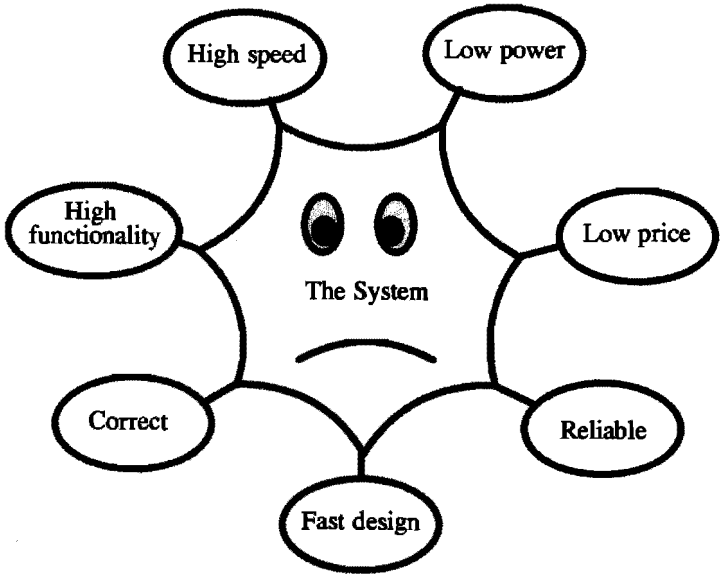


Figure 1-1: Demands posed upon system design

The design process always starts with a set of requirements which must be transformed into the operational system. The requirements are often given in an informal way. This means a fuzzy, incomplete and error-prone start of the design process. Using an operational computer program as 'formal' system specification provides no solution to this problem. A complex program is always derived from an informal specification. While being correct in itself, informal specification errors and derivation errors may cause the program to behave incorrectly under some obscure conditions.

Design processes should pay attention to the fact that the specifications may be incorrect. Working with incorrect specifications will yield systems which may conform to the specification, but do not operate as intended.

The best way to catch specification errors is by simulating the formally specified system. This allows the customer to check whether or not the system operates as intended. On-line interactive simulation allows a customer to check the system behaviour under conditions which he 'forgot to mention' in the informal specifications. Simulation will never *certify* a specification as being correct, it only provides a higher level of confidence. It is up to the customer to state that he is satisfied with the system behaviour *as simulated.* Once this is done, actual system design can start.

Chapter outline:

Chapter 1 states the problems found in general system design, compares some existing approaches and introduces the three-stage design path chosen in this Ph.D thesis.

Chapters 2 and 3 introduce an improved 'Object-Oriented' model (with extensions) which will be used to describe and simulate the system.

Chapters 4 and 5 show how this model can be used for high-level system behaviour analysis and high-level system architecture design.

Chapter 6 describes how the high-level architecture modules can be implemented in a mix of hardware and software.

Chapter 7 provides descriptions of the necessary tools. These provide an innovate mix of graphical and textual descriptions to combine designing and simulation.

Chapter 8 provides the results achieved so far and describes the work which remains to be done.

Appendix 1 shows the result of applying the model for system behaviour analysis.

Appendix 2 gives a list of results achieved with the tools which have been developed. This includes an ASIC which forms the major part of a switching network for a telephone exchange.

Appendix 3 provides a list of terms used in this Ph-D thesis.

1.1 Target: designing complex data processing systems

The design methodology described in this Ph.D thesis aims at complex data processing systems. The methodology should be generally applicable. It should not be limited to a single 'problem domain' like digital signal processing or real-time process control.

The methodology should be capable of designing system architectures at almost any level of 'module complexity'. Designing at computer-, board-, chip-, register or gate level does not differ that much. Parallels can be drawn in software design. The difference lies more in the complexity of the modules which constitute the system at each level. Within each complexity level, systems are built out of communicating modules. This makes it possible to use a common methodology.

As stated in the introduction, the specifications for a system are often fuzzy and need clarification. The informal specification must be formalised, if possible in a form which can be simulated on a computer. This allows checking whether the formalised behaviour matches the informal behaviour. Informal behaviour basically exists in the mind of the customer. The best way to check the formal behaviour is to show the customer what the system does. To ease checking, the formal behaviour should be stated in the terms which the customer used in his informal behaviour description.

When the formal system specification has been checked and approved, actual system design can start. A system architecture must be designed which exhibits the same behaviour as the formal specification. If necessary, this architecture can be refined to lower levels of module complexity until each of these modules can be implemented. Until that point, functionality can be moved freely between the modules, while modules may be combined or split up. When a module is implemented, the functionality is fixed. During implementation, a module may be split up into lower-level architectures, but the module's functionality remains the same. This is an iterating process - implementing high-level modules by decomposing them into lower-level modules. The iteration ends when pre-designed modules are used.

One of the major targets of our design methodology is to allow the use of *Application Specific Integrated Circuits* (ASIC's) as module of the designed system. ASIC's combine some very desired properties. They are small, fast, consume little power and can perform very complex functions. To allow ASIC design, the lowest level module library should contain elements which can be implemented in ASIC form.

The lowest level modules used in this design path are the *'basic building blocks'*. These model hardware structures like registers, memories (including queues, stacks and content addressable memories), arithmetic/logic operators and state machines. An implemented basic building block may contain thousands of gates, whose combined behaviour is described with only a few lines of text. This allows a much shorter design time than when the gates had to be connected manually. Combinations of basic building blocks may be stored as a single complex module, ready for re-use. This allows a designer to create a library of modules to build future systems from, decreasing design time even further. Translation of basic building blocks into lower description levels (*'gate'*, *'transistor'* and *'layout'*) is an automatic process.

1.2 Requirements for the design methodology

Before introducing our design methodology, we first give a list of requirements we think necessary for *any* design methodology. Section 1.3 gives some existing methodologies and checks their adherence to the requirements.

No architecture restrictions

A design methodology should not force the designer into designing more or less fixed architectures. Limiting the number of possible architectures immediately limits the range of applications covered by a methodology.

Applicable to all architecture levels

Limiting the range of architecture levels a methodology can handle also gives problems. When a methodology is only applicable to high-level architectures, a follow-up method must be applied to do the low-level architectures. Conversely, a methodology intended only for low-level architectures cannot give insight into the complete system.

Using a single methodology which covers a large range of architecture levels gives several advantages. Designers familiar with the methodology can do the high-level architectures and continue working on subsequently lower levels. Mixed level designs become natural - high-level modules can be used as test environments for low-level designs.

Possibility of simulation/execution

Being able to simulate a system has several advantages. An informal specification can be validated by formalisation followed by simulation. Interactive simulation allows building and verifying a design in small steps. This reduces the possibility of covering a mistake under a large number of quasi-simultaneous

changes made to the system. Simulation also allows stochastic analysis of the operational system.

Mixed-level simulation should be possible. A high-level architecture may have one or more elements modelled at lower levels. The remaining high-level elements act as test environment for the low-level designs.

Open to analysis

Systems described with any design methodology should be open to several kinds of analysis. Design decisions are based on analysis results. Both *static* and *dynamic* analysis should be possible:

Static analysis looks at system characteristics which can be deduced from the system description. For a software-implemented system, static analysis will provide the characteristics which are known 'at compile time', for instance:

- The complexity of the functions which are performed.
- Statically allocated data storage.
- Connections between system elements.
- Communication formats used.

Dynamic analysis looks at system characteristics which are exhibited during operation. For a software-implemented system, dynamic analysis provides the so-called 'run time' characteristics like:

- Data processing characteristics - the tasks which were performed, the amount of time needed.
- Dynamic data storage requirements.
- Data transfer characteristics - the amount of data transported, the amount of time needed, the occurrence of blocking or deadlock.

By applying analytical methods to the static system description, some of the dynamic characteristics can be predicted. Queueing theory can be applied to predict some of the data transfer characteristics. The Calculus of Communicating Systems (**[mil80]**) can predict other data transfer characteristics like deadlock (**[hui88]**). These analytical methods share a large disadvantage - they require the system to be abstracted to a degree that it does not adhere to the original behaviour anymore (data dependent traffic patterns are very difficult to take into

account). Where analytical methods fail, dynamic system analysis must be based upon statistics gathered during simulation.

Short design cycles

As stated in the introduction, a customer wants his system delivered 'ASAP' (As Soon As Possible). A design methodology should provide results fast, while maintaining high standards of quality and reliability.

Gaining speed in a design methodology can be done in two orthogonal directions:

- *Increase designer productivity*

  The number of correctly designed-in entities per time unit is almost independent of the entity complexity **[koo92]**. By using powerful entities, the number of entities (and time) needed to describe the system is reduced. Studies have shown that writing 100.000 lines of code requires between 50 and 75 times the effort of writing 10.000 lines of code **[ebe89a]**. This indicates that the gain achieved by using higher complexity entities is even larger than can be expected from the complexity ratio.

  Very powerful entities are found in previously designed system elements. Re-usability is a powerful way to increase designing speed. The number of errors introduced in a system relate to the amount of new code written. A design made out of library elements should therefore be of a much higher quality than one made from scratch.

  Designer productivity also depends on the tools used. Interactive design tools allow immediate checking of modifications and give a designer high confidence in his own work.

- *Support teams of designers working together on one project*

  A design methodology should make it possible to divide the work across several designers early on in the process. The methodology and tools should support this by allowing the exchange of finished design elements and multi-level simulation. This allows giving each designer a copy of a roughly subdivided system. Each designer is then assigned to one of the subsystems. Once (intermediate) results are produced, they can be distributed to all group members for inclusion in their system.

User friendly

User friendly-ness is both a question of tooling and of the design methodology.

A user friendly methodology models systems in a way a designer finds 'natural'. A model built with such a methodology closely follows our perception of 'reality'. Each of the model entities has a corresponding entity in the 'real world'. This makes it easier to describe the functions of the model to people unfamiliar with the methodology. This holds even when this 'real world' does not exist but is a more abstract concept.

Even when a methodology in itself is user friendly, having impossible-to-work-with tools will not make it very popular. Making a tool user friendly involves lots of ergonomic aspects which we will not describe here. The most important is that the tool's operation should be intuitive. It should also provide fast responses to design actions. Having a huge set of compilers, linkers and simulators cannot be considered user friendly methodology support.

1.3 Current design methodologies

This section reviews several existing design methodologies and compares their attributes to the requirements stated in the previous section. Tools are not evaluated, as these are always evolving and would blur the view.

This section is divided into three sections according to 'phase' of the design path (high-level system behaviour modelling, high-level system architecture, low-level component implementation/architecture). The existing design methodologies are checked for their usability for these phases. Some methodologies do not cover all phases and must be extended with other methodologies.

1.3.1 High-level system behaviour modelling

High-level system behaviour modelling is the first phase of the design path. It is used to analyse the requirements and transform an informal specification into a more-or-less formal description of the same system.

Hatley and Pirbhai ([hat87]) use a *'Requirements Model'* to fix the specifications of a system. The system is decomposed according to it's functions, leading to a layered description of processes, stores and data flows. Processes are described by *'Process Specifications'*, for which any description method may be used. A superimposed *'Control Model'* with control flows, control stores and Finite State Machine-like *'Control Specifications'* is used to control the data transformations in the system. Timing requirements may be specified. Data types used by the data and control flows are specified separately in a *'Requirements Dictionary'*, which allows static system checks to be performed.

The Hatley and Pirbhai methodology does not place many restrictions on the systems designed with it. In general, systems modelled with this methodology can not be simulated. No real semantics of the modelling entities are given. Also, no syntax and semantics of an entity specification language are provided (any language can be used, with a tendency to 'structured English'). System analysis is limited to static consistency checking (the requirements dictionary is checked for completeness, the connections drawn on the diagrams are checked for 'balancing' errors). Design speed is adequate, because complex entities can be described and work can be distributed. Behaviour errors cannot be detected by simulation which may lower the design speed. Hatley and Pirbhai use functional decomposition to build their system model. If this is done too strictly, it may become difficult to comprehend the mapping of the model to the problem.

Ward and Mellor ([war85]) base on the same work as Hatley and Pirbhai, namely that of DeMarco ([dem78]). The *'Essential Model'* describes the behaviour of the system independent of any architecture or implementation - this is the equivalent of the Hatley/Pirbhai *'Requirements Model'*. Ward and Mellor separate data flows into continuous data flows and event-type data flows. The same separation is done with control flows. Modelling identical processes is eased by directly indicating that there are multiple instances of a process. *'Entity Relationship Diagrams'* (taken from [che76]) are offered as 'data oriented' adjunct to the functionally oriented *'Transformation Scheme'* (which combines Process- and Control Specifications).

The Ward and Mellor methodology shares most of the problems and capabilities with Hatley/Pirbhai. This is not surprising, as they have the same ancestor. The separation of data flows types gives a better semantic base for the interconnections. It also models real-world communication methods more closely, as these can be synchronised ('event-type') or unsynchronised ('continuous'). Modelling with entity relationship diagrams helps building more comprehensible systems, as they decompose the system in a more 'natural' way.

VHDL (VHSIC Hardware Description Language, [iee88]) allows the definition of virtually any kind of *entity*, complete with input/output *ports* and an internal *architecture*. The entities themselves may form components of higher level architectures. True parallelism can be described because each entity behaves as a separate process. Unlike Hatley/Pirbhai and Ward/Mellor, VHDL is an executable language with syntax and semantics. The problem with VHDL as behaviour description language is stated in the first line of the IEEE Standard VHDL Language Reference Manual: *'The design entity is the primary hardware abstraction in VHDL'* (underline by author). The language is targeted at *hardware* architectures, not at formal descriptions of abstract problems. For this task, VHDL is comparable to other structured programming languages - it is possible to describe parallel tasks, but the level of abstraction is fairly low. Data types and behaviour descriptions are at the level of a language like Pascal. Interface methods are low-level hardware oriented, but can be

extended to include functions like buffering and multi-source communication channels. Extensions like these can be placed in libraries for common use. Another approach is taken by Benders and Stevens ([ben91]), who describe a VHDL preprocessor with high-level synchronization and communication constructs.

VHDL is a reasonably modern hardware specification language. In this section, VHDL can stand for other such languages like ELLA ([pra86]) and SID ([sag90]), all of which have similar capabilities.

> Hardware description languages like VHDL place a very stringent restriction upon the systems designed with them - they are fully targeted towards hardware (ASIC's). A large advantage of these languages is that syntax and semantics are fully defined. Simulation of the described system is possible, tools for gathering statistics are available. System consistency is defined by syntax and semantics and can be checked by compilers. Using hardware description languages for general system analysis is slow because of the low abstraction level. It is difficult to describe complex entities in a clear and concise way. The capabilities to describe complex data structures and communication protocols are limited. The way a system is decomposed using a hardware description language is up to the designer.

1.3.2 High-level system architecture modelling

Following rigid specification of the system's functionality, the functions must be allocated to actual operational modules. This defines a high-level system architecture and forms the second phase of the design path. At the end of this phase, an implementation is chosen for each of the architecture modules. High-level architecture design is very goal-oriented. The objective is to re-use already existing architecture module implementations. It is much cheaper to use an already existing implementation than to design a totally new one.

Hatley and Pirbhai ([hat87]) build an *'Architecture Model'* which defines the configuration of physical modules that perform all the required data and control operations. Each of the *'Architecture Modules'* is subdivided into four parts - user interface, input processing, output processing and functional/control processing. A fifth part may be added, containing maintenance, self-test and redundancy management processing. Architecture Modules may be layered. The lowest layers are specified by an implementation choice and the functions which must be performed (referring to the *'Process Specifications'* and *'Control Specifications'* stated in the *'Requirements Model'*, see section 1.3.1). *'Architecture Flow Diagrams'* indicate how data travels through the system. *'Architecture Interconnect Diagrams'* show the actual interconnections for the data flows. An *'Architecture Dictionary'* summarises the data types used in the system.

The Architecture Modules provide a good means to describe a system architecture. The problem remains that the system cannot be simulated and thoroughly analysed. Implementation choices are solely based upon the designer's experience. It cannot be checked whether or not a physical module can perform the functionality assigned to it. The subdivision of Architecture Modules is a functional one. It may not reflect actual implementations.

Ward and Mellor ([war85]) do not use a new modelling technique to build their *'Implementation Model'*. The framework of the *'Architecture Modules'* used by Hatley and Pirbhai is absent. The Implementation Model re-groups the functions and stores present in the *'Essential Model'* (see section 1.3.1). For each of the functional groups in the Implementation Model, an implementation method is stated. Ward and Mellor divide the Implementation Model into a *'Processor Model'*, which is subdivided in a *'Task Model'*. This shows a tendency towards software implementations. Separate sections are devoted to interface-, process management- and data management modelling.

The Ward and Mellor Implementation Model has the same shortcoming as the Hatley/Pirbhai Architecture Model - it cannot be simulated. The subdivision in Processor and Task models is tendentious.

VHDL ([iee88]) is better suited for high-level architecture design than for behaviour modelling. The rigid hardware orientation gives it the same shortcomings as stated in section 1.3.1. VHDL is not abstract enough to describe true behaviour.

SDL. The CCITT *'Specification and Description Language'* ([cci87]) is developed to describe the concurrent behaviour of processes in telecommunication systems. Using SDL, a system is described as a set of communicating state machines. Communication is performed by sending messages, which are buffered by the receiver. As shown by Hulzebos ([hul88]), the same language can also be used to describe parallel hardware processes. Implementation of such a (reasonably abstract) description is not a simple issue. Direct implementation is impossible, for instance because infinitely long buffers are assumed. The original SDL model must be changed by introducing artificial limits before implementation can start.

1.3.3 Low-level component architecture modelling

Once the functional-, interface- and implementation method specifications have been given for an operational module, it can be implemented. During implementation, lower-level architectures may be introduced, as long as the total functionality remains intact. Hardware as well as software (with an appropriate processor) and mixed hardware/software implementations can be chosen.

Hatley/Pirbhai ([hat87]) and Ward/Mellor ([war85]) stop when a high-level system architecture has been defined. Ward and Mellor state which tasks must be placed in a processor, but they do not generate any executable code for these tasks. Hatley and Pirbhai devote chapter 24 of their book to this phase (13 pages out of 401!). They state *'hardware decomposition stops when separate, deliverable units are identified'*. They do *not* state how large these modules are. From their text it appears that they mean functional groups of Integrated Circuits, like *'memory'* and *'CPU'*. Below that level, their architecture modelling technique becomes too cumbersome.

VHDL ([iee88]) is very well suited for intermediate- and low-level architecture design. Given the correct libraries, Register Transfer Level and Gate Level designs can be entered and simulated without much problems. Using VHDL as a tool for software module prototyping/programming is not the goal of the language and should be avoided.

Like other hardware description languages, VHDL is an input language for silicon compilers. Given a *'structural VHDL'* description of a system, these compilers can generate a silicon layout of a circuit which performs the system's functions. Currently, 'structural VHDL' is a (varying) subset of the complete VHDL language. Often, such a description must make use of predefined Register Transfer Level library entities, which themselves are defined in terms of logic gates. Most of these language systems have special constructs to define often used hardware entities like Programmable Logic Arrays or Finite State Machines (the ASA system described in **[sag90]** is an example here).

For more limited applications, specific silicon compilers can be generated. Examples are Cathedral ([man86]) and HiFi ([lan89]), which are both targeted towards signal processing algorithms. Cathedral allows designing a signal processing ASIC with only a transfer function equation (accompanied by some directives) as specification. This is, however, accomplished by using a few parametrisable architectures as foundation for building the ASIC. Although designs made with such a system look different, the basic structure is the same.

1.4 Our approach: Object Oriented Modelling

In section 1.3, several approaches towards system design have been examined. These all had one or more drawbacks. In this section, we will give a design methodology which tries to circumvent the problems encountered. The remainder of this Ph.D thesis will give more insight in the basic design model and methodology, in that order. Like in the previous section, the design path is split into three parts:

1) *High-level system behaviour modelling*, during which the problem statement is analysed and a complete (operational) behaviour model of the system is built.

2) *High-level system architecture design*, during which the system behaviour model is transformed into a high-level system architecture model.

3) *Low-level module architecture design and implementation*, during which the modules which comprise the system architecture model are implemented in hardware and software.

In our view, tooling is just as important as a solid model to build upon. The tools should be consistent throughout the design path, having similar interfaces and behaviour. This shortens familiarisation time for the designers. Operation of the tools should be *'intuitively'*, it should follow operational procedures which a user already expects. The same quality is something which we try to attain for the whole design methodology.

The basic model introduced in chapters 2 and 3 is a very complex model.

The tools support this model with an interactive graphical design and simulation environment. Setting up a design always starts with a bare and simple model. The designer gradually introduces the complexity needed to model the system. It is possible that a system design never uses all of the basic model's capabilities. All bookkeeping and consistency checking functions are performed by the tools while the design is built. These functions are normally invisible to the designer.

Using an interactive design environment allows the designer to concentrate upon his task: designing a system using those capabilities of the model which are actually needed, without being bothered with the intricacies of the model itself.

Conveying the operation of an interactive design environment on a static medium like paper is a nearly impossible task. There is no substitute for 'hands-on' experience, but that is something we cannot offer in this Ph.D thesis.

1.4.1 High-level system behaviour

During high-level system behaviour modelling, the problem statement is analysed and a complete (executable) behaviour model of the system is built. Our *Object Oriented Analysis* method is based mainly upon Coad and Yourdon ([coa90]), and Shlear/Mellor ([shl88]). Bailin ([bai89]) provided a suitable requirements specification method. Shlear and Mellor ([shl89]) described a method to analyse the problem statement. These methods have been merged into a single framework. We added timing and concurrency to describe and design real-time systems. Chapter 4 provides an in-depth description of the methodology, which can be outlined as follows:

- Compile a list of *'things'* which are found in the problem statement, the *'Problem Domain Entities'*.
- Give the system a structure by defining *'forms part of'* relationships between the Problem Domain Entities. These relations may be fixed or variable (Problem Domain Entities which *'travel'* through the system).
- Build a superimposing structure by finding *'is a kind of'* relationships between the Problem Domain Entities. This will later reduce the amount of work because similar behaviour need be coded only once.
- Define communication channels by tracing *'communicates with'* relationships between Problem Domain Entities.
- Determine the global operational *'aspects'* of the system. These will be used to keep a better overview of the system by concentrating on one aspect at a time.
- Define the *'message'* protocols to be used in the communication between the Problem Domain Entities. This includes describing in an informal way what a message is supposed to achieve.
- Implement the handling of the messages by the Problem Domain Entities. This includes defining variable storage timing aspects.

The result of the analysis phase is a system containing communicating Problem Domain Entities which exhibits the desired behaviour. This system can be simulated and shown to all interested parties (customers). The system behaviour described this way serves as a reference for the other phases, it is also the starting point for the next phase.

1.4.2 High-level system architecture

The high-level system architecture is designed by mapping the *Problem Domain Entities* onto a set of *Abstract Processing Entities*. Each of these form an abstract description of the operations performed by a (hardware and/or software implemented) processing unit. The problem-domain communication channels are re-mapped onto *Abstract Communication Channels*. Each of these represent a hardware communication medium (with accompanying protocols).

The re-mapping process is implementation directed. During high-level system architecture design, preliminary implementation choices are made. These are based upon the required capabilities of the Abstract Processing Entities and Abstract Communication Channels and a database containing *'profiles'* of actual processing units and communication channels. Operations of Abstract Processing Entities may be combined to make better use of a processing unit which is not fully loaded. An Abstract Processing Entities may be split to distribute a heavy processing load across several processing units. Abstract Processing Entities may be duplicated to design fail-safe systems. Communication channels may be split or combined for similar reasons.

The high-level system architecture design phase ends when the preliminary implementation choices have been fixed. The chosen Abstract Processing Entity implementations can handle the processing load they have been assigned to do. The chosen Abstract Communication Channel implementations can handle the data traffic. All functions of the analysis phase system have found a place in the Abstract Processing Entities. The overall behaviour adheres to the behaviour of the analysis phase system.

The result of the architecture design phase is a system containing communicating Abstract Processing Entities, connected by Abstract Communication Channels. For each of these system elements, a behaviour description and implementation strategy are given. The complete specifications for each of the Abstract Processing Entities and Abstract Communication Channels are given to the designers which will implement them in the next phase.

1.4.3 Low-level component implementation

During the low-level component implementation phase, the Abstract Processing Entities and Abstract Communication Channels are implemented. This is done in the form chosen at the end of the high level architecture design phase. Software implementations (programs) can be constructed from the behaviour descriptions which define an Abstract Processing Entity. Because these descriptions are executable, creating a first software implementation can be done by direct translation.

For hardware, the route to *Application Specific Integrated Circuits* (ASIC's, the main target of this design path) is somewhat more complex. Hardware implementations have a low-level architecture of their own. Several steps are needed to replace the abstract behaviour by more hardware oriented behaviour:

1) The Abstract Processing Entity is replaced by a *Low-level Simulation Entity*. The message interface is converted into a hardware compatible interface using *Interface Entities* and *Interface Primitives* (registers and memories, for instance). This fixes the interface between the hardware implementation and the 'outside world'. It includes a mapping of the behaviour level data types onto hardware compatible data types. Also, interface protocols must be defined.

2) The internal Abstract Processing Entity functions are translated into one or more *Algorithmic Level Entities*. During this translation, the behaviour level operations are converted into operations on the hardware compatible data types.

3) The Algorithmic Level building blocks which define a piece of data processing hardware are translated into real data paths and controller structures. These are specified in a 'language' which uses basic building blocks like *registers*, *memories* (including *queues* and *stacks*), arithmetic and logic *operators* and *state machines*. During this translation, the hardware interface primitive elements are included in the datapath.

4) The basic building blocks are converted into a suitable *Hardware Description Language* equivalent. This system description is converted into an ASIC using a set of standard design tools.

5) The system parts are actually built and connected together. Following system integration tests, the *design* path is concluded with delivery of the complete system to the customer.

Figure 1.4.3-1 shows the relations between the entities which are used during the first three steps of the ASIC implementation process. At the left, an Abstract Processing Entity is shown in it's original form. At the top-right, an Abstract Processing Entity has been converted into an Algorithmic Level equivalent. At the bottom-right, an Abstract Processing Entity has been converted into a basic building blocks design. The communication methods indicated in this figure allow all design elements to communicate with eachother.

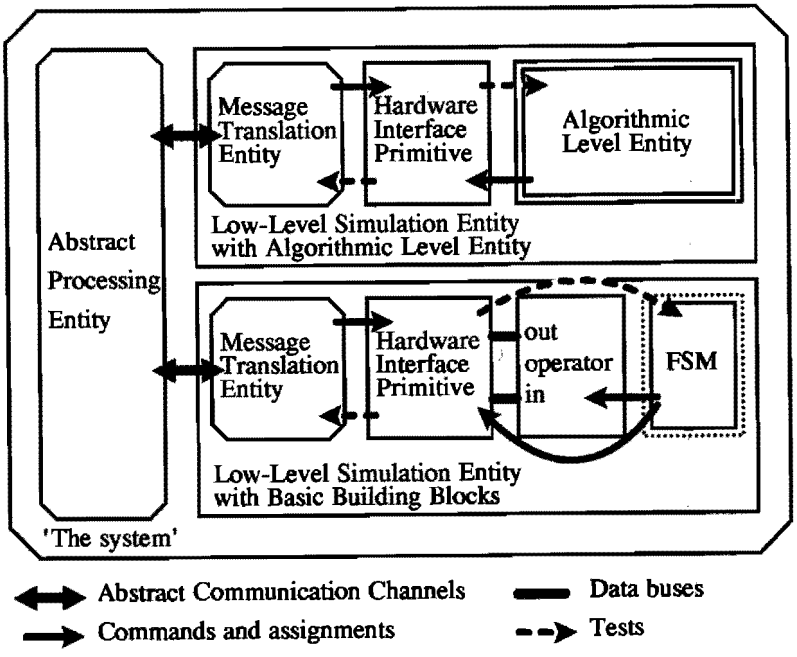


Figure 1.4.3-1: Three levels of abstraction in one system

Although not stated explicitly here, testing and testability plays an important role in the design process. An almost mandatory system aspect is *'testing and maintenance'*, which defines the global system functions related to keeping a high level of confidence in the system's operation. This aspect translates to built-in test hardware (scan paths) and extra testing code in software.

2. The Basic Model

This chapter describes the basic Object-Oriented model used for *Problem Domain Entities* and *Abstract Processing Entities* (during system behaviour analysis respectively high-level architecture synthesis). Extensions to the basic model are given in chapter 3.

The basic properties of the model are found by the following line of reasoning:

Problem stated in chapter 1:

> We need to analyse problem statements for complex information processing systems. When this has been done, architectures must be designed which implement these systems. These architectures range from very coarse grain (high-level system architecture) to fine grain (basic building blocks which can be implemented in hardware).

Solution:

> Provide a modelling technique which can be used to model and simulate the problem and any architecture derived from this problem model. The requirements for this modelling technique are the following:
>
> 1) *The modelling technique should provide an abstract way to describe concurrent systems of any kind.* Both the problem statement and all derived architectures contain concurrent communicating processes. A high level of abstraction is necessary to describe complex systems without going into fine details.
>
> 2) *The modelling technique should allow analysis and structure modifications.* Architecture design basically consists out of modifying the system structure. Analysis is needed to base modification decisions on.
>
> 3) *The modelling technique should provide simulation capabilities.* Comparing a model to an informal specification is only possible by simulation. The modelling technique is designed to be used in an interactive graphical design and simulation environment (see chapter 7).

Sections 2.1 through 2.1.6 gradually introduce and explain the basic model which fulfils these requirements. The remainder of this chapter describes the basic model in greater detail.

2.1 Ideas behind the model

The basic idea behind the model is to provide the designer with a fully Object-Oriented simulator for information processing systems. Standard Object-Oriented languages like Smalltalk ([gol89], [dig88]) solve problems by describing the problem as a set of communicating objects, each with their own behaviour. A user can 'ask' such a system to perform a function. This function is performed by all the objects in cooperation by sending messages back and forth. Eventually, a result message will be returned and the problem is solved.

Real information processing systems contain a multitude of elements (*'objects'* or *'entities'* in Object-Oriented parlance) which are all active at the same time. This *concurrency* aspect is lacking in most Object-Oriented languages. In Smalltalk, for instance, sending a message always makes the sending object wait for a response (even a dummy one). Only one object is active at a time. The system has no idea of time, which makes it virtually impossible to simulate real-time systems.

The predecessor of Smalltalk, *Simula* ([bir73], [fra77]) was a language specifically designed for simulation of general systems. This language allowed concurrency by having objects behave as processes. When a process object is created, it becomes operational and starts it's internal operations. A process object can put itself 'asleep' for a specified or indefinite time. If indefinite sleep is entered, it must be 'awakened' by another object. Actual calculations take zero simulated time. By inserting sleeping periods between calculations, an object simulates real calculations.

The model proposed in this Ph.D thesis builds upon Smalltalk. Timing and concurrency constructs are added to allow general system simulation. Redefined communication methods allow complex communication protocols to be modelled directly.

The following sections state more specifically what this model is based upon.

2.1.1 Standard Object-Oriented constructs

The Object-Oriented system model is built out of communicating objects. Each of these objects contains internal variables and the operations (*'methods'*) which manipulate these variables. The objects communicate by sending *'messages'* between eachother. Receiving a message invokes one of the methods in the receiving object. These methods may change the internal state of the object (the variables stored within it). They may also invoke the sending of other messages. A message may be a direct command (like a

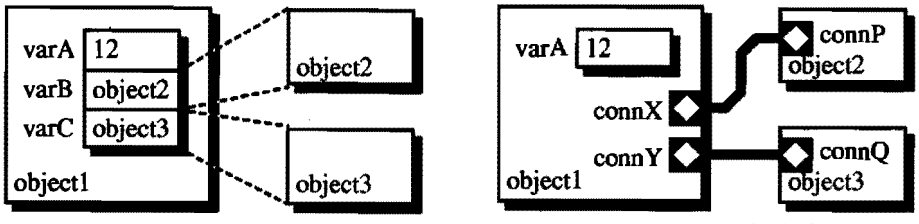
procedure in Pascal) or a request for information (like a function). In the latter case, a result message is returned.

Actual objects are instantiations of a *'class'*. The class defines which variables are stored within the objects and the operations which are performed when messages are received. Each *'instance'* (actual object created from such a class definition) has it's own set of variables. Identical messages sent to instances of different classes may provoke different reactions, something which is called *'polymorphism'*. For instance, both a circle and rectangle object may be sent the message **'surface'**. Both will calculate and return their surface area, but use radically different methods to do so.

A class description may be based upon another class description by a process called *'inheritance'*. A derived class (*'subclass'*) may add new variables and methods to those already defined in the original class (the *'superclass'*). It is also possible to re-define methods which are already present in the superclass. Inheritance allows classes to share common functionality. For instance, both the circle and rectangle classes may be based upon a class called 'displayable object'. This superclass may contain general purpose methods to display something on a computer screen. The rectangle and circle classes can refine these methods to display themselves.

2.1.2 Separation of object state from system structure

In most Object-Oriented languages, no distinction is made between variables stored *within* an object and *external* objects with which this object communicates. The internal variables contain the object's *'state'* and are often replaced by other values. References to other objects are stored in variables which are seldom changed. This network of references embodies the *'static system structure'*. Sending a message to a stored reference does not differ from sending a message to a variable (variables are objects too!). This situation is depicted in figure 2.1.2-1a: 'object2' and 'object3' are stored as variable within 'object1'. This is misleading because they are not actually *contained in* 'object1'.



a) "External" objects reached by storing them in variables

b) External objects reached via communication channels

Figure 2.1.2-1: Two methods for describing the static system structure

In our view, merging variables and external references is incorrect:

The state of an object is something completely different from the static system structure, and should be treated as such.

Some objects *'travel'* through the system (customers entering and leaving a waiting line, for instance). The presence or absence of these travelling objects should be treated as object state. Travelling objects entering another object must be stored in an internal data structure. They must be removed when they leave again.

We separate the static system structure from the variables by defining the structure using communication channels. These communication channels are connected to 'connectors' which are placed within the objects.

In our model, a message directed to an external object is sent to a local output connector. This connector places the message on the communication channel it is connected to. The channel forwards the message to other connectors attached to it. These other connectors transfer received messages to the actual object they are placed in. This situation is depicted in figure 2.1.2-1b: 'object2' and 'object3' can only be reached via communication channels attached to 'connX' and 'connY'.

By using local connector names, an object may be instantiated at different places within a system or even within separate systems. Each instantiation may have a completely different 'hookup' to other objects. No problems will occur when these other objects support the message protocols used by the replicated object. Seen from the instantiated object, it's environment must provide the services it requests by sending messages over the output connectors. *How* these services are provided does not matter.

2.1.3 Separation of communication from behaviour

In the previous section, we saw that sending and receiving messages between basic model objects is handled by connectors connected by communication channels. We now go a step further and basic model objects into a *'processing core'* and a *'communication shell'*:

- *The processing core* contains the state variables and the operations to be performed upon reception of messages. This core is defined by classes which may inherit behaviour from other classes.

- *The communication shell* selects, buffers and translates messages before passing them on to the processing core. It surrounds the behaviour core and protects the core from changes in the environment.

Separating behaviour from communication allows experimenting with several system structures, something which is necessary during architecture design. It is possible, for instance, to add a new communication channel without changing any behaviour core. This allows an architecture designer to study the effect of the new system structure on the system performance: Following this change, analysis and/or simulation may tell whether this extra channel removes a system communication bottle-neck.

2.1.4 Dynamic multiple inheritance

The way in which behaviour is inherited between classes differs between different Object-Oriented languages.

'Single inheritance':

Smalltalk ([gol89]) uses a tree-like class hierarchy, with class 'Object' as root. Each class has only a single direct 'superclass', which is repeated until the 'Object' class is reached. All classes in Smalltalk ultimately inherit behaviour from class 'Object', which directly translates to 'all objects are *a kind of* Object'.

Most Object-Oriented languages use this kind of inheritance, either with or without a single root class (C++ [str87], for instance, allows multiple root classes).

'Multiple inheritance':

In the Eiffel language ([mey88]), each class can inherit from multiple other classes. Inheritance relationships can be represented as a directed acyclic graph.

Multiple inheritance gives problems when two or more superclasses have variables with the same name or can receive the same message. When a subclass wants to update a variable, it must indicate unambiguously in which class this variable is located. When a message is received, it must be defined which superclass should handle it in case the subclass cannot do that.

Multiple inheritance is desirable. It allows defining the behaviour of an object as the combination of behaviours of other objects. The basic model uses a method which has most of the advantages of multiple inheritance. This method adds a very important advantage of its own - *'inheritance' may change over time*.

'Dynamic multiple inheritance':

Each basic object defines it's behaviour by the combined behaviour of a set of *'behaviour defining objects'*. These behaviour defining objects are stored in a user-defined number of *'slots'* within the processing core.

Message handling ambiguity is avoided by imposing an ordering upon the slots. This allows the interface shell to select the 'first' behaviour defining object which can respond to a received message.

The overall behaviour of a basic object can be changed by replacing one or more behaviour defining objects: '*Dynamic* multiple inheritance'. Behaviour defining objects are themselves normal Smalltalk objects, and allow only single inheritance.

2.1.5 Timing of objects

Each basic object should be seen as a self contained abstract processing unit. All basic objects in the system are capable of concurrent operation. They can all be handling messages at the same time. Synchronisation and cooperation are both achieved by sending messages.

Basic objects have two 'modes' of operation for handling messages:

1) *'Reactive' mode:* Objects are idle until a message arrives. At that time, processing is started by activating the corresponding method. This is the normal Object-Oriented way of handling a message. The object reacts to the messages it receives.

2) *'Imperative' mode:* The processing core actively retrieves messages which have been received and buffered by the communication shell. It is also possible to wait until specific messages arrive. The object itself determines which messages to receive and in which order they will be handled.

