EINDHOVEN UNIVERSITY OF TECHNOLOGY

Interactive modelling and simulation of heterogeneous systems

Citation for published version (APA):
Fleurkens, J. W. G. (1996). *Interactive modelling and simulation of heterogeneous systems*. [Phd Thesis 1 (Research TU/e / Graduation TU/e), Electrical Engineering]. Technische Universiteit Eindhoven. https://doi.org/10.6100/IR456563

DOI:
10.6100/IR456563

Document status and date:
Published: 01/01/1996

Document Version:
Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:

• A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the "Taverne" license above, please follow below link for the End User Agreement:

www.tue.nl/taverne

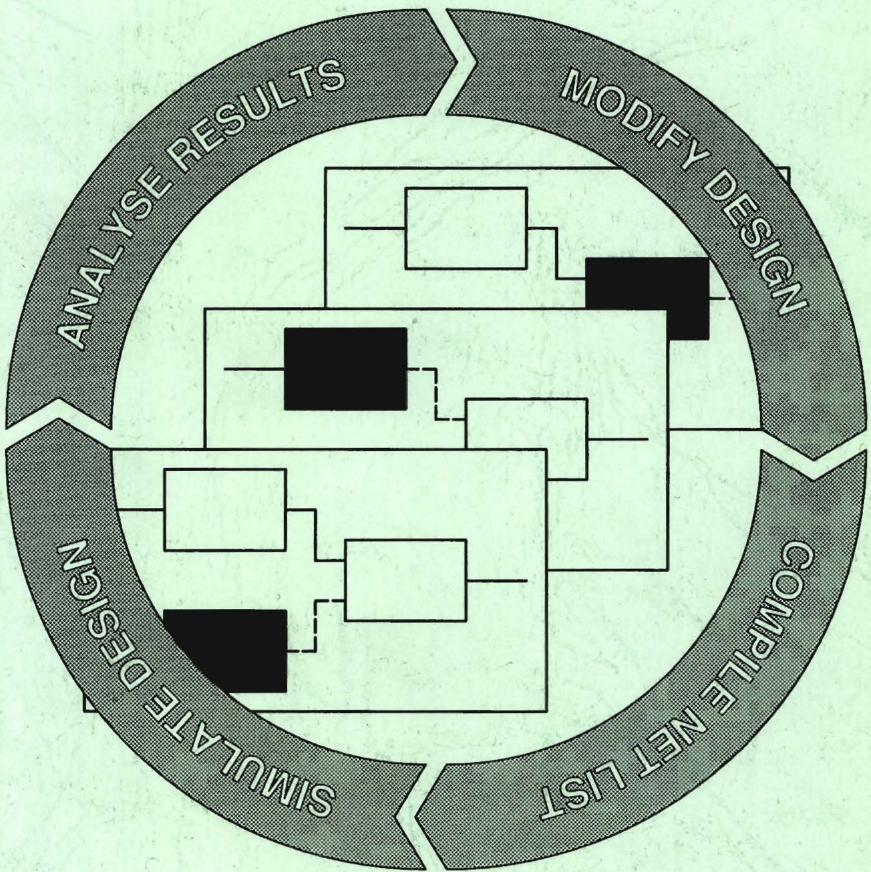
Take down policy
If you believe that this document breaches copyright please contact us at:

openaccess@tue.nl

providing details and we will investigate your claim.

Download date: 04. Oct. 2023

Interactive Modelling and Simulation of Heterogeneous Systems



Hans Fleurkens

Interactive Modelling and Simulation of Heterogeneous Systems

Interactive Modelling and Simulation of Heterogeneous 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 26 maart 1996 om 16.00 uur.

door

Johannes Wilhelmus Gerardus Fleurkens

geboren te Venray

Dit proefschrift is goedgekeurd door de promotoren:

prof.Dr.–Ing. J.A.G. Jess
prof.ir. M.P.J. Stevens

en de copromotor:

dr.ir. J.T.J. van Eijndhoven

© Copyright 1996 J.W.G. Fleurkens
All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without prior written permission from the copyright owner.

Druk: Wibro dissertatiedrukkerij, Helmond

CIP–GEGEVENS KONINKLIJKE BIBLIOTHEEK, DEN HAAG

Fleurkens, Johannes Wilhelmus Gerardus

Interactive modelling and simulation of heterogeneous
systems / Johannes Wilhelmus Gerardus Fleurkens. –
Eindhoven : Eindhoven University of Technology. – Ill.
Proefschrift Technische Universiteit Eindhoven. – Met lit.
opg. – Met samenvatting in het Nederlands.
ISBN 90–386–0377–0
Trefw.: elektronische schakelingen / simulatie.

Contents

Abstract .... **vii**

Samenvatting .... **ix**

Preface .... **xiii**

1 Introduction **1**

1.1 Background 1
1.2 Design and simulation in ESCAPE 4
1.3 Examples 9
1.4 Related work 12
1.5 Outline of this thesis 16

2 Event driven simulation **17**

2.1 Introduction 17
2.2 Simulation models 19
2.3 Discrete event simulation 21
2.3.1 Simulation approaches 21
2.3.2 Time advance mechanisms 24
2.4 Event driven simulation 25
2.4.1 The simulation state of the system 25
2.4.2 Event types 27
2.4.3 Examples using various delay models 31
2.4.4 Delta delay events 32
2.5 Event processing and management 33
2.6 Interaction between simulator and editor 36
2.7 Animation 42
2.8 Discrete event monitors 43
2.8.1 Discrete event automata 47
2.8.2 Examples and results 50
2.9 Hierarchy 51
2.9.1 Related work 53
2.9.2 Managing hierarchy 53
2.9.3 Design management 57

3 Multi–model simulation **59**

3.1 Introduction 59

3.2 Foreign language models 61
3.2.1 The foreign language interface 62
3.2.2 Compiled simulation 65
3.2.3 Experimental results 67
3.3 External simulator interface 68
3.3.1 Synchronization of simulation time 76
3.3.2 Example: The PLATO piecewise linear simulator 79
3.3.3 Simulation examples and results 82
3.4 Token flow models 83
3.4.1 Petri nets 85
3.4.2 Data flow graphs 89
3.4.3 Simulation of data flow graphs 92
3.5 Run–time configuration of the simulator 96

4 Simulation examples 99

4.1 Introduction 99
4.2 The inner product calculation chip 99
4.3 The bit blitter 102
4.4 Simulation of traffic on a road 103
4.5 Model of a railroad and block control system 106

5 Design process integration 113

5.1 Introduction 113
5.2 A programmable graph view 114
5.3 Inter process communication and the inter tool protocol 117
5.4 Configuration of the user interface 119
5.5 The interactive data access language 121
5.6 An example: the integration of NEAT and ESCAPE 122

6 Concluding remarks 125

A The LISP–like HDL 141

A.1. Introduction 141
A.2. The language 141
A.3. Summary of functions 141

B Simulating the bit blitter 147

C The inter tool protocol 151

Notation 155

Biography 157

Abstract

Simulation is used intensively to validate the functional behaviour of electronic systems. It may be used to make estimations on timing, performance and power issues. Simulation is also of great value during the prototyping and debugging of system descriptions in the specification phase. Both the increasing size and complexity of systems to be designed , the migration of the initial specification to higher abstraction levels as well as a reduction of the time to market put new demands on simulation tools.

Simulation performance and the ability to simulate different types of models are important parameters in developing a simulator that satisfies current and future needs. The key issue is to provide the flexibility to simulate a large variety of models without sacrificing simulation performance. Besides raw simulation performance, the design time itself should be reduced as much as possible.

This thesis describes the concepts and techniques that can be used to develop an event driven simulator that

- reduces the design cycle time to facilitate the prototyping and debugging of systems, and that allows to explore the design space;
- allows to simulate multiple types of models homogeneously.

All these concepts and techniques have been implemented in a tool called ESCAPE.

A reduction of the design cycle time has been achieved by embedding a graphics editor and discrete event simulator into a single tool. This allows a tight integration of the various phases of the design cycle, which reduces the overall design time significantly. The time consuming net list compilation phase is avoided by *incrementally* updating the simulation model with each modification of the underlying descriptions it is composed of. As opposed to other simulation tools, in which the compilation or elaboration of the simulation model consumes a large part of the designer's time, the simulation model can be re–simulated without perceivable delay to the designer. It is even possible to modify parts of the model during simulation, for instance to investigate the influence of errors on various aspects of the model.

The simulator also features various facilities to support the designer in analysing simulation results: the behaviour of the system may be animated or visualized during simulation. Furthermore, erroneous behaviour may be detected at a higher level of abstraction. This is achieved by hierarchically defining monitors that process sequences of events and replacing such sequences by events at a higher level of abstraction. These events do not influence the progress of the simulator itself.

This thesis also describes the problems and that are associated with multi–model simulations as well as some solutions. Since it is very hard to anticipate which models need to be supported in near future, the simulation kernel should be flexible enough

- to deal with multiple types of models at the same time;
- to be able to support relatively easy a new type of simulation model.

Performance and flexibility have been the dominating issues in the research of integration techniques that allow homogeneous simulation of multiple types of models.

The basic approach that has been taken here is that different types of simulation algorithms and different types of simulators are mapped onto the event driven paradigm: one event driven simulation kernel can handle different timing representations and different delay models. This kernel also orchestrates the execution of all models in a unified and proper way. Different types of events are used to accomplish this. To facilitate the integration of external simulators, both a simulation interface and a model encapsulation technique have been developed that deal with event conversion, value conversion and synchronization between the various models.

Finally, this thesis describes which techniques have been used to be able to customize the design and simulation tool and how it may be interfaced with other design tools. As a result, it can easily be incorporated into an existing design flow. When used in master mode, the tool may be used as a generic frontend, or as a simulation and visualization tool of other design tools, when used in slave mode. A very successful example is the integration with an architectural synthesis system.

The solutions presented in this thesis are substantiated by practical implementations and validated by a large variety of examples. They may be applied to various application areas, even in other disciplines.

Samenvatting

Simulatie is een intensief gebruikt middel om het funktionele gedrag van elektronische systemen te controleren. Het wordt gebruikt om schattingen te maken op het gebied van tijdsgedrag, prestatie en vermogensgebruik. Simulatie is eveneens een waardevol hulpmiddel bij het ontwikkelen van prototypes en het elimineren van fouten in systeembeschrijvingen tijdens de specifikatie van het gewenste gedrag. Omdat de grootte en de complexiteit van systemen toeneemt, systemen op een steeds hoger abstractieniveau beschreven worden en omdat tegelijkertijd systemen steeds sneller op de markt moeten komen, worden steeds hogere en nieuwe eisen aan simulatieprogrammatuur gesteld.

De snelheid waarmee gesimuleerd kan worden en de mogelijkheid om verschillende types van modellen te kunnen simuleren zijn belangrijke eisen die gesteld worden bij de ontwikkeling van een simulator die zowel aan de huidige als aan toekomstige behoeftes moet voldoen. Het is erg belangrijk om een groot scala van modellen te kunnen simuleren zonder de snelheid van de simulatie daaraan op te offeren. Behalve de invloed van de snelheid van de simulatie zelf moet ook de ontwerptijd zoveel mogelijk bekort worden.

In dit proefschrift worden een aantal concepten en technieken beschreven voor een *event* gestuurde simulator zodanig dat

- de iteraties tijdens het ontwerpen zo kort mogelijk worden en ook het aantal iteraties zo klein mogelijk is; dit bevordert de ontwikkeling van prototypes en maakt het gemakkelijker om fouten in beschrijvingen op te sporen en te verwijderen. Verder is het mogelijk om verschillende alternatieven af te wegen;
- verschillende soorten modellen tegelijkertijd op een homogene wijze gesimuleerd kunnen worden.

Al deze concepten en technieken zijn geïmplementeerd in een programma ESCAPE geheten.

De vermindering van de tijd om een ontwerpcyclus te doorlopen is mogelijk gemaakt door een grafische editor en een diskrete event simulator in één programma onder te brengen. Dit maakt het mogelijk om de verschillende fases van de ontwerpcyclus te integreren, waardoor de ontwerptijd aanzienlijk gereduceerd kan worden. De veel tijd kostende netlijst compilatie wordt verme-

den door het simulatiemodel *incrementeel* aan te passen, als één van de beschrijvingen waaruit het model is opgebouwd veranderd wordt. In tegenstelling tot andere simulatoren, waar het compileren of het uitwerken van het simulatiemodel een groot gedeelte van de tijd van de ontwerper kost, kan het simulatiemodel zonder enige merkbare vertraging opnieuw gesimuleerd worden. Het is zelfs mogelijk om tijdens de simulatie delen van het model te veranderen, bijvoorbeeld om de invloed van fouten op verschillende aspecten van het model te onderzoeken.

De simulator bevat eveneens een aantal eigenschappen die de ontwerper helpen bij de analyse van de simulatieresultaten. Het gedrag van het systeem kan nl. grafisch weergegeven worden tijdens de simulatie. Eveneens kan foutief gedrag ontdekt worden op een hoger abstractieniveau. Dit wordt bereikt door monitoren hierarchisch te definiëren die sequenties van events kunnen verwerken en deze vervangen door events op een hoger abstractieniveau. Deze events beïnvloeden het gedrag van het te simuleren systeem verder niet.

Daarnaast beschrijft dit proefschrift de problemen met betrekking tot, en een aantal oplossingen voor simulaties van een aantal verschillende soorten modellen. Omdat het erg lastig is om rekening te houden met de modellen die gebruikt gaan worden in de nabije toekomst, moet de kern van de simulator flexibel genoeg zijn om

- tegelijkertijd verschillende types simulatiemodellen aan te kunnen;
- relatief gemakkelijk een nieuw type simulatiemodel te kunnen toevoegen.

De efficiëntie en de flexibiliteit zijn de belangrijkste aspecten geweest bij het onderzoek naar integratietechnieken voor het op uniforme wijze simuleren van verschillende types modellen.

De basistechniek is dat verschillende algoritmes en simulatoren op hetzelfde event gestuurde model zijn afgebeeld: de kern van het simulatieprogramma kan omgaan met verschillende representaties van de tijd, verschillende vertragingsmodellen en controleert de uitvoering van alle modellen op een uniforme en correcte wijze. Dit is gerealiseerd door het gebruik van verschillende soorten events. Om de integratie van andere simulatoren te ondersteunen zijn een simulatie–interface en een encapsulatietechniek ontwikkeld. Deze zorgen voor de conversie van de events, de conversie van de signaalwaardes en de synchronizatie tussen de verschillende modellen en simulatoren.

Tenslotte beschrijft dit proefschrift een aantal technieken zodat het programma gemakkelijk aan de wisselende eisen van gebruikers kan worden aangepast en zodat dit programma kan samenwerken met andere ontwerptools. Deze technieken maken het mogelijk om het programma te gebruiken in een

bestaand ontwerptraject. Het programma kan met name gebruikt worden om andere programma's aan te sturen of voor het simuleren en het grafisch weergeven van gegevens van andere ontwerptools. Een goed voorbeeld is de succesvolle integratie met een architectuursynthesesysteem.

De oplossingen die aangedragen zijn in dit proefschrift zijn geïmplementeerd en gecontroleerd met een grote verscheidenheid aan voorbeelden. Ze kunnen gebruikt worden voor een groot aantal toepassingsgebieden, zelfs in andere vakgebieden.

Preface

This thesis is based on the research that has been performed in the Design Automation Section of the Department of Electrical Engineering of Eindhoven University of Technology in the Netherlands. In this period, an interactive flexible design and simulation tool called ESCAPE has been developed, in which a discrete event simulator and a graphics editor have tightly been integrated. The concepts and techniques that have been applied in this tool are described in this thesis.

Acknowledgements

I would like to thank Jochen Jess for giving me the opportunity to work in his research group and the Dutch Foundation on Fundamental Matter (FOM) and STW for sponsoring the research project under contract EEL 88.1427.

I would like to thank the following people for their contributions to my research work: Ronald Tangelder [Fle91], Pim Buurman [Fle93a] and Koen van Eijk [Fle95], and the students who helped me. Furthermore, I would like to thank the people who have provided some of the examples: Arnold Niessen who developed the basic components of the railroad model, Wim Philipsen who developed the model of the bit blitter, but in particular Geert Janssen.

I also would like to thank all other members of the Design Automation Section for providing a stimulating environment and a good atmosphere.

I am very grateful to Gjalt de Jong for thoroughly proofreading the draft version of my thesis and for the many discussions I have had with him.

And last but not least, I would like to thank my parents and my sister for giving me support throughout my study.

Chapter 1 Introduction

1.1 Background

Today's market, in which electronic products have a short life time, requires that new products are put on the market as soon as possible, while the production costs are minimized. The total costs of manufacturing electronic products are greatly reduced, if the number of components of these products are decreased. This can be achieved by integrating more components on a single chip. Although the progress in technology makes it possible to integrate more and more components on a single chip every year, the design process itself is becoming more and more a limiting factor for manufacturing ICs with complex designs.

To design an IC, which performs exactly as specified, is an extremely difficult task. Therefore, this task is usually divided into a number of implementation steps. A sequence of implementation steps transforms an initial specification of a system into a mask description of the layout of the chip, which can be processed in a foundry. Every implementation step has to be validated intensively. If the result of this validation does not satisfy, the implementation step has to be repeated. If the validation phase does not detect erroneous behaviour, the resulting design can serve as the specification of the next implementation step. In this thesis, an iteration of implementation and validation activities between two levels is referred to as the *design cycle* (see figure 1.1). To facilitate the design task, design automation tools have been developed in the last decades for implementing and validating designs.