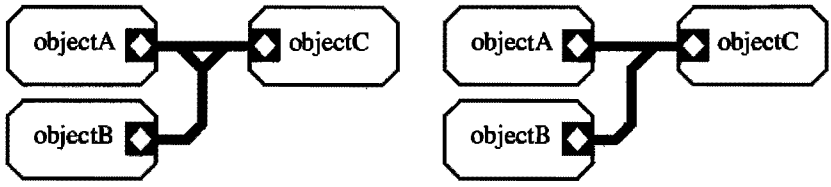
The simulation model is made time-conscious by allowing a basic object to specify processing times. Handling a message is always done at a specific *processing priority* level. Receiving a message with a priority level higher than the processing priority interrupts the running method and starts the method corresponding to this message.

Once the high priority message is handled, the low priority message method is resumed at the point of interruption.

2.1.6 Channel behaviour

A channel's behaviour is relatively simple. Multiple connectors may be connected to a single channel. Connectors are inherently bidirectional - they are capable of sending *and* receiving messages.

Channels generally *broadcast* any message placed upon them. Figure 2.1.5-1a) depicts this situation. It is possible to indicate that a channel concentrates messages towards a specific object. The other direction then automatically distributes messages. Figure 2.1.5-1b) shows the same situation with concentration and distribution towards- respectively from 'objectC'.



a) All objects intercommunicate (channel broadcasts messages)

b) Direct communication between objects A and B prohibited by channel message routing

Figure 2.1.5-1: Message channel routing

The channels should be seen as a simple communication medium. Routing messages based on contents or other system properties and/or states cannot be done by the channels themselves. Additional basic objects are needed to perform this function.

Timing of a channel is relatively simple. Only a single message can be transferred at a time. A channel may be in one of three states:

- *'Free'*: no message is present on the channel.
- *'Blocked'*: at least one of the receiving connectors refuses to handle the message which is present on the channel.
- *'Transferring'*: the message is actually transferred between the sending and receiving connectors.

Basic message transfer involves the following seven steps (it is assumed that the channel is in the *'free'* state before starting):

1) The sending basic object presents a message to one of it's connectors for transfer.

2) This connector offers the message to the channel.

3) The channel notifies all receiving connectors that a message has been offered.

4) The receiving connectors inspect the message offering and decide what to do with it. For now, the most important choices are to *receive* or *block* the message.

5) If any of the receiving connectors decides to block the message, the channel makes the transition to the *'blocked'* state.

6) If none of the receiving connectors blocks the message (anymore), actual message transfer starts. The channel makes the transition to the *'transferring'* state. This state lasts a designer-specified time.

7) Following message transfer, the channel goes back to the *'free'* state. The receiving connectors pass the message to their respective processing cores.

Other messages may be offered to the channel while it is *'blocked'* or *'transferring'*:

- *While 'blocked'*, a higher priority message offering is distributed immediately by the channel. The message transfer cycle is restarted at step 3. Lower- and equal priority message offerings have to wait until the current message has been transferred.

- *While 'transferring'*, offered messages are stalled in the sending connectors. Following transfer, the channel selects the highest priority message offered to it and immediately restarts the transfer cycle at step 3 (the channel does not become *'free'*). If there are multiple high-priority messages to choose from, a random choice is made between them (this gives all connectors a fair chance to use the channel).

2.2 Mapping of data transfers and events on messages

In Object-Oriented systems, all communication takes place in the form of sending messages.

Messages always contain a fixed part. This part identifies the message and may be used to differentiate between messages. It is also used to select the method which must handle the message in the processing core. This is the reason why the fixed part is called *'message selector'* in Smalltalk.

Messages may contain a variable part in the form of parameters. The number of parameters may vary. Objects of any class may be used as parameter.

Messages serve three purposes in an Object-Oriented system:

1) *Events are used for synchronisation purposes.* They are normally encoded by a message without parameters like **'resetKeyPressed'**. Here, the message selector indicates the type of event.

2) *Commands are requests to an object to perform a local function.* These may also be transferred by messages, either with or without parameters like **'stopCarriage'** or **'setSpeedTo: 37'**. Again, the message selector indicates the command given.

3) *Requests ask an object to return some information* which is present in (or can be obtained by) the receiver. These need two messages. The first issues the actual request. The second one contains the reply, going in the opposite direction. In the normal 'mode of operation' the sender waits for the reply, while the reply is only received by the original requester. The requesting message may or may not contain parameters, like **'giveSpeed'** (reply: **'37'**) or **'multiply: 3 with: 2'** (reply: **'6'**).

   Reply messages are special in that they do not actually require a message selector - the recipient is known and is anxiously awaiting. In most Object-Oriented languages, the reply 'message' consists out of a single object. When more results must be returned, it must be done in the form of a compound object like an Array.

Events and commands are no problem in the basic model. The sending object need not wait for anything to be returned, and can go on with it's operations immediately.

Requests and replies need special attention. Returning a reply may take some time. The behaviour of the sender during that time has to be specified. Two possibilities exist to model this behaviour:

1) *Standard behaviour.* The sender waits for a reply. The receiving object returns the result using the normal Smalltalk technique at end of it's method.

2) *Decoupled behaviour.* The sender does not wait for a reply. A separate result message is used to return the requested information.

Examples of both methods are given below.

Standard request message handling behaviour:

Example segment of a sending method:

```
...
result := floatAlu multiply: 2.0 with: 3.1 .
...
```

"'floatAlu' is the name of a connector with access to a floating point ALU. This way of sending a message within an assignment indicates a result is expected which must be awaited."

Receiving method:

```
multiply: x with: y

    ^x * y
```

"The caret indicates that a standard result message is to be generated and sent back across the connector at which the **multiply:with:** message was received."

This is normal (Smalltalk) message sending behaviour, no concurrency is used. The sender waits for the reply, the receiver stops processing after sending the result back. The syntax directly follows that of Smalltalk.

Decoupled request message handling behaviour:

Example segment of a sending method:

```
...
floatAlu multiply: 2.0 with: 3.1 .
...
```

"This way of sending a message without an assignment indicates no result is expected immediately. The sender simply continues with his operations after this expression (assuming message sending is not blocked)."

Receiving method:

```
multiply: x with: y

        floatRequester multiplyResult: x * y
```

"'floatRequester' is a connector which is used by the floating point ALU to send messages to the sender of floating point operation requests. It is assumed that this is always the same object. This way of sending a reply message back does not stop processing in the receiver. The receiver may continue with it's operations."

Method in requesting object:

```
multiplyResult: aFloat

        result := aFloat
```

"The original sender must be prepared to receive a separate result carrying message."

This looks like normal (Smalltalk) message sending behaviour, but there is a very important difference:

> After sending a message to another basic model object, the sender need not wait for a (dummy) result. This allows true concurrent operation of basic model objects.

The sending object may have several requests standing out at the same time. This means that it must be prepared to receive several result messages. Administration of these requests and handling the result messages can become very complex. Extra *'tag'* parameters attached to the request and reply messages may be needed to identify them.

Our modelling system has two extensions beyond these two methods:

1) *Following the caret expression which returns the reply message, a receiver may specify more operations to be performed.*

   The reply message may be used to indicate that processing has started. It can also contain a (receiver generated) tag which will be added to the actual result message. This tag may be used by the sender in his outstanding requests administration.

2) *A sender may be actively waiting for a reply message or poll the reception of such a message.*

   The latter allows the sender to do some background processing while waiting for a result. This 'background processing' may be polling for other messages. It is possible to await a caret result message (which contains no message selector) by referring to the original request message.

Some examples of polling and waiting are given below:

Example of waiting for a single message:

```
...
self waitFor: #multiplyResult: into: #(result).
...
```

"waitFor:into: lets the sending object wait for the reception of a message with the indicated message selector. The parameter values attached to this message will be assigned to the variables named in the indicated Array (between #()). By substituting a message connector for self, the channel from which the message must be read can be specified."

Example of polling for a single message:

```
...
pollResult :=
        self pollFor: #multiplyResult: into: #(result).
...
```

"pollFor:into: checks for the reception of a message with the indicated message selector. This returns a Boolean result, indicating whether or not the message was actually present (and received). If the message was present, the received parameter values will be assigned to the variables named in the indicated Array (between #()). By substituting a message connector for self, the channel from which the message must be read can be specified."

Example of waiting for a caret result message:

```
...
requestIdentifier :=
        floatAlu multiply: 2.0 with: 3.1 {identifier}.
...
self
        waitForResultOf: requestIdentifier
        into: #(result).
...
```

"Indications between {} following the sending of a message to a connector change the message sending behaviour, not the message itself. In this case, the keyword 'identifier' indicates that the value assigned to requestIdentifier should be an identification for the message just sent, and not the result value."

"waitForResultOf:into: lets the sending object wait for the reception of the (caret) result of the indicated message. The received value is assigned to the variable named in the indicated Array (between #()). By substituting a virtual message connector for self, the channel from which the message must be read can be specified. 'poll' may be substituted for 'wait' in waitForResultOf:into: to change the waiting into a polling operation. This returns a Boolean result indicating whether or not the reply has been received."

The extensions described above are the main (true) concurrency introducing methods in our modelling system. They make it possible to have more than one basic object active at the same time, without having to introduce an explicit multitasking concept with processes and semaphores as done in Smalltalk.

The automatic starting of a behaviour method upon the reception of a message is called *'reactive'* behaviour - the object reacts to a message.

The extensions described above allow a behaviour method to operate in *'imperative'* mode. The object itself determines which messages to receive and in which order they will be handled.

Stated in standard computer interface terminology, *'reactive'* behaviour models interrupts, while *'imperative'* behaviour models polling. Some extra extensions to this imperative mode will be given in section 2.5.

2.3 The interface shell

Each basic object consists out of two layers. This section deals with the *'interface shell'* which surrounds the *'processing core'*. The next section describes this core which contains data storage and performs the data processing functions.

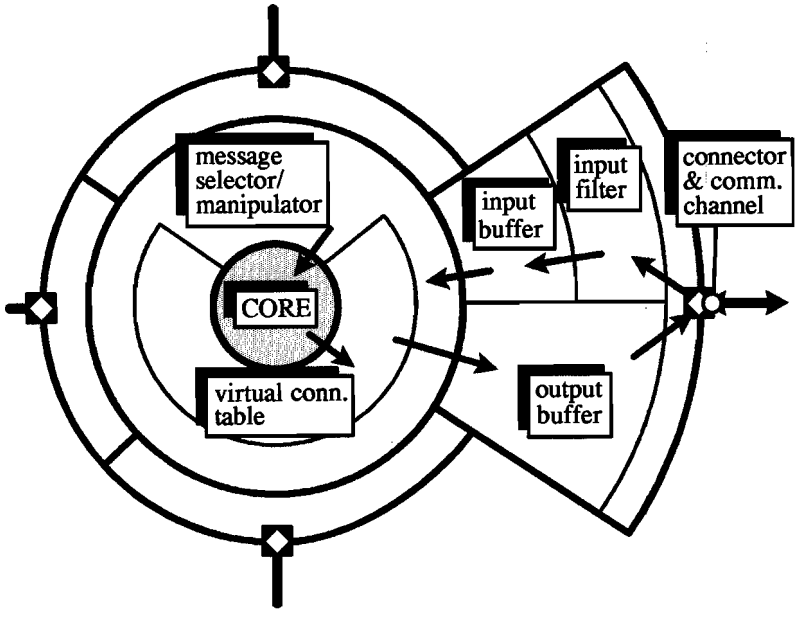


Figure 2.3-1: Elements of the interface shell

As depicted in figure 2.3-1, the shell is subdivided into two layers. The outer layer contains separate elements for each connector (only one connector is drawn expanded):

- *The input filter* is used to determine which messages on the channel will be received.
- *The input buffer* can be used to store incoming messages while the processing core is busy.
- *The output buffer* is capable of holding outgoing messages while the channel is occupied.

The inner layer is common to all connectors of a basic object. It contains the following elements:

- *The message selector/manipulator* selects the message to be offered to the processing core for handling. It can also perform some simple operations on these messages.

- *The virtual connector table* allows the core to use virtual names for the connectors. A single real connector may be known under several names.

Separate sections will be used to describe each of these elements.

Default behaviour of the interface shell elements following object creation is very simple: they are virtually absent. A system which doesn't use these elements to their full capabilities comes close to normal Smalltalk behaviour.

2.3.1 The input filter

Each message channel connector contains an input 'filter'. The purpose of this element is to make a selection of the messages on the channel. The messages which pass the filter will be handled by the input buffer and message selector and - ultimately - the processing core.

The filter's decision as to what should happen with a message are based mainly on the message selector. Furthermore, the decision may be influenced by the following variables:

- *Parameters of the message.* This includes 'hidden' parameters like message priority.

  Note that it is a bit strange that it is possible to check the parameters of a message which has not been received yet. The filter may decide to block the message transfer. Here, this decision is conditioned by the message contents. According to the channel behaviour described in section 2.1.5, the channel notifies all receiving connectors that a message has been offered. It is assumed that this notification includes enough data to let the message filters take their decision.

- *State of the input buffers.* This includes their filling level and the messages stored.

- *Variables stored in the processing core.* It is also possible to check the operating priority of the processing core.

Making a decision can be seen as a continuous process. Decision making starts when a message is placed on the communication channel. It stops when the message is thrown away, being transferred or removed from the channel. In the mean time the channel is in the *'blocked'* state (see section 2.1.5), and the initial decision may change.

The following decisions can be taken by the message filter:

- *Accept the message.* This opens the path from the channel to the input buffer. When message transfer starts while the filter accepts the message, the message will be stored in the buffer. If no buffer is used, the message will be transferred directly to the processing core via the message selector/manipulator.

  Accepting a message while the buffer is full will block the channel. The same happens when no buffer is used while the processing core is busy and/or the message selector does not select the message.

- *Absorb the message.* To the outside world, this is like accepting the message. Internally, the message is thrown away. It will not be stored in the buffer or transferred to the processing core.

  Stated in simple terms, absorbing a message means *'I know the message, but I do not handle it'.*

- *Hold the message.* This blocks the channel. The decision to hold a message should not be taken unconditionally as this may block the channel indefinitely. The only way out of that situation is when the message is removed from the channel, for instance by a timeout.

  As stated under *'accept'*, holding the message can also be done automatically.

- *Ignore the message.* This operation cannot be specified in the filter, it is automatically applied to all the messages which are not stated in the filter - the message is unknown.

  The *'ignore'* operation is basically the same as *'absorb'*. There is a significant difference - it is not allowed that a message is ignored by *all* connectors attached to a channel. When the latter happens, it indicates that no-one understands the message. Simulation tools which detect this situation will halt the system with an error message.

- *Reject the message.* Messages may be rejected because they are not allowed at all or because they are not allowed under the given conditions. During simulation, rejecting a message immediately halts the complete system with an error message. This option allows a designer to build his own error checking system.

Standard reply messages are always sent to a specific message connector (the one which was used to send the request). Therefore, *filters always accept a standard reply message.*

An input filter with an empty specification has special behaviour - it will accept *all* messages. Now it is up to the message selector and operational core to decide whether or not to handle the message. When the message selector's specification is empty too, at least one of the behaviour objects should have a method which is started with the message selector. The message will be automatically rejected if this is not the case.

As stated above, a non-empty filter specification causes unknown messages to be ignored. It is possible to change this is into rejecting all unknown messages. This can be used as a debugging aid during system simulation: All until then ignored messages must be specifically absorbed to prevent simulation halts.

A filter specification looks like a set of Smalltalk methods. For each of the message selectors, a separate text describes what to do with matching messages. An example:

```
terminal: address display: aString

"Display the given string on this terminal object if the address parameter
 matches the address variable stored in the 'interface' behaviour object.
 Block reception if the input buffer is full:"

self
    absorbIf:       "Address incorrect:"
        address ~= interface address.
self
    holdIf:         "Address correct but buffer full:"
        (address = interface address) &
        (self inputBufferFillLevel = self inputBufferDepth).
self
    receiveIf:      "Address correct and space in buffer:"
        (address = interface address) &
        (self inputBufferFillLevel < self inputBufferDepth)
```

2.3.2 The input buffer

Incoming messages can be stored temporarily in the input buffer. This element is located between the input filter and the message selector/manipulator. Each message connector has it's own input buffer. The following parameters are attached to each input buffer:

- *Buffer depth.* The number of messages which the buffer can store can be set between zero (no buffering at all) and - virtually - infinite.

- *Buffer algorithm.* The choice is limited to FIFO (First-In-First-Out) or 'priority FIFO'. The first keeps the messages in strict order of entry into the buffer. The latter sorts them so that higher priority messages are always in front of lower priority messages, with equal priority messages in normal FIFO order.

  The two algorithms inherently available in an input buffer are relatively simple. More complex algorithms can be simulated by the message selector, which can extract messages from the buffer at any point.

The default buffer depth is zero. When the buffer depth is set non-zero, the default buffer algorithm is priority FIFO.

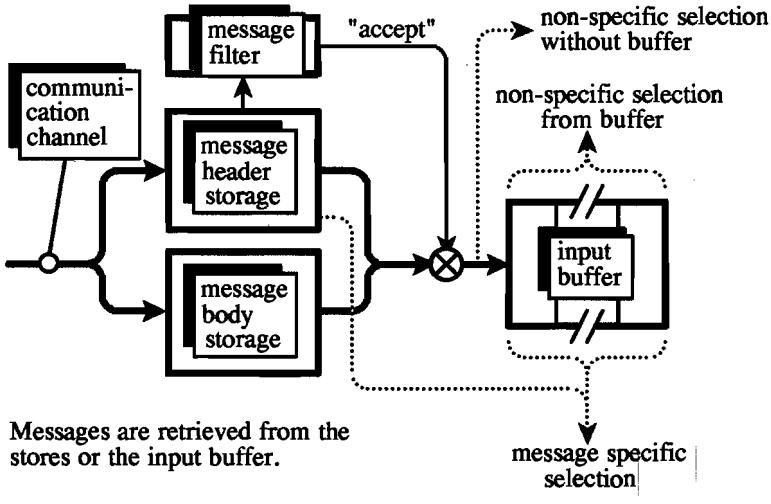


Figure 2.3.2-1: Conceptual diagram of the input buffer

Figure 2.3.2-1 should be used as a guide to describe the combined behaviour of the buffer and filter.

Conceptually, messages transferred across the channels consist out of a header and a body. The header contains the parameters needed to determine whether or not the message should be received. The body is the bulk of the message and is transferred when blocking is removed from the channel. Channels may select another (higher priority) message for transfer while the channel is blocked. When this happens, the header is invalidated and a new one is distributed.

Message selection is a combinatorial process. Any element selecting a message based upon the message header should be prepared for being offered a new set of data.

The headers which are distributed across the channel are loaded in a header holding store. Here, they can be inspected by the filter, which decides what to do with the message. Note that blocking is overruled when the message selector or processing core select the message:

- *Ignore/Absorb:* The channel is not blocked by this receiver.
- *Hold:* The channel is blocked by the receiver.
- *Accept:* Blocking depends upon the buffer. If there is space in the buffer, no blocking is done. If there is no space (or the buffer depth is zero), the channel is blocked.

Transfer of the message body starts once no receiver is blocking the channel anymore. The actual transfer takes a predefined time. When the transfer is complete, the body is loaded in a separate holding store. If there is space in the input buffer and the filter accepts the message, the complete message is copied to the buffer and the holding stores are invalidated. If the message cannot be copied to the buffer, it remains in the holding stores until one of the following happens:

- *The filter (again) accepts the message while space is (or becomes) available in the buffer.* At this point the message is copied to the buffer.
- *The message selector or the processing core retrieve the message.* The message never enters the buffer and is taken directly from the holding stores.
- *A new message is offered on the channel.* The header of this message is loaded in the header holding store. At the same time, the body holding store is invalidated. If, at this time, the header was selected (either by the filter, message selector or processing core), the message is lost. During simulation, loosing a message will normally lead to a system halt. In order to simulate real channel behaviour, a warning can be given instead.

Seen from the message selector and processing core, messages may be selected from the input buffer (head first) followed by the header holding store. Messages can be

retrieved from the input buffer and the holding stores. The method of message selection (see section 2.3.3.2) determines how the header holding store is treated. If selection is non-message specific, the header is selected when the filter accepts the message. If selection is message specific, the header is examined directly - messages may be selected which are not accepted by the filter. Message selection is indicated in figure 2.3.2-1 with the dotted lines.

2.3.3 The message selector/manipulator

Multiple messages may be waiting in the input buffer(s) for service by the processing core. The message selector/manipulator is the interface shell element which selects the message to handle. It can also perform simple manipulations and/or direct the resulting messages to a specific behaviour object (see section 2.1.4). Each interface shell has only a single message selector/manipulator, which is shared by all message connectors.

2.3.3.1 Requirements for the actual message transfer

All message selector/manipulator operations are performed in a 'combinatorial' fashion. Message selection and manipulation is a continuous process, resulting in offering one (manipulated) message to the processing core. Message selection merely indicates that the selected message has been checked to conform to specific requirements. Any message selection may be redrawn while actual transfer has not started yet. The actual message transfer is performed when the following two requirements are met:

1) *The message must be completely available.* The input buffers always store complete messages which can be selected immediately.

   While a channel is blocked, only the message header is available for selection. Selecting this header will override blocking introduced by a filled (or absent) buffer. When all blocking has been removed from a channel, message transfer starts and the body is loaded into the holding store. The message is then completely available. What happens at that point has already been described in section 2.3.2.

2) *The processing core is willing to handle the message.* This is the case when the priority of the offered message is higher than the operational priority of the processing core. The priority of messages is always higher than the 'core idle' priority. This is necessary to assure that a message can be received at all.

2.3.3.2 Message selection

The message selector/manipulator uses a two-layer scheme to select messages:

1) *Non-message specific selection.* Here, the actual message contents do not matter. If input buffers are present, only the messages at the head of the buffer can be selected. If the buffer is empty or absent on a channel, message headers loaded in the input holding store can also be selected, provided the filter accepts the message.

   Default behaviour for message selection is to use 'round-robin' message selection with priority. Each input channel presents a single message, from which the highest priority message is selected. If there are multiple messages with the same priority, 'round robin' selection is done, giving each channel a fair chance.

   A preference may be given to a specific channel. More specifically, it is possible to give a scanning order for the channels. This scanning order may be selected based upon buffer fill levels and/or core variable values. The scanning order may exclude channels and can specify whether or not message selection should take priority into account. When multiple scanning orders could be selected at the same time (because their selection conditions are all true), a default priority is used to select one of them.

   Non-specific message selection is specified by a list of texts. Each of these starts by stating under which criteria this selection is made. Following this, the channel scanning order and optional manipulations are specified, as in the following example:

```
"Use this non-specific message selection when the 'state' variable in
 the 'main' behaviour object indicates the 'shutdown' mode:"

main state = #shutdown

"Only receive from the 'control' connector in this case:"

receiveFrom: #(control).

"Increase the priority of all messages selected this way
 (section 2.3.3.4):"

increasePriorityBy: 20
```

2) *Message specific selection.* It is possible to select specific messages, based upon the following criteria:

- *The message selector.* The fixed part of the messages forms the main selection criterion.
- *Message parameters.* These include hidden parameters like message priority.
- *Input buffer fill levels.*
- *Operational core variable values*, including the core priority.
- *The priority of the message which would be selected by non-message specific selection.* This allows specific selection to be postponed when higher priority messages are waiting.
- *The input channel(s) on which the message may be received.* If not specified, all channels are open for reception. If a subset of channels is specified, the order of specification also dictates the scanning order.

  Selection of messages waiting at a channel starts at the head of the input buffer, proceeds to the tail of the buffer, and then looks in the header holding store. Message specific selection may select messages which do not pass the filter.

Message specific selection specifications look like a list of Smalltalk methods. Each of them start with the message selector which forms the main selection criterion. This is followed by an expression which indicates the other conditions which must hold for this selection to take place. Each specification ends with the operations to be performed on the received message, as in the following example:

```
setSystemModeTo: state

"If this message is received from the 'control' connector and the
 'state' parameter indicates 'reset'..."

(receivedFrom: #(control)) & (state = #reset)

"...then receive this message, and perform some manipulations:"

translateTo: #reset.        "Translate the message (sect. 2.3.3.3)"
setPriorityTo: #255.        "Change message priority (sect. 2.3.3.4)"
sendTo: #(main)             "Specify behaviour object (sect. 2.3.3.5)"
```

Message specific selection always has priority over non-message specific selection. When multiple specific selections hold, a default priority order is used to select between one of them.

Message specific selection is used by the processing core to receive standard reply messages. This way of operation ensures that the reply message is accepted and forwarded to the core as soon as it is offered on the channel.

As far as a channel is concerned, request and reply messages are simple normal messages which are handled separately. A channel is not kept free for the reply during the handling of a request.

A standard reply message cannot be selected by the message selector/manipulator, as it does not contain a fixed message part. Non-message specific selection skips reply messages. Without countermeasures, unexpected reply messages remain stored in input buffers or may even block a channel. To get rid of them, reply messages are flushed from the buffers and absorbed at the inputs while the processing core runs at the idle priority. This is done under the assumption that no requests are standing out during periods of inactivity.

2.3.3.3 Message translation

Re-using a behaviour object in a different basic model object may cause a mismatch between the message formats understood by behaviour object and the messages which are actually received. This mismatch can be removed by the message selector/manipulator by translating messages into another format. Message translation shields the behaviour objects from changes in their environment. This can only be done when message specific selection is used.

Translation may involve combining several messages into a single one. To do so, the source messages needed are specified as belonging together. To be able to select multiple identical messages, an already selected message is skipped for the remaining selection process. Actual selection of the messages is not done before all messages meeting the selection criteria are available. Actual translation is postponed until they are all present in their entirety. At that point, the selection criteria should still hold.

Translation always involves specifying a new message selector. Parameters for the translated message are either direct copies of parameters from the source message(s) or simple constants. Complex calculations on message parameters are not allowed, because this is a task for the processing core.

2.3.3.4 Priority manipulation

The message presented to the processing core may be given a priority which differs from the original message priority. The operations allowed are very simple - assigning a fixed new priority or raising/lowering the priority by a fixed amount. The resulting priority should lay within the allowed message priority range - from just above the 'core idle' priority to just below the 'no interrupts allowed' priority.

Unless specified otherwise, the priority of a message equals the processing priority of the sending behaviour method, limited to the range described above.

Translated messages may be based upon the combination of several other messages. In this case, the resulting priority is by default the maximum of the priorities of the original messages.

Priority manipulation can be applied to messages selected with non-message specific selection criteria. This makes it possible, for instance, to raise the priority of all messages coming from a specific connector.

2.3.3.5 Behaviour object selection

In order to obtain dynamic multiple inheritance, the processing core may contain multiple behaviour defining objects (see section 2.1.4). The slots in which these behaviour objects are stored have a predefined priority (the order in which they are named). By default, messages which are presented to the processing core are handled by the highest priority object which recognises the message selector.

For each of the message selection mechanisms described in section 2.3.3.2, specific behaviour object slot(s) may be selected for handling. If more than one slot is specified, then the scanning order is given by this specification - the first behaviour object which recognises the message selector will handle it.

2.3.4 The virtual connector translation table

The connector names used in the behaviour descriptions are virtual names. This allows more flexibility in placing behaviour objects in different environments. A virtual connector translation table placed within the interface shell is used to translate the virtual connectors names into the real connectors placed in the basic object. In addition to this, the table may also contain references to local behaviour object slots. For this reason, names of real connectors and behaviour object slots must be unique within a basic model object.

Behaviour objects can poll input buffers for the presence of specific messages. This is done through the virtual connector translation table.

The translation table is used to decouple behaviour from the actual communication architecture. Sending a message to a virtual connector may route this to a local or remote behaviour object:

- *Local behaviour objects* are reached by referring to virtual connector names which stand for a local behaviour slot. Messages sent to such a virtual connector never leave the basic model object.
- *Remote behaviour objects* are reached by referring to virtual connector names which stand for a real connector. Messages sent to such a virtual connector are sent across communication channel attached to the referred real connector.

The sender does not know where the receiver is located. Figure 2.3.4-1 shows how the behaviour slots and virtual connector table relate. This figure also shows that the message selector/manipulator does not use the translation table. The message selector/manipulator directly accesses the connectors and slots.

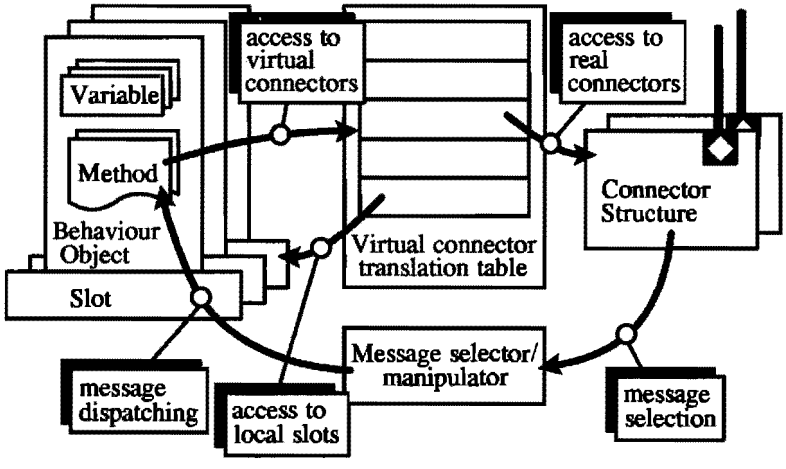


Figure 2.3.4-1: Use of the virtual connector translation table

When creating a basic model object, all the behaviour slot names are stored in the table. These names refer to their respective slots (a one-to-one translation). Each time a connector is placed in a basic model object, it's name is added to the table too, again translating one-to-one (unless this gives a name conflict). The table may be updated by the designer. Virtual names may be added, changed or removed. The table may contain several references to the same connector or behaviour slot. This is especially helpful for

connectors, as it indicates that a single channel provides communication facilities to different virtual objects.

The behaviour when a message is sent to a local behaviour object is relatively simple. The sending method is stalled until the started method has finished operation - the started method should be seen as a simple subroutine. This must be done because there can only be a single active method within a basic model object. The processing priority during the handling of the 'subroutine' is determined by the message priority assigned by the sending method. Because the default message priority is the processing priority, the processing priority does not change unless specific action is taken by the sender.

Note that the message selector/manipulator uses the real slot names when selecting behaviour objects. The virtual connector translation table is only used for messages sent by the behaviour objects.

2.3.5 The output buffer

Each communication channel connector may contain an individual output buffer. Without this buffer, a sending method blocks at the output action when the communication channel is occupied. With the buffer inserted, the messages are held in the buffer until the channel can transfer them. The sending method only blocks when the buffer is completely filled.

Like the input buffer described in section 2.3.2, the output buffer can be set to any length between zero and virtually infinite. The former is default and indicates 'no output buffer'. As with the input buffer, straight First-In-First-Out or priority FIFO message sorting algorithms can be selected (the latter is default).

Communication channel connectors are inherently bidirectional. Messages sent by a connector are not received by the same connector's input structure.

The behaviour of the output buffer can be explained by looking at figure 2.3.5-1. The processing core sends messages through the virtual connector translation table to the communication channel connectors. The messages are held in an output message holding store. Messages remain there until placed in the output buffer. If no buffer is used, messages remain in the output holding store until the message transfer is started on the channel. During the time that a message remains in the output store, the processing core is blocked.

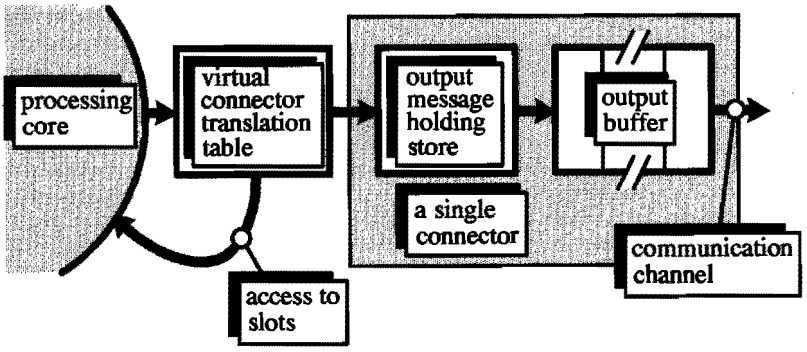


Figure 2.3.5-1: Conceptual diagram of output message handling

Seen from the channel, messages are presented for transfer by the head of the output buffer or (if no buffer is used) the output message holding store. When the channel is free or blocked, the channel arbitration algorithm selects between the messages offered by the attached connectors. The header of the selected message is distributed to the receivers for inspection. If there are no receivers blocking the message, the body of the message is transferred. Upon completion of the transfer, the message is removed from the output buffer or holding store. The latter will un-block the processing core again. If the message was removed from the output buffer and the holding store was filled, the message in the store is immediately moved to the output buffer. This will also un-block the waiting behaviour method.

The message selected by the channel's arbitration algorithm may change while the channel is blocked:

- If the output buffer is operated in priority FIFO mode, sending a high priority message may place this message at the head of the buffer immediately. This new message then becomes open to selection.

- A message may be sent with a transmission start timeout specified. If the transmission has not started when this timeout expires, the message is redrawn from the holding store or output buffer. If this message was at the head of the output buffer, the next message in the buffer becomes open to selection.

In both cases a new message is offered to the channel. This causes the channel arbitration algorithm to be run again. A new message is selected and the corresponding header is distributed for inspection by the receivers.

A message cannot be redrawn once body transmission has started. This is the main reason why the transmission timeout is related to the start of the message body transmission. The maximum blocking time for a sending processing core is the transmission timeout time plus the specified transmission time. This can only happen when no buffer is used.

When a buffer is used, the maximum blocking time is the timeout time. Blocking only occurs while the buffer is full. Once space becomes available, the message is placed in the buffer and the processing core is un-blocked.

The processing core may obtain the following three pieces of information regarding each output structure:

- *The channel status* - 'free', 'blocked' or 'transferring'.
- *Buffer fill level and buffer depth.* The latter is needed because this parameter is not under core control and may be needed for fill level control algorithms.
- *Whether or not specific messages are waiting in the buffer.* Searching for messages starts at the head of the buffer and proceeds towards the tail. A message which is not yet in transfer may be removed from the buffer. This has the same consequences as a timeout on that message.

2.4 The processing core

Actual data processing and data storage within a basic model object is performed by the processing core. The actual behaviour of the object is defined by a set of *'behaviour objects'*, which are stored in named *'slots'* within the core. Behaviour objects can be removed or replaced by others during system operation. This way, the overall behaviour of the basic model object can change dynamically.

Defining the behaviour of a basic model object as the 'sum' of the behaviours of the individual behaviour objects leads to a form of *'multiple inheritance'*:

- *True multiple inheritance* allows any object to inherit behaviour from several other object classes. The inheritance links between these classes are static.
- *In the basic model*, only the basic model objects inherit (combine) their behaviour from multiple behaviour object classes. The behaviour objects themselves use single inheritance. The basic model uses non-fixed ('dynamic') links, which is why we call this inheritance scheme *'dynamic multiple inheritance'*.

2.4.1 Priorities and interrupts

The processing core behaves like a processor with interrupt capability. Only a single message can be handled at a time, which is always done at a specific *'processing priority level'*. When a message is accepted by the core, the processing priority is set to the message's priority. While handling the message, the processing priority can be checked and manipulated by the processing core itself. Messages sent while handling a message by default receive the processing priority as message priority.

Handling a message is interrupted when the message selector offers a message with a higher priority than the processing priority. Handling the interrupted message is resumed when the high priority message has been handled. At that time, the original processing priority is restored too.

The processing core performs actual processing in 'zero' simulated time. The core interleaves moments of activity with waiting periods to create the illusion of being busy. All interrupts will happen during these waiting periods. An interrupt will temporarily halt the waiting period timer. This is necessary to prevent an interrupt from reducing the 'processing time' of another message.

The processing priority can be varied over a restricted range. At the lowest priority, *any* message will interrupt processing. This priority can be used for 'background processing', which is done when there is nothing else to do. At the highest priority, *no* message is capable of interrupting. This level is automatically selected when initialising a behaviour object, and can also be used to provide access protection (record locking in a database, for instance).

The processing core does not provide true multitasking capabilities. There were several reasons not to do so:

- *The separate basic model objects already behave as truly concurrent processes.* Adding another process does not slow down the system, something which happens in a real multitasking environment.

- *Multitasking is normally not needed in an Object-Oriented environment.* Multitasking is used to allow several large programs to run at the same computer. Switching between these programs is normally done when the running program must wait for input. An Object-Oriented environment is *event driven*. Methods are started automatically when something happens.

2.4.2 The behaviour object

The actual behaviour of a basic model object is defined by the behaviour objects which are stored inside it's processing core. The message selector/manipulator entity in the interface shell can send messages to any of these behaviour objects - if needed, to specific elements.

The behaviour objects are stored in named *'slots'* within the processing core. They themselves may contain a set of user-defined local variables, which remain intact during the lifetime of the behaviour object. A set of *'methods'* define the actions to be taken when a message must be handled. Each method is identified by a name which should match the message selector which starts the method.

2.4.2.1 Behaviour object variables ('state')

The state of a behaviour object is stored in a set of user-defined variables. These variables may be manipulated by the behaviour object's methods, either by assignment or sending messages to them. They cannot be accessed from outside the basic model object (they are 'hidden' from the outside world). External access to the variables is completely controlled by the behaviour object's methods. The input filters and message selector/manipulator have 'read only' access to all variables.

The state variables remain intact between the handling of the messages. They form the 'long term' memory of a behaviour object.

Type checking of behaviour object variables is possible by specifying the class to which a variable should belong. This class may be any of the standard system classes (like Integer or String) or a user-defined class. When assigning to a typed variable, the value should either belong to this type, or be 'no value'. When the type of a variable indicates a collection of some kind (Array or Set, for instance), the type of the objects stored in this collection can also be specified.

The default initialisation value for each of the variables is 'no value'. An initialisation method may store other values when the system is reset or a behaviour entity is created.

2.4.2.2 The behaviour descriptions ('methods')

The actual operations of a behaviour object are defined by a set of *'methods'*. Each of these methods is known by a symbolic name, which must be unique for a behaviour object. When a message is received by the behaviour object, the method which matches it's name against the message selector is started automatically.