The implementation steps have been automated to a large extent by developing synthesis tools for different abstraction levels. As a result, the level of initial specification is moved from the layout level to the system level. Currently, most research in this area is focussed on architectural synthesis, hardware software co–design, and system level synthesis tasks like specification, partitioning and interface synthesis. This thesis deals with the other phase of design activity: the validation part of the design cycle. The thesis especially focusses on validation at the highest levels of abstraction. Design validation is extremely important, because undetected errors may result in expensive redesigns and long delays before the resulting product can be put on the market.

Besides for validation purposes, simulation may also be used to make estimations on timing, performance and power issues.

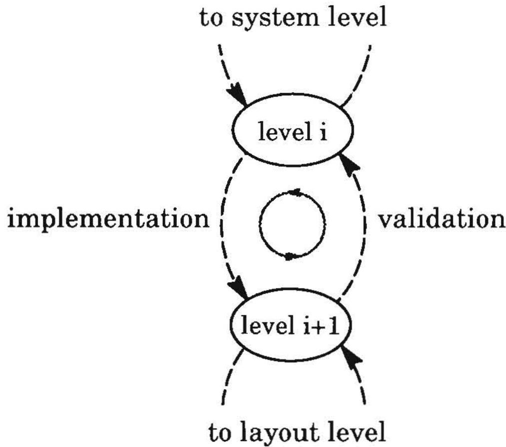


Figure 1.1. A step in the design trajectory

Simulation has been and currently is the most important technique to validate the functional behaviour of a system. Implementation steps are validated by comparing the simulation results at two different abstraction levels. Synthesis tools should make the validation process easier: in principle, they should produce results that are correct by construction. However, in practice the results of each synthesis tool have to be validated as intensively as the results of other approaches, because of bugs and deficiencies in those tools. Therefore, validation remains an important (and the most time consuming) design activity.

Simulation can not guarantee the correctness of a design in all circumstances, because the number of cycles required to get full coverage of errors is too large to perform this task in reasonable time. Despite this fact, simulation is fully accepted in the design community. Formal verification techniques are a promising alternative to simulation as a validation technique and their application often guarantees full coverage of errors [McM94]. Although much progress has been made in this area, these tools often fail to handle real–life designs and definitely can not handle full system specifications. Therefore, both simulation and verification tools should supplement each other to perform system validation. A simulator may also assist verification tools in locating and correcting design errors, e.g. by feeding the simulator with output traces of verification tools that lead to erroneous behaviour. But the most important application area of simulation will be its use during the prototyping, validation and debugging phases of an initial specification of a system.

A number of problems are associated with the simulation of large systems. The increasing complexity of systems makes simulation a difficult and time–consuming task. Even worse, the computational time increases exponentially with a decreasing level of abstraction, which is often required to accurately simulate a system. For instance, simulating a complete system at the circuit level is nowadays (almost) infeasible, because it may require several days or even weeks. A number of solutions exist for this problem: mixed–level simulators handle the problem by simulating various parts of the model at different levels of abstraction. This solution reduces the computational costs of the simulation task significantly. Other techniques like parallel or distributed simulation, or improvements on the simulation algorithm itself, only give minor (i.e. linear) performance improvements.

Not only much computational time is required to validate a system description, but the time spent by the designer to iterate through the design cycle is increasing rapidly as well. This is caused by both the complexity of interpreting simulation results, and the separation of synthesis and validation tools. The latter problem often causes expensive re–simulations for small design changes. Simulation time is also reduced by incremental simulation tools [Cho88], which restrict re–simulation to those regions of the system, that are affected by design modifications. This approach takes a large amount of memory to store simulation histories and is only useful in the debugging phase, when only small modifications are made to the design. Most work in this area still focusses on the lower levels of the design trajectory, in particular logic and switch–level simulation.

Validation at the system level introduces another problem: systems often contain subsystems of different nature and the design of each subsystem may involve a different design trajectory. Often, each subsystem is best modelled using the most natural and appropriate description language or, from simulation point of view, the most appropriate model of computation. The choice of an appropriate description language may also depend on the back-end design tools for that particular subsystem. Different descriptive means can also be used to model different properties or aspects of a system. For instance, a Petri net could model the communication behaviour between different components of a system: the operational semantics of a Petri net determine when a component needs to be evaluated. The (functional) behaviour of each component could be described using a hardware description language. In this case, the Petri net might be verified for properties like liveness and deadlock by dedicated analysis tools, while a simulator might be used to validate the overall system behaviour.

Validation of such heterogeneous systems can only be performed efficiently and accurately, if multiple models of computation are supported by the simu-

lation tool. Since it is very hard to anticipate on the models of computation that are going to be used to describe systems, it has to be relatively straightforward to include support for new models of computation in the simulation tool.

In the next section, an overview is given of a design tool called ESCAPE, which tries to cope with the problems related to the validation of complex heterogeneous systems.

1.2 Design and simulation in ESCAPE

ESCAPE is a highly interactive and flexible design environment. It can be used to capture complex heterogeneous system descriptions and verify these descriptions through simulation. Its open architecture makes possible easy adaptation to different application areas and simulation needs. The environment provides an incremental design and validation strategy and animation techniques to visualize the activity and other properties of the system under design. This allows fast exploration of the design space and easy debugging of a system description.

ESCAPE has evolved from a simple schematics entry program called ESCHER [Lod86] that has been extended with a discrete event simulator [Jan89] towards a sophisticated design environment, which combines textual and graphical design entry and the validation of systems composed of these descriptions. The role of schematics entry in the CAD community has partly been replaced by popular hardware description languages (HDLs) like VHDL [IEE88] and Verilog [Tho91]. HDLs are used to describe both structure and functional behaviour of systems, but can be combined with graphics entry as well. Often, a graphical representation of a system gives a better perception of the system's overall structure. Especially, if these graphical representations are used for visualization purposes by other tools, they are preferred over textual descriptions. Graphical languages and graphical design capture have gained in popularity lately, because of standardization of graphics libraries and fast graphics workstations. Graphical languages or descriptions are used for various purposes:

- structural decomposition of a description, which is analogous to the use hierarchy in schematics capture programs;
- functional decomposition of either hardware or software descriptions;
- design methodologies which rely on a graphics formalism.

In ESCAPE, a combination of graphical and textual formats can be used to describe a system. Systems are described hierarchically and consist of compos-

ite and primitive modules. A composite module is described using another graphical description, that again consists of other composite or primitive modules. Primitive modules require a textual description to document their behaviour. Such a description is not necessarily a pure behavioural one: it might contain structural data as well, if that is supported by the hardware description language. In this thesis, a primitive module is related to a textual description, mostly describing its behaviour, whereas a composite module is related to a graphical description, mostly describing structure.

The internal organization of ESCAPE is depicted in figure 1.2. ESCAPE is composed of several functional components, which are controlled through the user interface and which access the same data structures through the application procedural interface (API). The organization of ESCAPE will be discussed in more detail starting with the internal data structures. This will also give more detailed information on the capabilities of this interactive design and simulation environment.

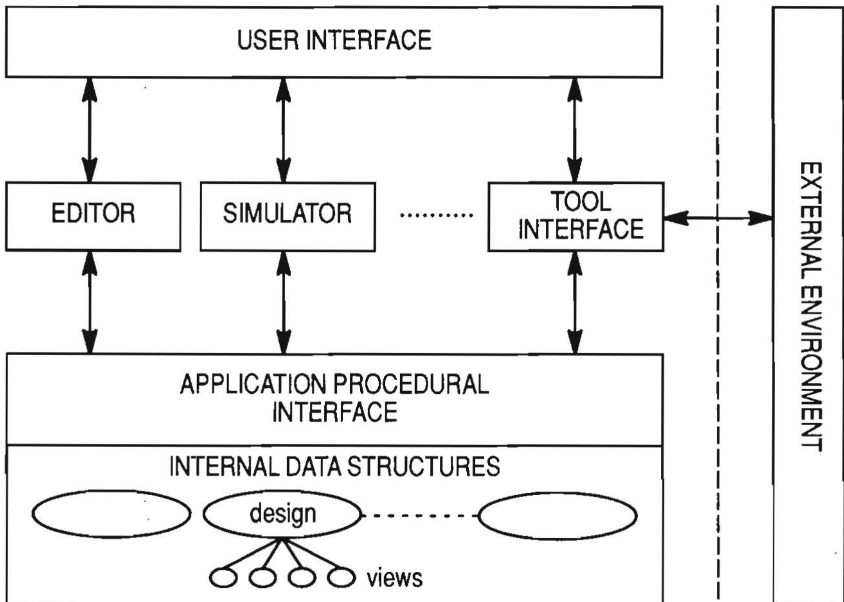


Figure 1.2. Schematic overview of ESCAPE.

The internal data structures store all data that is manipulated in a single session. In ESCAPE, a basic entity is called a *design*. Each design is a collection of different views, which represent various aspects and alternative implementations of a design. The following view types are defined:

- The *symbol* view which is used to define both the representation of a design and its interface.
- The *network* view which is used to capture structural descriptions of a design. Another design is instantiated in this view using its symbol view.
- The *graph* view which is used to capture graph models. Graph models are often used as an intermediate format in synthesis and verification tools. Many formalisms and representations in CAD applications are based on graphs. Examples are finite state machines, data flow graphs, Petri nets. A special definition language is developed to describe different classes of graphs: it allows to customize this view to be able to capture different types of graphs.
- The *text* view which is primarily used to capture behavioural descriptions of a design.
- The *link* view which is used to define a set of relations between different objects in views of one or more designs. In this thesis, such relations are referred to as *explicit* relations, i.e. explicitly defined by a user.

Besides explicit relations, some relations are directly modelled (hardcoded) in the underlying data structures: these relations are required by the built–in tools to correctly access and modify design data. They are referred to as *implicit* relations. This is opposed to the explicit relations which can be used without any restriction to model additional relations between various views. Multiple sets of explicit relations are used to capture different information. Each set is considered as a separate link view.

An example of an implicit relation is the link between a module in the network view and the symbol view of the design, of which this module is an instantiation. This relation is mandatory and models hierarchy. Explicit relations are for instance used to relate nodes in the three graph representations of the architectural synthesis system NEAT [Hei94]: ESCAPE is able to capture and visualize the design data used by this synthesis system by storing the relations between the various representations of synthesis data.

Each view consists of two data structures: the graphical data structure and the *network* data structure. Data in both of these data structures is strongly related. Data of particular objects (e.g. a module, a wire or some textual information) are represented in either data structure or both. The graphical data structure stores all data of an object related to its position, size and appearance on the screen. The graphical data structure is used to build and modify the network data structure in an incremental way. The network data

structure always represents an up–to–date net list of the system that is directly accessable by other tools, like the built–in event–driven simulator.

All functionality to create, modify, delete and access the objects in the internal data structures is accessable through an application procedural interface. Tools or new functions can be built which are compiled and linked into a single program (executable). In ESCAPE, a graphics editor and a discrete event simulator have been integrated tightly in a single program. This reduces the overhead of transferring simulation data using files or other mechanisms provided by the operating system to a minimum. Using the same concepts, more tools could be integrated in the same tool as well: this would further reduce the overhead between two or more design steps, and allow incremental updates and validation of multiple implementation or synthesis steps.

One of the components of ESCAPE is the event driven simulator, which is fully embedded within the tool or environment: it executes in the same address space as the editor. The performance of this simulator is not affected by the integration itself: the network data structures are optimized for access by the embedded simulator. The close interaction between simulator and editor reduces the design time: for instance, excessive net list compilation times for large net lists are avoided, because each modification immediately updates all data structures. In ESCAPE, it is even allowed to modify a design during a simulation experiment, which gives the opportunity to explore the design space and to simulate exceptions without any overhead.

With ESCAPE, the operation of a system can be *animated*. The simulator provides the functionality to highlight objects on specific conditions and to annotate simulation data on the screen. Although animation reduces the performance of the simulator, it greatly improves the understanding of the operation of the system. It also improves the debugging capabilities of the tool, since the location of errors can be isolated easily. Furthermore, the concept of *discrete event monitors* is provided to better support the validation of complex systems: a discrete event monitor analyses the event trace of a simulation run and can abstract from unnecessary low level details.

ESCAPE deals with the specific demands for system level modelling and simulation: models of different domains can be combined in a single simulation model and the result is simulated in a homogeneous way. To accomplish this, various techniques are used:

- the use of different event types that manipulate the behaviour of the simulator;

- the generation of code to embed models automatically into discrete event compatible models. The resulting model can contain a complete scheduler, which handles the event interface to the outside world;
- programmable simulation interfaces, through which an external simulator can be connected to the event driven simulator.

The same techniques are used to simulate partial implementations of a system: this may be required for analog subsystems and other critical parts of an electronic system. In principle, these techniques can be used for hardware software co–design as well (see section 1.4).

Currently, the behaviour of primitive modules can be described by

- a hardware description language based on the LISP [Ste84] language. Besides the statements found in other LISP languages, this language provides statements to define the animation of the system's components.
- programming languages like C and FORTRAN. A foreign language interface allows interactive and efficient simulation of compiled code models.
- a behavioural subset of VHDL as described in [Hou93].
- token flow models, in particular the ASCIS data flow graph.
- models simulated by an external simulator. An external simulator interface provides the necessary means to use ESCAPE as a simulation backplane. Different models can possibly be executed on different machines in the network, but this only improves performance, if the overhead of communication costs is small with respect to the computational costs of executing the models.

The *tool interface* gives access to the API from the external environment. This interface can be used in both master and slave mode. In slave mode, ESCAPE is invoked by another tool, which sends a series of commands to ESCAPE to visualize output or to perform other tasks. In master mode, external programs or tasks are started through the user interface and the results are read back by ESCAPE. These results can again be visualized in one of its view windows. In master mode, different tasks can be scheduled simultaneously, either on the same workstation or on different workstations in the network. This interface can be used for various purposes and provides users with all the flexibility and customizability they need to integrate various tools with ESCAPE and to use the functionality of ESCAPE without additional programming effort.

1.3 Examples

In this section, two small examples are presented that give an impression of the capabilities of ESCAPE:

- a delay insensitive two–place one–bit ripple buffer;
- a 5 x 5 Conway's Game of Life board showing a glider pattern.

A delay insensitive circuit called RIPP2 [Ber92] is a two–place one–bit "ripple" buffer, which is composed of a number of basic components. The set of basic components has been defined in [Ber88] and can be used to compile any program written in Tangram into a delay–insensitive circuit. Tangram is a HDL based on Communicating Sequential Processes (CSP) [Hoa78].

All basic components have been defined in ESCAPE as primitive modules. Defining a primitive module comprises of drawing a symbolic representation in the symbol view and editing a textual description representing the behaviour of the component. Some components have a more detailed implementation as well. For this example, the built–in LISP–like HDL has been used, which is described in more detail in appendix A. The basic component *variable* is depicted in figure 1.3 as an example.

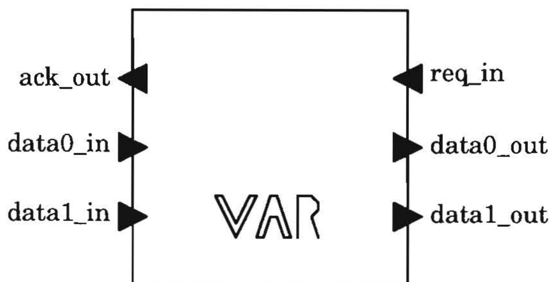


Figure 1.3. Symbol view of *variable.*

The corresponding behaviour is:

```
(behaviour var
    (term data0_in)
    (term data1_in)
    (term data0_out)
    (term data1_out)
    (term req_in)
    (term ack_out)
    (state value)
```

```
        ;; start simulation with random contents:
        (if (= (simtime) 0) (setq value (random 2)))

        (color (or req_in data0_in data1_in))

        (if (or data0_in data1_in)
            (progn
                (setq value data1_in)
                (delay 1 ack_out 1))
            ;else both low
            (delay 1 ack_out 0))

        (delay 1 data0_out (and req_in (not value)))
        (delay 1 data1_out (and req_in value))
        (write value)
)
```

In this description, the statements, which are used to visualize the activity of the module during simulation, can easily be identified. The *color* statement will highlight the module, if one or more of the module's inputs are high. Otherwise, the module will have its default colour. The *write* statement draws the current value of the state variable *value* in the module.

The implementation of this component can be modelled in the network view as well (see figure 1.4): the designer may choose to expand an instance of design variable and simulate its implementation instead of its behavioural description. For debugging purposes, a lower level implementation can be shown in a separate view window.

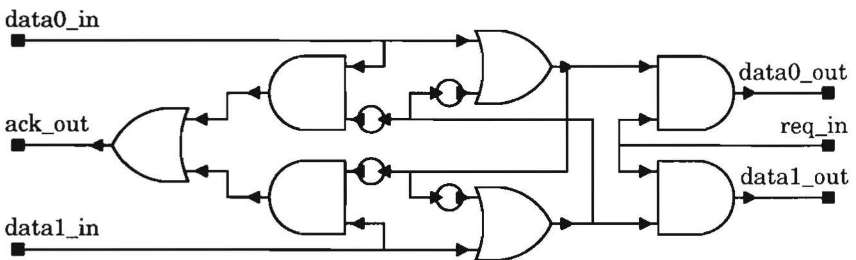


Figure 1.4. Implementation of basic component *variable.*

The next example is a network representing a world consisting of 25 cells of Conway's Game of Life [Gar70, Gar71]. This game is considered to be a cellular automaton: each cell of the automaton exchanges information with its 8

neighbours. Each cell has a state variable that indicates if this cell is alive or dead. The following rules are applied to each cell of the automaton:

- a cell will be alive in the next generation, if exactly three of its neighbours are alive in the current generation;
- a cell will stay alive in the next generation, if two or three of its neighbours are alive in the current generation;
- if none of the rules above applies to a cell, it will starve in the next generation.

Depending on the starting pattern of alive cells on the board, new generations will be generated from it. Some patterns are known to move itself along the board forever (e.g. gliders and fishes), other patterns oscillate with a specific period of generations before the orignal pattern is generated again. Large patterns have been developed for this type of cellular automaton. Examples are *blockpushers* and large *ships* having satellites. These patterns consist of numerous smaller patterns.

The behaviour of each life cell is described as follows:

```
(behaviour life_cell
  "Conway's game of life. description of 1 cell."
  (term nw_in) (term nw_out)
  (term no_in) (term no_out)
  (term ne_in) (term ne_out)
  (term ea_in) (term ea_out)
  (term se_in) (term se_out)
  (term so_in) (term so_out)
  (term sw_in) (term sw_out)
  (term we_in) (term we_out)
  (term init) ; used to set initial state
  (term clk) ; global control
  (state alive) ; holds state of cell: alive or death
  (local count) ; counts number of living neighbours

  (if (= (simtime) 0) ; initialize state
    (setq alive init))
  (color alive) ; color cell according state
  (if clk
    (progn ; determine new state of cell
      (setq count (+ nw_in no_in ne_in ea_in se_in so_in sw_in we_in))
      (setq alive (or (= count 3) (and alive (= count 2))))))
  ;; else
  ;; inform neighbours of this cell's current state:
  ;; refrain events: clock triggers evaluation
```

```
    (delay 0 nw_out alive 1)
    (delay 0 no_out alive 1)
    (delay 0 ne_out alive 1)
    (delay 0 ea_out alive 1)
    (delay 0 se_out alive 1)
    (delay 0 so_out alive 1)
    (delay 0 sw_out alive 1)
    (delay 0 we_out alive 1)
)
```

In the declaration section, the ports for communicating the state, the clock that controls the execution of the cell and the port that sets the initial state of the cell, are easily identified, as well as the state variable, that indicates its lifeness, and a local variable for counting the neighbours alive. The algorithm itself is straightforward: if the value of the clock rises to *high*, the rules of the game are applied to the cell using the values on its input ports. The result is communicated to the other cells using refrain events: these events do not cause the cell to be evaluated. Instead, if the value of the clock falls to *low*, the evaluation of all life cells is triggered.

Using a small world consisting of 5 x 5 cells, a glider pattern is initialized in the first generation. During simulation, the state of each cell is animated on the screen: the glider pattern starts to move from cell to cell, while it is rotating around. Some snapshots of this animation are presented in figure 1.5.

Larger worlds having more life cells have been developed as well. The underlying structural models for these examples have been generated using a special purpose program for placement and a routing program for schematics. Examples are:

- a 18 by 11 world starting with a oscillator as initial generation. This oscillator has a cycle of 16 generations;
- a 80 by 30 world starting with multiple glider patterns as initial generation.

The simulation speed for these examples is dependent on the number of cells, and it is not dependent on the total number of cells alive because in each cycle all cells are evaluated. The evaluation of a cell is triggered by the clock signal, which is an input of every cell. The behaviour of a model of which the speed mostly depends on the number of cells alive would be more complicated.

1.4 Related work

The functionality offered by ESCAPE has overlap with various kinds of systems. The most important one is Ptolemy. Simulation backplanes and mixed–level simulators are other types of simulators, which have some overlap. Another

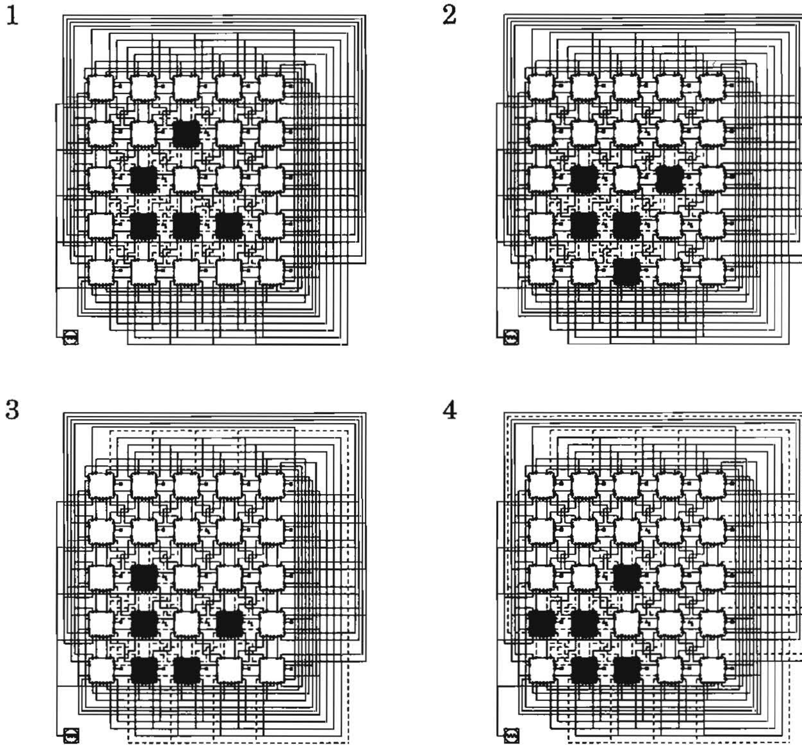


Figure 1.5. Animated simulation of Conway's Game of Life.

category of systems are development systems for DSP applications, which are readily available on the market. These and some other types of tools are explained in more detail in this section.

Ptolemy [Buc94] is a framework to capture and validate heterogeneous systems. It is focussed towards designing signal processing and telecommunication systems, but is used in other application areas as well. Ptolemy is based on an object–oriented programming paradigm. It consists of a kernel on top of which different models of computation (called domains) can be used. Among others, the following domains can be used: synchronous data flow, dynamic data flow and discrete event. Ptolemy has also been used for hardware software co–design: a dedicated simulator for a DSP microprocessor has been coupled to the system [Kal92]. If such a simulator would be available in public, such a simulator could easily be coupled to ESCAPE as well.

In Ptolemy, there is one top–level view of a simulation model. It is called the Universe and has a domain associated with it. The Universe contains a number of computational blocks, which operate in the same domain. These blocks

are called Stars. Stars are interconnected, which allows the transfer of data between them. In addition, the Universe contains a scheduler, which controls the execution order of these blocks. Models of other domains can be included in the Universe using a so called Wormhole, which behaves like a regular star. A Wormhole interfaces the foreign domain with the domain of the Universe using an interface called the EventHorizon. Each domain provides an interface to this EventHorizon.

In Ptolemy, different domains are hierarchically nested. Each foreign domain has its own scheduler, which controls the execution of the stars in this domain. During simulation, the current simulation time is known to all schedulers. Combined with a conservative scheduling approach, no deadlock can occur in the coordination of the various schedulers. However, this approach may restrict the feedback of inner domain particles to the outer domains or have a significant performance penalty.

The Simulator Coupling System (SiCS) [Nie92, Ocz91], is a simulation backplane, which can be used to simulate systems composed of different models of computation. SiCS consists of a coupling kernel and a user interface. Unlike ESCAPE, SiCS itself provides no support for simulating models of computation. SiCS contains a procedural interface for the interfacing between a simulator and the simulation kernel. This interface contains various functions for signal administration, signal manipulation and simulation control and enables the coupling of external simulators to the backplane. In [CFI94], a draft proposal for a standard on simulation backplane programming interfaces is described. In this document, much attention has been paid to the data types and the representation of values, the synchronization of the simulators, the user interface, and the resolution and conversion functions.

As opposed to a simulation backplane, other examples of multi–domain simulators are various mixed–mode simulators, which have been developed in the last decade. An overview of many mixed–mode simulators is given in [Sal90]. A mixed–mode simulator is oriented towards simulation at various well–defined levels of abstraction in order to reduce the computational complexity of simulating large systems. The main difference between these simulators and a simulation environment like ESCAPE, is that they do not provide the functionality to integrate new domains or levels of abstraction within the system, especially if the simulation paradigm is not event–based. This often implies that the integration of a new model of computation results in the development of a (partly) new simulator. An example of this are the various derivations of the ELDO simulator [Hen85], of which VHDL–ELDO is the latest one [Tah93].

STATEMATE™ [Har90] is a design framework, which can be used to specify and validate reactive systems. The design methodology is based on three graphics formalisms: module charts, activity charts and statecharts. Module charts represent the structural view of the system; this is analogous with structural descriptions in other tools, like the network view in ESCAPE. Behaviour is described using both activity charts and statecharts. Activity charts are used to specify the functional behaviour of components used in the module charts. The interaction between these components or, in other words, the specification of the control activities is described by statecharts [Har87].

Statecharts are a graphics formalism that offer some extensions to the FSM formalism to facilitate the description of complicated control activities. To reduce the complexity of this task, hierarchy is allowed by repeated decomposition of states into substates. Another difference is that the thread of control of a statechart is not necessarily sequential. Furthermore, statecharts allow broadcasting of messages to apply actions to multiple states, e.g. a reset.

STATEMATE is a good example of the fact, that visual formalisms are getting more important and that they gain in popularity. This effect is reinforced by the development of workstations with powerful graphics capabilities and the standardisation of graphics libraries. Like with ESCAPE, it is possible to animate the operation of the system in STATEMATE. The specification of the system can be analysed interactively or in batch mode. Various debugging commands are provided to control the execution of the model.

Related to the work described above is the SpecCharts language [Vah91]. used for specific system level synthesis tasks like partitioning and interface synthesis. The language is based on hierarchical and concurrent state diagrams (comparable to statecharts) and is extended with VHDL based constructs.

In [Hoe92], a design system for the specification and implementation of digital signal processor arrays is described. In this system, different applications like a network editor and a simulator are sharing a common data structure. A communication mechanism is used to notify other (passive) applications of modifications in the state of the system (caused by the execution of an application).

Visual hardware description languages are another example of design capture systems, which rely heavily on graphics. In [Gol93], a visual language for VHDL is described consisting of a graphics editor and a compiler that translates the graphical descriptions into a textual VHDL description for analysis and simulation purposes. Visual HDLs are also available commercially and often consist of graphical design capture, simulation and code gen-

eration tools. The graphics formalisms are often derivatives of FSM, flowchart and block diagram approaches. The output of such systems is a description in a standard HDL like VHDL or Verilog, which serves as input for other simulation and synthesis tools. The HDL code generation offered by these tools is tailored for commercially available synthesis tools. In general, these systems lack the extendibility of a system like ESCAPE. A designer is restricted to use the graphical languages provided by the system and can't include other models of computation.

1.5 Outline of this thesis

In chapter 2, the capabilities of the discrete event simulator are discussed in detail. It is also explained how the design cycle (edit – elaborate – simulate – analyse) time is reduced by applying incremental techniques and run–time analysis techniques. In chapter 3, various approaches are discussed that allow simulation of multiple models of computation and models at various abstraction levels by the discrete event simulator.

Throughout chapters 2 and 3 examples and experimental results are used to illustrate the various concepts and approaches applied in the prototype ESCAPE. In chapter 4, some additional examples are given that illustrate the flexibility, modelling capabilities and performance of the simulation environment.

In chapter 5, some additional features of ESCAPE are described. It shows how it is customized and integrated with other tools in the design flow. Finally, some concluding remarks are made on this research project and some suggestions are proposed to extend the prototype tool and to apply the techniques and concepts described in this thesis to other simulators.

Chapter 2 Event driven simulation

2.1 Introduction

Simulation is widely being used to validate the functionality and the performance of a description of a system under design at various levels of abstraction. At each level, a simulator meets different requirements with respect to model size, level of detail of the models and computational costs. The following levels are distinguished:

- *System level.* At this level, a system is described as a number of interacting components, which are described by a model of computation or a behavioral description. Subsystems may be of different nature: digital hardware, analog hardware, firmware or software components.
- *Architectural or functional level.* The function of a (sub)system is described as an algorithm, which receives data from its environment and issues data to its environment after calculation. Descriptions at this abstraction level often serve as the input specifications for architectural synthesis tools.
- *Register Transfer Level (RTL).* The system is modelled as a datapath and a controller. The datapath consists of computational blocks, multiplexers and registers. Data is read from the registers and fed to the computational blocks. After calculating the resulting output values, these values are transferred back to the registers. The multiplexers are used to direct the flow of data through the system. The controller is a finite state machine (FSM), that specifies how the data is transferred in each state.
- *Logical level (gate level).* The following classification is used for models at this level: combinational logic, sequential logic and asynchronous logic. Sequential logic can be handled in a similar way as combinational logic by cutting the nets at the flip–flops creating new inputs and outputs. Much research has been conducted at this abstraction level to increase the performance of logic simulation: optimization of the simulation algorithm for specific delay models, parallel logic simulators [Mat92] and hardware accelerators [Sas83, Bla84]. Note that raw simulation speed is not always the most important aspect of a simulator. A software implementation may be indispensable for debugging the model and may complement the use of a hardware accelerator [Bee90]. Many of the simulation techniques that have been developed for this abstraction level are being applied to higher

levels of abstraction as well, for instance hardware acceleration at the RTL [Tak90] and distributed simulation at the behavioural level [Gho95].

- *Switch level.* This level is used to model digital circuits as a network of nodes and transistors. Each transistor represents both a switch and a attenuator. At this abstraction level, many aspects MOS circuits can be modelled quite accurately. However, the computational costs are small compared to simulation at the circuit level. Examples of switch level simulators are MOSSIM II [Bry84] and COSMOS [Bry87]. In [Gen86], the model is refined using a network of switches, capacitors and resistors. In this model, analog values instead of a set of discrete values are used to model the strengths of the components. This enables to simulate a model fairly detailed with respect to timing aspects.
- *Circuit level (electrical level).* The model of a system is a set of differential equations, which are composed from the model equations of each component and from Kirchhoff's voltage and current laws. To simulate this model, the DC solution and the transient solutions have to be calculated from these equations. Simulation at this level produces the most detailed results. However, it is also most expensive with respect to computational costs. Spice [Nag75] and its numerous derivatives symbolize simulation at this abstraction level.

At various levels of abstraction, VHDL and Verilog are used as the standard hardware description languages. For these languages, many simulators are commercially available from various CAD tool vendors. Models written in these languages can be simulated together as well: this is referred to as *co-simulation*. There are also hardware accelerators available for VHDL simulation as well as techniques for parallel simulation [Vel92].

Simulation models at all abstraction levels can be divided in two classes: time discrete and time continuous models. The definitions are given below:

DEFINITION 2.1: A time discrete (simulation) model is a model of which the (simulation) state can only change at a *countable* number of time points.

DEFINITION 2.2: A time continuous (simulation) model is a model of which the (simulation) state changes continuously with respect to the simulation time.

Discrete and continuous models differ in the following ways:

The simulation time in a discrete simulation model is in general represented by an integer number, whereas the simulation time in a continuous simulation model is represented by a floating point number. In a continuous simulator, the simulation model is often described using differential equations; in

a discrete simulator, the model is composed of executable components, that are invoked by a scheduler. Events are used to model activity in such models. In this thesis, the term *discrete event simulation* refers to models, that are simulated by an event driven or a similar simulation algorithm, (e.g. a compiled simulator, see section 2.3.1).

This thesis primarily focuses on simulation of models at the register transfer level and higher levels of abstraction. These models are mostly modelled using a time discrete model. Models without an explicit notion of time and models of partial implementations at lower level of abstractions can be combined with time discrete models as well. Therefore, we restrict ourselves to discrete event simulation.

2.2 Simulation models

Systems are composed of computational blocks. These blocks are connected with each other by nets. Nets are used to communicate values from one computational block to others. A system S is described with the 7–tuple:

$$S = (\mathcal{M}, \mathcal{N}, \mathcal{P}, \mathcal{B}, P_{\mathcal{M}}, P_{\mathcal{N}}, M_{\mathcal{B}}) \tag{2.1}$$

where

- $\mathcal{M}$ is the set of modules or components.

- $\mathcal{N}$ is the set of nets or signals.

- $\mathcal{P}$ is the set of ports. This set contains both primary input and output ports of the system, and the input and output ports of the modules.

- $\mathcal{B}$ is the set of behavioural descriptions.

- $P_{\mathcal{M}} : \mathcal{P} \rightarrow \mathcal{M} \cup \{ \emptyset \}$ is a function that maps each port to a module. In case $p \in \mathcal{P}$ is a primary port, $P_{\mathcal{M}}(p)$ will return empty.

- $P_{\mathcal{N}} : \mathcal{P} \rightarrow \mathcal{N}$ is a function that maps each port to a net.

- $M_{\mathcal{B}} : \mathcal{M} \rightarrow \mathcal{B}$ is a function that assigns a behavioural description to each module.

The system model of S in (2.1) describes a structure. To complete this model, each module requires a behavioural description. The term *computational block* is used for a module, which has a behavioural description associated with it. A behaviour $B \in \mathcal{B}$ is described with the 5–tuple

$$B = (I, O, Q, init, exec) \tag{2.2}$$

where

- I is the set of input ports.
- O is the set of output ports.
- Q is the set of state variables (memory elements).
- *init* is a function that calculates the initial values of all state variables,
- *exec* is a function that calculates the values of the output ports for a time point greater or equal than the current time point from the values of the input ports and the values of the state variables. The new values of the state variables are calculated from the values of the input ports and the current values of the state variables as well. Time may also be used in the function to account for time–dependent behaviour.