The methods themselves are located in the class definition from which the behaviour object is an 'instance'.

When a method is changed, all behaviour objects belonging to the same class change their behaviour.

Behaviour objects are based upon Smalltalk objects. Using standard Smalltalk constructs, conditional execution and loops can be defined. The pseudo-variable *'self'* refers to the behaviour object itself. Behaviour objects add the pseudo-variable *'shell'* which provides access to the basic model object in which the behaviour object is stored. This is necessary to access and change the processing priority and to perform imperative mode operations like polling connectors for the reception of specific messages.

2.5 Concurrency and timing

The basic model allows true concurrency simulation. Timing aspects can be simulated. Timing simulation allows analysing whether or not a designed system meets some externally imposed timing constraints. This capability is needed for real-time system design.

We have already described several timing aspects, but these are summarised and completed in this section. There are three basic timing specifications which can be given:

- *Processing delays*
- *Data transfer time*
- *Timeouts*

These are described in more detail below.

Processing delays

In a real environment, the actual processing of data - performing operations - takes time. Even storing and retrieving data takes time, depending on the complexity of the data structure and the presence of dynamic-sized structures (for which space must be found).

Like in most simulators, this is simulated by interleaving bursts of processing (done in 'zero time') with periods of waiting ([**fra77**]). This creates the appearance that the processing is done during the specified waiting time. In the basic model, sending messages is seen as normal processing. Sending a message which requires an immediate reply automatically halts the processing core until the reply message is received.

Within the processing core of a basic model object, an interrupt caused by the reception of a high priority message may postpone handling another message. Due to the fact that actual operations take no (simulated) time at all, these interrupts *must* always come during periods of waiting (for reply messages or a specified time). This means that an actual processing burst is automatically protected against interrupts.

Data transfer time

Sending data across a communication channel takes time. In the basic model, a channel can transfer only a single message at a time. This means that messages have to compete for the channel - introducing extra waiting time. The actual transfer time can be specified when the message is created by the processing core.

Timeouts

Messages have to compete for the data channels. A busy channel introduces a waiting time for the messages, especially for those with a low priority. To limit this waiting time, a timeout may be specified by the processing core when creating the message. If the message transfer has not started when this timeout expires, a piece of user-defined processing is started to handle the timeout. This will normally invoke the setting of a flag or re-sending the message at a higher priority. The processing priority during timeout handling is automatically set to just below the 'no interrupts allowed' priority.

When a request message is sent, a different timeout can be specified. If reception of the result message has not started when this timeout occurs, a timeout just like the one described above will be generated.

2.6 System reset and initialisation

When the system is reset, it must be brought to a fixed default state. Following this, initialisation must be done before actual operations can occur. The default reset state is system defined, while initialisation is user-defined.

When the system is reset, the following happens:

- *All input and output buffers are cleared.* All messages present in the system are lost.

- *All behaviour objects are removed.* System state stored in these objects is completely lost.

  Individual behaviour slots may be protected from clearing. System state stored in these behaviour slots remains intact during system reset.

- *The first slot within each basic model object is loaded with an instance of a designer defined class.* This new behaviour definition object is subsequently sent the message **'initialize'**. While handling this message, it may create and store behaviour objects in other slots.

2.7 Object-Oriented aspects

The modelling system is implemented in Smalltalk, which is a fully Object-Oriented design environment. This does not imply that the model is fully Object-Oriented too. The following list describes the Object-Oriented'ness' of the major basic model elements:

- *The interface shell* does not fit in the Object-Oriented line of thought. Although messages are handled by it, no inheritance is present. For now, each shell is a unique element, although it may be a (modified) copy of another one. In the next chapter, we will see that it is possible to have a set of basic model objects which share exactly the same shell architecture (a *'multiple'*).

- *The processing core* is not Object-Oriented with respect to the behaviour slots. These are specified with the interface shell, and therefore do not inherit behaviour. This is not necessary either, because the slots themselves do not exhibit behaviour of their own.

- *The behaviour objects* are fully Object-Oriented. They are instances of user-defined classes, which have the same capabilities as normal Smalltalk classes. The only difference is that their methods are specified in an 'extended Smalltalk' language, which is translated into standard Smalltalk before compiling. They allow single inheritance, just like Smalltalk classes.

- *Dynamic multiple inheritance behaviour* is simulated by the shell, and depends upon the behaviour of the behaviour objects. The behaviour object classes are highly re-usable, and may need even less adapting and sub-classing than normal Smalltalk classes. This is due to the fact that a basic model object's behaviour is the sum of the behaviours of the behaviour objects. Several existing behaviour objects can be combined with a new element. This new element coordinates their actions and modifies their behaviour by 'preprocessing' and 'postprocessing' messages.

2.8 Mapping of other models on this model

The model proposed here is a general purpose system description method. Checking whether or not it is generally applicable can be done by comparing this model to other existing models. We will only indicate how the basic concepts of the other model can be simulated in our system.

We will not try to give a comprehensive list of modelling techniques. The two examples given in the next sections have been chosen because they have relatively complex semantics. Models like *'Communicating Sequential Processes'* (**[hoa78]**) and *Petri Nets* (**[pet81]**) are based on more mathematical grounds. They have simpler semantics and are therefore easier to simulate using objects.

Mapping the basic model onto other models can also be done. This is a far from trivial conversion because of the high abstraction level of the basic model. Being able to convert a system described with the basic model into a mathematically based description allows the system to be analysed with mathematical tools. The problem remains that even relatively simple abstract systems may be converted into very complex mathematical models. The complexity of these models often makes it impossible (or impractical) to use mathematical analysis tools.

2.8.1 The CCITT Specification and Description Language (SDL)

SDL ([cci87]) builds a system out of communicating processes. Communication takes place using *'signals'*, which are sent across predefined communication channels. The processes are described using a state machine-like syntax. State transitions are caused by the reception of signals. The current state, signal type and 'enabling conditions' determine *which* transition is made. Data is stored by processes, transported as signal parameter and processed during state transitions.

Mapping this model on the basic model is relatively simple. Basic objects, communication channels and messages take the place of processes, channels and signals respectively. Building an Object-Oriented state machine is trivial. Just store a state in a variable and use this variable to determine what to do with a received message. Part of this message handling will be setting a new state in the variable, preparing for the next message reception. The enabling conditions can be described by the message filters and/or the message selector.

There are some basic differences between the basic model and SDL which must be bridged some way or another:

- *Each SDL process has a single (infinite) signal buffer*, while the basic model uses separate buffers for each channel. The SDL buffering method can be simulated in the basic model by routing all incoming messages through a single connector. An alternative is to use non-channel specific message selection, which treats all input buffers as a single storage pool for messages.

- *Signals received by a SDL process in a state in which these signals are not expected are lost*, unless a specific 'save signal' construct is used. In the basic model, we try not to loose messages - we even stop simulation when this happens. Loosing a message can be easily simulated by receiving the message, then doing nothing with it. This can be performed by the filters, which can 'absorb' a message. When a message is handled by the processing core, being in a specific state may be a reason not to do anything with it.

SDL allows processes to be created and removed. While a basic model object does not allow this, the *'multiple object'* extension described in section 3.1 is an almost exact match for this functionality. It allows identical basic objects to be created and destroyed during the lifetime of the system. Like in SDL, these objects are all attached to the same set of communication channels. Creating a basic object is done by sending a message to a special 'management' connector. Selection between identical objects can be done either by a user-defined or system-defined identifier. Destroying a basic object can be done in several ways, the 'suicide' which an SDL process can do is one of them.

2.8.2 The Hatley/Pirbhai and Ward/Mellor models

The basic models used by Hatley/Pirbhai ([**hat87**]) and Ward/Mellor ([**war85**]) are both based on the work done by DeMarco ([**dem78**]). They differ more in the application of their models than in the modelling techniques themselves. We have chosen to provide a description of the mapping of the Hatley/Pirbhai model onto the basic model. Mapping the Ward/Mellor model onto the basic model follows the same methodology.

An important remark up-front: The Hatley/Pirbhai model does not require the processing to be described in an executable language. The *'Structured English'* they use *is not a programming language*. Unless a more rigid specification language is used, their model cannot be simulated.

The Hatley/Pirbhai *'requirements model'* uses separate processes and data stores. These are interconnected by *'data flows'* which may carry and route multiple items of information. Processes perform data manipulation when all the data items they require are present on the incoming data flows, generating data items on the output data flows. Data can be stored in the data stores, which normally have destructive write and non-destructive read (each item separately). A data store can store a subset of the information present on the input data flows.

Control of the Hatley/Pirbhai 'requirements model' is performed by a *'control model'*. This uses control flows, control stores and state machine-like descriptions of the actual control algorithms. The control flows behave almost exactly like the data flows. Control stores store events in a FIFO fashion. Control flows emanating from a process indicate data conditions, those pointing into a process can enable or disable the process. When disabled, a process will not handle data nor produce output, even when the necessary input data is present. Multiple processes may be collapsed in a 'transaction centre', which can perform multiple functions selected by the control flow entering it. The state machines are non-clocked, they make a state transition immediately when all conditions necessary for this transition hold.

Mapping the Hatley/Pirbhai model upon our basic model is done by replacing all processes, stores and control processes by basic model objects. Data and control flows are replaced by message channels. Each data item is represented by a separate message to indicate its presence and value. A separate message is used to indicate that a data item is not present on a flow anymore.

Simulating the continuous processes is done by sending a message each time a data item changes or is removed from a flow. Routing the data items is done by the filters, which determine the messages received by an object. Each time a message is received, a table of 'current' data items can be updated, and new messages generated from this table.

A data store mapped onto a basic model object is a very simple process. It relays data item change messages, but 'swallows' data item removed messages. Actual data storage is done by the processes, which each keep a copy of the data item. A control store simply buffers control event messages in a FIFO, removing a message each time the attached state machine 'consumes' one of them.

A Hatley/Pirbhai control process can be simulated just like a normal process, extended with a state variable. Each time a control flow changes, a table of 'current' control flows is updated. Following this, the stored state and control flows are evaluated, control flow messages are sent and a new state is entered.

Control flows entering a process are handled just like data flows. When a 'disable' message is received, messages are sent to invalidate data items. When an 'enable' message is received, stored input data items are evaluated and new data item messages may be generated.

The Hatley/Pirbhai *'architecture model'* basically uses the same operational model as the requirements model. The only difference lies in the system configuration. This difference is necessitated by the assignment of physical implementations to communication channels and operational modules.

Mapping the Hatley/Pirbhai model entities onto basic model objects is a straightforward task. 'Framework' objects can be designed which model the basic Hatley/Pirbhai entities. These can be subclassed to provide the necessary behaviour originally described in structured English. *The result will be an executable Hatley/Pirbhai model.*

3. The Extensions to the Basic Model

The basic model objects described in the previous chapter are not enough to model general information processing systems. Several important modelling techniques cannot be performed with the basic model objects alone:

- Decomposition of a complex system element into other less complex elements.
- Allowing a variable number of a specific system element during a system's lifetime.
- System elements which 'travel' through the system - their *'is contained in'* relation is not a fixed one.
- Continuous, non-synchronised communication methods.

Solutions to these problems are described in this chapter.

3.1 Groups

A complete system is composed out of a multitude of objects. Manipulating an large amount of basic model objects at once is very difficult for a designer. A way out of this problem is to group objects together and treat this group as a single unity.

Looking at it from the other side, the system is decomposed into a small amount of complex objects. These are again decomposed into less complex objects. At any level, 'simple-enough' objects are described by a single basic model object.

The latter way of decomposing complex objects into simpler ones is the preferred way of designing a system. It is generally referred to as the *'top-down'* design method:

1) *The behaviour of a complex object is described in rough terms.*
2) *This complex object is decomposed into a set of simpler objects.*
3) *For each of these a more detailed behaviour description is given.* The combined behaviour of the simpler objects should adhere to the original behaviour.

This process repeats itself until the desired level of detail is reached.

During the decomposition of a complex object, objects with which it communicates may remain intact. These may be used as test environment for the decomposed object.

Our modelling technique represents objects and their communication channels on *'Entity Communication Diagrams'*. Decomposing an object then becomes replacing it by a lower-level Entity Communication Diagram, which itself is represented by a symbol at the position of the replaced object. Communication channels which were connected to the original object remain connected to the symbol of the Entity Communication Diagram. They emanate on the lower-level Entity Communication Diagram through *'super connectors'*. These are modelling elements which have no other function than to signal that a communication channel is connected to another communication channel outside the Entity Communication Diagram. This conversion is depicted in figure 3.1-1: The basic object 'TOPENTITY' is decomposed into two basic objects named 'SUB1' and 'SUB2'. These are both placed within the Entity Communication Diagram which replaces the original basic object.

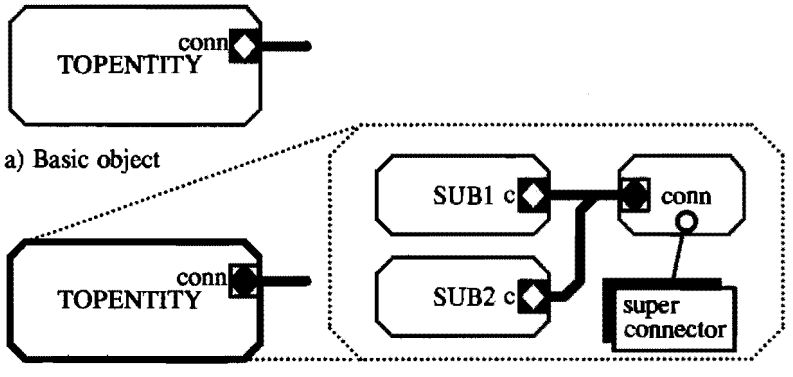


Figure 3.1-1: Replacement of a basic object by a group

A lower-level Entity Communication Diagram should be seen as a single object placed within a high-level Entity Communication Diagram. Because the symbol represents a group of other objects, we have given it the name *'grouping entity'* or simply *'a group'*.

Neither a grouping entity, nor a super-connector can do any processing. They are merely a technique to symbolise the decomposition of a complex object into less complex objects. The same hierarchy introducing technique is used by other graphically oriented system modelling methodologies.

The behaviour of a basic model object is characterised by the messages it receives and sends. The internal operations are 'hidden' from the outside world. A grouping entity which replaces a basic model object may send and receive the same messages, but have a different method of handling them. A basic model object behaves like a single process. A grouping entity contains several basic objects, which perform true

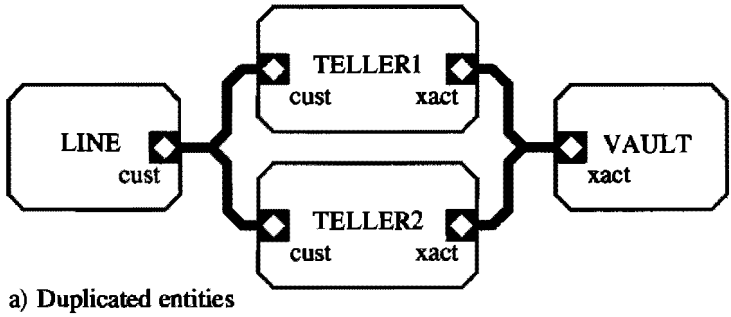
concurrent operations. This may lead to slightly different timing behaviour of a grouping entity.

In the single entity a message may have to wait for processing. Within a grouping entity it may be handled much earlier because there are multiple processes available. This capability can be used to gain performance by parallelising or pipelining a system at a high level of abstraction. This technique is used during the high-level system architecture design phase, described in chapter 5.

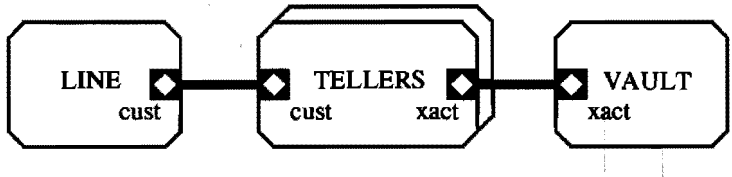
3.2 Multiples

Systems often contain elements which are copies of eachother. They exhibit (almost) the same behaviour and share connections to other entities. Examples are semaphores and mailboxes in an operating system, tellers in a bank or drilling machines in a workshop.

One can model these by simply copying a single prototype. These copies are attached to the shared channels as shown in figure 3.2-1a. With the channel routing depicted there, the duplicates cannot communicate amongst eachother.



a) Duplicated entities



b) Using a "Multiple" as replacement for the duplicates

Figure 3.2-1: Duplication versus multiple

Figure 3.2-1b shows a *'multiple'* which replaces the duplicated entities. The 'TELLERS' entity contains two separate tellers, each containing an identical communication shell. The actual contents of the buffers and the stored behaviour objects may differ. Using an identical communication shell provides each of the entities with the same interface to entities outside the multiple.

Each of the entities in a multiple has an identical set of named slots to store behaviour objects. Behaviour objects stored in identically named slots within individual entities normally belong to the same class. This provides all entities within the multiple with the same behaviour. Entities in a multiple behave differently when identically named slots contain behaviour objects belonging to different classes. This modelling capability is sometimes very handy. Some tellers may be used for depositing money, while others are for borrowing money.

The shell parameter settings in each entity of a multiple are the same. This means that the filter parameters are equal too. Each entity in a multiple will be offered the incoming messages. In order not to have every entity receiving all messages, these must be 'addressed'. Within the processing core, a unique 'address' of some kind must be stored (the teller number, for instance). Incoming messages must carry a matching address, which is checked by the message filters. Multiple addresses may be used, as well as 'group addresses'. The latter allow a subset of the entities in a multiple to receive a message.

The addresses must be assigned one way or another. Addresses can be assigned by the designer when the amount of entities in the multiple is fixed. This becomes different when the amount of entities varies during the lifetime of the system. The next section describes this situation.

3.2.1 Dynamic multiples

A multiple where the amount of entities changes during the lifetime of the system is called *'dynamic multiple'*. In a dynamic multiple, entities can be created or removed whenever necessary.

The amount of entities can vary from zero to an upper bound set by the designer. Upon system reset, all entities are removed. The *'management connector'* described in the next section is capable of creating an initial set of entities. An example of such an initial entity is the 'system mailbox' in the iRMX™ operating system (**[int84]**). This is a normal mailbox used for communication with the operating system, which is always available.

The difference between a normal and a dynamic multiple is the absence respectively presence of a management connector.

3.2.1.1 The management connector

Managing the set of entities in the multiple is a process which should be seen in separation from the actual functionality of these entities themselves. The management functions are provided by the *'management'* connector. These functions are available even when the multiple is empty. Figure 3.2.1.1-1 gives an example of the management connector (the 'M').

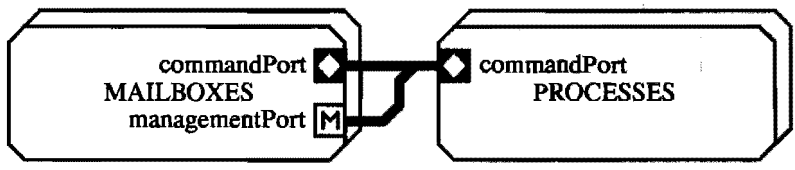


Figure 3.2.1.1-1: Example of a management connector

The management connector is actually a basic model object which is placed within the multiple. This basic model object can create and delete entities in the multiple. It also has access to the slots of the entities present in the multiple. A default behaviour object is included which can handle elementary entity creation and deletion messages. The designer may modify this behaviour object to enhance it's behaviour.

A management connector has limited access to the slots of an entity in the multiple. The contents of a slot can only be changed if it is not active (has no method in execution). The management connector may also send messages to a behaviour object stored in a slot. These are only accepted when the message priority is higher than the processing priority of that entity's core.

The management connector plays an important role in the initialisation of a multiple. During system reset, the management connector's behaviour object is sent an **'initialize'** message. This allows the behaviour method to create and initialise an initial set of entities in the multiple. The default behaviour object does nothing in response to this message - the multiple remains empty.

3.2.1.2 Entity creation

When creating a new entity, at least one of the behaviour slots must be filled with a behaviour object. Otherwise, it cannot do any processing. The method used is the same as described in section 2.6 for basic object initialisation. Upon creation of an entity, an instance of a designer specified class is loaded in the first behaviour slot. This instance is subsequently initialised. During initialisation, the other slots may be loaded with entities and initialised.

The management connector responds to two basic entity creation messages, '**new**' and '**newID: <anyObject>**'. Both create a new basic object with shell parameters and settings as specified by the designer. Both load an instance of the specified class in the first behaviour slot of the new object. The initialisation messages for this instance differ between the two creation messages:

- The '**new**' message uses a simple '**initialize**' message. A system generated identifier for the created basic object shell is returned to the sender of '**new**'. This identifier can be used for subsequent addressing.

- The '**newID: <anyObject>**' message passes the given object to the created behaviour object with an '**initialize: <anyObject>**' message. By storing the given object within the behaviour object, it can be used for subsequent addressing.

3.2.1.3 Entity removal

There are two ways to remove an entity from a multiple. They have in common that the entity is removed completely, including shell. If a channel was blocked by this entity, blocking is removed. If the shell was offering a message for transmission, this message is redrawn from the channel. Messages which were already in transmission will be completed. Messages held in buffers will be lost.

By default, the management connector's behaviour object responds to a single basic entity removal message: '**remove: <identifier>**'. The given identifier must match the system generated identifier for one of the basic object shells present in the multiple.

An entity may also remove itself from a multiple. This is done when a behaviour method sends the message '**remove**' to the pseudo-variable '**shell**'.

A dynamic multiple entity may be emptied by sending the message '**reset**' to the management connector. Following this, the standard '**initialize**' message may be sent to perform re-initialisation.

3.3 Multiple groups

A multiple group entity combines the functions of a grouping entity (grouping entities together on a lower-level Entity Communication Diagram) and a multiple (providing a set of identical basic model objects). It contains a set of identical grouping entities. As with normal multiples, dynamic multiple groups can be indicated by adding a management connector as described in section 3.2.1.1. This allows complete Entity

Communication Diagrams to be added to- and removed from the system during run-time.

Default initialisation of a newly created group follows system initialisation methods as described in section 2.6. The first behaviour slot of the basic object entities is loaded with an instance of a designer defined class. These instances are then sent initialisation messages, with as default a simple '**initialize**' message.

The management connector responds to two basic group creation messages, '**new**' and '**newID: <anyObject> for: <anArrayOfStrings>**'. Both create a complete grouping entity with contents as specified by the designer. For each of the basic objects in it, an instance of a designer specified class is loaded in the first behaviour slot. The initialisation messages for these instances differ between the two creation messages:

- '**new**' uses simple '**initialize**' messages. A system generated identifier for the created grouping entity is returned to the sender of '**new**'. This identifier can be used for subsequent addressing.

- '**newID: <anyObject> for <anArrayOfStrings>**' uses '**initialize: <anyObject>**' messages for the behaviour objects stored in basic objects whose names appear in the given Array of Strings. By storing the given object within these behaviour objects, it can be used for subsequent addressing. All other behaviour objects are initialised with a simple '**initialize**' message.

Like normal multiples, multiple groups have two methods for removing a group contained within them. Upon removal, anything contained within the group is lost, channel blocking is removed and offered messages are redrawn.

By default, the management connector's behaviour object responds to a single grouping entity removal message: '**remove: <identifier>**'. The given identifier must match the system generated identifier for one of the groups in the multiple.

An entity may also remove it's group from a multiple. This is done when a behaviour method sends the message '**removeGroup**' to the pseudo-variable '**shell**'. When dynamic multiple groups are nested, '**removeGroup**' will only remove the innermost nested group as seen from this entity.

A dynamic multiple group may be emptied by sending the message '**reset**' to the management connector. Following this, the standard '**initialize**' message may be sent to do any re-initialisation necessary.

3.4 Continuous data transfers

Message transfers carry an inherent synchronization aspect. Not considering transmission and buffering delays, a message received now was sent relatively recently. This provides extra information regarding the state of the sending object.

Another way of communication lets a sender produce a continuous value. Receivers may sample this value at any time they like. The obvious (and often given) example is the analog thermometer. It hangs outside, continuously displaying a temperature.

Ultimately, all forms of communication are based on continuous values. The value is polled regularly by the receiver. Changes in the value may indicate an event. When this has been detected, other values may be polled to determine *which* event has happened. It is also possible to insert a fixed time delay and poll the same information channel again. This assumes the sender has then placed a value on it which provides more information.

The model provides a way to specify continuous data transfers. A basic model object may be fitted with continuous data input and output connectors:

- *Continuous data output connectors* are accessed through the virtual connection translation table described in section 2.3.4. They can be directly assigned a value, which places this value on the channel. Assigning *'no value'* to the virtual connector name has the same function as placing a hardware output into *'three-state'*. The output connector always remembers the last value assigned to it. When the name of the virtual continuous data output connector is used in an expression, it provides the current channel value. Connector 'temp' in figure 3.4-1 is an example of a continuous data output.

- *Continuous data input connectors* are also accessed through the virtual connection translation table. These cannot be assigned a value, only used in 'read-only' fashion. When used in an expression, they provide the current channel value. Connector 'voltage' in figure 3.4-1 is an example of a continuous data input connector.

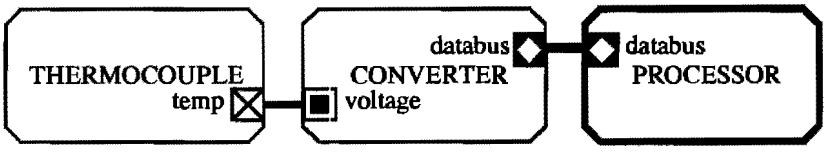


Figure 3.4-1: Example of continuous data transfer

These connectors are connected by special communication channels. These continuous data channels have some specific characteristics:

- *Continuous data channels are 'typed'.* The designer must specify a class (like 'Integer', 'String' or 'Array of Boolean') for transfer across the continuous data channel. The values which are transferred must be instances of this class or the *'no value'* object.

- *Continuous data channels provide no routing.* They behave like a hardware 'data bus'. A value placed on the channel by an output is immediately distributed to all connectors attached to the channel. The model does not introduce a time delay for this distribution.

- *Continuous data channels are multidirectional.* Multiple outputs and inputs may be connected to a channel. Each output may send values to any input. Only a single output may be active at a time. When no- or more than one output sends a value, the channel distributes *'no value'*. The designer may specify that sending more than one value at a time is an illegal operation.

3.5 Travelling Objects

Until now, we have only considered system elements which have a fixed 'position' in the system. The design elements described up to now define the 'static' system structure. Even dynamic multiples cannot move from their location on an Entity Communication Diagram.

In most systems there are objects which are not fixed in position. They can 'move around' like people walking through a building. There are two ways to control the operations and 'transfers' of travelling objects:

- *The passive travelling object.* Such an object can be seen as a simple data store which is shuffled around and operated upon. Data packets in a Local Area Network can be modelled as passive travelling objects. They are transferred between basic objects based upon their destination address until they reach their destination.

- *The active travelling object.* An active travelling decides for itself which actions it has to perform. If a specific action cannot be performed at the location where the object resides, it has to find out where to go and move itself overthere. Active travelling objects can be used to model processes running in a multiprocessor environment. These processes can move themselves to less busy processors when they receive to little processing time.

Travelling objects should actually *move* (and not be *copied*) between basic model objects. A travelling object which is located within several basic objects at the same time can receive a message within one object, responding by sending a message from within another object. *This would mean that a completely invisible connection exists between those objects.* To remove this possibility, we introduce the following general rule regarding travelling objects:

A travelling object may only be located within one basic object at a time.

In general, one should be very careful with data transferred between objects. If this is a complex data structure, *copies* should be sent instead of the original. Otherwise, updating the data structure within one object would also alter the data structure in another object - again a means for illegal communication.

In the real world, sending a file through a computer connection creates a copy of the file at the receiving end. It is then up to the sender whether or not the original file should be deleted. Unless a rigid update protocol is used, updates at either end are not reflected in the file at the other end.

The modelling system described in the previous sections provides all capabilities needed to model travelling objects. They can be modelled as behaviour objects which can be stored directly in a behaviour slot. They can also be stored within a data structure located in another behaviour entity. A travelling object may be transferred across a communication channel as message parameter.

The modelling system does not provide automatic protection against storing an object within multiple basic simulation entities. We think this is something which the designer must accomplish, because it is very difficult to remove something from a complex data structure automatically. Within a data structure, there may be several references to the object, all these references have to be removed 'manually'.

It is possible to check whether or not exactly the same objects are stored within more than one basic simulation entity. If this is an undirected search (no specification for the object to search for), it will take a large amount of time - something which should not be done on a regular basis.

3.6 Summary and conclusions

The model introduced in the previous chapters allows complex systems to be modelled as a set of communicating objects. The basic objects in this model operate concurrently. They communicate with- and synchronise to other objects by sending and receiving messages across communication channels.

Each basic object contains a *'communication shell'* which filters, buffers and selects messages before transferring them to the actual *'processing core'*. This core contains a set of *'behaviour objects'* which define the data storage and actual operations (*'methods'*) performed by the basic object. Behaviour objects can be replaced by other behaviour objects which changes the overall behaviour of a basic object.

Methods manipulate the data stored within the behaviour object in which they are defined. They can send messages to other basic objects across the communication channels. These outgoing messages may be buffered in the communication shell. Methods are also capable of sending messages to other behaviour objects in the same processing core. Methods send messages to virtual connectors which each represent either a real connector or local behaviour object. This allows the routing of messages to be changed without modifying behaviour objects.

Methods are started by the reception of a message which matches their name (*'reactive'* mode of operation). A priority system allows methods to be interrupted in order to handle a higher priority message. A running method may poll or wait for specific messages to be received by the communication shell (*'imperative'* mode of operation).

Messages may carry parameters which can be very complex objects in themselves. These *'travelling'* objects may be used to model system elements which do not stay at a fixed location in the system. Travelling objects can be stored inside basic objects as a behaviour object. They can also be stored inside the data structure within a behaviour object.

An alternative means of communication between basic objects is provided by the *'continuous data channels'*. These provide non-synchronised communication between objects. Methods can place data on an output connector connected to such a channel. Other methods can read the value on a continuous data channel through an input connector.

A system structure is specified by placing objects and communication channels on *'Entity Communication Diagrams'*. Hierarchy is introduced in the system by grouping objects together in *'grouping entities'*. Each of these contain a lower level Entity Communication Diagram, which is denoted by a single symbol on another Entity Communication Diagram.

Groups of basic objects which have an identical communication shell may be denoted by a *'multiple'*. Each of these basic objects operate independently, they may contain different sets of behaviour objects. The amount of basic objects in the multiple may vary during the operation of the system when a *'multiple management connector'* is placed in the multiple. By sending messages to this connector, complete basic objects may be created and destroyed within the multiple.

Groups of identical Entity Communication Diagrams may be placed in a *'multiple group'*. As with the multiple, a multiple management connector may be used to create and destroy complete Entity Communication Diagrams during the system's operation.

The modelling technique described above is designed to be used with interactive tools. Setting up a design always starts with a bare and simple model. The designer gradually introduces the complexity needed to model the system under design. All bookkeeping and consistency checking functions are performed by the tools. The tools combine design and simulation, which gives the designer immediate feedback on his design actions.

The basic model uses dynamic multiple inheritance to describe behaviour as a combination of several other behaviours. Coupled with the re-usability which is inherent in Object-Oriented techniques, this allows system behaviour to be described at a very high abstraction level.

The model can be used during system behaviour analysis as well as architecture synthesis. The high abstraction level of the model and use of interactive tools allow very complex systems to be handled.

4. Object Oriented Analysis using the Model

This chapter will describe an Object Oriented Analysis method which uses the model described in chapters 2 and 3. This method is an extension of the methods as described by Shlear and Mellor ([shl88]) and Coad and Yourdon ([coa90]). As stated in section 1.1, analysing the system requirements should be seen as the first step in hardware/software system design. This chapter is an elaboration of the Object-Oriented Analysis method described in section 1.4.1. Chapter 5 outlines how the same model can be used for high-level system architecture design.

The main objective of the Object Oriented Analysis method is to analyse the system requirements and build a solid foundation on which the remaining design steps can be based.

The basic model and accompanying tools provide a way to describe, simulate and analyse the behaviour of the system under design at a very high level of abstraction.

The ordering of this chapter follows the sequence of steps which should be taken to do successful Object Oriented Analysis. Analysing a system with OOA should follow normal 'top-down' design procedures. The behaviour of major system elements and their interactions should be described and analysed first. In subsequent steps, the major system elements are decomposed and analysed in greater detail. This means that the OOA steps described in this chapter may be repeated several times before reaching the required level of detail.

4.1 Finding the Problem Domain Entities

The first step in doing Object Oriented Analysis is to find and name the so-called *'Problem Domain Entities'*. The 'problem domain' includes the system to be designed and environment with which the system communicates. Including the environment in the problem domain allows a 'real life' test setup to be created for the system under design. It is not required (nor the intention) to analyse and describe the environment at the same level of detail as the actual system.

The Problem Domain Entities are all the entities which form part of the problem domain for any 'reasonable' amount of time. Entities which are continuously available within the problem domain certainly fit this requirement. Entities stored within and

transferred between these continuously available entities also fulfill this requirement. Temporary variables used during a calculation should not be considered a Problem Domain Entity.

Finding Problem Domain Entities is actually quite simple. If there is a written problem statement, simply look for the nouns used there, and you have a reasonable start. Another approach is to do a brain-storming session, and write down all 'things' which are related to the problem.

A system built with OOA is a simulation of an abstract reality.

This means that you can also find the Problem Domain Entities by imagining which operations have to be executed to perform a specific system function. Any entity involved in these operations will become a Problem Domain Entity.

The chance of finding an entity which later turns out *not* to be a Problem Domain Entity is much smaller than overlooking an element which *is* a Problem Domain Entity. Neither of the two is a real problem. A non-Problem Domain Entity will simply be removed. Elements overlooked are in most cases buried deep in the system. They turn up when performing detailed analysis of a system element. At that time, they can be added without any problem. If they had been 'visible' outside the sub-part, they would not have been overlooked. It may be better to skip these buried Problem Domain Entities during the first analysis steps, because they provide too much level of detail.

The result of the search for Problem Domain Entities is a list of names. This list need not contain a structure, providing structures is something which is done in the next steps of OOA. An example list for a multiprocessor system could be the following:

"Elements of a multiprocessor system (first detail step):"	
Processes	(variable amount)
Processors	(fixed amount)
Semaphores	(variable amount)
External events	(fixed amount)
Mailboxes	(variable amount)

4.1.1 Layering the Problem Domain Entities

In essence, all Problem Domain Entities are stored within other Problem Domain Entities. The problem domain itself is the topmost Problem Domain Entity, containing all others within it. The first step following the finding of the Problem Domain Entities is to place them in the correct *'forms part of'* relationships to eachother. This relationship states which entities are contained in another entity, thereby providing the

system with a spatial structure. As we will see shortly, the *'forms part of'* relationship need not be a fixed one.

> The *'system context diagram'* used by Hatley and Pirbhai ([hat87]) makes a firm distinction between the actual system and the 'outside world'. This outside world is represented by *'terminators'* which are connected to the communication channels which originate from the system. The terminators generally stand for those elements in the outside world which have a direct connection with the system. They provide the system with stimuli and receive responses in return.
>
> The problem with the Hatley/Pirbhai model is that the terminators have no interaction amongst themselves. This makes it very difficult to build an accurate model of the environment in which to test the system. For Hatley/Pirbhai, this is not a problem as the system model needs not be simulated anyway.
>
> We avoid this problem by making the environment a part of the problem domain. This means that a model of the environment should be designed which can be used during system tests.

Static *'forms part of'* relationships are easy to model and understand. Grouping entities (as presented in section 3.1) can be used to subdivide a complex Problem Domain Entity into several lower-level Problem Domain Entities.

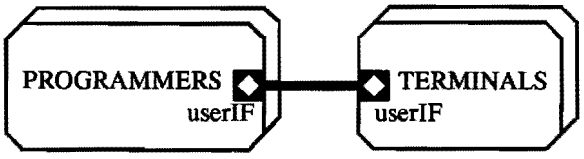
Multiple entities can be used when a fixed amount of a certain type of Problem Domain Entities form a part of another Problem Domain Entity. Each of the replicated entities has it's own internal variables, but is otherwise the same.

In general, the amount of entities in the multiple will change over time. If this time is larger than the life span of the system to design, then the amount can be considered a fixed value. The number of processors in a computer system is normally a fixed value, as it is not changed every day.

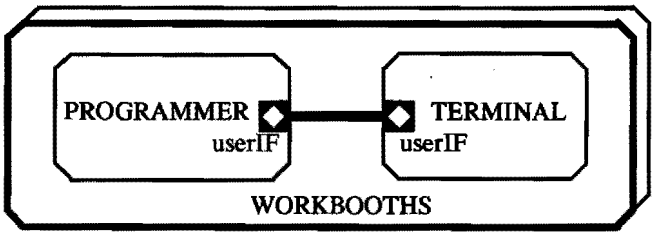
There should be a way to differentiate between the entities in a multiple. In the case of a fixed multiple, this must be done by the designer because the entities are created before the system starts operating. Each of the entities in a multiple has a different set of Problem Domain Entities stored inside of them. Their behaviour differs only because of these different Problem Domain Entities. Imagine what would happen if two processors on a network had the same address and were therefore indistinguishable.

Figure 4.1.1-1a shows the situation where entities in two multiples have a one-to-one relationship. For each entity in 'PROGRAMMERS' there is a corresponding entity in 'TERMINALS'. Modelled this way, all programmers share a communication channel towards their terminals. In such a situation, it is better to have a multiple group. Each of the groups within the multiple combines the related entities which were present in the separate multiples. This gives each of these relations a private communication channel. This situation is depicted in figure 4.1.1-1b. We will see later (during

architecture design), that it may be necessary to break up a multiple into several parallel multiples. This should not be done during system analysis, however.



a) A fixed one-on-one relation between multiples, channel shared



b) Using a "Multiple group" gives private communcation channels

Figure 4.1.1-1: Parallel multiples

Some Problem Domain Entities have non-fixed *'forms part of'* relationships. They move between Problem Domain Entities. An example of such a Problem Domain Entity is a process in a multiprocessor system. Such a process may be moved to another processor if it will receive more processing time there.