The exec function calculates the new values of the output ports at future time points. Note that more than one value may result from this calculation. However, the number of values produced should be finite. Otherwise, the function exec will not terminate after activation of the behavioural model. This leads to an erroneous condition during simulation, if more components need to be activitated, which is usually the case.

The behaviour of each module of (2.1) is described using (2.2). The inputs and outputs of behaviour B correspond with the ports of the module it is assigned to. An input gets its value from the net, that is connected to the corresponding port of the module. An output drives the net, that is connected to the corresponding port of the module. In figure 2.1, it is depicted how a module interacts with its environment. The behaviour of a module is executed or simulated, if activity occurs at one or more of the module's inputs.

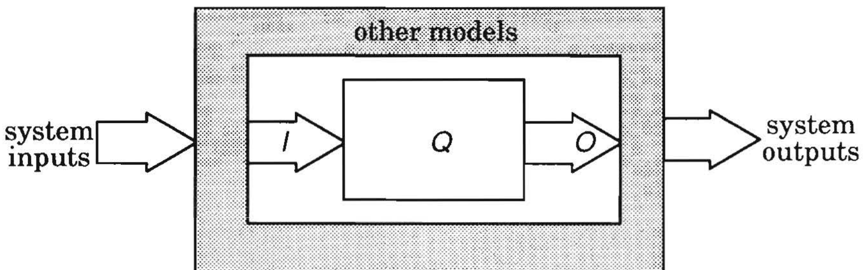


Figure 2.1. The execution of a model in its environment

A behaviour described using (2.2) closely resembles the description of a finite state machine. However, there are some differences:

- The data types associated with I, O and Q are not necessarily finite. The type of an element of these sets may for instance be a boolean, a natural number, a floating point number or some abstract data type.
- The model may have internal non–determinism. Even if a model has internal non–determinism, still its overall behaviour may be deterministic: the relation between the values at the input and output ports may still be deterministic.
- The behaviour may be dependent on time. Although it is not explicitly represented in (2.2), the *exec* function takes time as an argument.

Finally it should be noted that notation (2.1) does not account for hierarchical descriptions. This is not necessary, since its purpose is to represent a valid simulation model. It does not describe how the model is composed from a hierarchy of structural and behavioural descriptions! This is part of the internal representation of the tools used. The model that is actually simulated is represented by this notation. In section 2.9, it is explained in detail, how hierarchy can be dealt with.

2.3 Discrete event simulation

Discrete event simulators can be classified with respect to their overall simulation approach and their time advance mechanism. Depending on the type of models to be simulated, they determine its accuracy and performance.

2.3.1 Simulation approaches

Various approaches can be used to simulate a discrete model:

- *Event driven simulation.* A component of the system is evaluated depending on the activity in the system. Activity is expressed by events, which denote value changes in the simulation model.
- *Demand driven simulation.* An alternative approach to event driven simulation is demand driven simulation [Smi87]. This approach may reduce the number of component evaluations compared to event driven simulation, because demand driven simulation propagates requests for simulation values backwards through the circuit and through time. Event driven simulation propagates simulation values forward through the circuit in response to input port events.
- *Oblivious simulation.* All components of the system are evaluated at every time point. The execution order of the components is determined statically

and can be used to compile the simulation model. Therefore, oblivious simulation is often referred to as *compiled simulation.*

- *Process (interaction) oriented simulation.* All components of the system are executed independently and communicate with each other using events. Events, which are local to a component, are isolated from events occurring in other components and can be executed in arbitrary order. Communication between components leads to synchronization.
- *Cycle–based simulation.* In a cycle–based simulator, all components are evaluated at clock boundaries and there is no notion of time. Cycle–based simulators are used because of their high simulation performance for functional validation of synchronous designs. Traditionally, they can not support all features as provided by a popular popular hardware description language like VHDL. Often, they are used in conjunction with an event driven simulator to be able to support all design methodologies and styles, for instance asynchronous designs.

Event driven simulation versus oblivious simulation

The advantage of using event driven simulation instead of oblivious simulation is its flexibility:

- an event driven simulator handles both synchronous and asynchronous circuits or systems. In [Wan90], a technique is described that is able to compile asynchronous circuits by identifying strongly connected components in the circuit description;
- an event driven simulator handles multi–delay models, whereas most oblivious simulators can only simulate zero or unit delay models [Mau92];
- an event driven simulator handles different delay models. More specific, the possibility to cancel events scheduled for future time points, allows the simulation of more complex delay models;
- an event driven simulator reduces the number of component evaluations, which is dependent on the activity in the system under simulation.

Although an event driven simulator avoids unnecessary component evaluations, the overhead of managing and processing events reduces its efficiency. In general, compiled simulation is getting more efficient compared to event driven simulation with an increasing level of activity in a system. For logic simulation, activity levels of about 1% have been reported [Bar87]. In this case, the complexity of the behavioural descriptions of the components is very low and therefore the overhead of scheduling and processing events is relatively high. At higher levels of abstraction, event driven simulation is pre-

ferred because of its flexibility. The overhead of event scheduling and processing is reduced because of the complexity of the behavioural descriptions.

Computer hardware plays also a dominant role in the trade–off between event driven versus oblivious simulation. The performance of microprocessors is getting higher and higher every year. A compiled simulator can better exploit a cache in a computer architecture than an event driven simulator. Therefore, the use of a compiled simulator is getting more attractive with the progress in processor technology.

Process oriented simulation and process modeling

Process oriented simulation provides a higher conceptual view on the system and reduces the complexity of developing simulation models for large systems: each component in the model is considered as a separate process, which interacts with its environment. In figure 2.2, a process oriented simulation model is depicted with two processes, that are communicating with each other. Process A and B run independently. Each process executes its task: such a task is a repetition of a number of subtasks.

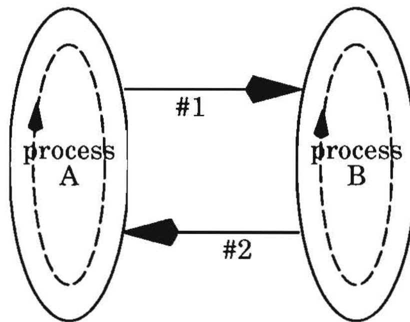


Figure 2.2. Process oriented simulation model

The behaviour of the processes is synchronized, if communication takes place between A and B. If process A is executing faster than process B, at some time point message #1 will be sent to B and A continues with its task. After a while, process A has to be suspended, because it requires a message from B. At some point in time, B requires a message from A, and detects its reception. B can immediately continue processing. After some time message #2 will be sent to A. After reception of this message, A can resume processing its own task.

Process oriented simulation can be emulated using an event driven simulator based on a scheduling technique. It is also possible to model various process

networks and communication channels, that can be simulated with an event driven model. An example of a process network with communication channels modelled in VHDL is described in [Sri92]. In [Rou89], VHDL is viewed as a process oriented simulation language. In general, a process oriented simulation model can be mapped onto an event driven simulation model, if each process cycle has at least one synchronization point, where it is suspended: this is the point, at which the process synchronizes with its outside environment. Otherwise, the process will not terminate after activation. This is not a requirement in a process oriented simulator: the simulator itself suspends and activates the processes of the model.

2.3.2 Time advance mechanisms

Another important aspect of a discrete event simulator is the time advance strategy: two strategies can be used to advance the simulation time: fixed–increment time advance or next–event time advance. Using a fixed–increment time advance strategy, events are processed at equidistant points in time. Each event is scheduled at a time point, that is an integer multiple of some time constant Δ. The length of the interval between two time points influences the accuracy and the performance of the simulator. With a next–event time advance, events can be scheduled and processed at arbitrary points in time. Especially for detailed models, this approach has to be used to guarantee accurate simulation results.

Note that a fixed–increment time advance strategy is a special case of the next–event time advance. If events are scheduled with delays, which are an integer multiple of some time interval, the performance of the simulator is increased using a fixed–increment time advance strategy: it reduces the time to insert a newly scheduled event in the event queue.

A fixed–increment time advance does not impose any restriction on the time interval that is associated with Δ: this is an issue during the development of the simulation model. A proper choice of Δ can increase the performance of the simulator. This will be explained later in this chapter.

2.4 Event driven simulation

An event driven simulation algorithm provides the flexibility required to control various types of computional models. Events also provide with an easy means for interaction between the simulator and its environment. In this thesis, the following definition is used:

DEFINITION 2.3: An *event* models a future change of a simulation variable.

An event E is described with the 6–tuple:

$$E = (\mathit{time},\ \mathit{insert},\ \mathit{type},\ \mathit{object},\ \mathit{value},\ \mathit{data}) \tag{2.3}$$

where

- *time* : the time point an event is scheduled for.
- *insert* : the insertion time of the event. It is used to order all events that are scheduled for the same time point. The implementation normally ensures a proper order of events scheduled for the same time point: this attribute is only used to describe this order. In this thesis, it is assumed that all events scheduled for the same time point are ordered using this attribute.
- *type* : the type of the event, which determines, how the event will be processed.
- *object* : the object in the model of (2.1) the event is scheduled for. This object is either a net, a module or a port.
- *value* : the value the object will be set to, when the event is processed.
- *data* : an additional data field needed for particular types of events.

2.4.1 The simulation state of the system

Let Q_m be the set of state variables, that are associated with the behaviour of a module m in (2.1). Let N be the set of *variables* associated with the nets of (2.1).Then V represents the set of all state variables in the model:

$$V = N \cup \left(\bigcup_{m \in \mathcal{M}} Q_m \right) \tag{2.4}$$

Note that the subsets, that compose V are disjoint. Let L_V be a mapping from the state variables to a value domain D:

$$L_V : V \rightarrow D \tag{2.5}$$

In case of all simulation variables are of type integer, the range of L_V would be N. Because the simulation variables may be of different types, the range of L_V is not defined explicitly.

Let E_D be the set of all events, that are defined using tuple (2.3). The *powerset* of E_D is defined as

$$P(E_D) = \{ x \mid x \subseteq E_D \} \tag{2.6}$$

Let E_q be a set of events

$$E_q = \{ e_1, e_2, ..., e_n \} \in P(E_D) \tag{2.7}$$

E_q represents the set of events that are scheduled at a particular simulation time point t_k. Thus $\forall e \in E_q : time(e) \geq t_k$. The *time* and *insert* attributes of the events in this set induce the ordering of the elements of E_q.

The simulation state of the system is then the tuple:

$$(L_V, E_q) \in (V \rightarrow D) \times P(E_D) \tag{2.8}$$

At the start of a simulation run, the state is initialized by executing the *init* functions of all behaviours and by setting the values of all nets. Note that this function merely sets the values of all state variables. Then, the *exec* functions of all modules is called: this schedules the events, which will be processed in the first simulation cycle.

Initialization results in the initial simulation state (L_0, E_0).

After initialization, the simulation state is updated each simulation cycle. Let (L_k, E_k) be the simulation state after k simulation cycles. First, the events are selected from E_k, that must be processed in the current cycle. This is a partitioning of E_k into two disjoint sets E_{sel} and E_{next}

$$E_{sel} = \{ e \in E_k \mid \forall e' \in E_k : (time(e) \leq time(e')) \} \tag{2.9}$$

and

$$E_{next} = E_k \setminus E_{sel} \tag{2.10}$$

Then the events of E_{sel} are processed in order – the *insert* attribute is used to determine this order. This results in the generation of new events and a modification of a subset of V. The new simulation state is then (L_{k+1}, E_{k+1}). E_{k+1} contains the union of E_{next} and the newly created events.

Note that the simulation time is not explicitly specified in this equation as time is changing implicitly as events are processed by the simulator. To ac-

count for the fact, that the value of the simulation clock is essential for correct processing of events[1], the simulation time is explicitly represented in the simulation state:

$$(T_{sim}, L_V, E_q) \in \mathbb{N} \times (V \rightarrow D) \times P(E_D) \tag{2.11}$$

where it is assumed that the simulation time is represented by a non–negative integer.

It is also possible to describe the overall system state using the system states of each submodel. This is interesting to describe parallel discrete event simulators or mixed–level simulators.

The simulation state of a submodel i is

$$(T_{sim}, L_V, E_q)_i = (T_{sim,i}, L_{V,i}, E_{q,i})$$

The overall simulation state is then (T_{sim}, L_V, E_q) with

- $T_{sim} = \min_i T_{sim,i}$
- $L_V = \bigcup_i L_{V,i}$
- $E_q = \bigcup_i E_{q,i}$

2.4.2 Event types

Event types are used to model different delays in the simulator and to orchestrate other activities during simulation. The following event types are available:

Nominal delay events

Nominal delay events are used to model a simple pure transport delay in the simulator. The following subtypes can be distinguished:

- A **normal** event will cause the net's fanout list to be processed, if and only if the new net value (as specified in the event) differs from the current net value.
- A **refrain** event will never cause the net's fanout list to be processed.
- A **trigger** event will always cause the net's fanout list to be processed.

The nominal delay model is associated with these event types, i.e. if an event is scheduled at time point t with a nominal delay of i time units, the effect of

1. In a distributed simulation environment for instance, it is essential that all events are processed in a non–decreasing order.

this event will always occur after i time units. In figure 2.3, a model of the nominal delay model is depicted.

The semantics of the various types of nominal delay events differ in the way the modules of the net's fanout list are processed. The use of nominal delay events will never result in the invalidation of *other* events already scheduled for a particular net. As a result, if the ports driving the net are using nominal delay models, it is not necessary to maintain an ordered list of events scheduled for this net. This is explained in more detail in section 2.5.

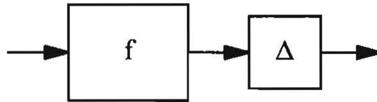


Figure 2.3. Model of a function f with a nominal delay of Δ.

Note that in literature on logic simulation, e.g. [Bre76], the nominal delay model is often referred to as a *transport* or *pure* delay model, which differs semantically from the transport delay model as used in VHDL. The nominal delay model is too simple to be used for modelling non–fixed delay characteristics of hardware components, but it is very efficient to be used for components with fixed delay characteristics.

The nominal delay event type can also be applied for modelling networks, in which data is transported at different speeds. For example, in a computer network packets may travel along different routes from the source host to the destination host and they may arrive in a different order. This is modelled by using a continuously changing delay for all nodes in the network: this may reverse the order in which packets are sent from one node to another node.

The transport and inertial event types

The nomimal event type is less suited for modelling the delay characteristics of hardware components and transmission line delays. Therefore, both the *transport* and the *inertial* delay models as defined in VHDL are supported within the simulator. In [Gho89], the limitations of the transport delay model are described and a preemptive scheduling mechanism is described to model inertial delays accurately. Note that the use of the word transport has a different meaning here as compared to the pure delay model. For this work, the transport and the inertial event type are defined to support the transport and inertial delay models respectively.

The semantics of these event types are described by operations on a set of events. Let E_s be a set containing all events scheduled for a net $s \in \mathcal{N}$ at a particular simulation time point t_k:

$$E_s = \{ e_1 , e_2 , \ldots , e_n \} , n \geq 0 \quad (2.12)$$

If $n = 0$, E_s is the empty set and thus no events are scheduled for this net. New events are added and deleted from this set during simulation. If the current simulation time equals t_k, the following property holds for all events in E_s:

$$\forall e \in E_s : time(e) \geq t_k \quad (2.13)$$

Note that the events of this set can be sorted using their time and insert attributes. Let e_{new} be a new event scheduled for this net.

After scheduling either a transport or inertial event, this set is updated using

$$E_{s,new} = (E_s \setminus E_{D,type}) \cup \{ e_{new} \} \quad (2.14)$$

with $type \in \{ transport, inertial \}$ and $E_{D,type}$ the set of events that is removed. Scheduling a transport delay results in

$$E_{D,transport} = \{ e \in E_s \mid time(e) \geq time(e_{new}) \} \quad (2.15)$$

Thus, scheduling a new transport event type for a net results in the removal of all events that are already scheduled for that net at time points equal to or greater than the time point the new event is scheduled for.

Scheduling an inertial delay results in

$$E_{D,inertial} = E_{D,prev} \cup E_{D,transport} \quad (2.16)$$

with $E_{D,prev} = \{ e \in E_s \mid time(e) < time(e_{new}) \wedge value(e) \neq value(e_{new}) \}$

Thus, scheduling a new inertial event for a net results in the removal of **all** events that are already scheduled for that net **except** if

- the event is scheduled at a time point that is smaller than the time point the new event is scheduled for **and**,
- the value of the event is equal to the value of the newly scheduled event.

This implies that an event is only processed, iff no other event with a different value is scheduled for the same net in the meantime. With an inertial delay model, pulses are not propagated, if the width of a pulse is shorter than the delay specified. Note that the actual delay value may vary for different conditions.

In [Ram92], a modelling technique for timing effects is presented as well as more complex delay functions like the edge triggered inertial delay function and the smoothing transport delay function. For many applications, these very detailed timing models are not required.

The cancelled event

An event of type **cancelled** is used to indicate that an event does not require further processing in subsequent phases of the simulation algorithm. Different phases in the simulation algorithm can convert an event of most other types into the type cancelled. Normally, this event type is only used internally by the simulator. Currently, cancellation is used in the preprocessing phase of the simulation algorithm for the nominal delay event types to prevent further processing.

The simulation control and synchronization event types

Several event types are defined to control and to synchronize the simulator:

- An **interrupt** event suspends the simulation run unconditionally. Simulation can be resumed afterwards without any restriction. Interrupt events are scheduled *out–of–band*, i.e. they are added in a special queue, which is processed before all other events are processed.
- An event of type **synchronize** is used to synchronize the internal time clock with external processes. This is for instance used if an external simulator is connected to the event handler using inter–process communication. Events of this type are processed in a different way than other events. More information on this can be found in chapter 3.
- A **timeout** event is used to notify that some condition has not occurred in time. Normally, the occurrence of this condition cancels the timeout event.

The action event

This type of event allows to extend the functionality of the simulation algorithm. It can be used to execute specific tasks , which are related to the simulation experiment, and, to extend the simulator with new domains or abstraction levels. The following data can be specified for this event type: the function that has to be called during processing, the time after which the event has to be called and an additional data field (which is often called a *client data* field).

In principle, all simulation tasks could be executed using the action event. However, this would reduce the performance of the simulator significantly, since the processing of each event would require an additional function call.

2.4.3 Examples using various delay models

Different delay models are associated with the various event types described in the previous section. The nominal delay model is the most simple one and is less expensive with respect to computational costs than the transport and inertial delay models.

Two very simple examples clearly illustrate the differences between nominal, transport and inertial delay. In the first example, a simple inverter is used having a constant delay of 8 ns. In figure 2.4, the output signals of inverters using different delay models are depicted for an arbitrary input signal. For a constant delay, the nominal and transport delay model exhibit identical behaviour. The inertial delay model has a different behaviour: it consumes all pulses having a length shorter than the inertial delay specified.

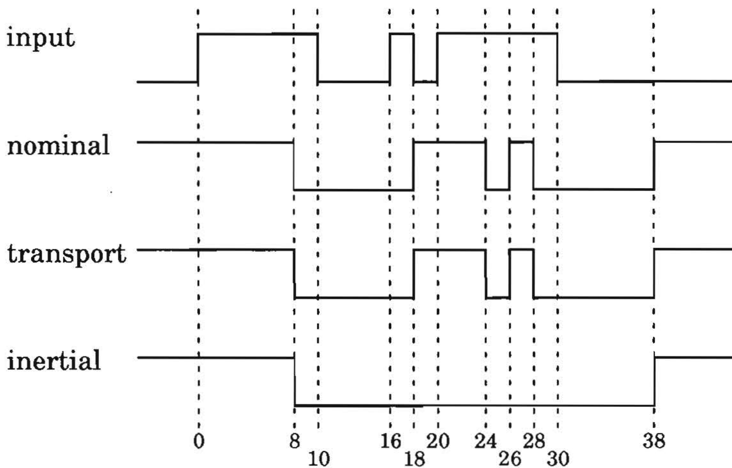


Figure 2.4. Example with nominal, transport and inertial delay

The second example better illustrates the differences between nominal, transport and inertial delay model. In this case, a simple inverter is used with the following characteristics:

- if the input raises to 1, the output will fall to 0 after 8 ns;
- if the input falls to 0, the output will raise to 1 after 12 ns;

In figure 2.5, the input signal and the output signals for the various delay models are depicted. The simple nominal delay model differs considerably from the transport and inertial delay models and is not really suited for modelling different delay conditions (e.g. a difference in rise and fall delay). For this particular example, the difference in rise and fall delay causes the output

to stay high, although the input stays high after $t = 20$. This is clearly the wrong behaviour, since this example models an inverter. The problem is caused by the reversed order in which the events of the pulse between $t = 18$ and $t = 20$ are processed. This annihilates the effect of the last input change.

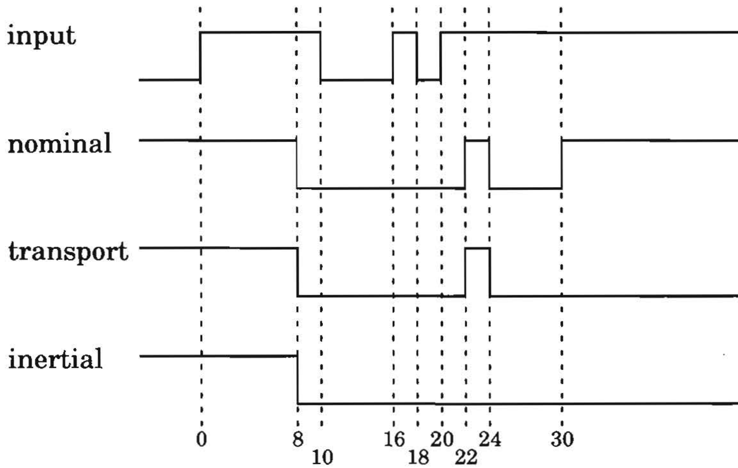


Figure 2.5. Example with nominal, transport and inertial delay

2.4.4 Delta delay events

During simulation, events are normally generated and scheduled with a delay value $t_\Delta > 0$. It is also possible to schedule events with a delay value of 0. Such a delay is called a delta delay. A delta delay is an infinitesemal delay, that does not forward the simulation time. The number of delta delay steps that may lie between two simulation time points is not determined. A complete simulation run could end at simulation time 0 after iterating through many delta time delay steps. In literature, the time points of the simulation clock and the delta time points in between are sometimes referred to as *macro time* and *micro time* [Bel93].

The delta delay is used in a simulator to be able to handle concurrency using a sequential algorithm: events that are scheduled for the same time point are handled sequentially in an event driven simulator. The processing order may not influence the assignment of values to variables in the model: therefore the assignment itself is delayed for an infinitesimal delay allowing that the current values of all variables will be used before any assignment is being done. Once, all assignments are scheduled using the proper values of the variables, the new values are assigned to the variables. This technique allows to swap the values of two variables without a temporary variable.

2.5 Event processing and management

During the simulation of a model, events are generated as various components of the model are executed. These events need to be stored in a data structure and are retrieved by the simulator at the proper time point. These data structures need to be optimized with respect to event insertion, deletion and traversal. Which data structure is more suitable, depends on the simulation algorithm used, the time advance strategy and the delay models of the components.

Various data structures can be used to implement an event queue. The most simple implementation is a linked list. The time complexity of inserting and deleting events is $\mathcal{O}(N)$. Other implementations of an event queue are binary search trees, heaps and Fibonacci heaps with better time complexity for the various operations.

There are also more specialized data structures, that are often used to implement event queues. The time of each event is used to store the event at the proper place in the event queue. An example of this is a linear indexed list [Kle84]: this list contains a number of headers that point to a list containing events for a particular time point or time interval. Depending on the organization and the resolution of the time, the time complexity of inserting an event is linear in the number of time point headers. This approach can be used for both time advance strategies.

If time can be represented by a natural number, which is the case if a fixed–increment time advance is used, then the use of a timewheel is an efficient way to store and retrieve events [Ulr69]. Each slot in this timewheel is called an event list header (ELH), which gives access to all events scheduled for a particular time point. Each ELH points to the first and last event scheduled for this time point using a *start* and *end* pointer (see figure 2.6). Besides regular events, the ELH may also point to other event lists, like the list storing out of band events.

Out of band events are used to control the simulation algorithm and to perform dedicated tasks. They could be coded in the simulation algorithm as well, but this would result in checking their occurrence with each pass of the simulation loop. As an example, the interrupt event is always scheduled out of band. Scheduling the interrupt event as a regular event, would require an additional preprocessing step to detect its occurrence. Alternatively, restoring the event queue after the detection of an interrupt requires to store their original and new types.

Insertion of an event in the timewheel is accessing the *end* pointer of the appropriate event list. This operation takes $\mathcal{O}(1)$ (constant) time. Deleting spe-

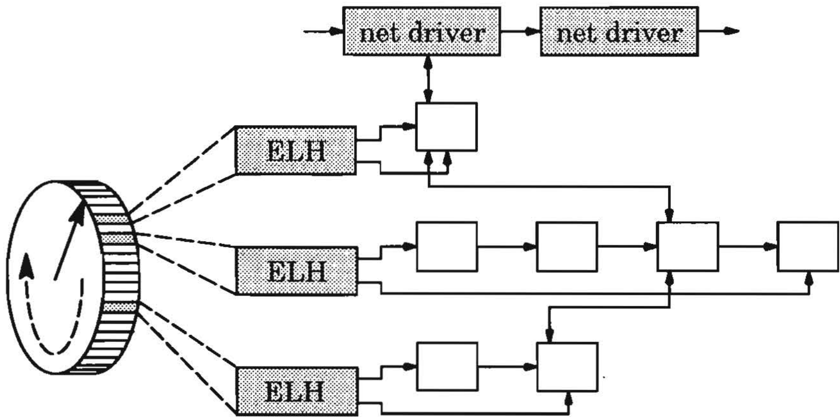


Figure 2.6. Data structure to store events

cific events from the timewheel is a time–consuming operation: deleting all events scheduled for a particular net implies that the whole data structure has to be searched. Therefore, all events scheduled for a particular net may be linked into a doubly linked list. Each net driver, as the head of this list is called, gives easy access to an ordered list of events (with respect to time) scheduled for that net. Only specific nets require this additional data structure for easy removal of events: this strongly depends on the delay models used for the ports, that drive the net.

Events, which are scheduled with a delay greater than the size of the timewheel, require an additional data structure for storage. These events are put on an *overflow list,* which is a doubly linked list for easy insertion and deletion of events. Insertion or deletion of events in the overflow list, is more expensive than inserting it in the time–wheel itself. Therefore, the events stored in the overflow list are not transferred to the time–wheel each time all events for a particular time point are processed, but periodically after processing all entries once (a complete turn of the wheel).

Algorithm 2.1 shows a simple version of a simulation algorithm using a fixed–time increment strategy. After (re–)initialization of the internal simulation model and the simulator, a loop is repeatedly executed processing events generated for a particular time point. Another loop is embedded into this loop to be able to process delta delay events.

Algorithm 2.1: Event driven simulation algorithm

```
1.  init_update_simulation_state (); /* initialisation of simulator */
2.  do { /* handle all events for current time point */
3.      transfer_overflow_list_events ();
4.      do { /* handle all events for current delta */
5.          process_oob_events (); /* out-of-band events */
6.          if (continue_processing) {
7.              unlink_event_list ();
8.              select_active_modules ();
9.              process_active_modules ();
10.         }
11.         handle_system_interface ();
12.     } while ('delta events scheduled');
13.     t_sim := t_sim + 1;     /* fixed-increment of one time unit */
14. } while ('events scheduled' or 'stop condition' = true);
```

In algorithm 2.2, the event processing routines for the nominal delay types are shown in more detail. In algorithm 2.1, the preprocessing is handled in line 8, and the processing in line 9.

Algorithm 2.2: Event processing for nominal delay types

```
1.  const wheelsize = 1024; /* size is arbitrary here */
2.  TimeWheel wheel [wheelsize];
3.  Queue evlist;
4.  Event event;
5.  ....
6.  evlist = wheel [t_sim mod wheelsize];
7.  wheel [t_sim mod wheelsize] = ∅;
8.  ....
9.  /* preprocessing phase */
10. foreach (event ∈ evlist)
11.     switch type (event) {
12.         normal:
13.             if ( value(event) == value(net) )
14.             then {
15.                 type(event) = cancelled;
16.                 break;
17.             }
18.             /* fall through */
```

```
19.         refrain, trigger:
20.             value(net) = value(event);
21.         default: /* error: unknown type of event */
22.     }
23. /* processing phase */
24. foreach (event ∈ evlist )
25.     switch type (event) {
26.         normal, trigger:
27.             process_fanout_list (net(event));
28.             break;
29.         default: /* error: unknown type of event */
30.     }
31. ...
```

This section described the kernel of the event driven simulator. The next section will describe how this simulator is embedded into ESCAPE and tightly integrated with the other parts of the simulation environment.

2.6 Interaction between simulator and editor

System validation not only requires much computational time, but also much time spent by the designer to iterate through the design cycle. Developments for simulators have mostly been focussed on the first aspect: building more powerful simulators for various abstraction levels. The second aspect, the reduction of the overall design time, is also very important for fast simulation. Examples of situations, where it helps to reduce design time, are:

- The prototyping of an initial specification of a system. Such a prototype is executable in the sense that it can be simulated. It serves as a master reference throughout the rest of the design process.
- Debugging and re–simulating a system description. Each iteration involves editing the description, compiling the underlying net list, simulating the system and analysing the results.

In [Hey88], the architecture of a highly integrated simulation system is described. The often used text–based interfaces between CAD tools are replaced by a persistent programming technique, that handles communication of data between tools. The tools have access to a common database. Files in the database are viewed as an extension of the memory, that is dynamically allocated by a tool. This mechanism is known as a persistent heap. The implementation of this heap uses the file system to realize the communication between vari-

ous CAD tools. The efficiency and performance of this technique is quite low, if fast interaction between two or more CAD tools is required.

In ESCAPE, the integration between the various tools, the graphics editor and the simulator to be more specific, is achieved by integrating them into a single tool: the editor and simulator run in the same address space and share all data kept in memory. However, the requirements of data structures to manipulate graphics data efficiently are completely different from the requirements of data structures to simulate a network of components with an event driven simulation algorithm. Therefore, both a graphics and a network data structure are used, in which design objects may be stored as two different data objects.

In figure 2.7, the relations between the data objects in the network data structure and the graphics data structure are depicted. In the graphics data structure, some objects are stored hierarchically in order to reduce the number of objects that have to be queried during a search: symbols and ports of an instance are stored as data of an instance.

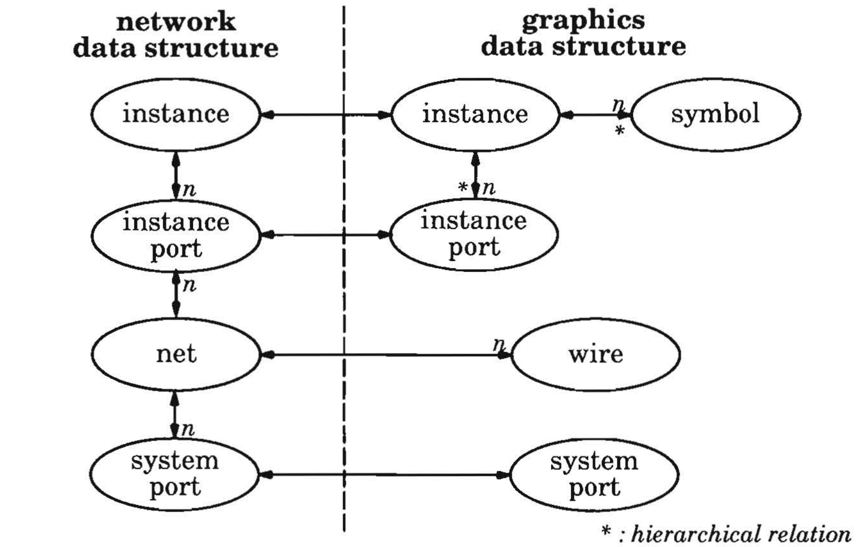


Figure 2.7. Relations between objects in network view

The graphics data structure stores all data related to the positions, sizes and relations of the graphical objects. The following operations are the most important ones:

- an insert of an object,

- a deletion of an object,
- a point query,
- a region query.

Performance and memory requirements are the most important aspects of a data structure storing geometrical data. In literature, various data structures have been described [Ros85]. In ESCAPE, three different data structures have been implemented and can be selected by the designer:

- a linked list;
- a multiple storage radix hash tree [Fon87];
- a storage minimizing automatic level sifting database for two–dimensional object location [She90].

The use of the *region query data structures* is often more useful in batch tools than in interactive editors. This is in particular true for schematics because the density of the objects is less than the density of objects in a layout. The performance of the linked list is sufficient for most applications. The operations applied to very large schematics (more than 500000 objects) can be executed without observable delay to the designer. The memory requirements of a linked list are also much lower than of the other two data structures. Therefore, for most examples the linked list has been used as the graphics data structure.

Algorithm 2.3 shows a simple example of an incremental update of the data structures after adding a new wire to the network view. First, a graphics object of type wire is created. This object is stored in the graphics data structure. In line 3, the graphics data structure is searched for objects that touch or overlap the bounding box of this wire. The objects, that are found with this query, are stored in a buffer.

The objects in the buffer are examined one by one. The relative position with respect to the wire and the type of the object determine how the network data structure is affected. This may lead to one or more of the following actions in the network data structure:

- the creation of a new net;
- the joining of two or more nets;
- the addition of an input or output terminal of a module to a net;
- the addition of a primary input or output to a net.

Combinations of the actions above may occur as well.

Each command supported by the editor updates both graphics and network data structures in a similar way. Although the overall structure of algorithm 2.3 seems simple, it is a quite complicated task to keep the data structures up–to–date and consistent for various editing commands or combinations of those commands. Despite this complexity, even complex operations, that manipulate hundreds of objects with a single command, allow incremental updates of all data structures without observable delays.

Algorithm 2.3: Example of updating data structures incrementally

```
1.  /* add wire */
2.  wire = CreateWire ( ..... );
3.  AddToGraphicsDb ( db, wire );
4.  /* use graphics data structure to update network data structure */
5.  RegionSearchGraphicsDb ( db, resbuf, wire->bounding_box );
6.  while ( ! objbuf_empty ( resbuf ) ) {
7.      object = get_from_objbuf ( resbuf );
8.      switch ( object->type ) {
9.          case wire:
10.             .....
11.         case instance:
12.             .....
13.         case primary_port:
14.             .....
15.     }
16. }
```

The incremental updates on the data structures required by the simulator allow fast interaction between the editors and the simulator integrated in the system. The following interaction models are implemented:

- *Tool context switching.* Using this interaction model, the simulator has to be halted before any modification of the design is made. After editing and debugging the design, the simulator can restart immediately without re–compilation of the underlying net list. This interaction model closely resembles the traditional way of editing and simulating a design. However, the tight integration of editor and simulator in a single tool strongly reduces the overall design time.
- *Run–time updating.* Using this interaction model, the design data, that is accessed by the simulator, can be modified while the simulator is running. Such modifications are applied to the design data after the processing of the events scheduled for a particular time point or, to be more specific, a delta

time point. In algorithm 2.1, this interaction is handled in line 11. After updating the design data, the simulator continues at its current state. Note that this state can be changed as a side effect: as an example consider the deletion of a net, which causes all events scheduled for that net to be deleted as well.