Modelling these 'travelling' Problem Domain Entities has been described in section 3.5. They can be stored as behaviour objects or within data structures which reside within a behaviour object. They 'travel' by being a parameter within a message.

4.1.2 Defining dynamic system structures

As stated in the previous section, multiples of Problem Domain Entities are almost always dynamic. A multiple is considered static if it does not change over the life-time of the analysed system. This means that there must be a way to model multiples which *do* change during the life-time of the system. The *dynamic* multiples described in section 3.2.1 are introduced for this purpose.

As with fixed multiples, the behaviour of all entities in a dynamic multiple is in principle the same. Their actual behaviour depends upon the Problem Domain Entities stored within them. Also, while present in the system, there must be a way to

differentiate between them. This is done by storing a *'tag'* value in each of the multiple entities. The tag attached to an entity should be known to any Problem Domain Entity which wants to send a message to it. The tag is assigned to an entity in a multiple during creation. Messages containing the tag are subsequently sent to those entities in the system which need to communicate with the newly created entity.

Within a Problem Domain Entity stored in a dynamic multiple, other Problem Domain Entities may be present. These can include other dynamic multiples, fixed multiples and/or single Problem Domain Entities. Each of these can be layered in itself. Whenever a Problem Domain Entity with internal structure is created, the internal Problem Domain Entities should be created too. The creation of such a complex entity may generate several tags. These all have to be stored and made available to the 'outside world'.

The 'travelling' Problem Domain Entities described in section 3.5 are another way to model dynamic system structures. They are created when necessary, and can then be stored in dynamic data structures (like Sets or Lists) within behaviour objects.

Figure 4.1.2-1 shows how the Problem Domain Entities of the multiprocessor system example relate to eachother. The processes have been modelled as travelling Problem Domain Entities which are stored in a Set structure within each of the processors.

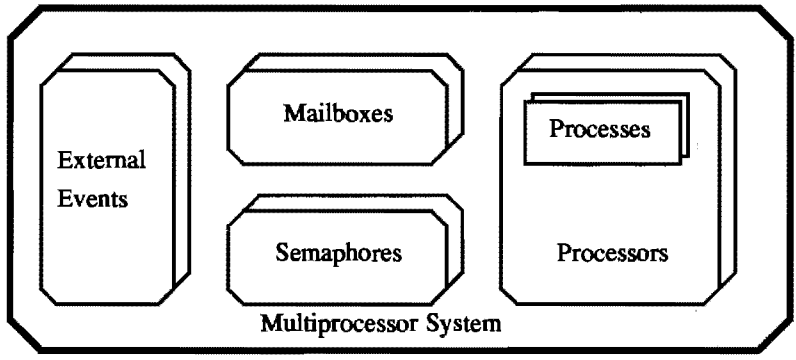


Figure 4.1.2-1: Example of a system structure

4.2 Classifying Problem Domain Entities

Aside from the *'is contained in'* relationships, Problem Domain Entities also form other kinds of relationships. A very important one is the *'is a kind of'* relationship.

It is possible to analyse each Problem Domain Entities behaviour completely on it's own. This would lead to a large amount of duplicated work. Detecting common behaviour and describing this behaviour once will lower the amount of work. Detection of common behaviour should be done as early as possible.

The actual implementation of *'is a kind of'* relationships is done during later analysis phase steps. The basic model described in chapter 2 uses behaviour objects to model actual basic model object behaviour. It is not necessary to model these behaviour objects in this stage of the OOA process. For now, it is sufficient to attach a note to a basic object stating that it's behaviour should be based upon some other behaviour(s).

4.2.1 Using inheritance

The simplest form of the *'is a kind of'* relationships are formed when a Problem Domain Entity adds new behaviour to- or modifies existing behaviour of another Problem Domain Entity. The new entity is said to *'inherit'* behaviour of the original entity.

Single inheritance as described above ties the Problem Domain Entity behaviours together in a tree-like hierarchy. The 'root' of this tree is the most general Problem Domain Entity behaviour available for a system. This behaviour includes functions which any Problem Domain Entity can use (like waiting for a predefined time or changing the processing priority).

Some Problem Domain Entity behaviours are never used on their own. A 'human' is always more than just a human, no actual Problem Domain Entities are created directly from the 'human' behaviour description. Such a behaviour is seen as a 'metaphor' to be built upon.

In Smalltalk ([**gol89**]), a behaviour template from which no actual objects are created is called a *'metaclass'*. A metaclass allows common object behaviour to be described at a central place. Objects which use this common behaviour are always created from subclasses of this metaclass. These subclasses extend the common behaviour and thereby create more specialised behaviour.

An example of inheritance can be found in the multiprocessor system when different types of semaphores are used. Processes waiting at semaphores may be ordered in different ways. The process restarted when the semaphore is triggered may be either the longest waiting or the one with the highest priority. The general semaphore behaviour can be modelled in a metaclass, the ordering methods can be implemented in subclasses of this metaclass. Actual semaphores are created from these subclasses.

4.2.2 Combining behaviour

A more complex form of the *'is a kind of'* relationships is found when a Problem Domain Entities behaviour is based on the behaviours of several other Problem Domain Entities. This form of behaviour linking is called *'multiple inheritance'*.

The basic model allows 'run-time' behaviour changes. In section 2.1.4 this property is called *'dynamic multiple inheritance'*. This is accomplished by storing behaviour objects within a basic entity. The interface shell within a basic entity is capable of automatically forwarding received messages to the correct behaviour entity. Behaviour may be changed by modifying the set of stored behaviour objects.

The mailboxes of the multiprocessor system example can use dynamic multiple inheritance to their advantage. A simple mailbox consists out of a queue to store messages and a semaphore to hold processes which are waiting for messages while the queue is empty. The semaphore can be a normal semaphore of any type (see the example at the end of the previous section). By changing semaphore types, the mailbox can be given different ordering methods for waiting processes.

4.2.3 Re-using previously defined behaviour

Until now, we assumed that the analyst builds a completely new system of Problem Domain Entities for each project. This would take a lot of time, even when inheritance is used. By re-using behaviour defined within other projects, a designer can save himself a lot of work.

Behaviour templates are stored in the design system's libraries and can be preserved for later projects. They can be exchanged between analysts and/or built by specialised firms ([**cox90**]).

4.3 Problem Domain Entity 'communicates with' relationships

The communication between the Problem Domain Entities in the system must be guided into the proper channels following the definition of the system structure:

- *Private 'point-to-point' connections* should be used to interconnect Problem Domain Entities which need to communicate without interference of other data sources.

- *'Bus'-like connections* may be used between groups of Problem Domain Entities which all have to communicate with eachother. Such a connection is a shared resource, which means that data sources may interfere with eachother (messages may be delayed because the channel is already in use).

- *Indirect connections* must be used when the sender does not know how the messages will be routed. This knowledge may be located in a *'routing'* Problem Domain Entity which receives messages and forwards them without changes. Multiple routing Problem Domain Entities may be needed to transfer a message from sender to receiver.

Communication channels can be connected to a multiple entity Problem Domain Entity. Messages sent to a multiple are presented to all the Problem Domain Entities present within it. The message filter within each of them must decide whether or not to handle the message. This decision can be based upon the message itself, the parameters of the message (tags!) and/or behaviour object variables. This means that a message may be handled by any subset of the Problem Domain Entities.

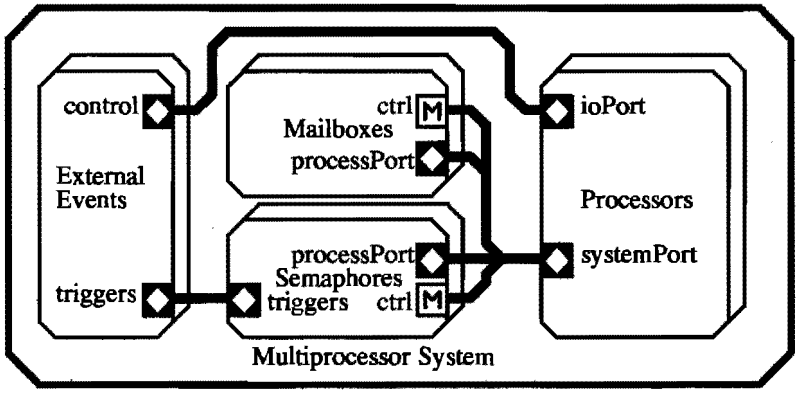


Figure 4.3-1: Communication channels in the multiprocessor system

Communication channels are specified graphically on an Entity Communication Diagram. Figure 4.3-1 shows the interconnections between the elements of the multiprocessor system. All processors have to communicate with the mailboxes and semaphores in the system. They use a shared bus to do so. The 'ctrl' connectors are used to create and remove mailboxes and semaphores. Private channels are used to control the external events and trigger semaphores. Controlling which semaphore is triggered by an external event can be done by sending messages through the semaphores. The addressed semaphore forwards this message to an external event, which makes this semaphore a 'routing' Problem Domain Entity. Note that the 'processes' are not visible in this Entity Communication Diagram - they are stored

inside the processors. The editing and simulation tools use other techniques to show and edit the contents of a basic model object.

4.4 Defining the system's operational aspects

The previous sections defined a system structure containing layered Problem Domain Entities and communication channels. The next two major steps are the definition of the message protocols and how the messages should be handled by the Problem Domain Entities.

For complex systems, the amount of messages can be massive. A human analyst quickly looses overview of the system's operation when all these messages are presented at the same time. To lower the amount of messages which are 'visible' to the analyst, they should be placed in groups which correspond to the different *aspects* of the system's operation. The next sections present some of these aspects:

- *System initialisation*
- *Normal system operation*
- *System reconfiguration*
- *System maintenance and testing*
- *Abnormal system operation*
- *System shutdown and restart*

The tools used for the analysis phase allow the designer to label each message with the aspect(s) it is used for. After choosing a 'current aspect', all messages which are not labelled with this aspect are kept out of sight. During simulation of the system, only messages labelled with the current aspect are allowed to be sent. This will help debugging the system.

Defining message protocols and implementing message handling should be done in an order which follows the 'natural' order of the system's operations. It is very difficult to simulate normal system operation if it has not been initialised!

4.4.1 System initialisation

During system initialisation, the elements which are needed to get the system operational must be brought into the system and linked together (they must know eachother's tags). This need only be done when the system contains dynamic structures like dynamic multiples. Single Problem Domain Entities and fixed multiples are defined by the designer and exist before the system is started.

System initialisation is a very important phase for systems which can be 'configured' or 'installed'. These systems contain variable amounts of specific Problem Domain Entities. These amounts vary from installation to installation, but remain static during the system's lifetime..

4.4.2 Normal system operation

A system is supposed to spend most of it's time in 'normal operation' - performing those actions which are needed for the main system functions.

There are a lot of different aspects which can be found in normal system operation. These are generally highly interrelated. They occur concurrently and may rely upon eachother (for instance because a system function uses a data structure which is updated by another function). Splitting 'normal operation' into too many aspects may give problems because these cannot be truly separated.

4.4.3 System reconfiguration

The system configuration as defined during initialisation may need to be changed during the system's operation. Designers should anticipate reconfiguration and build the necessary operations into the system right from the start. These operations should be able to change the system configuration without disturbing the running operations.

During system operation, data structures are built which depend upon the configuration. These data structures should be updated during reconfiguration. This makes reconfiguration a much more complex task than the initial configuration.

4.4.4 System maintenance and testing

During operation, a system's performance should be monitored and inspected by extracting specific information. This information may be used to 'tune' the system for optimum performance, for instance by choosing different cache sizes or message routing policies. Communication should be monitored to detect anomalies like increasing error rates or late responses. This allows the initiation of preventive maintenance before total breakdown of an entity or communication channel.

Complex interconnected data structures built during operation should be checked periodically for their integrity. External entities should be asked to perform self tests and report the results of these tests to the system.

All these operations can be initiated by a system user or by the system itself (scheduled in advance). System maintenance and testing functions should be designed into the system as early as possible: *'design for testability'* right from the start. The system specifications should already state that all Problem Domain Entities should be open to analysis. The functions introduced to do so will then automatically be incorporated in the final system.

4.4.5 Abnormal system operation

Certain parts of a system may break down. Communication channels may be disrupted and processors may fail. This is not so prominent during the analysis phase, where the analyst may assume the actual system will be error free ('the system' is then only an abstract description of the actual operations). The entities with which the system communicates may be expected to break down anytime. When restructuring the system during the architecture phase, the communication channels and processing entities within the system become more real. They only have a finite reliability, often expressed in 'mean time between failure' figures.

The abnormal system operation aspect has as objective to define how the system should react to failures. Workarounds should be found so that the system remains operable, possibly with slower responses or with a part of it's capabilities removed.

4.4.6 System shut-down and restart

Some systems do not operate continuously. A car's motor management system only runs while the motor is running. The easiest way to shut down and restart a system is to use *'persistence'*. This means that the data structures are stored in such a way that they remain intact while the system is shut down. With persistence, the system will restart as if nothing had happened. Unfortunately, this system-as-a-whole behaviour is difficult to achieve.

In most cases, systems are *implemented* to run on computers or computer-like processing elements. Each of these system components stores a portion of the system data structure in it's 'working memory'. This memory looses it's contents when the system component is switched off. It is therefore necessary to store all data structures in a safe place (for instance a battery backed RAM) during power down. This can become quite complex if only a few processing elements have the capability to store data in a safe place. All other elements must transfer their data structures to/from these 'safe keepers', using the already existing communication channels.

In the previous paragraph, the word *implemented* was highlighted. Determining which processing elements can be used as 'safe keeper' cannot be done before implementation

choices for the Abstract Processing Entities have been made. It is important to anticipate these operations in the analysis phase. The analyst should pin-point the vital data structures which must be saved (the amount of kilometers before the next oil change, for instance). For these vital data structures, storage and retrieval protocols must be defined.

4.5 Defining communication protocols

Behaviour description of the system's Problem Domain Entities starts with their externally observable behaviour. In an Object Oriented environment, this will consist out of the 'messages' sent between the Problem Domain Entities. These messages are the information carriers used for communication within the system. The messages come in several forms, depending on their main purpose. These main purposes are described in the following sections.

Messages are distinguishable by their 'format', which should be seen as the fixed part of a message (the *'message selector'*). Messages may carry parameters, which form the variable parts of the message. These parameters may be 'travelling' Problem Domain Entities or system defined variable types. Parameters can be used as 'tags' when a message is sent to a multiple Problem Domain Entity.

Messages are routed across the message channels which have been defined earlier. A message channel should be seen as a distribution medium without 'intelligence'. A channel simply transports messages between Problem Domain Entities. More than two Problem Domain Entities can be connected to a channel.

During high-level system behaviour analysis, the channels should be seen as 'broadcast' media. A message sent across a channel is received by all other Problem Domain Entities connected to that channel. During high-level system architecture design, the channels may be provided with simple message traffic distribution and concentration functions.

Each of the Problem Domain Entities connected to a channel inspects the messages sent across the channel. Within multiples (as described in section 3.2), each of the internal Problem Domain Entities inspects and handles the messages on it's own.

4.5.1 Trigger and synchronization messages

Messages which do not carry any parameters can be used to trigger events in Problem Domain Entities. They may also signal that a Problem Domain Entity has started a specific operation, asking other Problem Domain Entities to synchronise their operation to the sender.

Messages may include *'tags'* to route them to a specific Problem Domain Entity within a multiple. Tags should not be considered 'data' when a trigger or synchronization message is sent directly to the target Problem Domain Entity.

Within the multiprocessor system, the following (tagged) trigger message may be sent by an external event to a semaphore:

triggerSemaphore: <semID>

"Indicate the semaphore identified by <semID> that an external event has occurred. The semaphore may release a waiting process as a result of this."

Messages may have to be routed by other Problem Domain Entities before reaching their destination. In this case, the tags can be seen as routing information (data) which has to be used in the routing process. A routing Problem Domain Entity may even change the tags or the whole message structure. Routing information which has been used within a routing Problem Domain Entity may be absent in outgoing messages. In the multiprocessor system example, the following message may be sent by a process to an external event via a semaphore:

connectSemaphore: <semID> toExternalEvent: <eventID>

"Indicate the semaphore identified by <semID> to receive trigger messages from the external event identified by <eventID>. The semaphore forwards this message to the indicated external event."

4.5.2 Command and data transfer messages

The message format may specify a command which must be executed. Parameters can be used to further detail the operations to do. Within the multiprocessor system, the following command message may be sent by a process to a mailbox:

storeData: <aString> inMailBox: <mBoxID>

"If processes are waiting at the indicated mailbox, release one of them and give it the indicated String. Otherwise, store the String in this mailbox for later retrieval."

Data structures within a Problem Domain Entity may have to be kept up-to-date with data structures within other Problem Domain Entities. Messages carrying 'unsolicited' data (data which was not asked for) can be seen as commands to update an internal data structure. The message format and/or data itself indicates what to do with such a message.

The remarks on routing tags made in the previous section also hold for command and data transfer messages.

4.5.3 Messages requesting data or status

Messages may ask the recipient to return another message. This returned message can be a simple confirmation of message reception or a 'message handled' indication. The returned message may also contain other information, such as status information or data which was asked for in the requesting message.

This behaviour of a message may be present in trigger- as well as command messages. It is an orthogonal extension of the message types presented in the previous two sections. The following example message is sent in the multiprocessor system from a process to a mailbox:

pollMailBox: <mBoxID>

"If messages are present in the indicated mailbox, remove the first one and return it to the requesting process. Otherwise, return the 'no object' object."

4.5.4 Continuous data transfers

Not all data transfers in a system need to be made by messages. Message reception invokes an action, which is sometimes not wanted or possible (the receiving Problem Domain Entity may be busy handling another message).

If a data item must be kept up-to-date between different Problem Domain Entities, the normal way to do so is by appointing one of them to manage the original item. This manager handles update requests and distributes changes in the value with update messages. This is sometimes not the most obvious way to describe this kind of behaviour. When the data channel is one which is connected to external Problem Domain Entities, it may not be a realistic description *at all.* A thermocouple delivers a DC voltage, *not* a message each time the temperature changes by a tenth of a degree!

A continuous data channel as described in section 3.4 can be used to distribute small amounts of data. This channel is seen by other Problem Domain Entities as a read-only variable, which they can use whenever they need it. Receiving Problem Domain Entities are not informed that the value has changed. They should inspect (*'poll'*) the value regularly if they need to take action upon value changes.

Figure 4.5-1 shows a possible application of the external events in the multiprocessor system example. The generic 'external event' is replaced by an Analog-to-Digital converter, which measures the voltage generated by a temperature sensor. This voltage is transferred using a continuous communication channel.

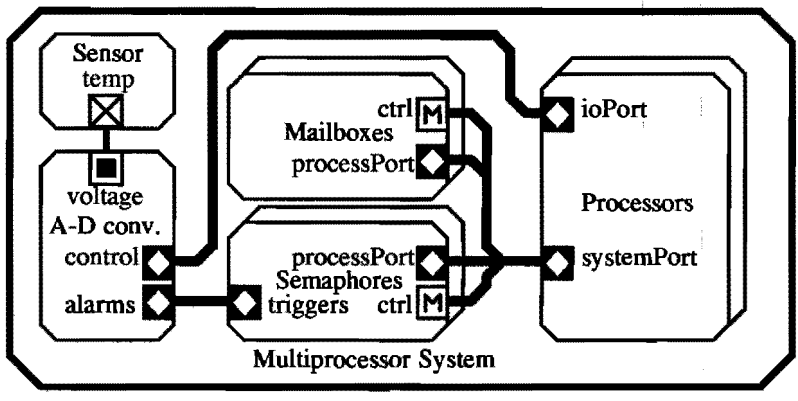


Figure 4.5-1: The multiprocessor system with an analog input

4.6 Implementing the handling of the messages

The last stage in system behaviour analysis is to describe the actual handling of the messages. In principle, each of the messages which is accepted by a Problem Domain Entity invokes an internal action. This internal action will be the starting of a Smalltalk-like *'method'* within one of the behaviour objects to handle the message. Parameters carried by a message are available to the method as read-only variables.

Within a method, it is possible to send messages across communication channels. It is also possible to manipulate local variables or send messages to other local behaviour objects.

4.6.1 Using a low level library

Behaviour objects may contain data structures built out of other objects. These objects can be travelling Problem Domain Entities and/or objects instantiated from classes provided by the design environment. The classes provided by the design environment

may themselves model data structures ranging from simple arrays to indexed and sorted lookup tables.

The behaviour object which models a processor within the multiprocessor system can be relatively simple when these complex data structures are used. It need only contain a 'Set of Processes' to hold all processes contained within the processor. A 'List of Processes' is used to store those processes which are ready to run. The process at the head of this list is the process which is actually running. The 'Processes' themselves are more complex objects which need to be modelled by the designer.

Aside from standard data structures, the design environment provides very complex objects which may be stored within a behaviour object. For instance, complete text editing windows can be attached to a behaviour object which models a terminal. The text editor window object is provided by the design environment, the behaviour object can exchange messages with it to receive and display text.

4.6.2 Inserting timing estimates

To be able to design time sensitive systems, several timing aspects must be modelled. Timing aspects are divided into two groups:

- *Problem Domain Entity internal operations.* The timing of Problem Domain Entity operations can easily be modelled by letting a Problem Domain Entity wait for a certain simulation time (specified by the designer), followed by performing the actual operations in zero simulation time. This will create the illusion that the actual operations took the specified waiting time to be performed.

  Operations can be broken down into smaller segments, each with their own timing specification. This can be used to model data dependent timing. A simple example is to attach a time delay to the internal operations of a loop construct. This way, the total waiting time depends on the number of loop iterations.

- *Data transfers across communication channels.* The time needed to transfer a message is specified by the designer in the sending behaviour method. This may be done either by stating a fixed time or by a 'message length' indication. In the latter case, the channel computes the actual transfer time (based upon channel speed and message overhead). It is possible to define a timeout when sending a message.

  Message transfer time specifications only provide an approximation of 'real world' communication channel behaviour. This is not a problem during high-level system behaviour analysis. Message transfer timing is specified only to

find communication bottlenecks (which have to be removed during system architecture synthesis).

A (somewhat complex) example of a method within the Analog-to-Digital converter of the multiprocessor system depicted in picture 4.5-1:

```
startConversions

"Start the endless loop of reading and converting a value and storing this
 value in the 'currentValue' variable. An alarm is sent to the semaphore
 indicated in 'alarmSemaphore' when the value exceeds the 'limit' variable."

[ self wait: 10 microSeconds.      "Simulate conversion time"
  currentValue := voltage.         "Read input value and store"
  (currentValue > limit)           "Check value against limit"
    ifTrue:
     [ alarms                      "Use 'alarms' connector..."
         triggerSemaphore:         "...to trigger the alarm semaphore"
           alarmSemaphore          "(see first example in section 4.5.1)"
       { transferTime:             "Specify message transfer time"
           2 microSeconds } ].     "(between {} not part of actual message)"
  self
    delay:                         "Start timer to wait for next conversion"
      990 microSeconds
    atPriority:                    "Wait at the lowest possible priority,"
      0                            "this allows other messages to be handled"
] repeat                           "Repeat the whole block over and over"
```

4.6.3 Exploiting concurrency

In the real world, Problem Domain Entities show a large amount of concurrency. Each Problem Domain Entity can be seen as a separate process, which is allowed to handle messages at the same time as other Problem Domain Entities in the system.

If a method within a Problem Domain Entity sends a message, the default behaviour is to wait until the message is being delivered (actual transmission has started). If the message did not request a result, the sending Problem Domain Entity method continues it's operations immediately following this. When an output buffer is used, the sending method need not even wait until the message is being delivered.Normally, a method which returns a result message stops at this point. It is possible, however, that a method returns a result and continues operating. The result can then indicate that message handling has been started. Actual results can be sent later as normal messages.

The example method in the previous section shows an example of how concurrency can be introduced in a system model. When started, it samples input values at regular intervals. While this is going on, processes running in processors may request the last sample value by sending messages like the following:

```
sampleValue

"Retum the value of the last sample. This message can be handled during
  the main waiting period in the 'startConversions' method. During this
  period, the processing priority is lowered to enable interrupts like these."

^ currentValue                  "Standard Smalltalk result retum method"
   { transferTime:              "Specify message transfer time"
      2 microSeconds }          "(between {} not part of actual message)"
```

4.7 Simulating the system

In the high-level system behaviour analysis phase, simulating the system is necessary to check if the behaviour implemented with the analysis tool matches the *expected* behaviour. We state *'expected'* here, not *'specified'*, because it is the analysis phase itself which is needed to specify the behaviour. Before the analysis phase, the behaviour may be *described*, but is normally not *specified* completely. Even mathematical equations may not be enough to specify a system completely.

During analysis, system elements are described with more and more detail. Each time this cycle is made, it is necessary to match the modelled behaviour against the specified behaviour. Checking the modelled behaviour against the specification can be automated using test sets or analytical methods. It is possible that the model exhibits unwanted behaviour at times that the specification specified 'unknown'. To find these anomalies, the analyst should be able to test the modelled behaviour with extra simulations.

Simulating the system also serves another purpose. It is done to show a customer what the system will do given certain inputs. It is not uncommon that during such a session the customer wants to see the system react to previously unspecified inputs. If the result is not what was expected, this will lead to a new piece of specification which has to be modelled in the system. This way, simulation can be used to remove ambiguities in the specified behaviour. Once the analyst and customer agree upon a certain system model behaviour, this is what the actually implemented system should do.

4.7.1 Gathering statistics

When system timing is hard-specified, critical path analysis can be used to check if the system adheres to this specification. If system timing is specified with statistical measurements, critical path analysis becomes less usable. Other analytical methods exist (queueing theory), but these only work for relatively simple abstract systems.

A common way out of this problem is to run a simulation of the system using close-to-real-life test value sets. The timing results of such a test session are collected in statistics. Mean and maximum system response times can be derived from these statistics.

The same timing results can be used to spot system bottlenecks during high-level system behaviour analysis. Communication channels and processing cores which are continuously active may slow down the complete system. These elements should be given extra attention during high-level system architecture synthesis.

Aside from timing measurements, collecting statistics can be helpful when limits must be imposed on dynamic storage structures like message buffers. Using large structures reduces the chance of overflow, but increases the cost of the implemented system. Statistics gathered during system simulation provide the analyst with the data needed to find an optimal storage structure size.

The design and simulation tools are capable of monitoring all communication channels and Problem Domain Entities during simulation. The tool user can specify exactly what must be monitored and how the data should be presented. Data gathered during simulation can be written to log files for later evaluation. It can also be presented in continuously updated charts on the computer screen.

4.8 System consistency issues

The following sections explain consistency checks which can be performed by the basic model design and simulation tools.

In an interactive design environment, consistency errors are flagged as *warning*. This is necessary because systems are modelled in small increments, an element may call upon elements which have not been added yet. During simulation of the system model, consistency errors will be flagged as a *fatal error* (which aborts simulation).

4.8.1 Static consistency checks

Static consistency checks are those checks which can be performed by looking at the system description alone. Some of these checks are so simple, that an interactive system can perform them 'on the fly' (while building the system model). Making these errors is simply not allowed by the design and simulation tools. Some examples:

- *Violating Problem Domain Entity- and communication channel naming rules.*
- *Designer specified tags for Problem Domain Entities in a multiple should all be different.*
- *Continuous data channels can only be connected to matching connectors.*

The semantics of the system structure are relatively simple: Basic model objects communicate across basic model communication channels. System structure related checks are therefore quite simple too, and are all related to the messages:

- *Messages sent across a channel should be known to other Problem Domain Entities connected to that same channel.* At least one of them should (conditionally) accept or absorb the message.
- *Messages which are known to a Problem Domain Entity should be sent by at least one other Problem Domain Entity connected to the message channel.*
- *Messages which require a result should be handled in a way that a matching result message is returned by at least one Problem Domain Entity.* To check this out, the internal descriptions of Problem Domain Entities must be examined.

4.8.2 Dynamic consistency checks

The dynamic consistency of a system is determined by values stored in the system. This kind of consistency is very difficult to determine from the system description alone. Analytical methods cannot handle complex systems without abstracting them to a level that their behaviour differs from the original specifications. The best way to perform dynamic consistency checks is by performing extensive simulations. Some examples:

- *Problem Domain Entities stored in a multiple must be distinguishable at all times.* Therefore, tags assigned to entities in a multiple must be different.
- *All designer specified limits* built into the system should be considered part of the dynamic consistency checks. These include the sizes of dynamic storage

structures (like message buffers) and time limits (like timeouts). Warnings should be generated when these limits are exceeded.

- *Variables of a Problem Domain Entity may not be used if they contain the 'no object' object.* Testing whether a variable is assigned 'no object' is allowed, but it is better not to *need* this test (it introduces an extra state for such a variable).

The designer may also use specific system operations to build his own consistency checks. Rejecting a message if a message buffer is full is an example. Within a behaviour method, aborting simulation can also be specified.

Performing dynamic consistency checks may require a more global view of the system than a single Problem Domain Entity or message channel. It is therefore necessary to provide a programmable consistency checking system which resides outside the design.

Global statistics generation and consistency checking have much in common. They both have access to the total system state. Statistics generation records selected parts of this state. Consistency checking stops simulation or generates a warning message if selected parts of the state fall outside specified limits.

4.8.3 Deadlocks

All Problem Domain Entities are separate processes which can operate concurrently. They may wait for specific events in other Problem Domain Entities. This means there is a danger for deadlock - two or more Problem Domain Entities waiting for eachother.

Detecting deadlock is done by tracing back messages which are blocked by a busy Problem Domain Entity. This trace must be extended to all Problem Domain Entities which are blocked by these blocked messages. This recursive method yields a tree of blocked Problem Domain Entities and messages. Each tree node is a blocked Problem Domain Entity, each branch is blocked message. The root of the tree is the Problem Domain Entity from which deadlock checking started. Deadlock is found when one of the nodes in the tree is again the root (the tree has become a directed graph with a cycle in it).

Deadlocks may resolve automatically when a timeout expires. This will remove one of the tree branches. If one of the branches in the cycle contains a message with a timeout, it means that deadlock was expected. In that case, simulation should not be aborted (giving a message may be useful, though).

A system may contain a specific Problem Domain Entity which is monitoring communications and tries to detect and resolve deadlocks. Aborting simulation if deadlock is detected would preclude testing this system function. The reaction of the design and simulation tools upon deadlock detection should therefore be programmable.

4.8.4 Meeting timing requirements

As stated in section 2.1.5, timing can play an important role in system design. Meeting specific timing requirements is *the* major objective in the design of real-time systems. Timing requirements form a part of the dynamic consistency checks. These can be specified by the analyst like any other consistency check.

A simple way to build timing restrictions into the system is to specify timeout periods for result messages. The method may abort simulation when the timeout expires. This gives the analyst a chance to trace the handling of the message. The statistics gathering tools can provide exact time traces of message handling. These traces allow the analyst to spot bottle-necks in the system *before* timing restrictions are violated.

4.8.5 Specification-to-implementation consistency

When analysing the system in a top-down fashion, behaviour descriptions serving as specification are implemented in lower level behaviour descriptions. In this process, more and more detail is added to the system. Adding detail means that the behaviour may change.

Specified timing is in most cases an estimate of the maximum response times. An implementation will therefore have slightly different timing characteristics.

A single Problem Domain Entity description normally handles a single message at a time. A sub-divided Problem Domain Entity contains concurrent Problem Domain Entities which can handle several messages at the same time. This means that message handling results may be returned at different times or even in a different order. In most cases, this poses no problems as long as maximum response times are adhered to and the actual results are identical.

If the ordering of results is important, then this must be stated in the specification. Any implementation should then adhere to this ordering. As with any other consistency check, the analyst may build message ordering checks into the system.

4.9 The result of the analysis phase: system behaviour

In this chapter, we have seen how Object-Oriented Analysis can be used to analyse a system under development. The system is described in terms of communicating Problem Domain Entities, which together possess the behaviour of the system. The analysis itself was done in a top-down stepwise fashion. Complex Problem Domain Entities were initially specified in global terms and subsequently analysed in greater detail.

The basic model described in chapters 2 and 3 is used for this high-level system behaviour analysis. This allows the designed system to be simulated to show how it behaves in response to external and internal events. The system analyst and the customer have agreed that this is the wanted behaviour. The remaining phases of the design path should all deliver systems which exhibit the same behaviour.

The next phase of the design path is the high-level system architecture phase. In this phase, the Problem Domain Entities must be mapped onto 'Abstract Processing Entities'. These will later become the actual hardware and/or software processing entities. The presence of an operational system behaviour description gives the designer several advantages:

- *Behaviour analysis can spot Problem Domain 'bottle-necks'* (Problem Domain Entities or communication channels which are heavily loaded). These need extra attention during architecture design.

- *The basic algorithms are already present in the system.* The complexity of these algorithms can be estimated. This allows the designer to make well founded decisions upon how to split or combine them.

- *The designer need not build a completely new system.* The analysis phase system can be modified step-by-step, *gradually* transforming Problem Domain Entities into Abstract Processing Entities. Each step is preceded by analysis of the current system configuration and followed by analysing the new system configuration. The objective is to utilise the Abstract Processing Entities to their full potential while meeting overall system requirements. Individual steps may even be suggested by an expert system ([**rov90**]).

5. High-Level System Architecture Synthesis

In the previous chapter, the behaviour of the system which must be built has been analysed. This was done in terms of Problem Domain Entities and their interactions. The problem domain does not describe the actual *architecture* of the system. The grouping of functions in Problem Domain Entities and their interconnections directly follow the original problem statement:

The system built during high-level system behaviour analysis is an *'architecture independent'* solution to the problem statement.

If a system is *architecture* independent, then it is of course also *implementation* independent. System implementation cannot begin before an architecture has been fixed.

High-level system architecture synthesis aims at providing an optimal system architecture, ready for implementation.

This architecture contains communicating *'Abstract Processing Entities'*. These provide abstract descriptions of the processing functions performed by the modules of the final system. For each of these, an implementation strategy will be selected based on their processing-, data storage- and communication requirements. An implementation strategy may use any mixture of hardware and software, ranging from ASIC's to standard software running on standard computer hardware.

The Abstract Processing Entities communicate across *'Abstract Communication Channels'*. These model the communication channels which will be present in the final system. The implementation strategy for the Abstract Communication Channels will be selected during high-level system architecture synthesis. This selection is based upon the communication requirements of the Abstract Processing Entities connected to these channels. The implementation strategy may range from simple point-to-point communication lines to complex Local Area Networks.

The system model which results from high-level system architecture synthesis directly models the actual processing operations of the final system. This model is therefore called the *'Processing Model'*.

High-level system architecture synthesis is performed by re-mapping the functions of the Problem Domain Entities into the Abstract Processing Entities. This re-mapping is done gradually, by modifying the system structure in small steps. Before each step, the

system structure is analysed to decide which modification step to perform. Following each modification, the resulting structure is analysed to determine whether or not this modification actually improved the system.

Throughout high-level system architecture synthesis, preliminary implementation choices are made. These label an Abstract Processing Entity or Abstract Communication Channel with actual capabilities. Preliminary implementation choices allow the architecture designer to match the functions of a Processing Model element to it's chosen capabilities. High level system architecture synthesis ends when all preliminary implementation choices are fixed.

Both the Abstract Processing Entities and Problem Domain Entities use the basic model as introduced in chapters 2 and 3. All the analysis methods described in the previous chapter remain usable during high-level system architecture synthesis.

System structure modifications are made with an *'architecture editor'* tool. This tool can split and combine existing basic model objects and communication channels. These operations are described in the next section.

Using the architecture editor, the Processing Model can be created out of the high-level system behaviour model. This keeps the amount of new objects that must be introduced in the system to the minimum, reducing the amount of errors accordingly. Using libraries of previously designed Abstract Processing Entities can be done as it was proposed for Problem Domain Entities in the previous chapter.

5.1 Architecture Editor operations

This section introduces the main tool used during architecture synthesis, the *'architecture editor'*. It's purpose is to modify the system model structure in ways indicated by the designer. The architecture editor provides four basic operations:

1) *Combining basic model objects*
2) *Combining communication channels*
3) *Splitting basic model objects*
4) *Splitting communication channels*

These basic operations will be described in the following sections. The remainder of this chapter uses the basic operations to convert the high-level system behaviour model into the Processing Model (the actual high-level system architecture).

Architecture editor operations do not change the *functionality* of the system. Performing these operations may change the system *timing* because concurrency is removed or added.

In general, combining communication channels or processing entities decreases system performance (concurrency is removed from the system). Splitting processing entities or communication channels usually increases the system performance (extra concurrency is introduced into the system).

5.1.1 Combining basic model objects

There are two methods to combine basic model objects:

1) *Place them in a grouping entity as described in section 3.1.*
2) *Actually merge the operations of the original objects into a new basic object.*

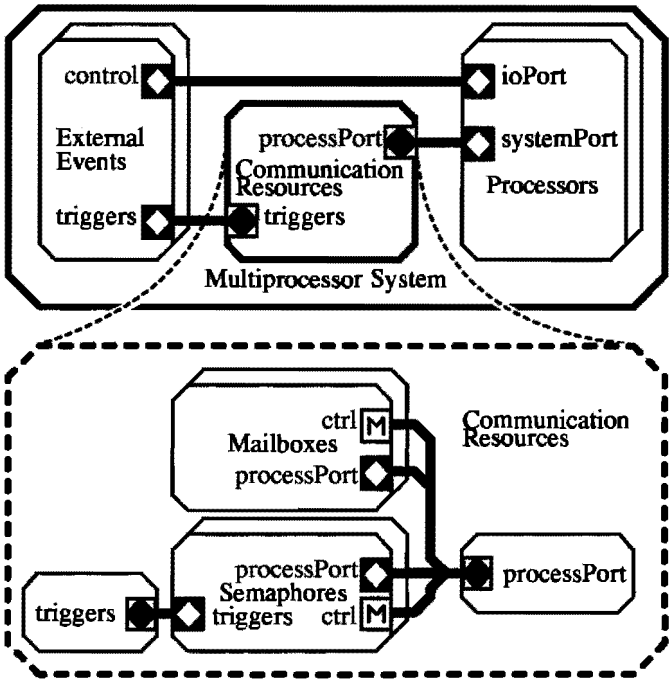


Figure 5.1.1-1: Combining basic model objects in a group

The first method combines basic model objects by placing them on the Entity Communication Diagram within a 'grouping' entity. The symbol for the grouping entity replaces them at their original Entity Communication Diagram. This is a relatively simple operation because no changes need to be made in the combined objects. Their behaviour remains exactly the same. Figure 5.1.1-1 shows the multiprocessor system model example after combining the mailboxes and semaphores

in the grouping entity 'Communication Resources' (figure 4.3-1 on page 73 shows the system structure from which this structure has been derived).