Run–time updating is a more interactive approach than tool context switching. It may be very useful, if a designer wants to change the behaviour of a module to explore the design space. For instance, a designer could change either the delay model or the delay values of a specific model. It is also possible to induce a fault in the model and to investigate its influence on the behaviour of the model. Run–time updating is a technique that adds some overhead during the editing and simulation of a system, since more consistency checks have to be performed and more data needs to be updated. Usually, run–time updating is allowed for modifying the behaviour of a primitive module, whereas tool context switching is used for modifying the structure of a composite module.

Using tool context switching, the design cycle time may be reduced even further by applying an incremental simulation algorithm [Cho89]. An incremental simulation algorithm only simulates those parts of a design that have been modified since a previous simulation run. This means that the simulation results of a previous simulation run are incrementally updated for those parts of the design that have been modified: the computational costs are proportional to the size of the modifications and not to the overall size of the system. This is possible by storing simulation state data from the previous simulation run as well as modifications during an edit session.

Various implementations are possible of an incremental simulation algorithm:

- an *incremental–in–space* algorithm is described in [Hwa87]. Components that are modified are marked by sending modification tokens to the simulator [Hwa88]. These components may affect other components in their fan-out trees as well: these nodes are modified by a traversal algorithm prior to simulation. After marking all possibly affected components, the simulator only simulates the marked components using the event histories of the components at the boundary of the affected area as input values.
- An *incremental–in–time* algorithm is described in [Cho88]. This algorithm only simulates those components of the system, whose input values or internal states are different from the values of the previous simulation run. Components, whose input values or internal states differ from the previous run, are called active and the other components are called inactive. Simula-

tion starts by scheduling all active components. Active components will repeatedly be scheduled until they become inactive.

In [Cho88], it is stated that the incremental–in–space algorithm is more efficient, if the components of the system have many internal state variables. The incremental–in–time algorithm is preferred for systems with feedback loops and busses and without components with many internal state variables. In [Jon92], a different incremental simulation technique is used in an zero/integer delay switch–level simulator. Like ESCAPE, the simulator is integrated with a schematic capture in a single tool.

At higher levels of abstraction, the application of an incremental simulation algorithm is less useful, since both time–shifts and the use of more abstract data types require massive re–simulation. The influence of time shifts on re–simulation is depicted in figure 2.8: it shows two signals, of which signal B is delayed in time with respect to signal A. It is important to note that signal A is the result of the first simulation run for a particular net and signal B is the result of the second simulation run for the same net. For instance, the delay of the component driving the net has been increased by ΔT. In an incremental simulator, this time–shift would have resulted in many events, that mark time points for re–simulation.

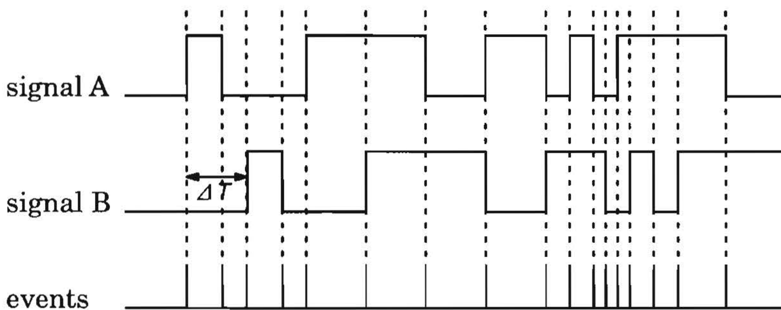


Figure 2.8. Events due to time shift between signals A and B

It is a topic for future research to investigate, in which cases it is more useful to re–simulate the system completely using a conventional simulation algorithm and in which cases an incremental simulation algorithm is best applied to re–simulate the system. It might even be possible to use a combination of both strategies for different parts of the system. A disadvantage of using an incremental simulation algorithm is the storage of a huge amount of simula-

tion data from the previous simulation run. This may limit the use of such a technique because of memory requirements.

2.7 Animation

Most simulators store their results in large files, which can be viewed as large listings or as signal diagrams using a signal viewer. Alternatively, design systems often provide back annotation onto the schematic as an additional means to examine simulation results. The increasing size and complexity of systems makes the analysis of simulation results a large and complicated task. Often, system designers create a high level specification of the system. This specification serves as a basis for other designers to further implement the system into silicon. A clear understanding of the system's functionality and of the interaction between the various subsystems reduces misunderstandings about the implementation.

In general, run–time animation is a very useful feature in a simulation environment, since

- it helps other designers to understand complex system behaviour more easily and more intuitively;
- errors are detected and corrected more easily; therefore, it facilitates debugging;
- communication between the increasingly large number of independent modules becomes more important. Visualization techniques help understand this communication behaviour.

In ESCAPE, animated simulation is supported in the LISP–like HDL. Special statements allow highlighting of nets and modules on specific conditions. This is used to show how the system operates. Additional modules can easily be added to the simulation model: such modules are used to model the environment of the system. An example is the railroad example presented in chapter 4.

Although animation greatly helps to understand the operation of the system, it has a disadvantage: it greatly reduces the performance of the simulation. Especially, in a tightly integrated system like ESCAPE, each animation request decreases simulation performance. Therefore, animation should only be used for high–level models, where performance is not a dominating issue. A designer can control the animation interactively during simulation and thus make a trade–off between performance and the level of animation.

Simulation results can also be presented in special purpose view windows. An example of such a window is a frequency analyzer display, which shows the

spectrum of a specific signal. Such a window is coupled to a signal by connecting the appropriate module to that signal. Such functionality is found in some commercially available DSP application development environments. However, most of these display windows are provided by the vendor and are only customizable through some parameters. In ESCAPE, new display windows are easily added to the system using the foreign language interface (see section 3.2.1).

2.8 Discrete event monitors

The validation of a design is an important activity in the design process, because undetected errors may cause expensive redesigns and delay the design project. Because of the increasing level of complexity, it is becoming more and more time–consuming for a designer to obtain an acceptable level of confidence in the correctness of the design. This problem can be alleviated by a tighter integration of evaluation and analysis methods into the design environment [Bus89, Aug90].

A complex and time-consuming part of the validation process is the analysis of the simulation results and the detection of erroneous behaviour. Therefore the ability to check if the simulated behaviour is consistent with a given set of requirements, would improve the efficiency of the validation process, especially if this is done during a simulation run. This involves the two subproblems of specifying how the design is supposed to behave, and comparing the simulated behaviour with the specified requirements. In this section the latter problem is discussed in the context of event driven simulation. This covers many simulators used today at various abstractions levels, including simulators for popular hardware description languages like VHDL and Verilog.

Most hardware description languages are primarily aimed at specifying a design by describing a model that implements the desired behaviour at some level of abstraction. Only limited support is provided to express the properties the model should comply with. For example, VHDL provides the *assert* construct to check constraints on the state of a model at specific points in time. More complex conditions involving time can be included by extending the description of the system with additional modules. It may even be necessary to extend the interface of existing modules in the design to make parts of the internal state observable from the outside. These modifications usually lead to less comprehensible models and may even introduce new errors in the design.

In [Aug90], the VAL language is proposed as an annotation language for VHDL. It extends the language with a small set of new constructs for abstract specification. For example, it is possible to express complex timing constraints on signals. A preprocessor is used to translate an annotated de-

scription to a regular VHDL description which automatically checks if the simulated behaviour complies with the abstract specifications. In [Gen92], the language is extended with so–called event pattern mappings to reduce the complexity of the simulation results. It is based on recursively recognizing and naming patterns of events. This enables the designer to view the simulation results on a higher level of abstraction, and therefore effectively reduces the amount of information that has to be inspected. The presented implementation analyses the event traces of the simulation in a post–processing step.

There exist numerous methods to specify requirements imposed on a design. For example, timing diagrams are frequently used to specify both qualitative and quantitative timing constraints. In [Bor92], a formal version of these diagrams is proposed, which also covers more complex aspects such as timing constraints on conditional and iterative event sequences. It would be very useful if such constraints could be combined with a design specification, independent of the languages used to describe this specification.

In a discrete event simulator, the behaviour of the simulation model is characterized by the event trace that is generated during a simulation run. To analyse this trace, a general mechanism is required to inspect the events scheduled for specific state variables and nets in the model. The main requirements for such a mechanism are that it does not influence the simulation results and that it has an acceptable effect on the performance of the simulator. Furthermore, it must be applicable to different simulation algorithms, models of computation and abstraction levels. Therefore, the concept of a discrete event monitor is introduced.

DEFINITION 2.4: A *discrete event monitor* (DEM) is an object which observes the events generated and processed for specific elements in the simulation model.

Although DEMs can be used as a post–processing step of a simulation run, they are intended to be embedded in the discrete event simulator itself. In an incremental simulation environment [Cho88, Jon92], this concept especially contributes to efficient validation of system behaviour, as it can be used to observe the events that model the differences with respect to the previous simulation run.

A DEM specification consists of three parts: an interface part, a declaration part and the part that defines the functions supported by the DEM (see figure 2.9).

```
dem ( net ..., net ..., dem ...) /* interface */
{
    /* declarations */
    state ...;

    /* internal functions */
    init { ... }
    exec { ... }
    exit { ... }
    report { ... }
}
```

Figure 2.9. Global structure of a DEM specification.

In the interface part, the variables which relate to the simulation model and other DEMs are specified. During instantiation of the DEM, each variable in the interface part is attached to an object in the simulation model or another instantiation of a DEM. This mapping is described separately to make the specifications reusable. Currently, the following types of variables are allowed in the interface part:

- A net variable is used to access the events which are scheduled for the net to which this variable is attached.
- A module variable is used to access the internal variables (for instance state variables) of the module it is attached to.
- A DEM variable is the port, on which higher level events can be scheduled by a DEM. Such an event will immediately activate the monitors that are attached to this variable. Note that a DEM can only schedule events that activate other DEMs, but it is not allowed to schedule events that influence the simulation model itself.

In figure 2.10, the relationship between the objects in the simulation model and the DEMs are depicted.

In the declaration part, state variables may be declared: they are used to store the state of the monitor and may have different (abstract) types. Each instantiation of a monitor allocates its own state variables, which can be accessed by the DEM's internal functions. Currently, the following definitions of internal functions can be specified:

- The *init* function, which is called before the first event is processed. This function can be used to initialize internal DEM variables and to allocate

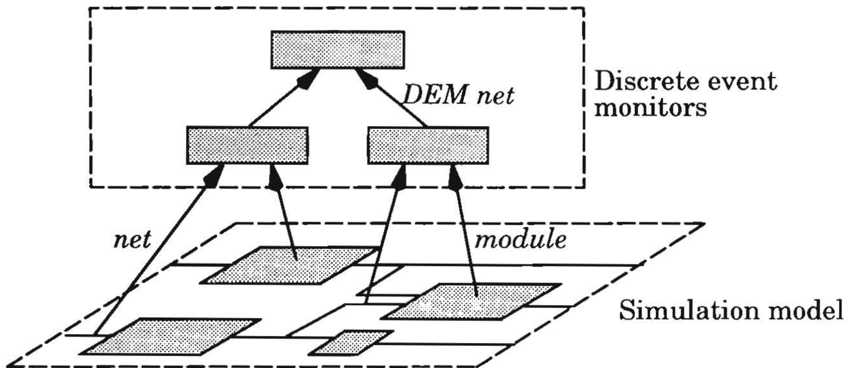


Figure 2.10. Relations between the simulation model and discrete event monitors

additional resources required by the monitor. Note that state variables are allocated by the *create* function during instantiation of the DEM.

- The *exec* function, which is called if an input variable of the monitor receives an event. In this function, new events can be scheduled, which activate other monitors at higher levels. In addition, special event types can be scheduled, which are handled by the simulator to detect time outs and other exceptions. Note that this does not influence the simulation model itself, but that this events are used to present this information to the designer.
- The *exit* function, which is called after the simulation run has finished. It is used to evaluate the final state of the monitor and to clean up resources.
- The *report* function of a monitor is used to evaluate the current state of a monitor and to collect statistical information of a simulation run. It may be called any time by the user but could also be embedded into the simulation algorithm.

DEMs can be specified using a conventional programming language. The DEM specification has its own syntax that is translated in C language constructs by a preprocessor. After processing, it can be compiled and linked with the simulator. In case of ESCAPE, this preprocessing is handled internally and the result may be loaded dynamically after compilation.

In algorithm 2.4, the operation of DEMs within a discrete event algorithm is shown. The algorithm presented here is a slightly modified version of algorithm 2.1. First, the model is initialized by initializing all nets and modules (steps 1 and 2). The behavioural descriptions of all modules are executed to activate the modules. This results in the generation of events, which will be

processed by the simulation loop. The last step of the initialization phase is the activation of all DEMs, just before the processing of events starts (step 3).

Algorithm 2.4: Simulation algorithm with DEM interface

```
1.  foreach ( n ∈ N) init_net ( n );
2.  foreach ( m ∈ M ) init_module ( m );
3.  foreach ( d ∈ DEM )  init_dem ( d );
4.  repeat {
5.      foreach ( event ∈ queue ( Δ ) ) preprocess_event ( event );
6.      foreach ( event ∈ queue ( Δ ) ) {
7.          execute_dems ( event );
8.          foreach ( m ∈ fanout ( net ( event ))) execute_module ( m );
        }
9.      Δ = Δ + 1;
    } until event queue empty;
10. foreach ( d ∈ DEM )  exit_dem ( d );
```

A DEM can pass information to another DEM by scheduling higher level events for that DEM. This raises the level of abstraction and therefore reduces the number of events that has to be examined by the designer. If an error is detected at the highest abstraction level, it can be traced back to its source in the simulation model (see figure 2.11). This is achieved by repeatedly annotating the source of the event into the event itself. Interacting DEMs can be represented as a directed acyclic graph. Usually, this graph is a tree. If a DEM schedules an event for another monitor, this monitor is immediately activitated. No additional delay model is associated with these events, since this could be the source of more misunderstandings about the interpretation of the simulation results.

2.8.1 Discrete event automata

DEMs are specified using a convential programming language. Although this does not impose any restriction on describing the DEM, the user needs to know some implementation details. Therefore, a specification language for DEMs has been developed. This language also serves as an intermediate language to facilitate the integration of other specification methods like timing diagrams.

The underlying model of this specification language is a deterministic automaton in which transitions are labelled with conditions and actions. The condi-

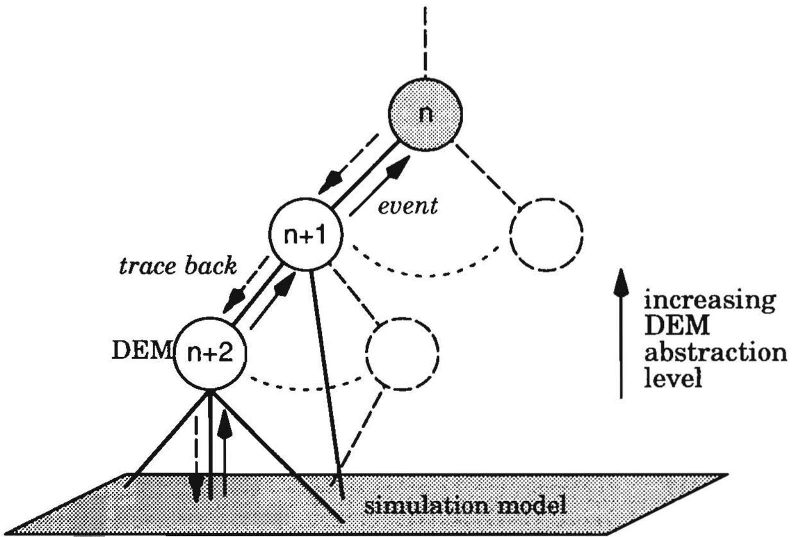


Figure 2.11. Event propagation and trace back to simulation model

tions define the events inducing a transition. The conditions can include the values of state variables and the attributes of an event. The actions describe the effects of a transition, which may include the generation of new events for other discrete event monitors. This model is expressive enough for describing a large class of constraints, and can be executed efficiently during a simulation run. A description in this specification language is called a *discrete event automaton* (DEA).

In figure 2.13, an example is given of a DEA which validates the correctness of a dual rail encoded interface. The states of this automaton are depicted in figure 2.12. This example is used to illustrate the main aspects of the language. The header specifies the name of the DEA and declares its parameters. In this particular case, the parameters are the three nets on which the communication takes place. The body of the description consists of two sections, which respectively declare the state variables and specify the behaviour of the automaton. Optionally, a third section can be added to customize the way the information about the state is reported to the user.

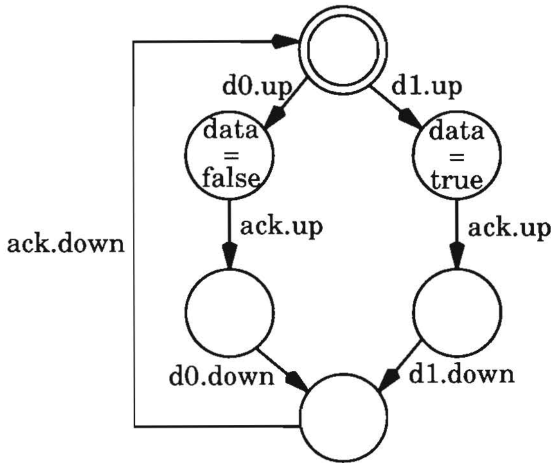


Figure 2.12. Discrete event automaton of dual rail encoded interface

```
monitor dual_rail_decoder (net d1, net d0, net ack)
 /*  This monitor checks the correctness of a dual rail
     encoded interface and translates the data transfers
      to higher level events. */
{
   variables {
     int data;
   }

   process {
     while (true) {
       select { /* valid data */
         d1.up : data = true;
         d0.up : data = false;
       };
       event (data); /* create high-level event */
       select {ack.up}; /* acknowledge */
       if (data == true) /* invalidation of data */
         select {d1.down};
       else
         select {d0.down};
       select {ack.down}; /* invalidation of acknowledge */
     }
   }
}
```

Figure 2.13. An example of a discrete event automaton.