Merging basic object operations into a single basic object is a much more complex operation which requires the following four steps:

1) *Place the elements of the original communication shells and processing cores in the new basic object.*

2) *Combine connectors which are connected to the same communication channels.*

3) *Re-route messages which were originally sent between now combined basic objects.*

4) *Remove behaviour objects whose functions are already provided by other behaviour objects.*

These steps are explained in more detail below.

Merging basic objects replaces two (or more) concurrent processes by a single process. This means that functions which were first executed in parallel now have to be performed in sequence. *Timing* of the function handling may change as a result of this (the actual *functionality* remains the same).

Step 1 - Combining communication shells and processing cores:

All connectors and behaviour slots of the original entities are placed within the combined entity. Connectors remain connected to their communication channels. Existing behaviour objects move with their slots.

The message filter and buffer specifications are simply copied. Message selector/manipulator entities are specified with several lists of selection criteria and corresponding actions. The designer must indicate how these (priority ordered) lists should be merged.

Combining virtual connector translation tables may generate naming conflicts. Identically named entries may be collapsed into one if they all refer to connectors which are connected to the same communication channel (like the 'processPort' connectors on the mailboxes and semaphores of the multiprocessor system example). In all other cases, one of the virtual names must be changed. The behaviour methods referring to such a changed name must be updated.

Step 2 - Combining connectors:

All connectors which were connected to the same channel may be collapsed into a single one. The message selector/manipulator specifications and virtual connector translation table must be updated to reflect these changes.

Message filter specifications are given as lists of message selectors (see section 2.3.1). Each of these has a textual specification of what is done with the messages which match the selector. Designer intervention is needed when specific messages are received by more than one of the original connectors. Following combination, this message will only be received once. This situation can be detected when the logical conditions for accepting a message have a non-empty intersection.

Combining message buffers is a trivial problem if they use the same mode ('straight FIFO' or 'priority FIFO', as indicated in section 2.3.2). The designer has to intervene when the buffer modes differ. Buffer depths can be added.

Step 3 - Re-routing inter-object messages:

Messages which were sent between two combined entities can be routed internally (these may have been sent via other basic model objects). This may increase system performance because no communication channel is needed for these messages. Internal re-routing is done by changing the virtual connector translation table. A connector reference is replaced by an internal slot reference.

Messages may be sent to more than one entity. The designer should intervene when at least one of these entities is now combined with the sender. The connector cannot be removed as long as external entities remain in the set of receivers.

Following this step, one or more connectors may be left unused. These connectors can be removed from the combined entity.

Step 4 - Removing obsolete behaviour objects:

Behaviour objects may be removed when their functionality is available in another behaviour object. This can be done, for instance, when a data cache is combined with the actual data. Using fixed classes (like a 'Cache' class) for this purpose allows tools to detect this situation. Otherwise, detecting this possibility relies on the designer's experience. Messages sent to the removed behaviour object may be re-routed by changing the virtual connector translation table.

5.1.2 Combining communication channels

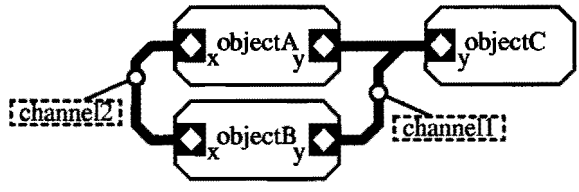
Communication channels may be combined to save interconnection hardware. System timing may change because a single channel can only carry a single message at a time (where the original separate channels were capable of transferring messages in parallel). When combining channels, two problems must be solved:

1) *Identical message formats may have been sent over the separate channels.*

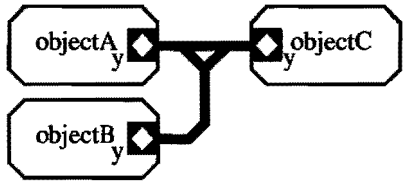
   In general, these messages must be made distinguishable to prevent them from being received by the wrong receivers. The sending behaviour object methods must be changed because they define the message format (message selector and parameters).

   The situation is simplified when these messages were *intended* to be received by receivers on the separate channels. Such a message now only needs to be sent once. This requires changing the sending behaviour object method.

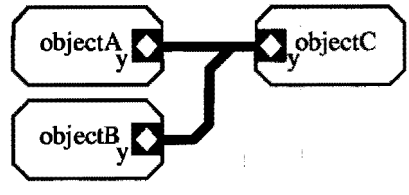
2) *Channels may have different topologies.*



a) Original situation. 'objectA' can directly communicate via 'channel2' with 'objectB'. Communication via 'channel1' must always involve 'objectC'.



b) Combination 1, 'objectC' is not needed anymore as intermediate. Direct communication is maintained.



c) Combination 2. 'objectC' is still needed as intermediate. Direct communication between 'object1' and 'object2' is prohibited.

Figure 5.1.2-1: Combining different channel topologies

When channels are used as broadcast media, they can be connected at any point and continue operation. Other channel topologies (tree-like structures, for instance) may cause problems when they are combined. Combining a direct link with a tree-like channel topology is illustrated in figure 5.1.2-1. Selecting between the two combinations depicted there is up to the designer.

Combining channels may cause multiple connectors of an Abstract Processing Entity to become connected to the same channel. These can be collapsed into one as indicated by step 2 of the previous section.

5.1.3 Splitting basic model objects

Splitting basic model objects increases the amount of concurrency in the system. This allows the system to work on more tasks at the same time. Splitting basic model objects also increases communication overhead because data which was once directly available is now distributed across several objects. The extra communication overhead can be minimised by keeping data and operations which manipulate this data together as much as possible.

When splitting an basic model object, the system architect should distribute the object's internal behaviour objects and connectors across the group of new objects. The internal elements of a basic model object form an intricate network of dependencies. This network should remain intact after the split. The following two dependencies can be found in a basic entity:

1) Connector-behaviour method dependencies:

Behaviour methods are started by messages which are received by specific connectors. Methods also send messages across specific connectors. Splitting a method from it's connector necessitates relaying these messages between basic model objects. The basic model object which contains the original communication channel connector becomes a message router. A new communication channel may be needed to transfer this message.

Another solution is to provide the method carrying object with an extra connector to receive the message directly from the original communication channel. This reinstates the original situation regarding the connector-behaviour method dependency (at the cost of an extra communication channel interface).

2) Inter-behaviour object dependencies:

Communication between behaviour objects which are distributed across different basic model objects must remain possible. Messages originally sent directly must be routed across a channel to which the different basic model objects are

connected. If such a channel is unavailable, a 'private' channel must be created with accompanying connectors. Message routing is done by changing the virtual connector translation tables.

Continuous data inputs and outputs have their own problems when basic model objects are split:

- Reading a continuous input which is placed in another basic model object can be done by sending read request messages. Another solution is to provide the basic model object with the method with it's own continuous data input and connect this input to the continuous data channel.

- Updating a continuous output located in another basic model object can be done by sending update request messages. It is possible to use the three-state capabilities of continuous outputs to provide the basic model object with the method with it's own connection to the continuous communication channel. Controlling which of these connectors is active may be more complex than sending update messages.

5.1.4 Splitting communication channels

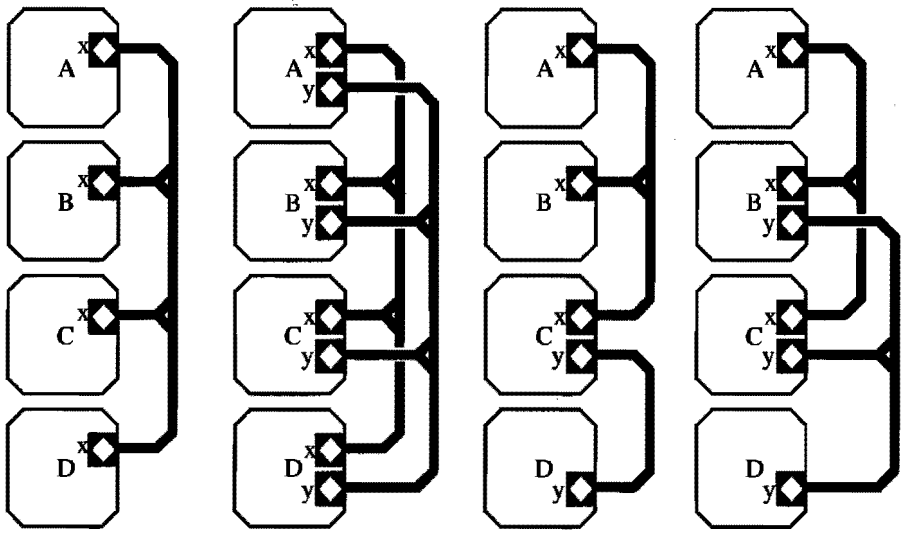
A message channel may be split in multiple channels to increase the communication bandwidth between Abstract Processing Entities. There are several ways to split a channel:

- *Fully parallel split* (figure 5.1.4-1b): The channel is duplicated. Each basic model object connected to the original channel receives an extra connection to the new channel. Messages may be distributed across both channels, which means that two messages may be transferred in parallel.

- *Segmenting* (figure 5.1.4-1c): The channel is split in separate segments, each connected to a subset of the original basic model objects. The segments can transfer messages in parallel, without disturbing eachother. These messages do not automatically appear on other segments. Specific basic model objects have connections to two or more of these segments. These are used to route messages between segments (object 'C' in the figure). In a truly segmented channel there is always only a single path between segments.

- *Partially parallel split* (figure 5.1.4-1d): A new channel is connected in parallel to the original one, but not *all* basic model objects get a connection to this new channel (like object 'A' in the figure). This may be combined with segmenting by disconnecting some basic model objects from the old channel (object 'D' in the figure). The latter case needs routers to transfer messages between the

channels. The routing function can be performed by all basic model objects with access to both channels (objects 'B' and 'C' in the figure).



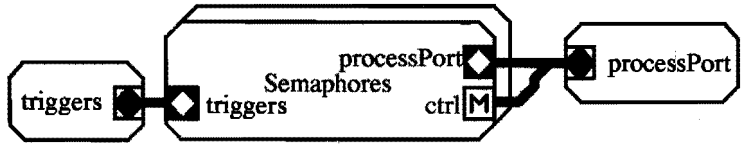
a) Original channel | b) Fully parallel split | c) Segmented channel | d) Partially parallel split

Figure 5.1.4-1: Several ways to split a channel

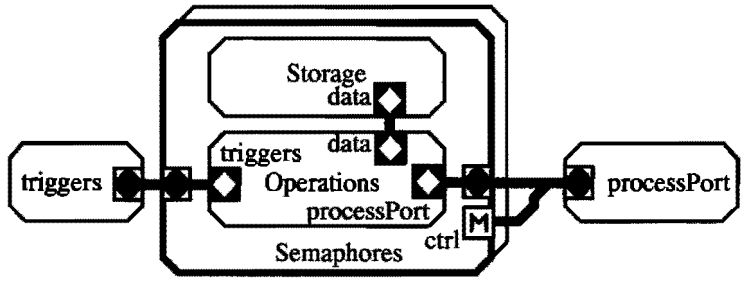
5.2 Removing dynamic structures

One of the operations which must be done during high-level system architecture synthesis is finding ways to implement the dynamic multiples which are present in the system model. These model entities from which there are a varying amount present in the system. The system itself can decide to create new entities or remove old ones.

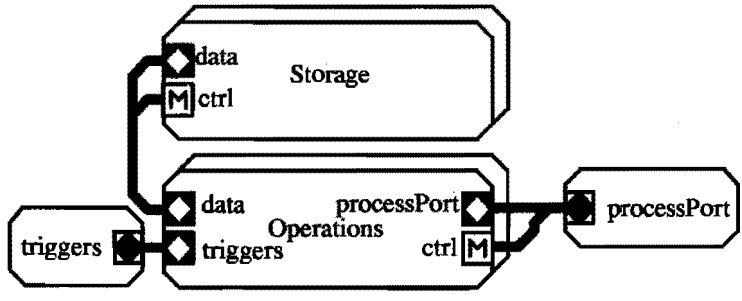
The basic software implementation of dynamic multiples is to use dynamically allocated memory. This memory is used to store the data structures of the entities in the multiple. The operational parts of the entities (the methods) are shared by all entities in the multiple. Each time a method is started, it is provided with a pointer to the data structure it has to work with. The basic software implementation allows only a single entity to be active at a time.



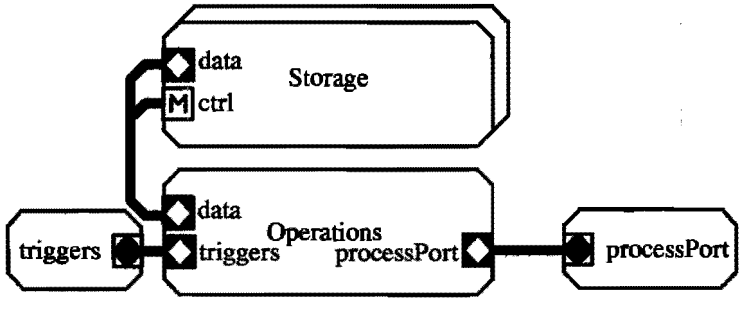
a) Original situation: a dynamic multiple (the 'M' connector is the multiple management connector described in section 3.2.1.1)



b) Result after splitting Storage and Operations in separate objects



c) Result after splitting multiple group in dynamic multiples



d) Result after changing the operations in a single entity

Figure 5.2-1: Steps in converting a dynamic multiple into hardware

A hardware implementation of dynamic multiples may follow the same approach. Figure 5.2-1 shows how the 'Semaphores' dynamic multiple in the multiprocessor system example can be converted:

1) Each basic object in the dynamic multiple is split into it's operations and the actual variables. The dynamic multiple is replaced by a dynamic multiple group to contain these objects. The result is depicted in figure 5.2-1b. The 'data' connectors are used by the 'Operations' basic object to retrieve and store the actual status information of the semaphore.

2) The dynamic multiple group is split into two parallel dynamic multiples. This situation is depicted in figure 5.2-1c. For each entity in the 'Operations' multiple, an accompanying member of the 'Storage' multiple must be present. Each time a new entity is created in the 'Operations' multiple, it immediately creates a new entity in the 'Storage' multiple to hold the status information of the semaphore it represents.

   The communication channel between the multiples is shared between all semaphores. To ease communication, identical tags should be used to address entities in both dynamic multiples. This makes it unnecessary to perform tag translation in the 'Operations' multiple.

3) The 'Operations' multiple is reduced to a normal basic object. This results in the situation shown in figure 5.2-1d. The tag present in the messages sent to this object is used to address the entities in the 'Storage' dynamic multiple. The entity creation and removal messages must now be handled via the 'processPort' connector. These messages are simply forwarded to the 'ctrl' connector of the 'Storage' dynamic multiple. The tag which results from the creation operation is returned via the 'processPort' connector.

4) The remaining 'Storage' dynamic multiple models a dynamically allocated memory. This can be split in a memory manager and an actual 'dumb' data memory. Implementations of these can be pre-designed and placed in a library.

Some remarks can be made:

- Using only a single 'Operations' basic model object allows only a single semaphore to be active at the same time. By changing the 'Operations' dynamic multiple into a normal multiple, a specific amount of semaphores can be active at once. The entities in this multiple are indistinguishable, they can all perform the same operations. Distributing the incoming message across this multiple must be done either statistically (for instance by hashing the tag) or deterministically (for instance by a 'manager' object which chooses one of the idle 'Operations' entities).

- The dynamically allocated memory may be shared with information for other data structures. This way, the mailboxes of the multiprocessor system example can be merged with the semaphores. Whether or not the 'Operations' basic objects are merged too is up to the architecture designer.

- The basic behaviour of the converted dynamic multiple remains the same. Message handling delays occur when the 'Operations' entity (or entities) cannot handle the processing load which was handled concurrently by the entities in the original dynamic multiple.

5.3 Building fail safe systems

Like removing dynamic structures, introducing 'fail safe' system structures is an operation which is common in high-level system architecture design. These structures allow a system to remain operational even when parts of it do not function properly. This is normally achieved by providing the system with spare parts or by re-allocating tasks. There are numerous possibilities to do this, some of which are:

- *Hot standby operation.* Each system element which must remain operational has at least one copy which is performing exactly the same operations in parallel. One of the copies takes over immediately when the active element fails.

- *Cold standby operation.* A system element which must remain operational has a spare copy which is capable of taking over it's tasks. The active element regularly saves it's internal state. When taking over, the spare loads the last stored state and starts from there.

- *Function re-assignment.* Systems with interconnected identical processors can use function re-assignment to increase their reliability. When a processor breaks down, it's functions are distributed across the remaining processors.

Duplicating basic model objects and communication channels is the first step towards modelling these strategies. Checking for errors and managing the error recovery process can be done by introducing new basic model objects in the system. Function re-assignment can be modelled by using travelling objects to model the functions.

An example of hot standby operation modelling is given in figure 5.3-1. The 'Manager' entity distributes all messages which are received on the 'command' connector towards the processors. It compares the results which it receives back from the processors. Only one of these results is actually returned across the 'command' connector. A malfunctioning processor delivers results which differ from the other two. This processor will not be used anymore.

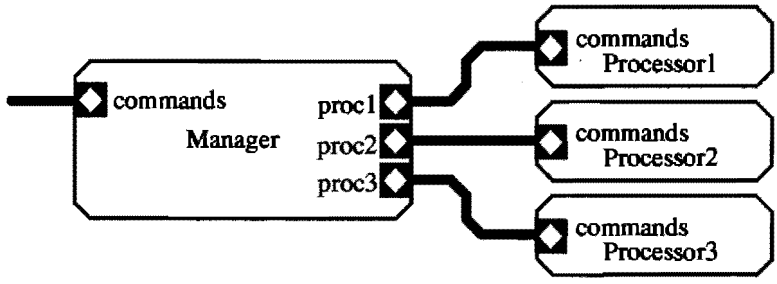


Figure 5.3-1: 'Hot standby' operation modelling

5.4 Preliminary implementation choices

High-level system architecture design has a specific goal: The final architecture must be implemented in hardware and/or software modules.

Direct implementation of high-level system architecture elements in the Processing Model may be impossible or too expensive. In these cases, architecture optimisations are necessary as described in section 5.5. Following these optimisations, another attempt at selecting an implementation strategy may be performed. Designing a new Processing Model element implementation is left as last resort when optimising does not yield the required results. This 'new' element may be an adaption of an already existing implementation.

Selecting an implementation strategy starts by generating the *'profile'* of the Processing Model element which must be implemented. Such a profile forms a standardised highly abstract description of the requirements of the Processing Model element. The contents of basic model object and communication channel profiles are described in the next sections.

Requirements profiles can be matched against a database containing profiles of actual implementations. A database search may come up with implementations which are not optimal:

- *An implementation may be over-rated.* The excess capabilities may be used to let this system element perform additional functions.
- *An implementation may be slightly under-rated.* These can only be used if it is possible to reduce the processing, data storage and/or data transfer requirements.

The profile of a chosen Abstract Processing Entity or Abstract Communication Channel implementation can be attached to the Processing Model element. This profile acts as an extra system constraint when architecture changes are executed. It also allows a more precise modelling of the element's operation.

The starting point for moving towards system implementation is by selecting those elements of the system which need special attention (the 'bottle-necks' mentioned in section 4.9). When implementation choices for these elements have been fixed, the remaining system elements can be handled. Excess capacities in the already fixed elements may be used to perform these remaining element's functions.

5.4.1 Selecting communication channels

Choosing implementations for Processing Model elements should start with the Abstract Communication Channels. They form the network which interconnects the Abstract Processing Entities. Choosing the channel implementations first fixes the interface requirements of the Abstract Processing Entities.

Channel *'profiles'* are used to describe the characteristics of existing channel implementations. They contain the following information:

- *Communication protocol used by the channel*
- *Channel topology*
- *Data/event rate(s)*
- *Physical characteristics*
- *Error probability and handling*
- *Cost*

Existing communication channel implementation profiles are placed in a database within the architecture design toolbox. A channel's requirements for the first three elements of the profile can be deduced from the Processing Model. With this data, a computer can search the database with implementation profiles for suitable channel implementations. The final channel implementation decision is taken by the architecture designer.

Following the selection of a channel implementation, the Abstract Processing Entities connected to this channel should be taken under consideration. This is the subject of the next section.

5.4.2 Selecting processing entities

In the previous section, communication channel capabilities were abstracted into a *'profile'*. These profiles were matched against channel requirements to provide an initial implementation choice. The same method can be applied to processing entity implementations.

The profile of an implemented processing entity should contain the following information:

- *Data (and program) storage capabilities*
- *Interface capabilities*
- *Processing capabilities*
- *Cost*

The first three elements of the required profile for an Abstract Processing Entity can be deduced from the Processing Model. The required interfaces are given by the communication channel implementation choices made in the previous section.

5.5 System architecture optimisation

The main objective of system design is to build a system which performs as specified while the total cost is as low as possible. System architecture optimisation plays an important role in reaching this objective.

Once preliminary implementation choices have been made, architecture optimisations can be applied to use these Processing Model elements to their full potential. Architecture optimisation steps may also be performed when direct implementation of Processing Model elements proves impossible or too costly.

Architecture optimisation and making preliminary implementation choices forms the core of high-level system architecture design. Optimisation may cause different implementations to be chosen for certain system elements. These choices may lead to new optimisations. This loop is broken when a near-optimal solution is found.

System architecture optimisations do not change the *functionality* of the system. System *timing* may change when architecture optimisations are performed. In general, combining communication channels or processing entities decreases system performance. Splitting processing entities or communication channels generally increases the system performance. These timing changes may be compensated by choosing different communication channel or processing entity implementations.

The next sections outline how the architecture editor functions (described in section 5.1) can be applied to perform system architecture optimisation.

5.5.1 Combining low bandwidth data channels

Multiple communication channels may exist between two communicating Abstract Processing Entities. Some of these may be combined to lower system costs. The combined channel should be capable of handling the increased communication load.

If multiple parallel communication channels remain, the message traffic may be re-distributed amongst them. Having multiple paths to choose from gives the architecture designer extra freedom in optimising the system. Message distribution decisions will be influenced by channel characteristics.

5.5.2 Combining processing entities

Processing entities which are not fully utilised may be given extra functions to perform. If it is possible to move a processing entity's functions into other processing entities, then this processing entity can be removed.

When entities are fully combined, communication *between* the original entities will become internal communication *within* the combined entity. This may be a major driving force to combine entities. It can improve system speed by removing communication overhead.

5.5.3 Splitting high bandwidth data channels

As indicated in section 5.1.4, there are several ways to distribute message traffic across multiple channels. This may become necessary when the original channel cannot handle the communication load. The way in which the channel is split depends upon the message traffic patterns:

- *All Abstract Processing Entities communicate amongst eachother.* The channel should be duplicated to distribute the message traffic.
- *A small group of Abstract Processing Entities keep the channel occupied with messages sent between members of this group.* A new channel should be connected to the entities within the group (partially parallel split).

- *There are several groups of Abstract Processing Entities with relatively little communication between the groups.* The channel should be segmented so that each of these groups has it's own segment.

5.5.4 Splitting and duplicating processing entities

Splitting and duplicating processing entities is done when no implementation can be found which provides the required processing and/or data storage capabilities:

- *Splitting a processing entity* provides a set of processing entities with different characteristics. These can be connected in sequence to form a processing 'pipeline' when the original entity had very complex functions to perform. These can be connected in parallel when the original entity had different functions to perform. Splitting processing from data storage provides the opportunity to choose a much larger separate data store.

- *Duplicating a processing entity* provides a set of processing entities with the same characteristics. These are connected in parallel and share the processing and data storage load.

Long-term data storage may form a problem when processing entities are split or duplicated. Keeping data and operations on this data combined in a single entity is not always possible. Separating data storage and operations before splitting or duplicating the actual operations provides a good solution to this problem.

5.6 Final implementation choices

The preliminary implementation choices will be made final when a satisfactory system architecture has been found. The following sections give an indication of what can be expected from these final implementation choices.

Rigid interface specifications are needed to ensure that implemented processing entities can be interconnected. Section 5.7 provides an overview of the possible interfaces.

5.6.1 Communication channels

Abstract Communication Channels can be implemented with very diverse methods. Communication channel implementations may be selected based upon the type (continuous data or message driven) and use of the channel as present in the Processing Model:

- *Continuous data communication channels* have no implicit protocols. Their implementation ranges from analog two-wire connections to simple (three-state) data buses.
- *Channels which carry events* can be implemented with very simple methods. A two-wire connection suffices for a single event, multiple events can be encoded to reduce the needed number of wires.
- *Point-to-point communication channels* can be implemented as simple serial or parallel connections with handshake protocols.
- *Multi-source and multi-destination channels* can be implemented by standard computer buses or communication networks. These include arbitration and error recovery protocols.

Direct communication between software modules is regarded an interface, which will be described in section 5.7.1.

5.6.2 Software implemented processing entities

Software implementations of an Abstract Processing Entity consist out of two parts:

1) *The hardware on which the software must run.* The choice for this hardware is based upon the processing, interface and data storage capabilities needed by the Abstract Processing Entity. The hardware may range from a processing core embedded in an ASIC, via single chip microcomputers to complete (board-level) computers.

2) *The software itself.* This should be a translation of the behaviour description of the Abstract Processing Entity into the machine 'language' of the chosen processor. This translation may be done via intermediate 'high-level' languages.

5.6.3 Hardware implemented processing entities

Hardware implementation of an Abstract Processing Entity means that a (new) data processing architecture must be designed. This architecture is specifically tailored to the functions which must be performed. Examples of functions which can be implemented in highly specialised hardware are fast Fourier transforms, data compression and decompression, data encription and decription, image generation and communication channel switching.

Designing new hardware should only be done when existing implementations cannot be used. This only happens for extreme situations, like very high data processing speed or ultra low power requirements.

5.6.4 Mixed hardware/software implementations

Mixed hardware/software implementations are obtained when 'general purpose' programmable elements are combined with specifically designed hardware:

- *An already existing processor's instruction set is changed.* This can be done to tailor this processor to the processing requirements.
- *New hardware is added to an existing processor.* This hardware may perform specific processing or interface functions.
- *A new general purpose processor is designed.* Although designed to implement a specific Abstract Processing Entity, this processor may be re-programmed for other purposes.

5.7 Interface specifications

Interfaces must be specified to define the interconnections which will be present in the final system. The different Abstract Processing Entities are implemented separately. The interface specifications must be very thorough to ensure that implemented processing entities can communicate when they are interconnected in the final system.

There are various types of interfaces which can be present in the final system. The next sections describe these types, where they are used and what is needed to specify them.

5.7.1 Software-software interfaces

Direct interfaces between software implemented entities can only occur when these software modules run within a single processor. The interface between application modules and an operating system is a very important software-software interface. The two major types of software-software interfaces are the following:

- *Shared variables.* These are generally used for non-event driven communication (for instance to distribute status information). Shared variables can be specified by giving their storage format, memory space and address.
- *Procedure and function calls.* These are used to indicate an event or transfer a request. Their specification includes the formats of variables given and returned and how these are transferred. The way to invoke the actual routine should also be specified (for instance direct call, table look-up or software interrupt).

5.7.2 Software-hardware interfaces

Software-hardware interfaces are needed when software interacts with input/output hardware. The software initiates an interactions through input and output instructions. External devices initiate an interaction with software through interrupts.

Interrupts and input/output instructions provide only a primitive means of communication with external devices. It is possible to raise the hardware-software interface level by using more complex processor hardware and instructions.

A good example of an advanced hardware-software interface is the T9000 'transputer' ([pou91]). This processor is capable of sending and receiving data packets with a single instruction. A built-in multitasking operating system runs other tasks while a task waits for the completion of such a transfer.

5.7.3 Hardware-hardware interfaces

Interfaces between hardware elements occur in a multitude of places. Each time there is an Abstract Processing Entity to Abstract Communication Channel connection, a hardware interface must be defined and built.

Section 6.2.3 provides the primitives which can be used for hardware-hardware communication. These 'primitives' include registers, queues and multiport memories. A multitude of different communication forms and protocols can be built using these primitives.

5.7.4 Sensors, actuators and adaptors

Systems are not built using direct digital interfaces only. Special interfaces must be applied whenever special voltage or current levels are needed or non-electrical communication forms are used. Three basic types of these 'special interfaces' can be found in a system:

- *Adaptors* are used to convert the voltage and current levels used within the data processing equipment into other levels and vice verse. Examples of adaptors are power drivers, input protection networks, digital-to-analog and analog-to-digital converters.

- *Sensors* are used whenever values or states must be input into a data processing system. These are often non-electrical in nature. A sensor generally includes an adaptor for connection to the digital processing hardware.

- *Actuators* are used whenever a data processing system must effect changes to it's environment. These changes are mostly of non-electrical nature. Like sensors, actuators generally include an adaptor at the processor interface.

Designing interfaces like these is a far from trivial and often specialised job. Their importance is visible in the problem statement when the external interfaces of the system are non-electrical. Problem Domain Entities must be introduced during high-level system behaviour analysis to convert these interfaces into something more manageable. These Problem Domain Entities remain visible in the architecture phase. Their basic function does not change at all and will later be implemented in hardware.

5.8 Design consistency issues

All design consistency issues which were described in section 4.8 are valid during high-level system architecture design. The methods used to check for inconsistencies remain operational because the same basic model is used.

Making preliminary implementation choices introduces three consistency issues which are not inherent to the basic model as described in chapters 2 and 3:

- The processing and data storage requirements of an Abstract Processing Entity should remain within the profile of an implementation choice made for this entity.
- The communication performed on an Abstract Communication Channel should be allowed by the profile of an implementation choice made for this channel.
- All Abstract Processing Entities attached to a channel should be able to interface with it.

5.9 Design for testability

Design for testability is the follow-on of these *'maintenance and testing'* functions described in section 4.4.4. These functions monitor and maintain the operational characteristics of the system. Design for testability focuses upon the final architecture and it's components. Two major goals must be met:

- *Testing the system components in isolation.* This can be used prior to system integration to make sure that the individual components are operational.
- *Testing the system as a whole.* This can be done following integration to check the communication channels and the component's adherence to the specified protocols.

To reach these goals, the set of existing test messages should be expanded with messages which concentrate upon the architecture components. The actual implementation of the handling of these messages remains to be done during implementation of the system modules. Standard design for testability measures like *boundary scan* (board level) and *scan testing* (device level) should be used whenever possible. These methods can be applied automatically by integrating them in the design tools (this will be described in section 6.5.4).

Both boundary scan and scan testing allow *'structural'* tests to be performed. They are capable of checking all elements of the system structure down to the logic gate level. Structural tests are preferred above *'functional'* tests, which only test the basic functions of a system. A functional test may leave some system elements untested. Functional tests have an advantage over structural tests in that they can be performed while the system is operational. The 'maintenance and testing' functions will therefore in general be performed by functional tests.

5.10 The result of system architecture design: the Processing Model

The end result of high-level system architecture design is a network of processing entities interconnected by communication channels. This network models the partitioning of processing in the final system, and is therefore called the *'Processing Model'*. The Processing Model performs the same functions as the high-level system behaviour model from which it has been derived.

Implementation strategies have been chosen for each of the processing entities and channels. Interface- and functional specifications are available for each of the processing entities. These will be used during low-level module architecture design and implementation, which is the subject of the next chapter.

6. Implementing the Processing Model in Software and Hardware

The last phase of the design path handles low-level module architecture design and implementation. During this phase, Abstract Processing Entities and Abstract Communication Channels are converted into operational hardware and software.

This chapter concentrates upon hardware implementation. An executable language is used to define the behaviour of Abstract Processing Entities. This allows the generation of software to be more or less automated by compiler-like tools.

The target of this chapter is the design of digital Application Specific Integrated Circuits (ASIC's). These have the following advantages:

- *ASIC's allow the highest functional density of all implementation techniques.*
- *Because of the tight packing, interconnection delays are minimised.* This gives ASIC's an advantage in overall system speed.
- *The low level components in an ASIC can be matched exactly to the requirements.* This allows highly optimised architectures to be the built.

ASIC's also have their problems. The major one is the following:

- *Design errors cannot be corrected within finished devices.* A design which is not 100% correct is virtually useless.

Building a prototype to verify the design is very expensive. Functional verification is therefore done by extensive simulations. Simulating complex ASIC's at gate- or lower levels is very time consuming. The amount of simulation runs is reduced to the bare minimum to shorten this time. Building a comprehensive test set under these constraints is very difficult.

Using *'correctness by construction'* methods removes the necessity of low-level functional verification of the design. Silicon compilers take a high level description of the circuit and convert it into an ASIC layout. Creating an ASIC this way depends on the availability of a correct (proven or simulated) high level description.

System requirements analysis and high-level architecture design use behaviour descriptions of the system components. The behaviour of these system models has been checked and approved. The abstraction level of current 'silicon compiler' languages is too low to match these behaviour descriptions. Several intermediate descriptions of the system are needed to close this 'semantic gap'.

The design path described here uses two intermediate levels:

- *Algorithmic level.* Design modules described at this level contain hardware oriented interfaces and algorithms. This level is used to remove most of the abstract-ness of the behaviour description.

- *'Basic building block' level.* The algorithmic level description is translated into real data paths and controller structures. These are specified in a 'language' which uses basic building blocks like *registers, RAM's, ROM's, queues, stacks, Content Addressable Memories, operators* and *state machines.* This level is used to optimise the implementation of the algorithms.

Conversion between the levels is done by compilers. Non-optimal conversion results may be improved manually. The resulting basic building blocks can be handled by silicon compilers.

The algorithmic level entities and basic building blocks can be seen as very specialised behaviour descriptions. The behaviour level model as introduced in chapters 2 and 3 can be used to simulate these entities. This is not done because simulation would be slowed down enormously. The specialised low-level entities have much less simulation overhead than behaviour level entities. This is needed because a large amount of them is necessary to describe the system.

This chapter outlines the steps to be taken to turn the high-level system architecture components into silicon. The next section describes software implementation issues for new hardware.

6.1 Software implementation issues

Complex Abstract Processing Entities will be implemented as a processor-like architecture, driven by a 'machine language' program. Development tools are available for standard machine languages. These allow the actual algorithms to be specified in a more or less abstract language. This helps the programmer to concentrate on the algorithms and disregard the hardware. Section 6.1.1 outlines the tools necessary to perform program development for a new processor.

These programs must be interfaced to the hardware which surrounds the processor core. This is the subject of section 6.1.2.

6.1.1 Toolbox construction for new processors

A set of software design tools must be created for each new processor architecture:

- *An 'assembler'* to translate symbolic descriptions of the processor's instructions into the 'machine code' bit patterns which are understood by the actual hardware.

- *One or more 'compilers'* which translate high-level programming languages into machine language.

  > The ideal compiler would take the behaviour description of the Abstract Processing Entity and turn this into executable machine code.

- *Machine level simulators* which are able to simulate the execution of the machine language. This allows the running of programs before any actual hardware has been designed.

- *Debuggers* which allow testing of- and removing errors in programs written in any of these languages. To aid debugging of programs running on the actual processor, extra hardware may be added. This hardware finds it's roots in the high-level system behaviour aspect *'system maintenance and testing'* (section 4.4.4).

The most important step in this process is the definition of the machine language - the actual instructions which will be executed directly in hardware. An assembler can be created when symbolic names have been assigned to these instructions. Compilers can be created when the semantics of these instructions have been defined in a machine readable form. This information can also be used to create machine level simulators and debuggers. Building all these tools may be done by specialised programs (**[you88]**).

Generating a new machine language from a behaviour description of an Abstract Processing Entity is a far from trivial task. The Algorithmic Level description of a processing entity provides a better starting point. This description must be analysed to find the necessary short-term storage locations and common operations. The storage locations become registers and on-chip memories. The common operations become the instructions of the processor. The actual machine language is formed by assigning bit patterns to each of these instructions.

6.1.2 Interfacing languages with hardware

The basic interface between the software and hardware domains is the assembly language. All instructions coded in the assembly language are one-to-one translated into machine code. The bit patterns which comprise the machine code are directly executable by the processor's hardware.

Input and output is performed by a few instructions which access specific memory locations or 'ports'. Synchronising to external events is done with interrupts. Controlling devices this way requires some extra 'intelligence' in the form of a device control program. A device control program is a set of machine language routines which can be called from any language.

Complex (input/output) operations may also be performed directly in hardware. Special instructions are used to initiate these operations. The T9000 'Transputer', for instance, can transfer information packets with a single instruction ([pou91]). Instructions like these are disguised as procedure or function calls in a high level language. Some languages call such operations *'primitives'*. This is a very appropriate term because the operation of these instructions is defined in lower-level languages only.

6.2 Coupling behaviour to lower language levels

The first step to be taken when converting an Abstract Processing Entity into hardware is to define the exact interfaces to be used. These interfaces couple the processing entity with the chosen communication channel implementations.

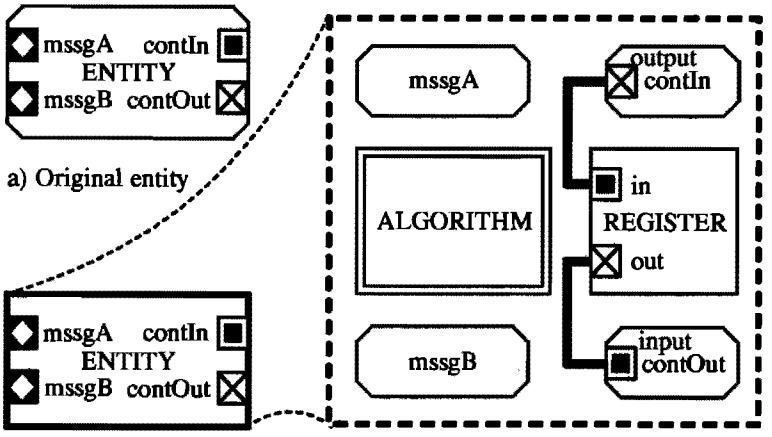
Three language abstraction levels are used in the design path which is described in this text. A behaviour level language is used during high-level behaviour analysis and architecture synthesis. This chapter adds an algorithmic level language and a 'basic building block' level.

Simulation of a system described at several abstraction levels should be possible. This allows abstract system elements to function as test environment for the more detailed system elements.

Mixed level simulations like these give problems in interfacing the different language levels. The approach taken here is to use very hardware-oriented interface primitives like registers, queues and dual port memories. These primitives are themselves basic building blocks. They can be directly accessed from within algorithmic level descriptions. The next section describes the interface to the behaviour level.

6.2.1 Low-Level Simulation Entities

Replacing an Abstract Processing Entity by a *'Low-Level Simulation Entity'* starts the implementation process. A Low-Level Simulation Entity resembles a grouping entity because it contains other elements. Connectors for message channels and continuous data channels may be present at it's boundary. The function of the 'Entity Communication Diagrams' is taken over by *'schematics'*. As shown in figure 6.2.1-1, a Low-Level Simulation Entity schematic may contain the following elements:



b) Low-Level Simulation Entity replacement with contents

Figure 6.2.1-1: A Low-Level Simulation Entity and it's internal structure

- *Interface entities.* These couple the internal entities to the message driven environment (entities 'mssgA', 'mssgB', 'contIn' and 'contOut' in figure 6.1.1-1b).

- *Interface primitives.* These are actually basic building blocks used to interconnect the different abstraction levels (like entity 'REGISTER' in figure 6.1.1-1b).

- *Algorithmic level entities.* These are depicted by a symbol like entity 'ALGORITHM' in figure 6.1.1-1b.

- *Basic building blocks.* These are introduced in the Low Level Simulation Entity when the algorithmic level entities are converted into datapaths and controllers (section 6.4).

The interface entities and interface primitives are described in the following sections. Algorithmic level entities are described in section 6.3.

6.2.2 Interface entities

All connectors placed in the symbol of a Low-Level Simulation Entity are internally provided with an interface entity. These interface entities perform four separate functions:

1) *They handle the messages which are received by a connector.* Message reception is translated in the manipulation of the interface primitives. The message interface entity has all the capabilities of a message filter as described in section 2.3.1. This function is described with method-like texts like the following example (which may be placed in the 'mssgA' interface entity in figure 6.2.1-1):

```
replaceValue

"Tell the 'REGISTER' interface primitive entity to load the value
 which is present at it's input connector."

REGISTER load "this is a standard register command"
```

2) *Interface entities monitor interface primitives for specific changes.* When one of these occurs, a message is formatted and sent across a message connector. The following example shows how the 'mssgA' interface entity checks the 'REGISTER' in figure 6.2.1-1:

```
"Specify the condition under which the message must be sent:"

(REGISTER = 255)

"Send the 'limit reached' message when this occurs."

limitReached
```

3) *Values input on a continuous data input are decoded and placed on internal data buses.* This may be done with true three-state outputs. The following example is the description of the 'contIn' interface entity in figure 6.2.1-1:

```
"Convert an external integer value into  an 8 bits wide integer:"

output := contIn width: 8
```

4) *Values present on internal data buses are encoded and output on a continuous data output.* Full control of the 'three-state' capabilities of the output is available (see section 3.4). The following example is taken from the 'contOut' interface entity in figure 6.2.1-1:

```
"Output the REGISTER value as an integer when it is not 255:"

contOut : =
  (input = 255)              "check for limit"
    ifTrue:  [nil]           "place in three-state condition"
    ifFalse: [input asInteger] "convert to normal integer"
```

There are two reasons for choosing this interface method between the between the behaviour level and algorithmic/basic building block levels:

- *Fixing the interface.* Interfaces must be well defined to allow the interconnection of the system elements at the end of the implementation phase. This definition is formed by the hardware interface primitives and the interface language specifications. The hardware interface primitives themselves are basic building blocks. These remain unchanged when moving down the chain of language levels.

  The data type used by both the algorithmic level and basic building blocks represents an integer with a specified (and fixed) number of bits. The interface languages have the means to convert high level data types into these integers and vice verse.

- *Language compatibility.* The interface language is a modified version of the language used for the behaviour entity methods. The way this language tests and manipulates the interface entities is almost the same as done by the algorithmic level language. The basic building blocks allow the same constructs and add the possibility to transfer data across 'physical' data buses.

6.2.3 Hardware interface primitives

Hardware interface primitives provide a consistent interface between the behaviour level, algorithmic level and basic building block level entities in a design. Together with the interface entities described in the previous section, they define how abstract messages are received and sent by hardware constructs. There are three basic interface methods between the different abstraction levels. These all base upon elements of a basic building block design:

1) *Reading and writing data stores located within basic building blocks.*

2) *Reading the value on a data bus.*

3) *Sending commands to basic building blocks to control the handling and storage of data.*

All operations within a Low-Level Simulation Entity are timed by a 'clock' signal. These operations include the hardware interface primitives.

The next sections describe the hardware interface primitives in more detail. A complete description of the basic building blocks is given in **[ver90c]**.

6.2.3.1 Direct bus connections

Data buses provide a means to transfer data values. These data values are of the standard fixed-bit-width integer type. A bus can only transfer a single value within each clock cycle.

- **Behaviour level bus manipulation:**

  Direct bus connections are the equivalent of the behaviour level continuous data channels. These two may be coupled by special interface entities, as indicated by functions **3)** and **4)** in section 6.2.2.

  An interface entity which receives a message can send commands to hardware interface primitives (see section 6.2.3.7). Such a command may force a specific value on a bus.

  A message may be sent when an interface entity detects a specific value on a bus.

- **Algorithmic level bus manipulation:**

  Algorithmic level entities have no direct connections with data buses. Placing a value on a bus must be done indirectly by sending commands to blocks which are connected to the bus. Algorithmic level entities can check the value on a bus just like the behaviour level interface entity.

- **Basic building block bus manipulation:**

  All actual data transfers between basic building blocks are performed by the data buses. State machines which control algorithms are capable of testing bus values just like the behaviour level and Algorithmic Level entities.

6.2.3.2 Synchronization with signals

'Signals' are single bit semaphore-like storage locations which are attached to a basic building block design. The 'scope' of a signal is the complete Low-Level Simulation Entity in which it is defined. They can be tested and manipulated by all language levels. Signals are normally used for synchronization purposes.

6.2.3.3 Registers and synchronizers

A registers is a data store which can hold a single integer with a fixed bit width. Each register model includes a 'flag' bit. This bit is set automatically when the register is loaded with a new value. It can be reset on command.

All language levels can read the contents of the register and it's flag bit. The behaviour level and algorithmic level languages can directly assign values to a register. These assignments are actually executed at the next clock. Other basic building blocks use a bus to present data to a register. A **'load'** command will then load this value into the register at the next clock. A bus may also be used to distribute a register's value to other basic building blocks.

Each Low-Level Simulation Entity has it's own clock signal. This clock assures that data transfers within a Low-Level Simulation Entity occur reliably. The clocks are not synchronized between different Low-Level Simulation Entities present in a system, they may run at different frequencies. Values transferred from one Low-Level Simulation Entity to another may reach this other Low-Level Simulation Entity just before or after it's clock. The same may happen with messages received from basic behaviour level entities.

Normal register implementations show unwanted behaviour when input signals change within a short time interval around the clock instant. This behaviour may range from abnormally long settling times to the destruction of the register.

The hardware solution to this problem is called a *synchronizer*. Synchronizers are special registers which are capable of handling signals which are not synchronised to the clock. They may take several clock cycles to load a signal coming from the outside world. This behaviour can be modelled in the behaviour level interface entities.

6.2.3.4 Queues

Registers provide a single storage location. The *First-In-First-Out* (FIFO) memory provides a queue which can hold multiple data words. Data is read from this memory in the same order it was written.

The basic building block FIFO memory allows reading and writing one word per clock cycle. Reading and writing can be done at the same time. The number of words in the memory can be checked to allow the implementation of flow control algorithms.

6.2.3.5 Multiport Random Access memories

A FIFO imposes a strict ordering upon the handling of the data written into it. It is also an inherently *unidirectional* communication method.

Both these restrictions can be lifted by using *'Random Access Memories'* (RAM's). The storage locations within a RAM can be written and read in any ('random') order. Multidirectional communication can be implemented by giving communicating processes independent access ports to the RAM.

The basic building block RAM model has separate read and write ports. Each port allows a single data transfer per clock cycle. No flags are available to indicate whether a storage location is free or written. Controlling the traffic through a multiport RAM has to be implemented separately.

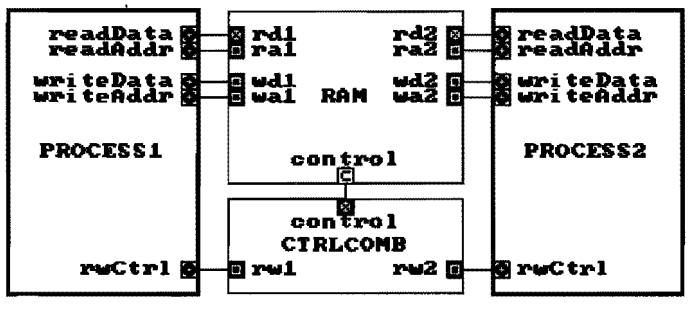


Figure 6.2.3.5-1: Dual-port RAM communication example

Figure 6.2.3.5-1 shows two processes interconnected by a dual-port RAM. Each process has a separate read and write port, controlled by 'rwCtrl' buses. These buses are combined in the basic building block 'CTRLCOMB' before being presented to the 'control' connector at the RAM.

6.2.3.6 Content Addressable Memories

Data words which are transferred between communicating processes may encode commands in their bit pattern. The reading process(es) may have a preference for some

commands. Searching these commands in a Random Access Memory must be done sequentially and takes a long time.

This can be avoided by using a *'Content Addressable Memory'* (CAM). All storage locations within a CAM can be matched against a given data word in a single clock cycle. Specific bits in the data words may be ignored during the search. Each search action provides the number of matched words and the first matched word. Bits in the stored data can be modified as a result of the matching process.

The basic building block CAM model allows a single search-and-modify action per clock cycle. The stored data words may be extended with flag bits to indicate their status. The logic to generate and mask these flags must be provided external to the CAM.

6.2.3.7 Controlling low level entities directly

Basic building blocks are controlled by sending them commands. These commands may be generated by several sources:

- *State machine controllers.* These are themselves basic building blocks.
- *Algorithmic level entities.*
- *Behaviour level message interface entities.*

The basic building blocks check all commands which are received. Some commands may only be given when specific connectors are present. Other commands may not be combined within a single clock cycle.

6.3 Converting behaviour to hardware oriented algorithms

The first step towards hardware implementation of an Abstract Processing Entity is the conversion into an *Algorithmic Level* equivalent.

The hardware interface architecture was defined in the previous section. This included the conversion of the abstract data types into hardware compatible integers and vice verse.

This hardware interface provides the outer shell of the Abstract Processing Entity implementation. The Algorithmic Level entity introduced in this section describes the internal operations. This description shares it's operators and 'data type' with basic building blocks.

An Algorithmic Level entity provides a way to describe algorithms which can be implemented in hardware. The actual implementation in datapath and controller architectures is postponed until later.

The next section describes the capabilities of the Algorithmic Level entity. Section 6.3.2 outlines the behaviour level to algorithmic level conversion.

6.3.1 The Algorithmic Level entity

An Algorithmic Level entity provides local data storage and procedures which operate on this data. These procedures also have access to basic building blocks placed in the 'environment' of the Algorithmic Level entity. This environment includes all elements drawn on the same schematic as the Algorithmic Level entity. Each of the Algorithmic Level entities in a design is a separate concurrent process. These characteristics are described in the next sections.

The Algorithmic Level entities are based upon the *'Hardware Oriented Design and Simulation System'* described in **[hul90]**.

6.3.1.1 Local data storage

The data storage located within an Algorithmic Level entity can only be accessed by the internal algorithms. These local data stores represent simple registers and Random Access Memories which can be written and read just like their interface primitive counterparts. There is a significant timing difference between the interface primitives and the local data stores, which is explained in section 6.3.1.5.

Complex 'local' data storage can be provided to an Algorithmic Level entity by placing basic building block memories in it's environment. Complex data stores like these remain intact when the Algorithmic Level entity is converted into basic building blocks.

6.3.1.2 Interfaces

The procedures placed within an Algorithmic Level entity have access to the basic building blocks placed in their environment. External registers and RAM's can be written directly. Other basic building blocks must be controlled by sending them commands. Obtaining values from external entities (including buses) can be done by direct name reference.

6.3.1.3 Basic Algorithmic Level language constructs

The language used for Algorithmic Level entity procedures provides most of the language constructs found in ordinary algorithmic languages:

- *Assignment and control expressions.* Expressions operate on values stored in local variables or basic building blocks placed in the environment of the Algorithmic Level entity. The results of these expressions can be assigned to (local) registers or RAM locations. The expression operators are the same as those used in the basic building block descriptions.

  Commands can be sent to all basic building blocks placed in the environment. Parameters for these commands may be given in the form of an expression.

- *Flow control.* Conditional evaluation of program segments is possible by the use of a 'CASE'-like construct. The same construct is also used for 'IF-THEN-ELSE' tests.

  All standard loop constructs are available, including a special 'endless' loop. Special constructs allow breaking out of a loop.

- *Subroutines.* Large programs can be broken up into a main routine and one or more subroutines. There are two kinds of subroutines:

  - '*Local*' subroutines can only be called from within the Algorithmic Level program itself.

  - '*Global*' subroutines can also be called by other Algorithmic Level entities.

  Parameter passing between routines must be done with the normal external variable storage entities.

6.3.1.4 Concurrent programming

Each Algorithmic Level entity is capable of running a single 'program' at a time. All Algorithmic Level programs placed within a Low-Level Simulation Entity can be running concurrently. Communication between Algorithmic Level entities is performed using the hardware interface primitives described in section 6.2.3.

An Algorithmic Level entity can call global subroutines within other Algorithmic Level entities placed in it's environment. There are two ways to request a remote subroutine call:

- *A subroutine call* blocks the calling entity until the external call has been handled.
- *A co-routine call* allows the calling entity to continue as soon as the call has been posted.

Each Algorithmic Level entity provides a priority system like the one described in section 2.4.1 for the behaviour level processing core:

- *The entity is always operating at a specific priority level.* This priority can be manipulated by the internal procedures.
- *The main routine starts at the highest possible priority level.* The priority is lowered to the lowest possible level when no routine is running.
- *Global subroutines are started when their request has a higher priority than the operating priority.* The operating priority is set to the priority of the request. Other requests are queued in priority FIFO order.
- *Remote global subroutine calls by default receive the operating priority as priority.*

These capabilities can be used to design complex client-server architectures at an algorithmic level. Each basic behaviour level entity can be broken up into a set of cooperating Algorithmic Level entities.

6.3.1.5 Timing

Algorithmic Level entities are placed in an environment which contains basic building blocks. The Algorithmic Level entities must be synchronised to the clock used by these basic building blocks. This synchronization is only needed for external accesses and timing purposes. Timing characteristics of an Algorithmic Level entity can be summarised as follows:

- *Clock synchronization.* An Algorithmic Level program contains 'wait' statements. Execution of such a statement stops the program until at least the next clock. The waiting period may be stretched by specifying specific conditions which must be true to exit the wait state.

- *Interfacing external entities.* Reading external entities always returns steady state values (as if it was done just before the clock). Manipulation of external entities is buffered. The actual changes are postponed until the next clock. This gives the external entities the opportunity to check for illegal command combinations and multiple assignments.

- *Handling local variables.* Assigning a value to a local register or RAM is done immediately. Reading a local variable always reflects the last value assigned to it.

Figure 6.3.1.5-1 shows an Algorithmic Level entity which is being used to calculate the Fibonacci series ($X_N = X_{N-1} + X_{N-2}$, $X_0 = X_1 = 1$, $N > 1$). The resulting values are placed in the external variable 'OUTPUTREG'. The actual algorithm shown in the right window makes use of the fact that assignments to external variables are not executed before the next clock. 'OUTPUTREG' is first assigned X_{N-1}, which becomes X_{N-2} in the next loop. The 'OUTPUTREG' value used in the addition has been assigned during the previous loop.

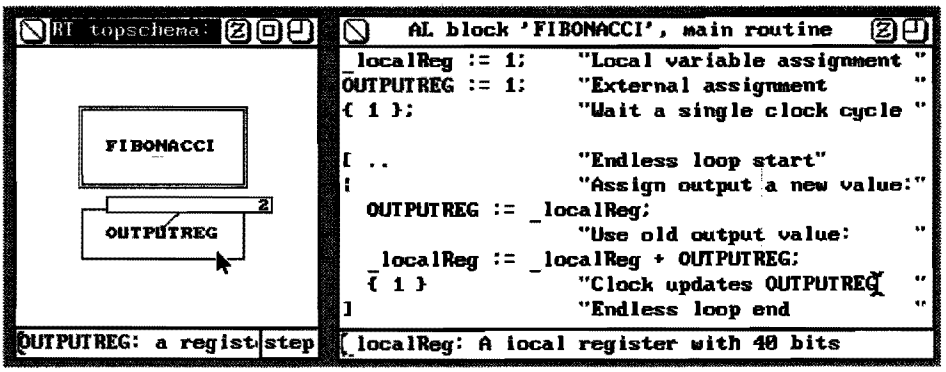


Figure 6.3.1.5-1: Example of an Algorithmic Level entity

6.3.2 Converting behaviour to algorithms

The interface entities, interface primitives and Algorithmic Level entities have been introduced in the previous sections. This section describes how a behaviour level Basic Model entity can be converted into an Algorithmic Level equivalent. This conversion is a four-step process:

1) *Convert the data types*
2) *Create the hardware interface*
3) *Convert the message handling shell*
4) *Convert the processing core*

These steps are detailed below.

Step 1: Data type conversion

The behaviour level uses all kinds of data types for parameters and stored values. Equivalent integer-based data types must be defined for use by the algorithmic level. The actual conversion between complex data types and integer values is handled by the interface entities. Equivalent integer-based operations must be formulated for the operations performed on the behaviour level data types.

Step 2: Interface boundary conversion

The format of messages received and sent by the Algorithmic Level system must be defined down to the bit level. This includes the placement of the parameters attached to the messages.

Messages may contain such complex data types that they cannot be handled as a whole. It is also possible that the channel implementation choice enforces a sequential protocol. In both cases, messages have to be broken up into pieces which must be handled sequentially by the interface primitives.

Message format conversions are handled by the message connector interface entities. These entities also handle accepting, absorbing, ignoring, blocking and rejecting messages. Any conditional decision in this respect must be based upon the state of interface primitive entities.

Step 3: Message interface architecture conversion

The interface shell present in a behaviour level entity must be converted into an Algorithmic Level equivalent:

- *Input filter* functionality should be implemented in Algorithmic Level entities and/or basic building blocks. Forwarding the handling decision to the communication channel must be performed by the message interface entity.

- *Input and output buffers* are implemented with interface primitives. A simple buffer may be implemented with a register or FIFO memory. Multiport RAM's, Content Addressable Memories and local Algorithmic Level entity memories may be used in complex situations.

- *The message selector/manipulator* is complex enough to warrant the use of a separate Algorithmic Level entity. This separates message selection and handling.

The virtual connector translation table is a static resource within the message interface shell. The contents of this table do not change. It's functionality can be completely incorporated within the implementation of the processing core.

Step 4: Processing core implementation

The behaviour objects can be implemented with one or more Algorithmic Level entities:

- *Variables* used in the original behaviour objects can be placed in local Algorithmic Level variables or basic building blocks.

- *The main routine* can be used to initialise these variables and select messages when this is not done by external means.

- *Subroutines* can be used to implement the different methods. Global subroutines can be used when these must be invoked by an external message selector.

- *The operations* on the original abstract data types have been converted into integer handling equivalents in step 1. Complex operations should be placed in local subroutines.

- *Input and output* actions are performed by the appropriate manipulation of the interface primitive entities.

The conversions described above are relatively straightforward. Some optimisations may be done during the conversion:

- *The operations have been detailed to a much deeper level.* This allows a re-scheduling of the behaviour level timing specifications. Reducing peaks in the processing load will also lower the amount of hardware needed.

- *Extra concurrency may be brought into the system by applying multiple Algorithmic Level entities.* These can perform complex operations and message handling in parallel.

6.4 Converting algorithms to datapaths and controllers

An Algorithmic Level entity describes a piece of data processing hardware. The operations and storage structures present in an Algorithmic Level entity are already in the realm of digital hardware. What is lacking is the scheduling and assignment of variables and operations:

- *It is not specified which operations are done within the same clock cycle.* Only a rough indication is given by their placement between the wait statements.

- *The Algorithmic Level entities' local variables are only a rough indication of the needed data storage.* Data stores may be combined when they never contain active data at the same time.

- *Nothing is said regarding the actual architecture of the final system.* No indication is given of the use of data buses and the sharing of operational hardware.

The goal of basic building block hardware design is the following:

Create an actual architecture containing storage, data processing and control elements out of a description which only provides a functional specification.

The basic building blocks are the lowest-level design elements which need human intervention. They can be converted into logic gates by logic synthesis programs. Layout fragments for these gates are stored in libraries. These fragments can be placed and interconnected by placement and routing programs. The end result is a complete ASIC layout. This process can be performed fully automatically (**[baa91]**, **[zan91]**). Appendix 2 gives some examples of these conversions.

6.4.1 Basic building block design elements

The next sections describe the basic building blocks in somewhat greater detail. A complete description is given in **[ver90c]**. Section 6.4.2 outlines how an Algorithmic Level design can be converted into basic building blocks.

6.4.1.1 Data storage

All communication primitives which provided storage (sections 6.2.3.3 through 6.2.3.6) are available for data storage in the basic building block design:

- *Registers.*
- *Random Access Memories.* These can have multiple read and write ports.
- *First-In-First-Out memories.* These implement a multi-level queue. Data is read from the 'head' of the queue and written at the 'tail'.
- *Content Addressable Memories.* These allow searching all words in parallel by matching bits with a given reference word. Multiple words can be changed within a single clock cycle.

The next memory type is a very common data storage structure which is seldom used for communication purposes:

- *Last-In-First-Out memories.* These implement a multi-level *'pushdown store'* or *'stack'*. Data manipulations are always performed at the 'head' of this store.

The *signals* introduced in section 6.2.3.2 can be used as single bit data storage.

6.4.1.2 Data transfer with buses

Data transfer between basic building blocks is performed by data buses. A bus must be connected to the basic building blocks by *'connectors'*. Each bus carries a fixed number of bits and can only be connected to connectors which have the same width. The values transferred by a bus are fixed bit width integers with the specified number of bits.

Multiple output connectors may be attached to a bus. Only one of these may be active during a clock cycle. The other output connectors on that bus must be held in a 'three-state' condition (like the 'bBusOut' and 'cBusOut' outputs shown in figure 6.4.1.3-1).

The state of output connectors is controlled by sending commands to the basic building blocks they are placed in.

6.4.1.3 Data manipulation with operators

'Operators' are basic building blocks which can perform arithmetic and logic operations. An operator can have multiple input and output connectors. The values of the output connectors are defined by a set of expressions. These expressions use the names of the input connectors as parameter. The operators used by the expressions are the same as those used by the Algorithmic Level entity assignment expressions.

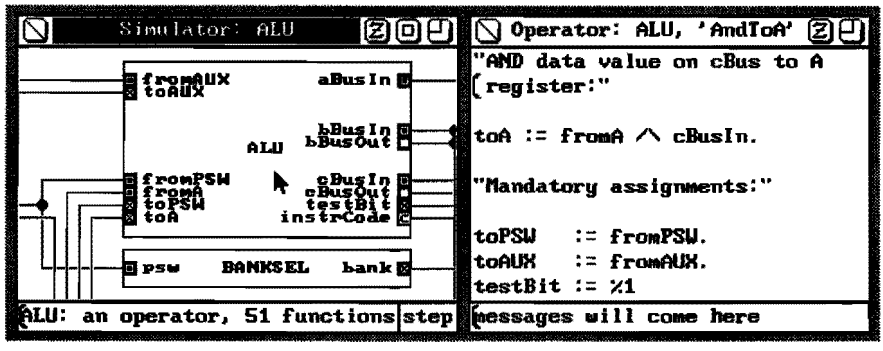


Figure 6.4.1.3-1: Basic building block operator example

Each set of expressions defines a *function*. An operator may contain multiple function definitions. Only one of these may be active during a clock cycle. Figure 6.4.1.3-1 shows an operator symbol ('ALU' in the left schematic window) and the description of one of it's functions in the right window. Each function is described by a separate text.

6.4.1.4 Control with state machines and microprograms

Finite State Machines can be used to control other basic building blocks. Controlling is done by sending abstract commands to these blocks. A FSM can test values present in other design elements. The results of these tests may influence the generation of control signals and state transitions. The FSM basic building block can be used to describe the following three basic control structures:

- *'Mealy' state machines:* Both control signals and state transitions can be conditional.

- *'Moore' state machines:* The control signals generated in a state are always the same. Only state transitions can be conditional.

- *Microprograms:* The control signals generated in a state are always the same. Each state has a default 'next' state. Only a single conditional state transition per state is allowed.

The FSM basic building block can model a subroutine stack. Interrupting a FSM is possible when the stack is available.

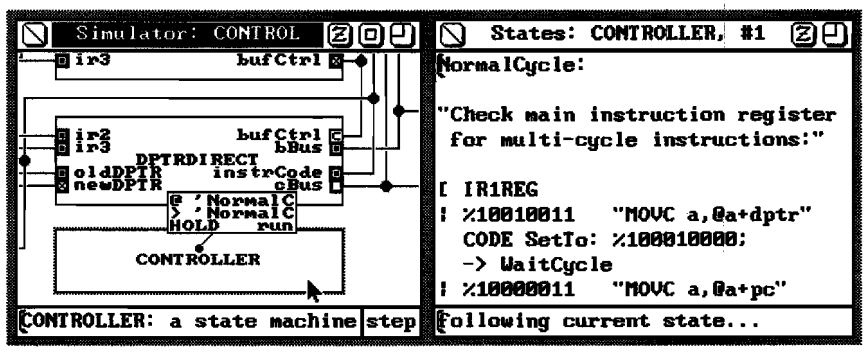


Figure 6.4.1.4-1: Basic building block FSM example

Figure 6.4.1.4-1 shows an example of a basic building block Finite State Machine symbol ('CONTROLLER' in the left window) and its description (right window). Each state is defined with a separate text. This text starts with a symbolic state label ('NormalCycle' in the example). The text may contain symbolic commands which are sent to other basic building blocks in the design. The example shows how the basic building block named 'CODE' is sent the command 'SetTo:' with parameter %100010000. State transitions are indicated with symbols (like '->') followed by state labels. Conditional constructs start with the value which must be tested (the contents of the 'IR1REG' in this example), followed by a list of possible values and accompanying actions.

6.4.1.5 Distributed control structures

Distributed control architectures use a data bus which carries encoded control values. This bus is attached *'control connectors'* placed in all system elements which must be controlled. Each of these control connectors generates local control signals based on the bus value.

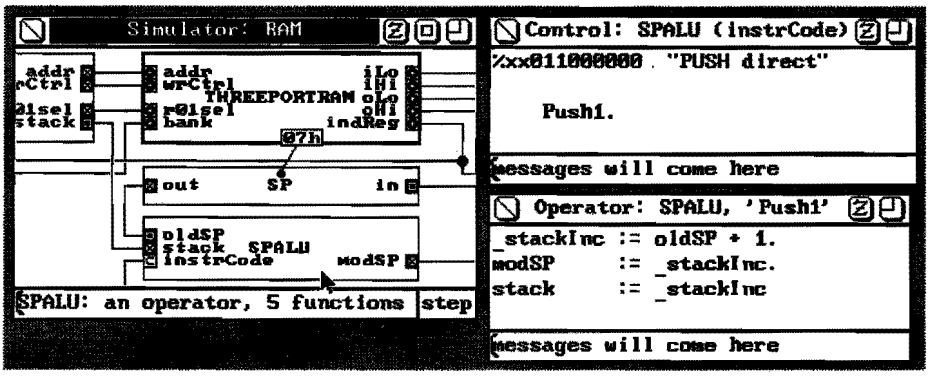


Figure 6.4.1.5-1: Basic building block control connector example

Figure 6.4.1.5-1 shows an example of the use of a control connector. The bottom right window shows one of the functions which can be performed by the operator 'SPALU' in the schematic window. The top right window shows the description of the 'instrCode' control connector which controls this operator. It shows the value (each 'x' stands for a 'don't care' bit) which must be present on the control bus to let the operator perform the 'Push1' function.

6.4.2 Converting algorithms to register transfers

An Algorithmic Level entity defines algorithms in a very hardware oriented form:

- *The only data type is an integer with a fixed number of bits.* Basic building blocks use the same data type.
- *Interface primitives are used for communication with the environment and other Algorithmic Level entities.* These interface primitives are basic building blocks.
- *An Algorithmic Level entity is aware of the fact that it's environment is timed by a clock signal.* Interactions with entities placed in the environment are synchronised by 'wait' statements.

Two operations remain to be performed to convert an Algorithmic Level entity into basic building blocks:

- *Scheduling:* The Algorithmic Level expressions have to be assigned clock cycles during which they will be performed.

- *Assignment:* The Algorithmic Level operations and local data stores must be placed in basic building blocks. These must be interconnected to form a *'data path'*.

These operations are outlined in the next two sections.

6.4.2.1 Operation scheduling

Assigning operations to happen during specific clock cycles is a very complicated optimisation problem. Assigning a large amount of operations to each clock cycle will result in a fast system which uses a large amount of hardware. Finding a sensible balance between speed and hardware cost is a difficult task. The approach taken here is a two step process. First, the algorithm is speeded up as much as possible. Second, the designer is allowed to reduce the hardware cost by removing parallelism. These steps are outlined below.

Phase 1: Optimise for speed

The Algorithmic Level entity description is converted to a FSM-like form. All high-level conditional and looping constructs are transformed in state transitions. Several techniques are then applied to optimise this state machine (**[bud92]**):

- *Expression forwarding.* Expressions which may be executed earlier are moved forward to other states. This may remove temporary variables.
- *Expression optimisation.* This may reveal that some expressions evaluate to a constant value. Conditions may become fixed as a result of this.
- *State elimination.* States without expressions are either removed or combined with other states.
- *Variable lifetime analysis.* Storage locations which never contain active values at the same time may be combined.

Basic building block interactions should be handled with care. Moving these around might crash externally defined communication protocols.

Phase 2: Reduce hardware requirements

The speed-optimised FSM uses a large amount of hardware to perform many operations in parallel. This is the fastest possible implementation of the original Algorithmic Level entity. The designer is given two methods to reduce the hardware requirements:

- *Specify limits for the amount of operations which may be performed in parallel.* The designer cannot influence how the operations will be re-scheduled to make use of the reduced hardware.

- *Increase the number of clock cycles allowed to perform operations.* How these operations will be distributed across the extra clock cycles is not under control of the designer.

Applying these methods interactively allows a designer to fine-tune the final schedule. Manual intervention should be possible.

6.4.2.2 Data path synthesis

The operations must be performed by interconnected basic building blocks, the so-called *'data path'*. Defining this architecture is done in four steps:

1) *Optimise the operations*
2) *Create storage entities*
3) *Build the actual data transfer paths*
4) *Handle Algorithmic Level co- and subroutine calls*

These steps are outlined below. The operations performed in the FSM description are replaced by control commands. This FSM can then be replaced by a basic building block state machine.

Step 1: Operation optimisation

The basic operations described in the scheduled FSM description can be optimised to share hardware:

- *Bit width conversions may be used to make better use of operator hardware which is needed anyway.*

- *Complex operations can be broken down into smaller parts.* These may be used separately or in other combinations during other clock cycles.

Step 2: Creating storage entities

The local Algorithmic Level storage facilities are converted into basic building blocks. The maximum number of read and write operations in a single clock cycle is determined. This value defines the number of necessary read and write ports. The actual read and write operations are assigned to these ports.

Step 3: Building data transfer paths

The assignments specified in the FSM description must be performed by connecting buses between read and write ports. Operators are inserted in these transfer paths to perform the necessary operations found during step 1.

Step 4: Algorithmic Level co- and subroutine calls

Standard data paths are appended to handle the external co- and subroutine calls. This hardware should handle priority related functions.

Operations which were 'atomic' in the original Algorithmic Level description may now be distributed over several clock cycles. Intermediate results are held in temporary storage locations. Interrupting such a sequence may cause problems.

6.4.3 Re-using old designs

Basic building block designs need not always be created from scratch as done in the previous sections. Hardware architectures ranging from very specific to general purpose processors can be placed in libraries. Combinations of these architectures expand the solution space even further. The processes of selection and modification are outlined in the next two sections.

6.4.3.1 Selecting a design for re-use

Selecting a basic building block design from a library follows the same principle as outlined in section 5.4.2. Each of the designs is given a 'profile' which states it's capabilities. A similar profile is deduced form the problem statement. An expert system can be used to match profiles and select an initial set of architectures to choose from.

Some simple changes to a design may make it the best solution. This flexibility is very hard to incorporate into a profile. Human intervention is needed to guide the expert system and make the final selection.

6.4.3.2 Parametrization and modification

An existing basic building block design may be parametrized at the following points (without changing the actual architecture):

- *Width of system elements.* Changing the number of bits in a word alters the precision of calculations.

- *Number of words in storage entities.* Memory sizes may be 'cut-to-fit' to a particular problem.

- *Functions in an operator.* Changing operator functions modifies the operations performed in the data path.

- *FSM states.* Modifying the control sequences allows a data path to perform totally different algorithms.

Modifications to an architecture can be made by simply adding and removing basic building blocks. Complete hierarchies of basic building blocks can be combined or split with only a few design actions.

6.5 Converting basic building blocks into ASIC's

Basic building blocks are meant to be implemented in hardware. All storage structures can be converted into parametrisable standard architectures. Gate-level equivalents exist for each of the basic operators. These can all be described in standard Hardware Description Languages like VHDL ([iee88]), ELLA ([pra86]) and SID ([sag90]). This conversion can be performed fully automatically by a compiler ([sim90], [baa91], [zan91]). Once converted, standard tools can be applied to generate several forms of hardware implementations:

- *Small and Medium Scale Integration circuits.* This implementation methodology uses of-the-shelf components. Logic circuit density ranges from 1 to 100 gates.

- *Programmable logic.* These are off-the-shelf components whose function can be programmed. Each of these may contain up to several thousands of gate 'equivalents' (actually usable gates).

- *Application Specific Integrated Circuits (ASIC's).* These are specially built circuits which may contain several hundreds of thousands of gates. Around the year 2000, well over ten million gates can be integrated in a single package ([she91]).

The design methodology described in this Ph.D thesis targets very complex systems. Most of these systems have to be implemented by using ASIC's for at least major system parts.

Appendix 2 shows some examples of ASIC's which have been designed with the basic building blocks design and simulation tools. These have been fully automatically converted into a silicon layout via SID.

Several optimisations are performed during the conversion of the basic building block design into Hardware Description Languages. These are described in sections 6.5.1 through 6.5.3.

Testing complex ASIC's is complicated by the fact that only a few connections to the outside world are available. Section 6.5.4 describes how low level testing facilities can be incorporated within the design.

6.5.1 Removing unused functionality

An obvious way to reduce the size of the generated circuit is to remove those parts which are not going to be used. There are two kinds of functionality which can remain unused in a basic building block design:

- *Model introduced overhead.* Basic building block models contain functionality which may remain unused in a specific design.
- *Designer introduced extraneous functionality.* Designers may simply forget to remove some parts of a design.

Parts of a design which do not interact with other design elements may be removed as a whole. Operator functions and state machine states which are never used may be removed too. These situations can be detected by examining the static design description.

6.5.2 Operator optimisation

Operators are used to describe the actual arithmetic and logic operations performed in a basic building block architecture. The functionality of an operator is defined by the functions it can perform. Each function is entered as a set of expressions. Each expression defines an output connector value as a function of the input connector values. Expressions use a set of basic operators to describe the actual arithmetic and logic operations. Each of these basic operators can be directly implemented as a gate level design.

An operator can be directly implemented by interconnecting the gate level equivalents of the basic operators. Selection of the executed function can be performed by multiplexers placed in front of the outputs. The result will be a large network of

interconnected gates. Logic optimisation of this network is a very complex task when thousands of gates are involved.

Optimisation at the expression level is needed before converting the basic operators into gates. The following list gives some examples of what can be done at this level:

- *Propagate and generate constants.*
- *Make use of normally unused functionality in the standard gate level implementations* (for instance the carry input of an adder).
- *Find common constructs in the different functions.* Their hardware can be shared between functions by inserting extra multiplexers.