The *select* statement is the main construct to define the behaviour of a DEA. This statement specifies a single state of the automaton and the transitions leaving that state. Each transition is defined by a set of event labels and the corresponding actions. An event label specifies the object for which the event must be scheduled, the value of the event and optionally the type of the event. By default, a transition sets the automaton to the next sequential state. Control structures are provided to support the description of conditional and repeating patterns of behaviour. As a result, many requirements can be described very concisely by a DEA. Every state can be given a name, although this is not shown in the example.

The *event* statement can be used to generate events for other DEAs. In the example, every successful data transfer results in the generation of a new event, that can be processed by a DEM at a higher level, if no explicit destination is given. This is normally the parent in the hierarchy of DEMs (as in this example). It also shows how DEAs can be used to implement event pattern mappings [Gen92]. These mappings are very useful for comparing a detailed simulation model with a specification at a higher level of abstraction.

2.8.2 Examples and results

DEMs have been used to validate event traces of various examples. The primitive components of all examples are modelled using the LISP-like hardware description language supported by ESCAPE. The following examples are presented in table 2.1:

- An implementation of the asynchronous two place ripple buffer presented in section 1.3. For this example, the absence of unexpected request and acknowledgement events has been validated.
- An abstract model of a railroad. The model is composed of two abstraction layers: one layer models the environment, in particular the transportation of the trains, and the other one models the control aspects of the system. The handshake protocol used to move the trains has been validated. This example is presented in chapter 4.
- An architectural model for a Kulisch inner product chip . The timing of the adder stations in the model has been validated. This example is also presented in chapter 4.

These examples differ considerably in size and complexity, both in the structural and the behavioural respect. The complexity of the DEMs is comparable to that of the DEA shown in figure 2.13.

Table 2.1. Effect of DEMs on simulation performance

Model			#DEMs	%DEM time
name	#mods	#nets		
ripple buffer	14	38	4	5.4
			8	5.5
			16	6.3
railroad	483	1480	4	4.9
			8	5.5
			16	6.2
Kulisch inner product chip	679	754	2	2.3
			4	2.6
			8	3.0

The results show that little overhead is introduced by extending the simulation algorithm to support DEMs. Of course this also depends on the complexity and internal structure of the simulation model and the connected DEMs. For these particular examples, the total decrease in performance is less than 7%. Comparable results have been found for other simulation models. This demonstrates the efficiency of the proposed methods and the applied implementation techniques.

2.9 Hierarchy

In the previous sections, a system is described using a two–level representation: the system is composed of a number of subsystems or modules, which are interconnected by nets: one top level description of a composite module consisting of a number of primitive modules. Each primitive module has a behavioural description. However, most systems use structural hierarchy to manage the complexity of describing a large system and to reduce the total amount of design data. Hierarchy can also be used for behavioural decomposition of complex system at high abstraction levels.

In ESCAPE, the management of hierarchy presents some additional problems. In section 2.6, the interaction models between the editor and the simulator have been described: a modification incrementally updates all data structures. Using hierarchy, the composite module, in which the modification is made, may be used multiple times or even in a hierarchy of another top level description. The basic problem is how to propagate these modifications through the hierarchy and which instantiations of the modified module should be replaced or modified as well.

With a hierarchical approach, a system is represented as a collection of two–level hierarchical designs in an editor. In the ESCAPE terminology, a network view of a design contains instantiations of symbol views of other designs. If one selects one design as the top level of the description, it is possible to compose a hierarchical tree from these parent/child relationships. Figure 2.14 depicts a very simple representation of such a hierarchy showing the relations between various modules. In this figure, R is the root design, 3 and 5 are composite modules, and the other modules are primitive.

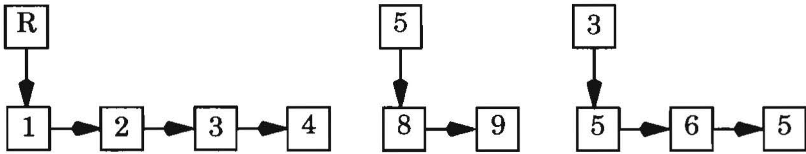


Figure 2.14. Two–level (hierarchical) representation

Simulation or verification tools usually require that such a description is *flattened* (for instance, VHDL simulators may compile the hierarchy into a nested function call hierarchy). Flattening means that the hierarchy is expanded into a single two–level description (see figure 2.15). This expansion recursively replaces each composite module by its network view description, which consists of a number of modules connected by nets, while connections are properly updated. After flattening, the resulting system description only contains primitive modules with a proper model of computation.

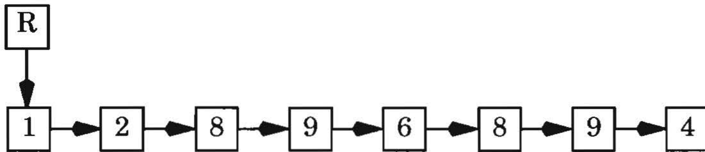


Figure 2.15. Simulation representation of design R after expansion.

Flattening is often embedded into a net list compiler, which compiles a hierarchical description into a format or data structure, that can be accessed by a simulator. In a traditional environment, netlist compilation is performed by a separate tool or by the simulator prior to a simulation run. Small modifications in a description often result in long compilation times of the underlying net list, because the hierarchy has to be expanded and flattened again. This problem is prevented by keeping the resulting hierarchy up–to–date and by

propagating all modifications in a description through the whole hierarchy. This results in an incremental approach for the complete hierarchy.

2.9.1 Related work

Last years, some attention has been paid to incremental net listing to reduce excessive re–compilation times. In [Jon89a], a technique is described that allows incremental net list compilation. The technique is fast enough to update designs, that are being modified in the testing phase, without observable delays. The underlying data structure for this compilation technique is a design DAG, which is a more compact representation of a hierarchical tree. This representation is described in more detail in [Jon89] together with online/offline netlist compilation techniques.

In [Cae93], a process–level debugger for the GRAPE environment is described. This debugger animates program behaviour of complex parallel programs using a hierarchical graphical representation. During debugging, a top–down approach is used: the level of detail of erroneous processes can be expanded. A record–replay mechanism guarantees reproducible program behaviour. Although the designer can select the level of detail in the graphical representation (by expanding the hierarchy), the underlying model is always completely executed. This zooming approach is not possible, if the underlying model is modified as well, for instance replacing a model with its more detailed implementation. The record–replay mechanism can only reproduce the original behaviour, if no state variables are present or if the execution of the model is fully repeated.

2.9.2 Managing hierarchy

In section 2.6, the interaction strategies between editor and simulator have been explained for a flat model. In this section, it is explained how hierarchy is represented that allows easy interaction between editor and simulator. It is based on the idea to keep both the (partial) flattened net list and the hierarchical data, from which this net list is derived, up–to–date.

Some additional advantages of these methods are

- it allows easy switching between different simulation configurations: a structural and hierarchical description of part of the system is replaced by a behavioural one or vice versa.
- it allows interactive expansion or reduction of (parts of) the hierarchy.

In figure 2.16, the relations between various objects of a composite module and its implementation are depicted using dashed arrows. These are required

to keep the hierarchy up–to–date. Note that the composite module is normally embedden into another composite module and that its ports are connected to nets in that module. These relations may be used to build two alternative representations of a hierarchical design.

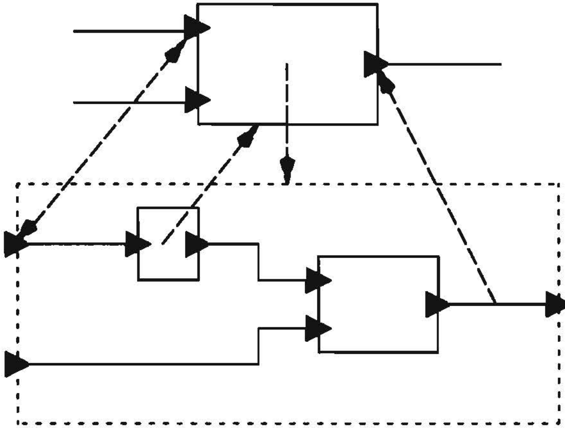


Figure 2.16. Relations used to maintain hierarchical information

In the first alternative, hierarchical trees are built for the modules and nets, and additional data is stored in the ports of these objects to be able to traverse the hierarchy. Note that ports are stored at the modules and that nets refer to these ports for their fanin and fanout lists. During expansion, a composite module is replaced by a copy of the objects described in its network view (circuit). The newly created objects are merged with the existing data structure using parent/child relationships. This procedure is repeated for all modules and stops if a primitive module is selected.

After full expansion, a hierarchical tree is built for the modules and one hierarchical tree per net. This method expands the hierarchy downwards: the primitive modules are pushed downwards and constitute the leafcells of the module tree.

All modules, nets and ports can be accessed traversing the various trees and using the relations between the various objects. It is difficult to present all the relations of figure 2.16 in a single figure for the hierarchy of figure 2.14. To give an impression, figure 2.17 depicts the child relations between the various modules after expansion. Note that to access all modules connected to a single net, the top parent of the net has to be found and then all subnets can be traversed using the parent/child relations of the subnets it is composed of. It is also clear, that sometimes one has to descend in the hierarchy to access a

primitive module. In the flattened net list of figure 2.15 each module is immediately accessible at the top level.

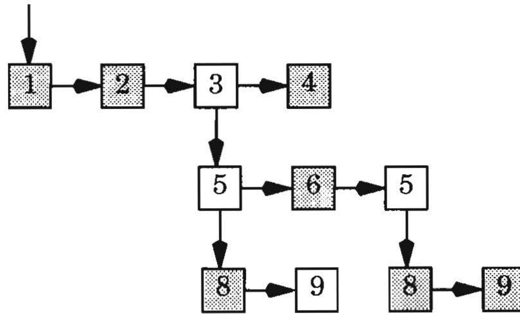


Figure 2.17. Expanding the hierarchical tree downwards

This representation of a hierarchy requires the simulation algorithm to be changed to be able to simulate a hierarchical description. Modules at any level in the hierarchy are generally connected to modules at various other levels in the hierarchy. Therefore, the event scheduling and processing routines have to be changed. If an event is scheduled for a net that spans more levels in the hierarchy, it will be scheduled for its top parent net, which can be found by traversing parent/child relations. At the time the event is processed, the modules on this net's fanout list can easily be accessed by traversing the subnets in a recursive way (see algorithm 2.5).

Algorithm 2.5: Processing a hierarchical fanout list

```
process_fanout_list (Net net)
{
    Port p;
    foreach (p ∈ net–>fanouts)
    {
        if (p–>child)
            process_fanout_list (p–>child–>net);
        else
            process_event (net, p);
    }
}
```

Clearly, this approach decreases the performance of the simulator, if deep hierarchies are used to model the system. In that case, many child relations

have to be traversed to access all modules connected to the net. A solution for this problem is the introduction of *skip pointers*. Intermediate nodes in the hierarchy, which are fully expanded, are skipped. As a result, the leaf nodes of the net tree representation will directly point to the top node of the net. During simulation, at most one additional pointer has to be traversed to access each module connected to a particular net.

Therefore, an alternative representation of both net list and hierarchical information may be used. To the simulator, the net list is similar to a normally flattened net list (figure 2.15). However, it is still possible to traverse the hierarchical data the net list is built from.

In this representation, the composite modules are pushed downwards. To represent the net list and original hierarchy, the same relations are used between modules, nets and ports (see figure 2.16). The original hierarchical relations are accessible using the parent/child relationships of the modules, nets and ports respectively. In figure 2.18, the resulting tree after expansion is depicted. Again just one of the relationships is shown in this figure and only for the modules: it allows access to all modules in the expanded list and the composite modules they are expanded from.

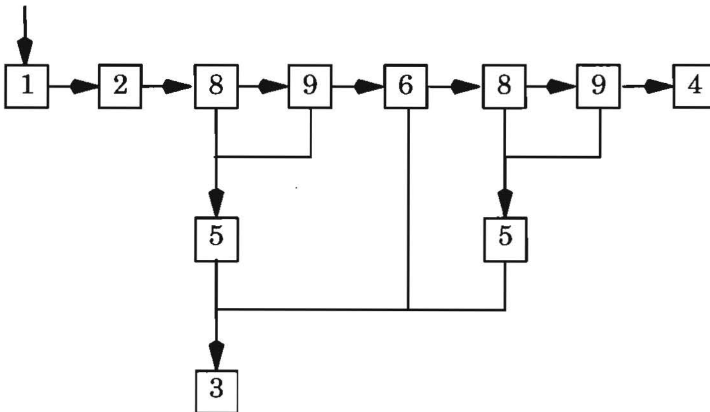


Figure 2.18. Expanding the hierarchical tree upwards

The second expansion method does not require any modifications in the simulator, but it is much more complicated to keep the resulting data structure consistent, if modifications are made to the circuit after expansion. Many commands provided by the API, mostly editing commands, have to perform complicated actions to keep the resulting data structure up–to–date. The hi-

erarchical information, which can be accessed using the various parent/child relations of the objects, has to be traversed frequently.

An additional improvement is the replacement of a hierarchical tree by a directed acyclic graph (DAG), in which a subgraph may represent multiple copies of isomorphic circuitry. However, each copy has to store some unique data (e.g. the value that a particular signal carries), that has to be represented explicitly in the data structure of a node. A DAG representation would facilitate incremental updating of the hierarchy after modifying a subdesign.

2.9.3 Design management

In the previous section, the management of hierarchy using an incremental approach has been described. In addition to managing hierarchy, each component of a system can have different implementations, simulation models and status during design. A component may even have multiple descriptions that need to be simulated using different algorithms or another simulator (see chapter 3). Before each simulation run starts, it has to be determined, how the overall simulation model is built using the hierarchy, versions and *simulation views* of various components. But there are more problems associated with editing and simulating designs. A few examples are:

- A child design is used as an instance of two other parent designs. After simulating one of these designs, the child design has to be modified. Have these changes to be effectuated for the other parent design as well?
- A specific component has multiple models of computation or behavioural descriptions. The system needs to track, which description is used for each instantiation of the design. How can this be done in a consistent way?
- The time representation of a component may be different for various types of models or in different simulation runs. This is especially true, if different types of models are simulated in a single run (see chapter 3). How can the system maintain time parameters of a model in a consistent way?

In general, the problems related to design data management are handled by a CAD framework [Wol93]. Such a framework manages different relations and aspects of design data.

As an example, the relation between hierarchy and versioning is presented in figure 2.19. Consider, the hierarchical description of a nand gate, which is composed of an *and* gate and an inverter (*not*). If a new version is created of the inverter, there are two approaches to update the versions of its ancestors. These approaches are referred to as dynamic and static binding of hierarchy.

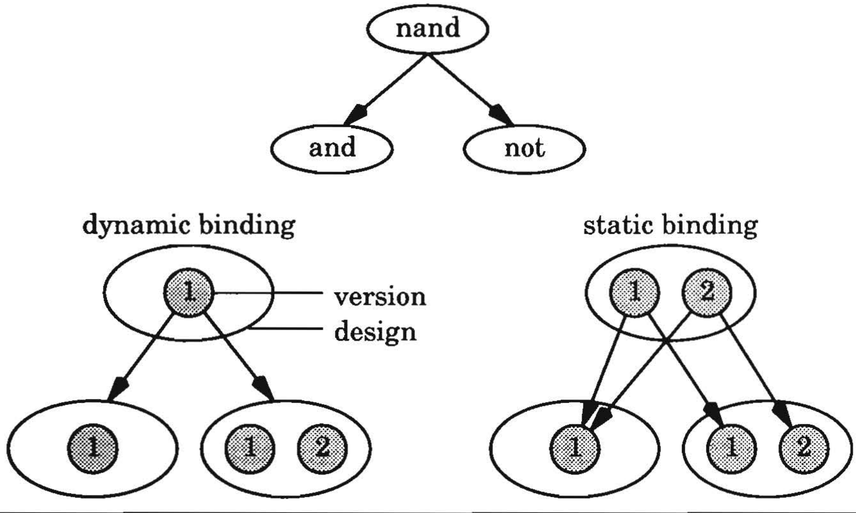


Figure 2.19. Dynamic and static binding of hierarchy

The problem is that on the one hand the management of these problems should be an integral part of the simulator, but on the other hand part of these problems are solved by a CAD framework. Most frameworks are able to manage at least hierarchy and versions in a consistent way. In [Bin94], the functionality of a CAD framework is used to build system simulation environments.

Although a CAD framework may facilitate the use of tools and the management of simulation data, it does not solve the problem of building a simulation model that is composed of a number of submodels, in particular if these submodels are of a different type. Therefore, the management of data related to the design and the underlying simulation model is an important parameter in the development of a simulation tool or environment itself.

Chapter

3 Multi-model simulation

3.1 Introduction

In the previous chapter, the discrete event simulator and its embedding into ESCAPE have been discussed in detail. In this chapter, it is explained how the discrete event simulator of ESCAPE is extended to support simulation of different models of computation. This support is required

- to be able to simulate different models of computation (at the highest levels of abstraction, in particular the system level);
- to be able to simulate partial implementations at lower levels of abstraction.