6.5.3 State machine optimisations

Basic building block state machines are described in a very abstract form. State transitions are specified using symbolic state names. Controlling an entity is done by sending symbolic commands. This allows several optimisations to be performed:

- *State assignment.* Actual state numbers have to be assigned to the abstract state names.
- *Control vector assignment.* Different bit vectors must be assigned to the abstract commands sent to a controlled entity.
- *Constant encoding.* Some commands carry constant parameters. These can be generated directly or in an encoded form. In the latter case, a constant decoder is needed at the controlled entity.
- *Control vector combination.* A single control vector suffices when several entities always receive identical commands. Control vector space can be shared between entities from which only one is active within a clock cycle. In this case, a separate indication is needed to indicate for which entity this control vector is meant.

6.5.4 Incorporating low level test facilities

Hardware is generally built without redundancy. Gates and interconnections which will never be used are removed during optimisation. It is therefore of prime importance to test *all* design elements. Only *'structural'* testing methods provide a systematic means of reaching all design elements.

Structural testing methods work at the gate level only. They explicitly do *not* test higher level functions.

Basic building blocks hide actual gate level implementations as much as possible to achieve technology independence. This means that structural tests cannot be specified in a basic building block design.

Testing logic must be added during the translation of a basic building block design into Hardware Description Language:

- *A scan chain* which connects all registers.
- *Self test logic* for complex storage structures.

With these standard additions, test vectors can be generated automatically by specialised programs in the ASIC design environment.

6.6 Integrating the system

The final acts in the design path are the assembly of the system followed by system integration tests. A system which passes these test can be delivered to the customer.

The system parts which form the final system should be pre-tested before integration. This allows the system integration tests to concentrate on the interconnections between the system parts. These interconnections were specified during high-level system architecture design. Implemented modules must adhere to these interconnection specifications.

System integration is a hierarchical process. System parts are built out of smaller parts. Each of these can be tested separately. This directly follows the hierarchical system decomposition as present at the end of the high-level system architecture phase.

6.7 Summary of the design path, final remarks

The last three chapters described a design path which starts at a - possibly informal - problem statement. The end result is an operational system which solves the problem more or less economically. The design path consists out of three separate phases:

1) *High-level system behaviour analysis*
2) *High-level system architecture synthesis*
3) *Low-level module implementation*

These phases can be summarised as follows:

Phase 1: High-level system behaviour analysis

The original problem statement is analysed. An operational behaviour model of the system is built. An *Object Oriented Analysis* method is followed which results in a set of communicating *Problem Domain Entities*:

- Compile a list of entities which are found in the problem statement, the Problem Domain Entities.
- Give the system a structure by defining *'forms part of'* relationships between Problem Domain Entities.
- Build a superimposing structure by finding *'is a kind of'* relationships between Problem Domain Entities.
- Define communication channels by tracing *'communicates with'* relationships between Problem Domain Entities.
- Determine the global operational *'aspects'* of the system.
- Define the *'message'* protocols to be used in the communication between Problem Domain Entities.
- Define how messages are handled by Problem Domain Entities.

The system behaviour described this way serves as a reference for the other phases. The behaviour has been simulated and is approved by the customer.

Phase 2: High-level system architecture synthesis

Architecture synthesis is performed by re-structuring the system behaviour description. The Problem Domain Entities found during system analysis are mapped onto a set of *Abstract Processing Entities*. The problem domain communication channels are mapped onto *Abstract Communication Channels*.

Preliminary implementation choices are made during architecture synthesis. These are based upon the required capabilities of the Abstract Processing Entities and Abstract Communication Channels and a database containing *'profiles'* of actual processing units and communication channels. Abstract Processing Entities and Abstract Communication Channels may be split and combined to optimise the system.

High-level system architecture synthesis ends when the preliminary implementation choices have been fixed. The result is called the *'Processing Model'* of the system. This model defines how data is processed in the system. It also defines what kind of modules will be used and how these are interconnected.

Phase 3: Low-level module implementation

During this phase, the Abstract Processing Entities and Abstract Communication Channels are implemented in the form chosen at the end of the previous phase. Software implementations can be derived from the Abstract Processing Entity behaviour descriptions. The route to hardware in the form of *Application Specific Integrated Circuits* is as follows:

- The Abstract Processing Entity is replaced by a *Low-level Simulation Entity*. The message interface is converted into a hardware compatible interface using *Message Translation Entities* and *Hardware Interface Primitives*.
- The Abstract Processing Entity-internal *interface shell* and *processing core* are translated into one or more *Algorithmic Level* entities.
- The Algorithmic Level entities are translated into *data paths* and *Finite State Machine controllers*. These are specified by parametrisable *basic building blocks*.
- The basic building blocks are converted into a suitable *Hardware Description Language*. This system description can be converted into an ASIC using a set of standard design tools.

The system modules are actually built and connected together. Following system integration tests, the *design* trajectory is concluded with delivery of the complete system to the customer.

Some final remarks:

- The actual 'life' of a system starts after delivery. Maintaining, upgrading and extending an operational system can be as complex as designing it. These post-delivery operations are covered by the operational aspect *'system maintenance and testing'*.
- Testing plays an important role in the design trajectory. Periodic testing of the system components is necessary to detect gradual degradation and initiate preventive maintenance. Integrated testing facilities are convenient when the system has been modified 'in the field'.
- The Algorithmic Level entities and basic building blocks are not Object-Oriented. They are not data abstractions and lack characteristics like inheritance and polymorphism.

7. Tools

The system design path described in chapters 4, 5 and 6 must be supported by computer based tools. This chapter describes the tools which will be available in the complete 'toolbox'. The emphasis will be on the actual design and simulation tools. These provide the framework to which other analysis, manipulation and conversion tools will be attached.

The next section provides an overview of the needed tools. Section 7.2 gives some general ideas which apply to all of them. Section 7.3 outlines how complex projects can be supported.

7.1 Overview

All tools will be placed within a single design environment:

- *Design and simulation tools.*
- *Analysis tools.*
- *A generalized monitoring tool.* This can be used to generate log files, signal specific events and set 'breakpoints' during simulation.
- *Expert systems.* These guide the designer in taking the correct design steps. They also do optimisations.
- *Data bases.* These contain re-usable design entities.

The next sections describe the tools which are needed for the different design abstraction levels.

7.1.1 Behaviour Level design

The Behaviour Level toolbox is used during high-level system behaviour analysis to model the *Problem Domain Entities*. The same tools are used to model the *Abstract Processing Entities* during high-level system architecture synthesis.

The basic Behaviour Level design and simulation tool is a graphical editing window which allows the drawing of *Entity Communication Diagrams*:

- Symbols are placed on the 'working sheet' to denote basic simulation entities, grouping entities, multiples and multiple groups.
- Connectors are depicted by small symbols located at the boundary of the entity symbols.
- Communication channels are defined by drawing lines between connectors.

Grouping entities are edited by opening a new graphical editing window.

Any entity drawn in an Entity Communication Diagram immediately receives a default behaviour and starts behaving as such. Definitions of message filters, message selectors and behaviour objects are entered in separate edit windows. Other specifications are changed with menus and 'fill-in-the-blanks' forms. Each behaviour change is immediately reflected in the system simulation.

The major operations done during high-level system architecture synthesis are the splitting and combining of entities and channels. These operations will be performed by an *'architecture editor'* tool under direction of the designer.

7.1.2 Algorithmic Level design

An Abstract Processing Entity which must be implemented in hardware is replaced by a *Low-Level Simulation Entity*. Low-Level Simulation Entities are indicated on an Entity Communication Diagram by a special symbol.

The contents of a Low-Level Simulation Entity are defined by the top-level 'schematic' of a basic building block design. This schematic is edited in a separate window. The Algorithmic Level entities, hardware interface primitives and interface entities are all indicated by symbols on this schematic. Sub-schematics may be used to group together several other entities.

Connectors and data buses are depicted in ways similar to Behaviour Level connectors and communication channels. Textual descriptions of Algorithmic Level routines and interface entities can be edited and compiled in separate windows. Drawing something on a schematic or otherwise changing an entity's behaviour is immediately reflected in the simulation.

Entities within a Low-Level Simulation Entity are timed by a clock signal. Each Low-Level Simulation Entity clock runs at a fixed frequency defined by the designer. Simulation of the Low-Level Simulation Entity contents is done clock-by-clock. The system outside the Low-Level Simulation Entity is advanced in time with each clock

tick. Simulating the Behaviour Level system sends the correct amount of clock ticks to each Low-Level Simulation Entity embedded in it.

7.1.3 Basic building block design

Basic building blocks are used to design actual data paths and controllers out of Algorithmic Entity descriptions. These are all depicted with symbols on the schematics which were already used during Algorithmic Level design. Textual descriptions of operators, state machines and control connectors are edited and compiled in separate windows. Memories, signals and subroutine stacks can be monitored and changed using their own dedicated windows. Fully interactive simulation is retained at this level of design. There is no need to stop or even reset simulation when changes are made in the system.

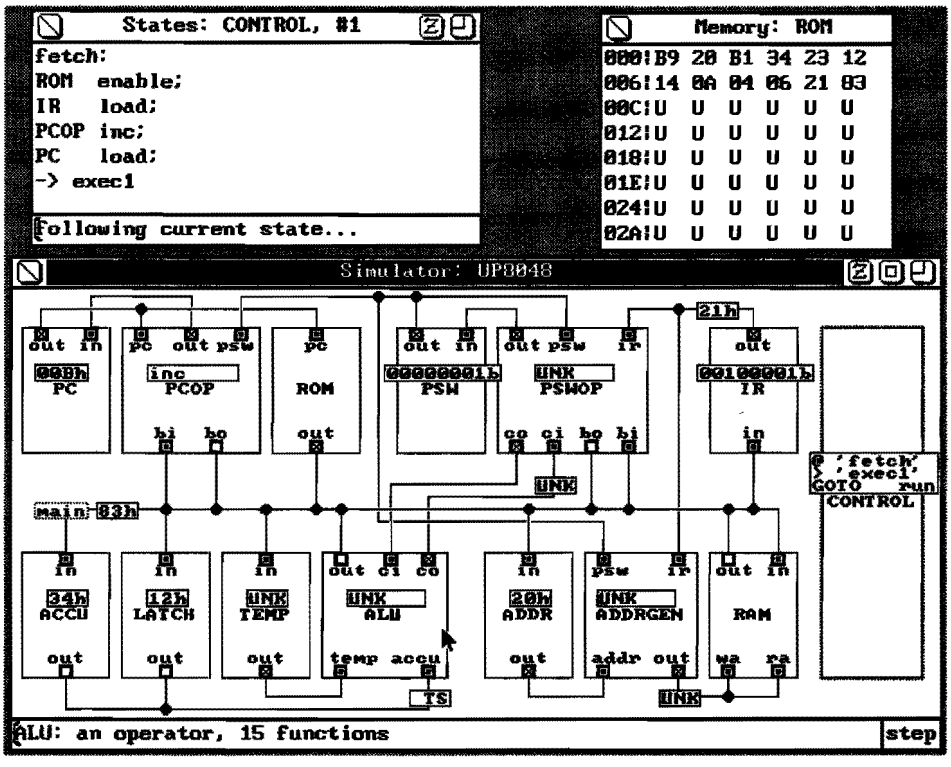


Figure 7.1.3-1: Basic building block design in action

Figure 7.1.3-1 shows a snapshot of a basic building block design session. The bottom window shows the schematic of a microprocessor core. The top right window shows

the contents of the ROM. The top left window shows the first state of the 'CONTROL' Finite State Machine.

7.1.4 Gate Level (and lower) design

The basic building blocks are directly translatable to real hardware structures. The last tool in the system design toolbox translates basic building blocks in a standard Hardware Description Language. The remaining conversions (HDL description to logic gates, gates to transistors and transistors to layout) are performed with standard tools outside the interactive design environment. This allows the system design path to be implementation technology independent.

7.2 Common tool behaviour

The tools used in the design path will be highly interactive. Simulating during entry and modification of a design gives immediate feedback to the designer. This allows an exploratory and stepwise design approach.

These capabilities are obtained by using a mixed graphics/textual description method:

- *Graphical methods are used to define the overall system structure.* An entity is placed in the system by drawing it's symbol on an 'Entity Communication Diagram' or 'schematic'. Hierarchical layering of the design is possible by using symbols which denote complete Entity Communication Diagrams and schematics.

  Connections between the entities are defined by drawing lines on the Entity Communication Diagrams and schematics. These lines start and end at connector symbols located at the edge of the entity symbols.

- *Detailed descriptions of internal entity operations are given in a textual form.* Where possible, the internal operations are broken up into separate parts. This allows an entity's behaviour to be defined, compiled and checked in small steps.

Entities placed on a schematic immediately take part in the simulation. New entities are given a simple default behaviour which can be modified later on. Data transfer starts immediately when a connection is drawn.

Compiling a textual description instantly installs the described behaviour in the system. These compilers thoroughly check for syntax errors. Unresolved references only generate warnings. This is necessary because a textual description may reference an entity which is not yet present in the system. Trying to use such a reference aborts simulation with an error message. The design can be test-compiled to check for unresolved references.

Entities may be designed and tested in isolation. Integration into the system is done by simple cut-and-paste operations. Similar operations are available to move entities between the design and libraries.

The state of the simulated system can be checked by requesting the entities to display their contents or operations. Continuous display of values or operations is done by attaching probes and special inspection windows. A system-wide event monitoring tool will be available. This allows three operations to be performed in case a specified event occurs:

- *Generate a user-defined message.* These can be used to draw attention to very obscure conditions.
- *Place an entry in a log file.* The resulting file can be used to compare other designs to the monitored design. The log file can also be formatted as a set of test vectors and responses.
- *Abort simulation with an error message.* The associated event can be seen as a breakpoint or termination condition for a long simulation run.

Interactive simulation is done by advancing the system step by step. The state of the system is displayed after each step. Modifying the system state and/or structure is done between steps. It is not necessary to re-run a complete simulation after making a modification to the system.

Free running simulation is possible by specifying the number of steps or time to simulate. Setting this value to infinity allows the simulation to run until a termination condition or error is encountered.

7.2.1 Ergonomics and Ease-Of-Use

The systems which can be designed with this toolbox are very complex. Several measures must be taken to keep the system manageable for a designer. The learning curve for the description methods and toolbox use must be as low as possible. The following sections sum up some of the measures taken to make this possible.

7.2.1.1 Multi-windowing and 'viewers'

The system is described by a tree-like hierarchy of Entity Communication Diagrams and schematics. The amount of information on the screen must be kept under control. Edit windows need only be attached to those design elements which are of interest to the designer. Other elements may have no window attached or have their window 'collapsed' so that it takes up a minimum amount of screen space.

The schematic windows are used to define the system structure. They are also the basic user interface for obtaining state information during simulation. The designer can simply point the cursor at a schematic element and request status information. Status information is displayed continuously by attaching small *'viewer'* windows to the schematic elements. Viewers form the simulated equivalent of a probe.

The few characters of information offered by a viewer are not enough to display the contents of a complex Behaviour Level data structure or basic building block memory. Special windows can be opened which allow the inspection of these information sources. These windows act bidirectionally. Changes made during simulation are reflected immediately in the window. Manual editing of the window contents alters the stored information.

Those elements which have to be defined textually can be edited with a private edit window. A built-in text editor allows the inspection and editing of these definitions. Two methods are used to decrease the number of needed text windows:

- All texts belonging to a single basic entity must be edited in one window.
- A *'universal compiler'* window can be used to edit and compile all textual descriptions which reside in the system.

7.2.1.2 System aspects

The system aspects as defined in section 4.4 are an integral part of the Behaviour Level design tools. During editing and simulation, the designer may select a set of 'current' aspects. The amount of visible design information is reduced by displaying only those parts of the system which are needed for the selected aspect(s). Generating messages which do not belong to one of the selected aspects aborts simulation.

7.2.1.3 Menus and help

All tool operations are menu and 'fill-in-the-blanks' form driven. The menus are design state sensitive. They display currently impossible selections in a different color on the screen.

The toolbox will be fitted with an extensive context-sensitive help system. This system provides help for anything which can be displayed on the screen. A hypertext-like text linking system provides further information. The help system contains all syntax and semantics rules of the languages used in the toolbox.

7.2.1.4 Language consistency

The toolbox uses several languages for the different description levels. Each of these languages is targeted towards it's application. The needed constructs are provided in a concise syntax. These syntaxes try to borrow as much as possible from eachother. This has two advantages:

- *Conversion between abstraction levels is simplified.*
- *Users of the system have less syntax to learn.*

The main expression syntax is taken from Smalltalk. Expressions used in Algorithmic Level entities and basic building blocks operate on the same data type and use the same operators. Controlling and testing basic building blocks is done in a syntactically consistent way. The commands generated by a state machine are also used by control connectors, Algorithmic Level routines and message interface entities.

7.2.2 Documenting the designs

Working with the toolbox, a designer can obtain all information by looking at the screen and clicking some mouse buttons. An example of this tool behaviour is given in the bottom line of the left window in figure 7.2.2-1. Complete and up-to-date documentation in a readable format is needed when the toolbox is not available.

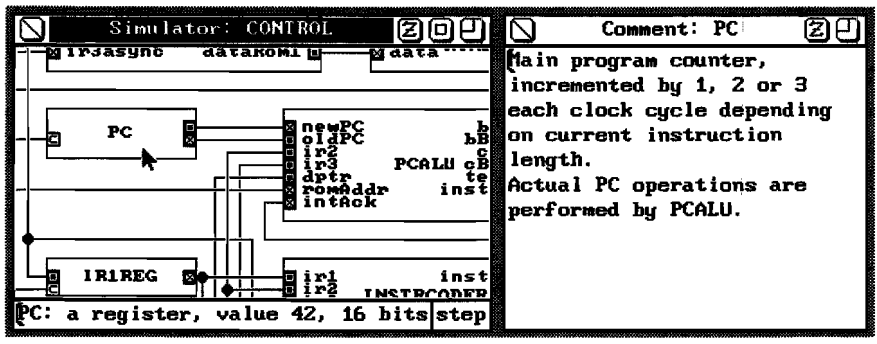


Figure 7.2.2-1: The 'comment editor' window

All structural and most of the functional design information is present in the design itself. A *'comment editor'* window allows adding extra functional information. These comments become an integral part of the design. They complete the functional design information. The right window in figure 7.2.2-1 shows the comment attached to the 'PC' register shown in the schematic window on the left.

Documentation can be generated automatically from the information present in the design environment. The documentation is a plain English text which follows the design hierarchy as closely as possible. All textual definitions and comments are included. This textual information can be enhanced by adding printouts of the Entity Communication Diagrams and schematics. Figure 7.2.2-2 shows the documentation as generated for the 'PC' register shown in figure 7.2.2-1. Note the inclusion of the text in the comment window under the heading 'Designer comments:'.

7.3 Working on large projects

Large projects cannot be done by a single designer. The design must be split into more or less separate parts with a defined 'interface'. This must be done as early as possible in the design path. Each of the parts is given to a designer for detailed design. Intermediate results of this detailed design must be combined at crucial points in the design path. This allows checking their interoperability and adherence to the original specifications.

'MCS8052\MCS8052CORE\CONTROL\PC' is a register.

This register is 16 bits wide and is controlled by an unnamed control input.
The default function is 'load'.
This register is loaded with value 65535 (FFFFh) following system reset.

The value loaded for the 'reset' command is 0.

Designer comments:

Main program counter,
incremented by 1, 2 or 3
each clock cycle depending
on current instruction
length.
Actual PC operations are
performed by PCALU.

This register has the following connectors:

Control connector without a name:
Has a width of 11 bits and is connected to bus 'instrCode'.

Control specification:

"Hold PC when executing non-first
cycles (INTCALL is regarded a
single, first cycle):"

%xx1xxxx1xxx,
%xx1xxx1xx0x,
%xx1xxx1x1xx Hold

Input connector without a name:
Has a width of 16 bits and is connected to bus 'newPC'.

Continuous output connector without a name:
Has a width of 16 bits and is connected to bus 'oldPC'.

Figure 7.2.2-2: Example of automatically generated documentation

The design path described in this Ph.D thesis provides the necessary framework to work with groups of designers:

High-level system behaviour analysis

The first split is made when the message interface and global functionality have been defined for the major Problem Domain Entities. The first task for the system analysts is to generate a minimum working model for these Problem Domain Entities. The working models are combined into a complete system to check if they work together. The complete system is distributed as test environment for the detailed Problem Domain Entity analysis. This is a recursive process which ends when the system description contains enough detail and is approved by the customer.

High-level system architecture synthesis

High-level system architecture synthesis starts out like system behaviour analysis. The system behaviour description is split into a very crude major system architecture. Each of the architecture components is given to an architecture designer. Architecture design is performed by moving functionality between design elements. The architects should keep in touch so that they can offer and request processing facilities. Deciding on a communication channel implementation should be done together by architects working on Abstract Processing Entities which connect to this channel.

Low-level module architecture synthesis and implementation

Implementation starts when implementation strategy choices have been fixed for all Abstract Processing Entities and communication channels. An Abstract Processing Entity can be given to different kinds of designers:

- *Programmers* are needed for the software parts of any implementation strategy.
- *ASIC designers* are needed for the custom hardware parts.
- *Board-level hardware designers* are needed to interconnect standard hardware and ASIC's.

Each of the designers of a (sub) component is responsible for delivering it *'operational as specified'*. This allows integration tests to focus upon overall system behaviour and interfaces.

Tool support for group designs

The basic functions to support designing with a group have already been described:

- *Entities can be filed out for distribution.* This includes complete hierarchies of design entities.
- *Entities can be replaced by others.*
- *Documentation forms an integral part of a design.* This eases the transfer of design elements between designers.

The group design process can be streamlined by adding the following functions to the design environment:

- *Version management.* This allows designers to track the changes made in the design entities and restore an entity to it's former state.
- *Automatic notification of major updates made by other designers.* The actual incorporation of these changes should remain under local control.
- *An automated 'marketplace' for exchanging functions and deciding on implementation choices.*

8. Conclusion

This Ph.D thesis is a snapshot of the status of a very complex project. Some historical background is needed to judge the results up to this point. The next section provides this background. Section 8.2 summarises the results which have been reached so far. Section 8.3 outlines the work which still has to be done.

8.1 History of the project

The original goal of this Ph.D work was to build a multitasking operating system in an ASIC. It was intended as follow-up of the work described in **[ver87]** and **[vos87]**. No tools were available which were capable enough to allow analysing the problem statement. A survey of tools revealed two different tool 'worlds':

- *The world of high-level software design tools.*
- *The world of low-level hardware design tools.*

Tools which spanned the complete system design path were not found. The focus of the Ph.D work shifted towards the definition of a complete system design path. Tools should be provided to support this design path.

A several year's old interest in Object-Oriented techniques provided the first ideas. A system built with these techniques solves problems by simulating some kind of 'reality'. The objects themselves mimic the behaviour of 'things' which are found in the problem specification.

Directly simulating the problem specification is not enough for hardware systems. These need a rigid architecture in which the operations are spatially and timely separated. An architecture designer needs to build a system containing communicating processes. Again, objects can be used to simulate these.

The original goal of the Ph.D work was the generation of an Application Specific Integrated Circuit. This also became the major goal of the design path. Designing at gate level was thought to be too cumbersome. Silicon compilers were capable of handling higher level elements like registers, memories, ALU's and state machines. These became the lowest level building blocks of the design path.

Architecture design can be seen as a transformational process in which system functions are continuously relocated and refined. A high-level abstract object model was needed to do problem analysis and the first steps of architecture design. This model evolved in the basic model described in chapters 2 and 3.

The gap between abstract objects and basic building blocks was too large. It resembled the gap between the two tool-'worlds' described above. The Algorithm Level was introduced to bridge this gap with an extra translation and optimisation step.

The highly interactive Smalltalk environment ([gol84]) provided the ideas which shaped the tools. A first version of the basic building blocks design and simulation tool was built within a few months. Tool construction started at that level because the models needed there were very well understood. At that time, the behaviour level model was not refined enough to build a tool around it.

The work diverted into several directions at the same time:

- The tool prototype went through several revisions. New features were added which could also be used for other description levels.
- An Algorithmic Level language was defined. A prototype tool for it was built ([hul90]).
- The behaviour level model was refined. Prototype tool construction has been started.

8.2 Results

Nearly five years of Ph.D work were not enough to reach all project goals. This section describes what has been completed (as of March 22, 1992).

- A complete design path for complex data processing systems has been specified.
- The system description models needed along the design path have been specified.
- A basic building block level design and simulation tool has been built and extensively tested ([ver90a], [ver90c]). A prototype Algorithmic Level tool has been built ([hul88]). A 'mock-up' version of the Behaviour Level design tool is operational.
- A number of designs have been completed with the basic building block tool (see appendix 2). The Object-Oriented Analysis method described in chapter 4 has been applied to one complex system ([hu91] and appendix 1).
- A prototype automatic conversion tool to convert our basic building block designs into the Hardware Description Language ELLA ([pra86]) has been completed and tested ([baa91]). A more capable converter for the SID language ([sag90]) is available ([zan91]). Several ASIC layouts have been generated with

this converter (see appendix 2). Preliminary work for a VHDL ([iee88]) conversion tool has been done ([sim90]).

8.3 Future work

As stated in the previous section, work is not complete. There are several things which still have to be done:

- The basic building block design and simulation tool must be completed.
- The Algorithmic Level entity must be completed and incorporated into the basic building blocks tool.
- The Behaviour Level 'mock-up' tool must be converted into a true prototype.
- The high-level system architecture synthesis operations must be defined and tools must be generated to support them.
- Library management for designs at different levels must be introduced in the toolbox. The design process can be supported by knowledge bases and expert systems like the ones described in [rov90].
- The basic building blocks to Hardware Description Language conversion tools must be completed. The subset of the basic building blocks they currently support should be extended to cover most designs.

A1. Analysis example: X-25 protocol

The high-level system behaviour analysis method outlined in chapter 4 has been applied to a few systems.

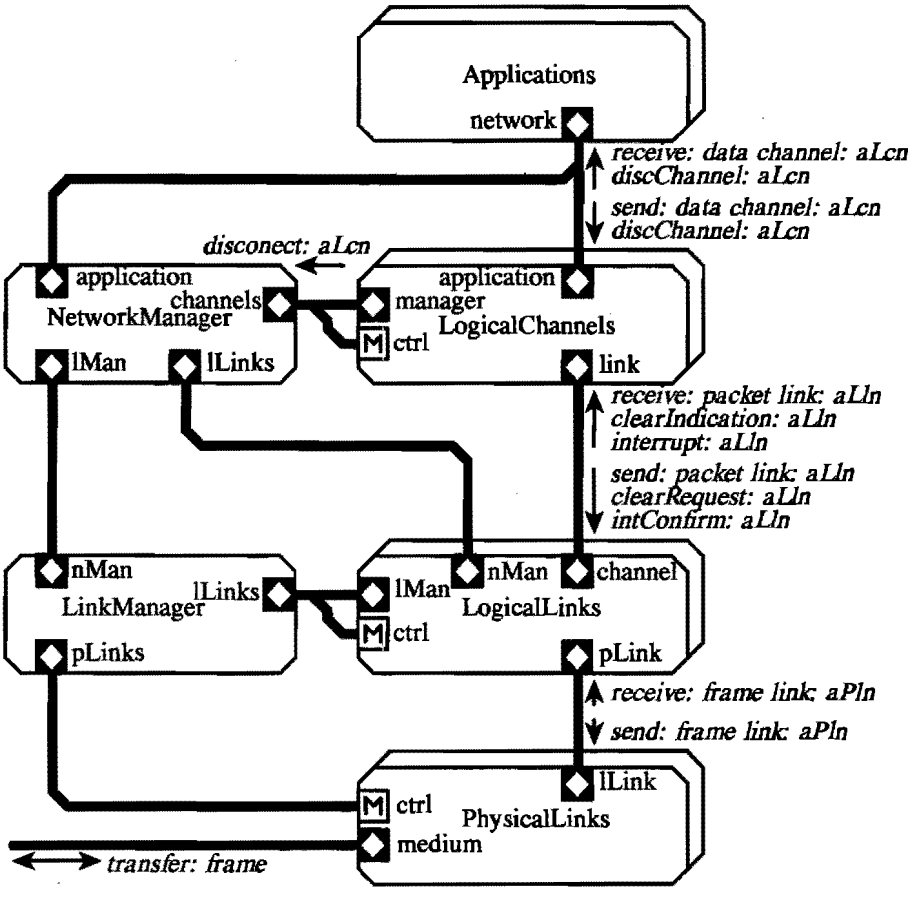


Figure A1-1: X-25 protocol analysis example

Y.C. Hu ([**hu91**]) gives a first-level decomposition of the ISDN X-25 protocol. The basic entities which are stated in the problem domain are directly mapped onto Problem Domain Entities (depicted in figure A1-1):

- *Applications* model the 'users' of the network. An application can request a logical channel for communication with another application somewhere on the

network. Once this channel is established, data can be sent and received across it. A channel can be disconnected when it will not be used anymore.

- *Logical channels* form the endpoints of communication for the network of communication links. A logical channel may route the data traffic generated and received by a single application across several parallel links. The channel must keep the order of the packets sent and received across different links intact.

- *Logical links* provide error recovery and frame contents formatting procedures.

- *Physical links* place frames which must be sent on the communication medium and check received frames for errors. Frames received with errors in them are simply discarded.

- *Network and link managers* are used to control the set of channels and links respectively. These have common behaviour which they inherit from a common superclass called 'Manager'.

Packets and frames are modelled as travelling Problem Domain Entities. The services described in the standard are mapped onto messages. Figure A1-1 indicates some of these messages. Sending is done with the *send:...* messages which go in the direction of the 'PhysicalLinks' multiple. Receiving is done with the *receive:...* messages which move in the direction of the 'Applications' multiple. The figure shows most of the messages related to the 'data transfer' and 'disconnect channel' aspects of system operation.

The variables stored within the behaviour objects of the different entities can be directly derived from the state variables used in the protocol description (for instance the receive and transmit window counters in the logical links). Coding the actual protocol behaviour in Smalltalk-like methods is a relatively straightforward task. The example below is taken from the 'LogicalChannels' dynamic multiple:

```
send: data channel: aLcn

"Handle the data which is sent by an application. The message filter has
 made sure that the logical channel number (lcn) matches aLcn."

link                    "Create and send packet to logical link"
  send: (Packet for: lcn sendSeq: pS recvSeq: pR data: data)
  link: linkNr.
pS := pS + 1            "Increment send sequence number"
```

A2. Some Low-Level Designs

The basic building block level design tool *'IDaSS for ULSI'* (Interactive Design and Simulation System for Ultra Large Scale Integration, **[ver90c]**) was originally conceived to test the ideas behind interactive designing. The original goal was to build a prototype tool and evaluate this with a few designs.

The IDaSS for ULSI tool generated a lot of interest. Work started March 1988. By August 1990, a paper was published (**[ver90a]**) and a conference tutorial was given (**[ver90b]**). The tool is being evaluated by several companies (Philips P-ASIC, IBM Zurich and others). It has been used for numerous designs by students within our group. Short descriptions of some of these designs will be given below. The figure at the end of each header estimates the amount of working days needed to complete the design. Note that these students never worked with IDaSS before. Most of them had no prior experience with digital system design.

Instruction cache for a RISC processor [hu89] (100 days)

The first complex design done with IDaSS for ULSI was a high performance instruction cache based upon work presented in **[bor90]**. This is a very complex two-way set-associative cache, capable of delivering two words per clock cycle to the attached processor core.

Intel 8048 microcomputer core [mae90] (25 days)

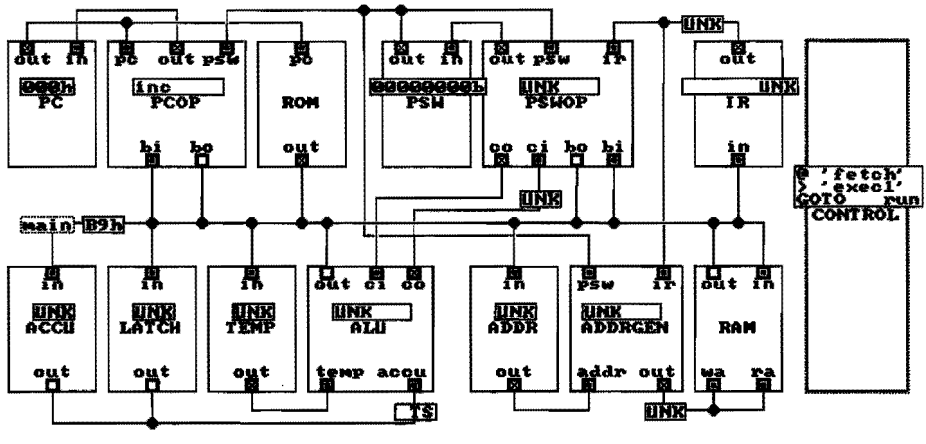


Figure A2-1: Schematic of an 8048 compatible microprocessor core

This is a re-design of the Intel 8048 microcomputer core. The design contains an 8-bit processing core with 64 bytes RAM and 1024 bytes program memory. It closely follows the original single bus architecture as defined by Intel. Instruction execution takes one to five clock cycles. The complexity of this design is very modest, a single schematic containing 14 basic building blocks (shown in figure A2-1). The main elements of the 8048 processor are immediately visible in this design, like the program counter ('PC'), ROM, RAM, accumulator ('ACCU') and program status word ('PSW').

This design has been converted into an ASIC layout using the ASA silicon compiler ([zan91], [sag90]). The result took 21 mm^2 and ran at 15 Mhz clock frequency. The result of this conversion is shown in figure A2-2. Memories are implemented by parametrisable macrocell generators (ROM at the bottom and RAM at the far right side). Standard cell technology is used to implement the other design elements (like the CONTROL state machine above the ROM and the ALU between ROM and RAM). This layout was generated without any designer intervention.

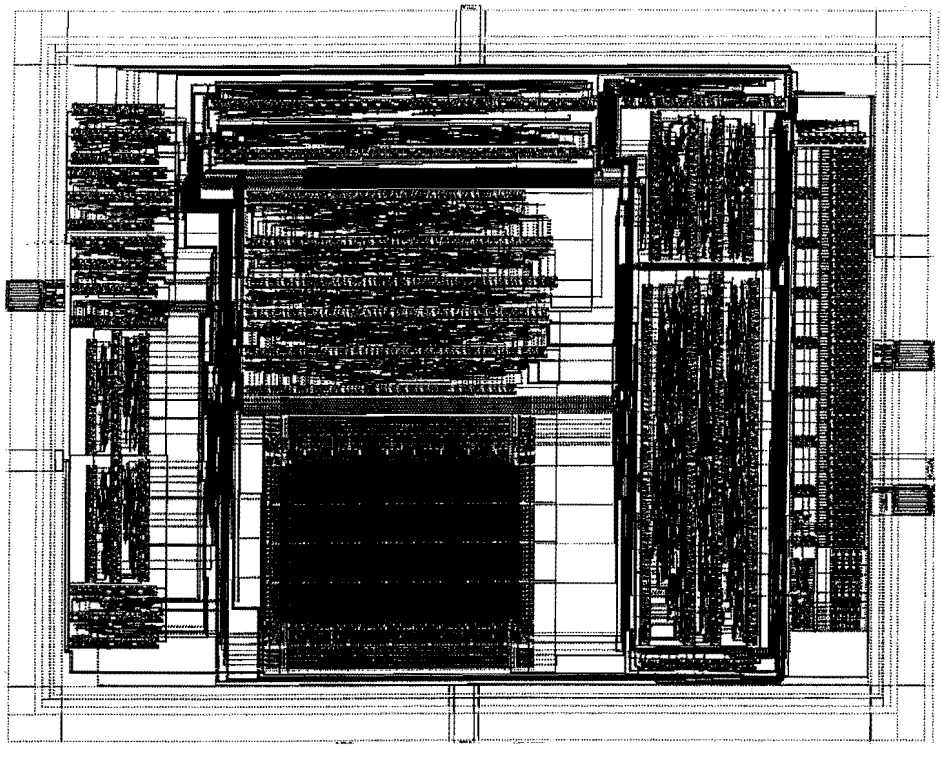


Figure A2-2: Layout of the 8048 compatible microprocessor core

Pipelined Intel 8052 microcomputer [lec90] [bek91] (100 days)

This design is based on the pipelined version of the Intel 8052 microcomputer's processing core designed by W. Lecluse. The architecture differs radically from the original Intel design and manages to execute most instructions in a single clock cycle. R. den Bekker added the input and output components which are standard on the 8052 microcomputer. Work on an instruction cache is almost finished.

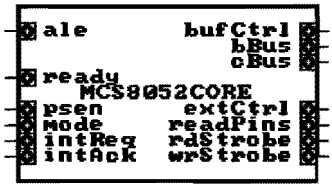


Figure A2-3: Processor core symbol of 8052 microcomputer

This is a complex design with several levels of nested schematics and more than eighty basic building blocks. Figure A2-3 shows the symbol of the actual processing core of this microcontroller. This core communicates with external interface registers using the 'bufCtrl', 'bBus' and 'cBus' connectors. 'bufCtrl' addresses the interface registers and specifies the operation to be performed. 'bBus' is used to read the addressed interface register, while 'cBus' can be used to write modified contents back into the same register (in the same clock cycle).

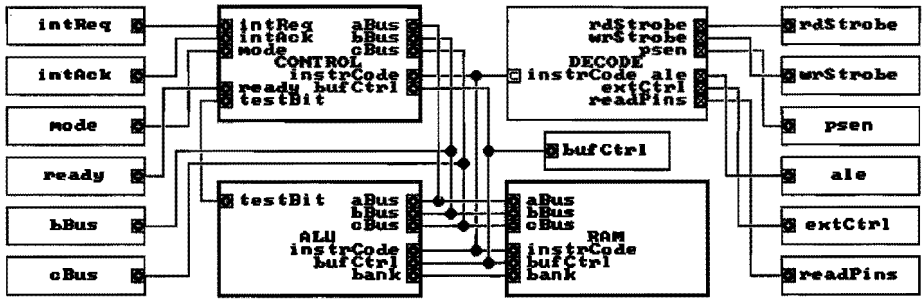


Figure A2-4: Processor core architecture of 8052 microcomputer

Control is completely distributed by control buses and control connectors. The 'instrCode' bus present in figure A2-4 forms the main control mechanism. The 'CONTROL' schematic contains all elements of the design which are needed to generate the actual control signals on this bus. Three data buses ('aBus', 'bBus'

and 'cBus') are used to transfer data between the system parts in each clock cycle. 'aBus' is generated by 'CONTROL' and carries immediate data constants.

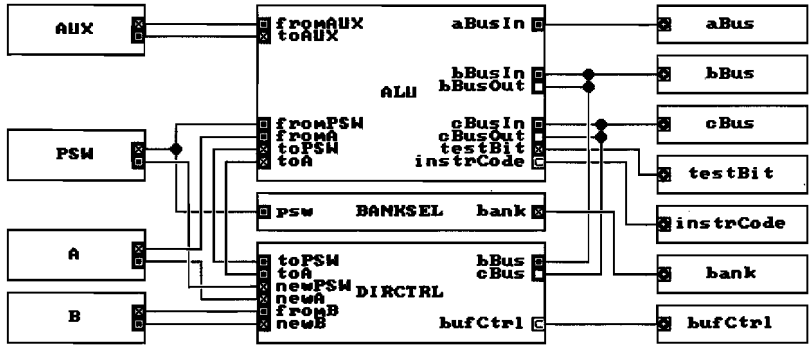


Figure A2-5: 'ALU' schematic of 8052 microcomputer core

Figure A2-5 shows the contents of the 'ALU' schematic in figure A2-4. This schematic contains the actual Arithmetic Logic Unit (the 'ALU' operator), which performs 51 different functions (all byte and bit manipulations). Selecting between these operations is done by the 'instrCode' control connector. The Program Status Word ('PSW'), accumulator ('A') and 'B' registers are also located on this schematic. Reading and writing these registers as interface registers is controlled by the 'DIRCTRL' operator (in turn controlled by the 'bufCtrl' control connector). 'AUX' is a temporary data storage register used during multi-cycle instructions (like multiply and divide). 'BANKSEL' is a very simple operator which extracts two 'bank select' bits from the program status word for use by the 'RAM' schematic in figure A2-4.

Floating point core [bal91] [hot91] [kor91] (75 days)

This design represents an IEEE standard compatible floating point core. Multiplication, addition and subtraction are implemented fully combinatorial. Eighty bit precision division takes up to seven clock cycles using the 'Newton-Raphson' algorithm. Sine and cosine are calculated in approximately 200 clock cycles using the 'cordic' algorithm. The architecture of the design is not very optimised because the emphasis of this project was on algorithms and precision.

Based on this design, a floating point adder and multiplier have been described in a single basic building block operator.

A 'Grammar Processor' [jac91] [lun91] (50 days)

The Ph.D work of R.J.H. Bloks is centered around a so-called 'Grammar Processor'. This is a processor which is intended for the execution of communication protocols. The protocols are described in the form of an extended grammar.

The hardware for the processor contains several cooperating processors. A central processor ([jac91]) stores and executes the grammar 'rule base'. Tests needed to decide which grammar rule to execute are evaluated by one or more attribute evaluation processors ([lun91]). The latter design uses a five-stage pipeline which is invisible to the grammar compiler.

A complete protocol engine requires a grammar processor, packet storage memory and a low-level hardware interface. The Token Ring ([iee85]) controller core described in [lee90] can be used as basis for such a low level engine.

Neural network [verz91] (25 days)

The first test in designing massively parallel architectures was a simple digital neural network. The basic neuron is shown in figure A2-6. A network of these neurons (as shown in figure A2-7) can be taught to discern between several patterns of zeroes and ones.

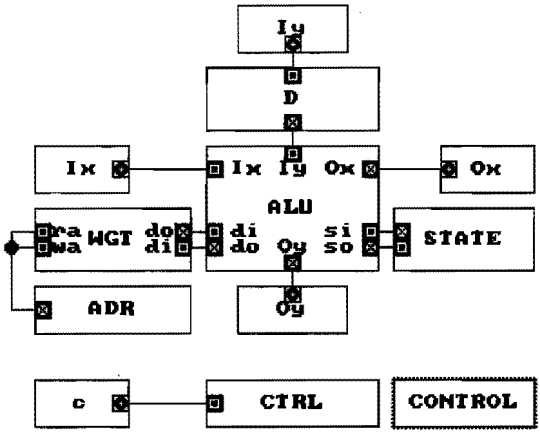


Figure A2-6: A single neuron implemented in basic building blocks

Patterns which must be recognised are presented in parallel via the 'Ix' inputs. During recognition, each neuron calculates it's own excitation value in the

'STATE' register. Excitation values of other neurons are transferred sequentially via the 'Iy' and 'Oy' inputs and outputs (these are buffered in the 'D' registers). The 'WGT' RAM contains the weights of the connections to each of the other neurons, and is addressed sequentially via the 'ADR' counter register while excitation values pass through the neuron. The 'WGT' RAM is updated automatically during the learning of the patterns which must be recognised. The resulting patterns are output in parallel via the 'Ox' outputs. The operation of the neurons is controlled by placing values in the 'CTRL' register of each neuron. This value is decoded by the 'CONTROL' state machine.

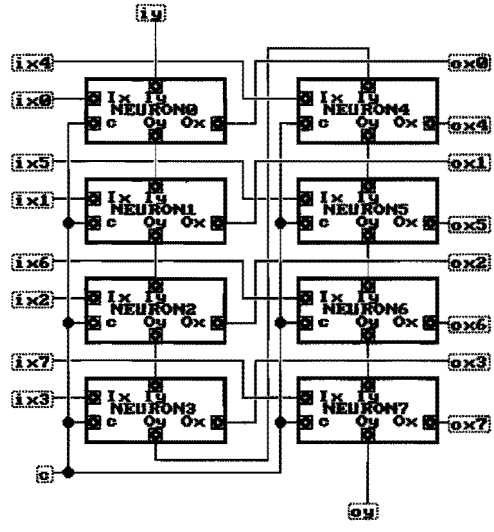


Figure A2-7: A simple neural network

PCM switching network [laa91] (100 days)

Two different architectures for a digital PCM switching network were designed and evaluated in approximately six months. Four of these networks in parallel form the switching network of a non-blocking fault-tolerant telephone exchange which supports 40.000 subscribers. One of the architectures has been converted into an ASIC layout with ASA (scaled down to 10.000 subscribers). The result was a chip measuring 51 mm^2 (figure A2-9). This chip contains six identical 512 words by 4 bits RAM's, a 512 words by 36 bits RAM and control logic implemented in standard cells. Four of the smaller memories contain the data for the 'Time' switches (located in the 'T_SWITCH' on figure A2-8). The remaining small memories control the two 'Space' switches ('S_SWITCH1' and

'S_SWITCH2' on figure A2-8). The large memory controls the 'Time' switches.

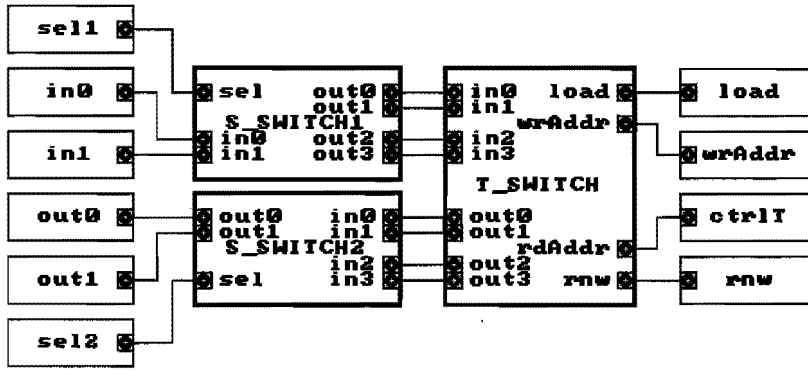


Figure A2-8: PCM switching network Space-Time-Space switches core

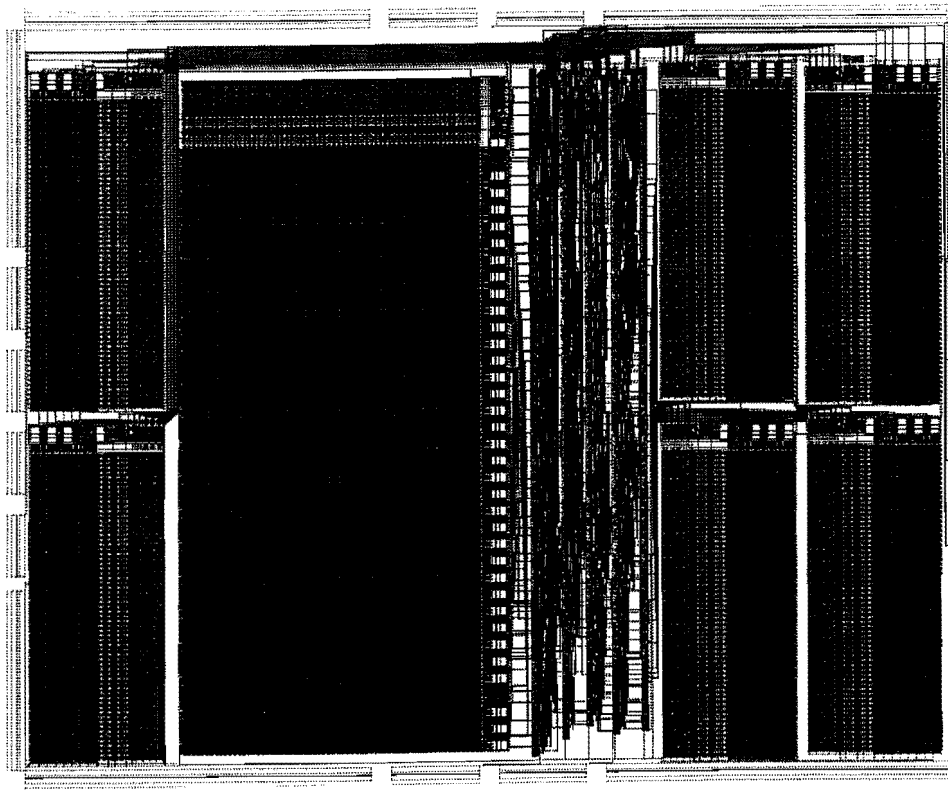


Figure A2-9: PCM switching network ASIC layout

A3. Terminology

The major terms used in this Ph.D thesis are listed in alphabetical order in this appendix. The number between braces is the page number where the term is described in the text.

A

Abstract Communication Channel: an abstract model of a *communication channel* which will be present in the final system (89)

Abstract Processing Entity: an abstract model of a *processing entity* which will be present in the final system (89)

Algorithmic Level entity: a design entity which describes a piece of hardware in an algorithmic (Pascal-like) language (123)

Architecture editor: a tool which can combine and split *processing entities* and *communication channels* (90)

B

Basic building block: a low level design entity which can be handled by a silicon compiler (130)

Basic model entity: one of the entities which can be used in the basic model. This includes *basic model objects, groups, (dynamic) multiples, (dynamic) multiple groups* and *Low-Level Simulation Entities.*

Basic model object: an abstract entity which describes data storage, data operations and communication within the basic model (17)

Behaviour object: an object capable of data storage and operations, stored in a *behaviour slot* within the *processing core* of a *basic model object* (46)

Behaviour slot: an element of the *processing core* within a *basic model object* which can hold a single *behaviour object* (44)

C

Communication channel: the means to transfer *messages* between *basic model objects,* can perform simple routing functions (23)

Continuous data: an alternative means of communication between *basic model objects,* has it's own channels and connectors (61)

Control connector: a special connector placed within a *basic building block*, which allows controlling this block with a value present on a data bus (132)

D

Dynamic multiple: a group of *basic model objects* with an identical description but different state, where the amount of objects varies during the operation of the system (57)

Dynamic multiple group: a group of identical *Entity Communication Diagrams*, where diagrams can be added and removed during the operation of the system (59)

Dynamic multiple inheritance: the way in which a *basic model object* inherits behaviour from the *behaviour objects* stored in it's *processing core* (22)

E

Entity Communication Diagram: a diagram which graphically describes a system with symbols for all *basic model entities, communication channels* and *continuous data* channels (55)

G

Group: an object in the basic model which can be used to group other *basic model entities* together on an *Entity Communication Diagram* (54)

I

Input buffer: a storage place for *messages* which have been received but cannot be handled yet by the *processing core* of a *basic model object* (34)

Input filter: an entity in the *interface shell* of a *basic model object* which decides what to do with the *messages* which are present on the *communication channel* it is connected to (31)

Interface shell: that part of a *basic model object* which contains the *input filters, input buffers, message selector/manipulator, virtual connector translation table* and *output buffers* (30)

Interface Entity: an entity which forms the interface between the abstract basic model communication methods (*messages* and *continuous data*) and the hardware oriented *interface primitives* within a *Low-Level Simulation Entity* (117)

Interface Primitive: a *basic building block* which is used to describe the interface hardware between a *Low-Level Simulation Entity* and other *basic model entities* (118)

L

Low-Level Simulation Entity: an entity which replaces a *basic model object* which has to be converted into hardware (116)

M

Management connector: a special connector on a *dynamic multiple* or *dynamic multiple group* which is used to control the creation and removal of entities in the multiple (58)

Message: the main means of communication between *basic model objects*, sent across *communication channels*. Contains a *message selector* and an optional set of parameters (25)

Message selector: the fixed part of a *message* which may be used to differentiate between different messages (25)

Message selector/manipulator: an element of the *interface shell* within a *basic model object* which selects the *messages* to be handled by the *processing core* (36)

Method: a textual description of the handling of a *message* within a *behaviour object* (46)

Multiple: a group of *basic model objects* with an identical description but different state, where the amount of objects is fixed during the operation of the system (56)

Multiple group: a group of identical *Entity Communication Diagrams*, where the amount of diagrams is fixed during the operation of the system(59)

O

Operator: a *basic building block* which describes combinatorial operations as a set of designer-defined functions (131)

Output buffer: an element of the *interface shell* within a *basic model object* which can be used to store *messages* which cannot yet be transferred by a *communication channel* (42)

P

Problem domain: the system to be designed and the environment with which the system communicates (66)

Problem Domain Entity: any entity which forms part of the problem domain for any 'reasonable' amount of time (66)

Processing core: that part of a *basic model object* which contains the *behaviour slots* and *behaviour entities* and defines the actual behaviour (44)

Processing entity: the generic name for an *Abstract Processing Entity* which has been assigned an implementation strategy during high-level system architecture synthesis (91)

Processing Model: the system model which describes the actual operations as they will be performed in the final system (89)

Profile, communication channel: an abstract description of the capabilities or requirements of a *communication channel* (102)

Profile, processing entity: an abstract description of the processing, data storage and interface capabilities or requirements of a *processing entity* (103)

S

Schematic: a diagram which graphically describes a *Low-Level Simulation Entity's* contents with *interface entities, interface primitives, Algorithmic Level entities* and *basic building blocks* (116)

Signal: a single bit semaphore-like *interface primitive* which is used for synchronisation purposes within a *Low-Level Simulation Entity* (120)

Super connector: a graphical element used to denote that a *communication channel, continuous data* channel or data bus is continued across an *Entity Communication Diagram* or *schematic* boundary (55)

T

Tag: an object which is used to address specific entities within a *(dynamic) multiple* or *(dynamic) multiple group* (70)

Three-state: the condition in which no values are placed on a *continuous data* channel (61) or data bus (130)

Travelling object: object which does not have a fixed location within a system, moves between *basic model objects* as *message* parameter (62)

V

Virtual connector translation table: a table which translates virtual names used by behaviour object *methods* into real connector or *behaviour slot* names (40)

References

The references are first ordered according to year, then to the surname of the principal (or first-mentioned) author.

[bir73] Birtwistle, G.; Dahl, O.-J.; Myhrhaug, B. and Nygaard, K.: *Simula Begin*, Auerbach, Philadelphia, 1973

[che76] Chen, P.P.: *The Entity-Relationship Model - Toward a Unified View of Data*, in: ACM Transactions on Database Systems, Vol 1, No. 1, 1976, PP. 9..36

[fra77] Franta, W.R.: *The Process View of Simulation*, Elsevier North-Holland, Inc., New York, 1977, ISBN 0-444-00221-9

[dem78] DeMarco, T.: *Structured Analysis and System Specification*, Yourdon Press, New York, 1978

[hoa78] Hoare, C.A.R.: *Communicating Sequential Processes*, Comm. ACM, Vol 21, No 8, PP. 666..677, 1978

[mil80] Milner, R.: *A Calculus of Communicating Systems*, Lecture Notes in Computer Science, Vol 92, Springer, Berlin, 1980

[pet81] Peterson, J.L.: *Petri-Net theory and the modelling of systems*, Prentice-Hall, 1981

[gol84] Goldberg, A.: *Smalltalk-80: The Interactive Programming Environment*, Addison-Wesley, 1984

[int84] Intel Corporation: *iRMX™86 Programmer's Reference Manual, Part I (For Release 6)*, Intel Corporation, 3065 Bowers Avenue, Santa Clara, CA 95051, 1984

[iee85] IEEE: *Standards for Local Area Networks: Token Ring Access Method and Physical Layer Specifications*, ANSI/IEEE Std 802.5-1985, March 1988

[man86] Man, H. de; Rabaey, J.; Six, P. and Claesen, L.: *Cathedral-II: A Silicon Compiler for Digital Signal Processing*, IEEE Design and Test Magazine, December 1986, PP. 13..25

[pra86] Praxis Systems plc: *The ELLA User Manual*, Issue 3.0, 1986, Praxis Systems plc, 20 Manvers St, Bath, Avon, BA1 1PX, England

[cci87] CCITT: *Specification and Description Language (SDL)*, CCITT/Recommendation Z.100, January 1987

[hat87] Hatley, D.J. and Pirbhai, I.A.: *Strategies for Real-Time System Specification*, Dorset House Publishing, New York, NY 10014, 1987, ISBN 0-932633-04-8

[str87] Stroustrup, B.: *The C++ Programming Language*, Addison-Wesley, Amsterdam, 1987, ISBN 0-201-12078-X

[ver87] Verschueren, A.C.: *A Coprocessor for Hardware Multitasking Support*, Master's thesis, Eindhoven University of Technology, 1987

[vos87] Vos, H.J.M.: *Interprocess Communication in a Multiprocessor Environment; Specification of a Communication Controller*, Master's thesis, Eindhoven University of Technology, 1987

[dig88] Digitalk Inc.: *Smalltalk/V 286 Tutorial and Programming Handbook*, May 1988, Digitalk Inc., 9841 Airport Boulevard, Los Angeles, CA 90045

[hui88] Huis in 't Veld, R.J.: *A Formalism to Describe Concurrent Non-Deterministic Systems and an Application of it by Analysing Systems for Danger of Deadlock*, Research report 88-E-200, Eindhoven University of Technology, August 1988, ISBN 90-6144-200-1

[hul88] Hulzebos, R.M.: *Description of Parallel Processes in Hardware Using SDL*, Master's thesis, Eindhoven University of Technology, 1988

[iee88] IEEE: *IEEE Standard VHDL Language Reference Manual*, Std 1076-1987, March 1988

[mey88] Meyer, B.: *Object Oriented Software Construction*, Prentice Hall, Hertfordshire, 1988, ISBN 0-13-629049-3

[shl88] Shlear, S. and Mellor, S.J.: *Object-Oriented System Analysis: Modelling the Real World in Data*, Prentice Hall, Englewood Cliffs, NJ, 1988 ISBN 0-13-629023-X

[you88] Young, J.L.: *QTC makes it easy to design Custom Processors*, Electronics, January 21, 1988, PP. 49..51

[bai89] Bailin, S.C.: *An Object Oriented Requirements Specification Method*, Communications of the ACM, Vol 32, No 5, May 1989, PP. 608..623

[gol89] Goldberg, A. and Robson, D.: *Smalltalk-80: The Language*, Addison Wesley, 1989, ISBN 0-201-13688-0

[hu89] Hu, Y.C.: *An Instruction Cache in IDaSS*, Master's thesis, Eindhoven University of Technology, 1989

[lan89] Lange, A.A.J. de; et al: HiFi: *An Object Oriented System for the Structural Synthesis of Signal Processing Algorithms and the VLSI Compilation of Signal Flow Graphs*, in: Proceedings of the IMEC-IFIP International Workshop on Applied Formal Methods for Correct VLSI Design, Elsevier Science Publishers, Amsterdam, Vol 2, 1989, PP. 462..481

[shl89] Shlaer, S. and Mellor, S.J.: *An Object Oriented Approach to Domain Analysis*, ACM Sigsoft, Vol 14, No 5, July 1989, PP. 66..77

[bor90] Bormans, J.E.H.M. et al: *Designing High Performance Instruction Caches in VLSI*, in: Microprocessing and Microprogramming, Volume 30, Numbers 1-5, August 1990, PP. 135..142

[coa90] Coad, P. and Yourdon, E.: *Object-Oriented Analysis*, Prentice-Hall, Englewood Cliffs, NJ 07632, 1990, ISBN 0-13-629122-8

[cox90] Cox, B.J.: *Planning the Software Industrial Revolution*, draft for publication in: IEEE Software Magazine, November 1990

[hul90] Hulzebos, R.M.: *A Hardware Oriented Description and Simulation System*, IVO report, Eindhoven University of Technology, 1990

[lee90] Leermakers, H.P.: *Designing a Token Ring Controller with IDaSS*, Master's Thesis, Eindhoven University of Technology, 1990

[lec90] Lecluse, W.: *A Fast 8051 Compatible Microcontroller Processor-Core in IDaSS*, Master's thesis, Eindhoven University of Technology, 1990

[mae90] Maessen, R.: *An IDaSS 8048 Design*, practical working period report, Eindhoven University of Technology, 1990

[rov90] Rovers, W.M.H.M.: *A Theoretical Preliminary Study for a Design Assistant*, IVO report, Eindhoven University of Technology, 1990

[sag90] Sagantec Europe B.V.: *ASA Version 4.6 User Guide*, Sagantec Europe B.V., P.O. Box 2102, 5600 CC Eindhoven, The Netherlands

[sim90] Simons, P.W.: *Compiler from IDaSS to VHDL (Principles of IDaSS to VHDL Conversion)*, practical working period report, Eindhoven University of Technology, 1990

[ver90a] Verschueren, A.C.: An *Object Oriented Design and Simulation System for VLSI*, in: Microprocessing and Microprogramming, Volume 30, Numbers 1-5, August 1990, PP. 241..246

[ver90b] Verschueren, A.C.: *Object Oriented Design of ASIC's*, Tutorial given at the Euromicro 90 conference, Amsterdam, August 27, 1990

[ver90c] Verschueren, A.C.: *IDaSS for ULSI manual*, can be obtained from: Eindhoven University of Technology, Digital Systems group, P.O. Box 513, 5600 MB Eindhoven, Netherlands

[baa91] Baars, E.A.: *IDaSS to ELLA conversion - Automatic conversion of a design from one design-tool into another*, master's thesis, Eindhoven University of Technology, 1991

[bal91] Balen, M. van: *Ontwerp van een floating-point ALU m.b.v. IDaSS (Design of a floating-point ALU using IDaSS)*, practical working period report, Eindhoven University of Technology, 1991

[bek91] Bekker, T.J.A.M. den: *Peripheral Extensions for the Fast 8051 Compatible Microcontroller Processor Core in IDaSS*, practical working period report, Eindhoven University of Technology, 1991

[ben91] Benders, L.P.M. and Stevens, M.P.J.: *Task Level Behavioural Hardware Description*, in: Microprocessing and Microprogramming, Volume 32, Numbers 1-5, August 1991, PP. 323..331

[hot91] Hotho, G.: *Het ontwerp van een floating-point deler en vermenigvuldiger als onderdeel van een floating-point ALU met behulp van het ontwerpprogramma IDaSS (The design of a floating-point multiply/divide unit as sub-component of a floating-point ALU using IDaSS)*, practical working period report (in Dutch), Eindhoven University of Technology, 1991

[hu91] Hu, Y.C.; Verschueren, A.C. and Stevens, M.P.J.: *Object Oriented System Analysis for VLSI*, in: Microprocessing and Microprogramming, Volume 32, Numbers 1-5, August 1991, PP. 101..108

[jac91] Jacobs, M.J.F.: *Ontwerp van een Push Down Control als onderdeel van een Grammatica Processor (Design of a Push Down Controller as sub-component of a Grammar Processor)*, practical working period report (in Dutch), Eindhoven University of Technology, 1991

[kor91] Korsten, E.: *Ontwerp van een floating-point ALU - een sinus- en cosinus generator m.b.v. het cordic algoritme (Design of a floating-point ALU - a sine- and cosine generator using the cordic algorithm)*, practical working period report (in Dutch), Eindhoven University of Technology, 1991

[laa91] Laat, R.F.C. de: *Analysis and design of a digital switching circuit*, Master's thesis, Eindhoven University of Technology, 1991

[lun91] Lunteren, J. van: *Ontwerp van een Attribuut Evaluator als onderdeel van een Grammatica Processor (Design of an Attribute Evaluator as sub-component of a Grammar Processor)*, practical working period report (in Dutch), Eindhoven University of Technology, 1991

[pou91] Pountain, D.: *The Transputer Strikes Back*, in: Byte, Volume 16, Number 8, August 1991, PP. 265..275

[she91] Sheldon, K.M.: *The Intel Micro 2000*, in: Byte, Volume 16, Number 4, April 1991, PP. 132..133

[verz91] Verzijl, P.G.A.: *Neurale Netwerken - Een IDaSS Ontwerp (Neural Networks - An IDaSS Design)*, practical working period report (in Dutch), Eindhoven University of Technology, 1991

[zan91] Zandvoort, R.M.L. van: *Converting IDaSS to ASA*, master's thesis, Eindhoven University of Technology, 1991

[bud92] Budzelaar, F.P.M.: *Internal operation of the APDL compiler*, personal communications

[koo92] Koomen, C.J.: *The Design of Communication Systems*, Kluwer Academic Publishers Group, 1992, ISBN 0-7923-9203-5

The following texts are additional reading material:

Berard, E.: *Understanding the Object-Oriented Life-Cycle*, network message in newsgroup comp.object, 1991

Berard, E.: *A Closer Look at the Recursive/Parallel (O.O.) Life Cycle*, network message in newsgroup comp.object, 1991

Bezooijen, H. van: *The Evaluation of Suitable Object-Oriented Development Techniques in order to make a Prototype Point of Sales Terminal on a Personal Computer*, Master's thesis, Eindhoven University of Technology, 1990

Cole, B.C.: *Object-oriented VHDL package goes graphical*, in: Electronic world news, November 18, 1991, PP. 1&4

Duff, C. and Howard, B.: *Migration Patterns*, in: Byte, Volume 15, Number 10, October 1990, PP. 223..232. This text describes the advantages and problems of introducing Object-Oriented programming methods.

Gibson, E.: *Objects-Born and Bred (Object Behaviour Analysis)*, in: Byte, Volume 15, Number 10, October 1990, PP. 245..254

Hoerbst, E.; Mueller-Schloer, C. and Schwaertzel, H.: *Design of VLSI Circuits based on VENUS*, Springer-Verlag, Berlin, 1987, ISBN 3-540-17663-2

Huis in 't Veld, R.J.: *The Role of Languages in the Design-Trajectory*, in: Microprocessing and Microprogramming, Volume 30, Numbers 1-5, August 1990, PP. 177..184

Ledgard, H.F.: *Ada, An Introduction/Ada Reference Manual (July 1980)*, Springer-verlag, New York-Heidelberg-Berlin, 1981, ISBN 0-287-90568-5

Loy, P.H.: *A Comparison of Object-Oriented and Structured Development Methods*, in: Thayer, R.H. and Dorfman, M. (eds.): System and Software Requirements Engineering, IEEE Computer Society Press, 1990

Meulen, P.S. van der; Ming-Der Huang; Uzi Bar-Gadda; Lee, E. and Baltus, P.: *EXIST: an Interactive VLSI Architectural Environment*, invited paper ICCD, 1988, PP. 312..319

Nestor, J.A; Soudan, B and Mayet, Z: MIES: *A Microarchitecture Design Tool*, 22nd Annual Workshop on Microprogramming and Microarchitecture, August, 1989

Nippon Electronic Company: *Product description uPD7281/uPD9305 dataflow processor*, 1986

Stevens, M.P.J. and Budzelaar, F.P.M.: *System Level VLSI Design*, in: Microprocessing and Microprogramming, Volume 30, Numbers 1-5, August 1990, PP. 321..330

Yourdon, E.: *Modern Structured Analysis*, Prentice Hall, Englewood Cliffs, NJ, 1989, ISBN 0-13-598632-X

Yourdon, E.: *Auld Lang Syne*, in: Byte, Volume 15, Number 10, October 1990, PP. 257..264. This text describes how Object-Oriented programming methods can be introduced in an organisation.

Summary

The increasing complexity of the societies' data processing systems makes it very difficult to design them within the given time and cost constraints.

The Object-Oriented design paradigm is gaining acceptance in the software world. Data processing is performed by a set of 'objects' which communicate by exchanging messages. Each of these objects behave like elements in an (abstract) 'real world'. Designing such a system can be done very quickly by re-using and modifying already existing behaviour.

This Ph.D thesis describes how Object-Oriented design methods can be applied to systems which contain a mix of software and hardware modules. An improved object model is introduced which allows designing systems which contain a high degree of concurrency. The model can be simulated on a computer in an interactive design environment. Timing can be specified and simulated to check system performance. The complete system design path is split in three phases:

1) *High-level system behaviour analysis.* An extended Object-Oriented Analysis method is used to obtain an operational system defined in terms of 'Problem Domain Entities'. This is an architecture-independent description of the system which serves to fix and complete the system specifications.

2) *High-level system architecture synthesis.* A system architecture is synthesized by gradually transforming the Problem Domain Entities into 'Abstract Processing Entities'. Implementation choices are made for these processing entities and the communication channels which connect them.

3) *Low-level module architecture synthesis and implementation.* The Abstract Processing Entities are implemented in a mix of hardware and software. The path towards Application Specific Integrated Circuits is formed by several lower level description tools which co-reside in the design environment.

This Ph.D thesis gives characteristics common to the tools used for this design path. These tools are highly interactive and combine design and simulation. Simulation will not be interrupted while the designer modifies the system structure or entity behaviour. Shortening the design-simulate-debug cycle gives a designer immediate feedback on design actions.

Several design and simulation tools have been implemented. Using these, some ASIC's have been designed (including some processors and the switching network for a telephone exchange which supports 40.000 subscribers).

Samenvatting

De toenemende complexiteit van de data verwerkende systemen gebruikt door de samenleving maakt het zeer moeilijk om deze tegen acceptabele kosten te ontwerpen binnen de gegeven tijd.

De Object-Georienteerde werkwijze wordt in toenemende mate geaccepteerd binnen de 'software' wereld. Hierbij wordt data verwerkt door een verzameling 'objecten' die met elkaar communiceren door het uitwisselen van berichten. Ieder van die objecten bootst het gedrag na van een element uit een (abstracte) realiteit. Zo'n systeem kan zeer snel ontworpen worden door het hergebruiken en veranderen van reeds bestaand gedrag.

Dit proefschrift beschrijft hoe de Object-Georienteerde ontwerp methoden gebruikt kunnen worden voor systemen die een mengeling van 'hardware' en 'software' bevatten. Een verbeterd object model wordt geintroduceerd dat een hoge graad van parallellisme toelaat. Dit model kan op een computer worden gesimuleerd in een interactieve ontwerpomgeving. Tijdsvertragingen kunnen worden gespecificeerd en gesimuleerd om systeem prestaties te kunnen controleren. Het totale systeem ontwerp pad is opgesplitst in drie fasen:

1) *Hoog-niveau systeemgedrag analyse.* Een uitgebreidde Object-Georienteerde analyse methode wordt gebruikt om een operationeel systeem te definieren in termen van 'Probleem Domein Entiteiten'. Dit is een architectuur-onafhankelijke beschrijving van het te ontwerpen systeem die gebruikt wordt om de systeem specificaties vast te leggen en te completeren.

2) *Hoog-niveau systemarchitectuur synthese.* Een systeemarchitectuur wordt gesynthetiseerd door Probleem Domein Entiteiten geleidelijk te transformeren in 'Abstracte Dataverwerkings Entiteiten'. Voor deze entiteiten en de verbindende communicatie kanalen worden implementatie keuzes gemaakt.

3) *Laag-niveau module architectuur synthese en implementatie.* De Abstracte Dataverwerkings Entiteiten worden geimplementeerd in in een mengeling van hardware en software. Het pad naar Applicatie Specifieke Geintegreerde Circuits wordt gevormd door verscheidene laag-niveau ontwerp hulpmiddelen die in de ontwerp omgeving aanwezig zijn.

Dit proefschrift geeft de gemeenschappelijke karakteristieken voor de bij dit ontwerppad gebruikte ontwerphulpmiddelen. Deze zijn sterk interactief en combineren ontwerpen en simuleren. Simulatie wordt niet onderbroken als de ontwerper de systeem structuur of entiteit gedrag verandert. Het korter maken van de ontwerpen-simuleren-foutzoeken cyclus geeft de ontwerper een directe terugkoppeling over de genomen ontwerpacties.

Verschillende van deze tools zijn geimplementeerd. Hiermee zijn enige ASIC's ontworpen (waaronder enkele processoren en het schakelnetwerk van een telefooncentrale voor 40,000 abonnees).

Curriculum Vitae

Adrianus Cornelis Verschueren was born July 26, 1960 in Goirle, the Netherlands.

He attended the Paulus Lyceum in Tilburg from 1972 to 1978. There, he developed an interest in designing and building electronic systems.

Starting September 1978, he studied at Eindhoven University of Technology. He graduated with honors under the supervision of Prof.ir. M.P.J. Stevens in August 1987. Four days later, he started the Ph.D work which resulted in this thesis - in the same group and under the same professor as his graduation. During this Ph.D work, he spent some time in Switzerland at the IBM Zurich research laboratory.

As from January 1992, he has been appointed scientific staff member in the digital system's group at the Eindhoven University of Technology.

Theses/Stellingen:

1) The best way to catch specification errors is by simulating the formally specified system. (this Ph.D thesis)

   De beste manier om fouten in de specificatie te vinden is door het formeel gespecificeerde systeem te simuleren. (dit proefschrift)

2) The number of correctly designed-in entities per time unit is almost independent of the entity complexity. (**[koo92]**, page 224)

   Het aantal correct in het ontwerp gebruikte elementen per tijdseenheid is vrijwel onafhankelijk van de complexiteit van die elementen. (**[koo92]**, pagina 224)

3) The problem with VHDL as behaviour description language is stated in the first line of the VHDL language reference manual: *'The design entity is the primary hardware abstraction in VHDL'* (underline by author). (this Ph.D thesis)

   Het probleem van VHDL als gedragsbeschrijvingstaal is aangegeven in de eerste regel van de VHDL taal referentie handleiding: *'De ontwerp eenheid is de primaire hardware abstractie in VHDL'* (onderlijning door auteur). (dit proefschrift)

4) Modelling methods, that are used to define system architectures, should make a sharp distinction between system state and system structure.

   Modelleringsmethoden die gebruikt worden om systeemarchitecturen te definieren moeten een duidelijk onderscheid maken tussen systeemtoestand en systeemstructuur.

5) Innovative research can be seriously hampered by policy makers who do not see the tightly coupled multi-disciplinary nature of system design.

   Innovatief onderzoek kan ernstig gehinderd worden door beleidsmakers die niet inzien dat het ontwerpen van systemen een sterk gekoppelde multi-disciplinaire aangelegenheid is.

6) System optimisation and making preliminary implementation choices form the core of architecture design. (this Ph.D thesis)

   Systeem optimalisatie en het maken van voorlopige implementatie keuzes vormen de kern van het ontwerpen van architecturen. (dit proefschrift)

7) Describing the purpose of a language in a few lines is far more important than giving pages full of syntax definitions.

   Het beschrijven van het doel van een taal in een paar regels is veel belangrijker dan het geven van pagina's vol met syntax definities.

8) Interactive design and simulation tools reduce system design time by shortening the design-simulate-debug cycle.

Interaktieve ontwerp en simulatie gereedschappen brengen de systeem ontwerp tijd terug door het verkorten van de ontwerpen-simuleren-verbeteren cyclus.

9) Fully automated high-level architecture synthesis will never realize the results which can be achieved by a skilled and creative designer.

Volledig geautomatiseerde hoog-niveau architectuur synthese zal nooit de resultaten realiseren die kunnen worden bereikt door een getrainde en kreatieve ontwerper.

10) Viewing design processes as modifiable architectures of interconnected tool operations allows the design process itself to be optimised just like the designs made with it.

Het beschouwen van ontwerpprocessen als modificeerbare architecturen van gekoppelde ontwerpgereedschap-operaties laat het toe het ontwerpproces te optimaliseren op dezelfde manier als de ontwerpen die ermee gemaakt worden.

11) Common-sense organizational principles like reusability and interchangeability are still the exception rather than the rule (in software design). ([**cox90**]) The basic model provides system design with these principles.

'Gezond verstand' organisatorische principes zoals herbruikbaarheid en uitwisselbaarheid vormen nog steeds de uitzondering op de regel (bij het ontwerpen van software). ([**cox90**]) Het 'basic model' maakt het mogelijk deze principes the gebruiken bij het ontwerpen van systemen.

12) Fuzzy logic is more than just a buzzzzzword.

Stellingen behorende bij het proefschrift van A.C. Verschueren: 'An Object-Oriented Modelling Technique for Analysis and Design of Complex (Real-Time) Systems'. Eindhoven, 19 mei 1992.

About the cover:

The front cover shows the layout of a PCM telephone exchange switching network ASIC, designed with tools which are described in this Ph.D thesis and the ASA silicon compiler. Four of these 50 mm^2 chips connected in parallel form the core of a fault-tolerant telephone exchange capable of handling 10.000 subscribers.