The motivation for the ability to simulate different models of computation is that in near future complete systems like telecommunication and multimedia systems are going to be integrated into a single IC. These systems consist of digital components such as DSP cores, general purpose processor cores and standard logic cells, analog components and embedded memories, and, the software that is mapped into these memories. There is also a tendency that processor cores are bought from other parties together with simulation and verification models. These models have to be integrated with models of other parts of the system to create an overall model that is executable or simulatable.

The motivation for the ability to simulate partial implementations is that it is impossible to simulate a complete system at a low abstraction level, each time a part of that system has been implemented or refined. If this partial implementation can be simulated together with the rest of the system modelled at a higher abstraction level, the operation of that implementation is more easily validated: the system itself provides test vectors during simulation.

Furthermore, design flows are often adapted to particular needs for different projects. Different tools are used to accomplish a specific design task. Each tool may use its own descriptive means to model behaviour and other properties of a design. A simulator has to be flexible enough to deal with a rapidly changing design flow and the descriptions used by the tools in this flow.

Therefore a simulator must provide facilities for fast integration or coupling of new types of simulation models. Furthermore, the simulator must simulate these types of models together in a homogeneous way. Simulation of various types of simulation models is often referred to as *co–simulation*. Examples of types of simulation models that should be handled by the simulator are:

- models documented in HDLs, that are not supported by ESCAPE;
- models based on other computational paradigms;
- analog models;
- digital models with a different accuracy;
- optimized versus non–optimized digital models;
- software models (hardware/software co–design).

The basic idea to combine these models into a single overall simulation model is to use *events* as the unifying mechanism. The event handler of the built–in event driven simulator fulfils a central role: it controls the execution of all types of simulation models. The following aspects have to be dealt with:

- different granularity of time and the representation of time in the simulator. A token flow graph may even have no notion of time at all, but this restricts its embedding in an overall (timed) simulation model;
- synchronization of the local simulation time in various components;
- different data types and representations for the values of nets in different simulators.

The following techniques are used to handle various types of models in a homogeneous way:

- by using different types of events;
- by embedding models with local control;
- by using an external simulator interface.

The terminology used in this chapter is explained by figure 3.1. A simulation model of a system is composed of a hierarchical description of components. This hierarchy is depicted as a tree in the picture to the left. All leaf nodes in this tree are primitive components, whereas all other nodes are composite components. Each primitive component has a description of its internal behaviour. This may be a description in a language supported by ESCAPE or in an input language of an external simulator integrated with ESCAPE's simulator. The latter description could even contain hierarchy, if that is supported by the external simulator. This hierarchy is hidden from escape. Note that the beha-

viour of a composite node in ESCAPE is implicitly described by its structure and the components in that structure.

A primitive component together with its behaviour description is often called a model. The class of behaviour descriptions that are executed using an external simulator or a dedicated simulation algorithm is often called the *model of computation* or the *type of model.* The picture to the right shows how a variety of built–in models and external simulators simulating models is simulated together using the event handler of the built–in simulator in a central role.

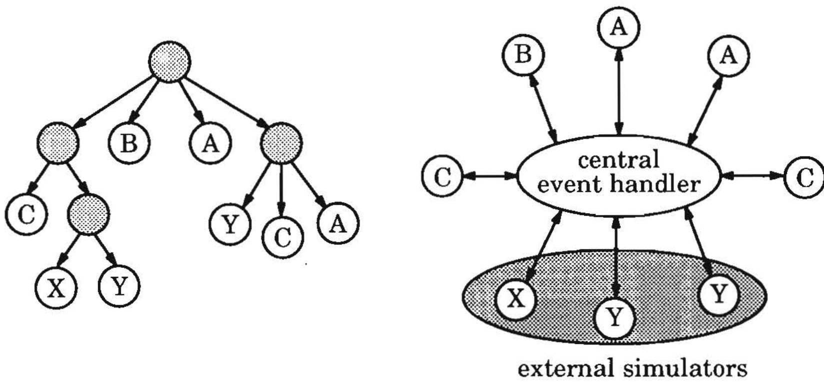


Figure 3.1. Simulating models both internally and externally.

In the remainder of this chapter, it is first explained how a foreign language interface can be used in a flexible and interactive simulation environment. Next, a generic interface to external simulators is described. Then the simulation of data flow models and the co–simulation of discrete event models and data flow models is explained in more detail. Finally, it is described how a simulator is configured at run–time to improve the overall simulation performance.

3.2 Foreign language models

The incorporation of foreign language models is an important feature for a simulator. It allows to write models in a regular programming language like C. This is important because

- prototype programs are often written in a programming language;
- it provides access to the operating system and to other programs.

Nowadays, most commercial simulators provide an application programmable interface to include models written in a programming language. Most of the time this interface is based on the language C [Ker88]. In this section, a technique is described, which may be used to specify an interface of a simulation model written in a foreign language, that is independent of the simulator the model will be simulated with. This feature is used in ESCAPE to be able to detect the necessity of re–compilation, if the compiled model gets out of date. It also allows to re–use the model with multiple simulators that are able to process the syntax of the language.

A foreign language interface of a simulator may be used for various purposes:

- To specify the behavioural description of a module in a programming language like C, Pascal or Fortran. Models may be used for both simulation and execution as part of a regular program written in software.
- To connect an external simulator to a module. The module is executed like any other module by the event handler of the internal simulator. It behaves like an internal simulation model.
- To import other discrete models. A separate compiler translates a description into C files and an interface file. For instance, a (symbolic) description of a finite state machine is translated into regular C code and an interface. After loading, the resulting compiled model is executed very efficiently during simulation.

3.2.1 The foreign language interface

Many foreign language interfaces are provided as a C library, which allows open access to the simulator and its internal data structures. Such a C library is very tedious to use and highly dependent on the simulator. Instead a language has been developed, that allows to specify the interface between a foreign language model and a simulator. This language relieves the designer from the task to write an interface using the functions provided by a C library and the macros defined in include files. A preprocessor built into the simulator translates the language into the appropriate interface functions. This hides many implementation details from the designer. The language in which a foreign language interface is specified, is called ESCAPE–C.

Another reason to develop a language is the highly interactive nature of ESCAPE. As described in section 2.6, there are two interaction strategies between the editor and the simulator: objects in the model may be modified anytime, even during simulation. With both strategies, it is very important that references in the loaded object are consistent with the interface of the design. Otherwise, severe run–time errors may abort the program. The preprocessor easily detects modifications with respect to the last time it processed the interface file. If necessary, the model is compiled and loaded again. An example

may illustrate the importance of using up–to–date models: if a port of a design is deleted, it still may be referenced in the object that is currently loaded for simulation. Trying to access the port will result in a fatal run–time error.

The steps that have to be taken to load a foreign language model dynamically into the address space of ESCAPE are depicted in figure 3.2. First, the interface file is processed by the simulator, in our case ESCAPE, that will load the resulting object code after compiling and linking the C files of the model. In ESCAPE, the steps are invoked through the user interface or automatically in the initialization phase of a simulation run. Second, all C files are compiled and linked with those options enabled, that allow dynamic loading of the object code[2]. Finally, the resulting object is loaded run–time by the program and the model loaded may be called during simulation as part of a larger simulation model.

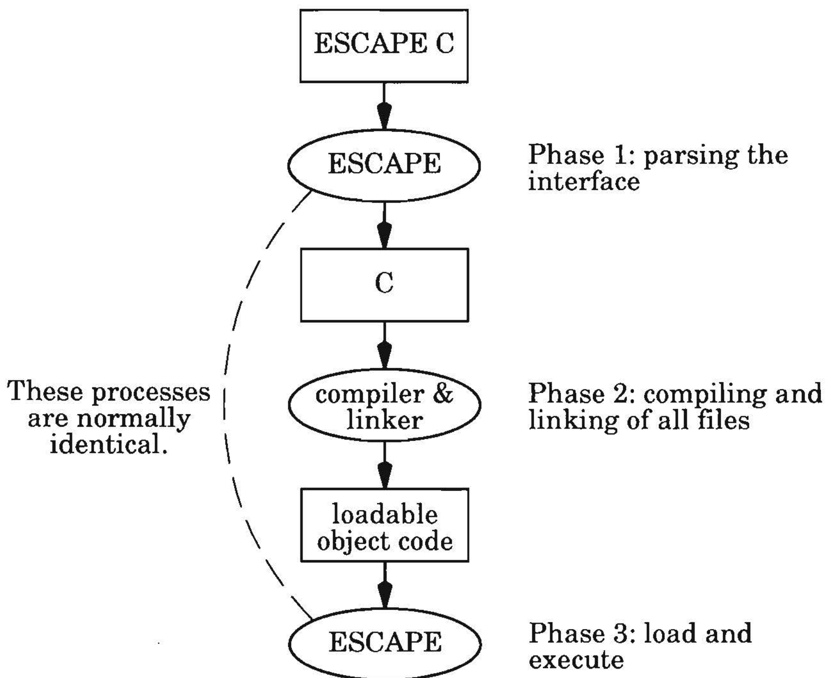


Figure 3.2. Flow to dynamically load a foreign language model.

In figure 3.3, a very simple example of a module's interface written in the ESCAPE–C format is shown. The section between the %% delimiters is processed

2. Dynamic or run–time loading is a technique that allows to add new functionality to an executing program. It requires compilation and linkage using special options. After linking, functions and variables in the object code can be accessed like any other function in the program without any overhead.

by ESCAPE and converted into the C language, that can be compiled by an ANSI C compiler. The other sections of the file and all other files are plain C and are not processed. Therefore, C programs modelling behaviour are easily interfaced with the discrete event simulator. Similar parsers could be developed to be able to incorporate the model into another program or to execute it as a stand–alone executable. Such a parser has to be written only once, whereas normally each model has to be adapted to a particular foreign language interface.

```
/* regular C */
%%
nand2  (  input in1,
          input in2,
          output out )
{
   delay (out, 1, !(in1 && in2), NORMAL);
}
%%
/* regular C */
```

Figure 3.3. Example of an ESCAPE–C file

During pre–processing, the terminals found in the interface are searched for in the port lists of the appropriate design. If the corresponding port has been found, it is replaced by a C variable, which references the data structure of the corresponding port stored with the instance. After compilation, the port can immediately be accessed during the execution of the instance's simulation model. If the port is not found, a warning is issued and the corresponding port is considered to be dangling. Note that these steps have to be performed only once for each design, not for each instance of that design.

The format may be used to define state variables as well. Note that static variables as defined in ANSI C would be shared by multiple calls for different instances. However, all state variables of a model have to be stored uniquely with each instance using that model. Therefore, state variables of a model have to be declared explicitly in the interface. During preprocessing, a data structure containing all state variables is defined together with a *create* function that is called during initialization of the simulation model. Calling the function creates a data structure that allows private access to state variables for each instantiation of the model.

To be able to load a compiled model dynamically, it has to be compiled and linked to create either a special object file or a shared library. This is highly dependent on the operating system. After compilation, the object can be

loaded by the program. As a result, the function in the object code is callable using a special function provided by the operating system. If the function is loaded properly, the simulator may call it without any overhead during simulation. Dynamic loading of object code is a technique that improves both flexibility and performance.

3.2.2 Compiled simulation

The foreign language interface may be used to integrate other simulation models with the simulator, e.g. optimized logic models used at the gate level. A frontend has been built to allow compiled simulation of logic models as submodels of a larger system. The interface of the models will be generated in the ESCAPE–C format. A logic model is a number of input variables, output variables and expressions that describe relations between logic variables (input, output and intermediate variables). It is the representation of a model using boolean expressions during logic synthesis.

In a compiled logic simulator, the input description is analyzed before simulation to determine the execution order of each expression. A graph is built from the logic expressions constituting the model. Each expression describes a relation between an output variable and a number of input variables. Each variable is represented in the graph as a node. In the graph, there is an edge from the node representing an input variable of an expression to the node representing the output variable of that expression. As a result, a directed acyclic graph (DAG) is built from the expressions of the logic model.

Let $G = (V, E)$ be the DAG representing a logic model. The model may also be at a higher level of abstraction, but the restriction is that it is not asynchronous. The input variables are represented in this graph as nodes which have outgoing edges only. The output variables are represented as nodes which have incoming edges only.

The set of predecessor nodes of a node is defined as:

$$pred(v) = \{w \mid (w, v) \in E\} \tag{3.1}$$

The set of successor nodes of a node is defined as:

$$succ(v) = \{w \mid (v, w) \in E\} \tag{3.2}$$

The function *level* assigns to each node of V a number

$$level : V \rightarrow \mathbb{N} \tag{3.3}$$

in such a way, that the levels assigned to all predecessors of a node are smaller than the level assigned to the node itself. This process is often called *rank or-*

dering or *levelizing* in simulation literature. In general, this algorithm is called a *topological sort*. The function *level* is defined as follows:

$$level(v) = \begin{cases} 0, & \text{if } pred(v) = \emptyset \\ \max\limits_{w \in pred(v)} (level(w) + 1), & \text{if } pred(v) \neq \emptyset \end{cases} \qquad (3.4)$$

The code generator generates C statements from the logical description in the order determined by the rank ordering. The rank ordering ensures that the expressions can be calculated with the proper values for all variables. An example of a small part of generated C code is depicted in figure 3.4.

```
void sim_logic (int lv[])
{
  ...
  lv[514] = !((lv[126])&&(lv[7])&&(lv[9]));
  lv[575] = !((lv[126])&&(lv[7]));
  lv[647] = !((lv[7])||(lv[8]));
  lv[857] = !((lv[7])||(lv[11]));
  lv[864] = !(((lv[129])||(lv[128]))&&
            ((lv[7])||(lv[8])));
  lv[884] = !((lv[7])||(lv[8]));
  lv[894] = !((lv[128])||(lv[7])||(lv[9]));
  ...
}
```

Figure 3.4. Part of generated C code from logic expressions.

The code generator can generate statements using either logical or bitwise operators. The advantage of using logical operators is that the evaluation of the resulting operands of an expression is aborted, if the evaluation of the previous operand results in a fixed value, for instance if one of the operands in a logical and evaluates to 0.

The advantage of using bitwise operators is that it allows the parallel simulation of multiple testvectors. This implies that if there are multiple modules with the same logical description as behaviour, the calculation of their output values can be performed in parallel with no performance penalty. Using a computer with 32–bit arithmetic, it is possible to execute 32 instantiations of the same type in parallel. Of course, it is also possible to calculate 32 testvectors in parallel for specific simulation experiments.

3.2.3 Experimental results

The compiled logic simulator has been evaluated using a few benchmark circuits. These benchmark circuits consist of

- the two largest examples of the ISCAS '85 benchmark circuits [Brg85].
- the *primes* examples. The circuit **primes *n*** calculates, if the number at its n inputs is prime or not. If the number is prime, it will set the first output bit to a logical 1. Otherwise, it will set its first output bit to a logical 0 and set the remaining output bits representing the smallest divisor of the number at the input bits. These primes circuits have been used as logic synthesis benchmarks in [Ber92].

In table 3.1, the system resources used by the compiled models are listed. The size of the shared library depends on the number of expressions and the complexity of these expressions. This is illustrated best by the primes examples: the number in the name of the example is related to the number of expressions (it equals the number of input bits of the circuit). Before logic synthesis, the number of expressions equals 2^n, where n is the number in the name of the example. Also, the complexity of the expressions depends on the number of input variables, which is equal to n. This is still roughly true after logic synthesis. The primes circuits used in these benchmarks are the expressions after logic synthesis.

The compile times of these circuits are also listed. In [Bry87], the overhead of compiling into C code is estimated at least 70 percent. It can be reduced by generating executable target machine code. This is often referred to as *native compiled code* generation.

Table 3.1. Usage of system resources

name	size of library [kbytes]	compile time [sec]
c6228	120	10
c7552	160	11
primes9	30	8
primes12	320	38
primes14	1500	165
primes16	5900	641

In table 3.2, some characteristics of the benchmarks are listed as well as the number of simulation vectors that can be executed per second. The benchmarks have been run on a HP 9000/755. The variation in benchmarks clearly

illustrates the difference in time it requires to perform exhaustive simulation for a model giving a 100% error coverage. For instance, the c7552 benchmark circuit has 207 inputs: exhaustive simulation of all possible input patterns requires 2^{207} input patterns. Given a simulation speed of roughly 1400 vectors per second, it will take about 10^{50} years to simulate it completely, even if 32 vectors are simulated in parallel. However, the largest circuit (primes16) having 16 inputs requires only 65536 input patterns and can be simulated completely within 3 minutes (simulating 32 vectors in parallel).

Table 3.2. Simulation results of compiled logic simulation models

name	inputs	outputs	expressions	vectors / sec
c6228	32	32	2416	1607
c7552	207	107	3513	1397
primes9	9	5	464	7764
primes12	12	7	2960	402
primes14	14	8	17133	46
primes16	16	9	60080	13

In table 3.3, the simulation performance of the LISP–like HDL descriptions and compiled C code is compared. Note that it only compares raw simulation speed: the time to build the internal data structures for the LISP–like HDL description and the time to compile the C code is not included here[3].

Table 3.3. Comparison of interpreted and compiled models

name	Number of evaluations per time unit		speed up
	lisp	compiled	
primes9	1102	2568	2.33
primes12	400	2396	5.99
primes14	84	1228	14.62
primes16	–	227	–

3.3 External simulator interface

Although the built–in simulation models and the foreign language interface allow many models to be simulated, it is very important to be able to simulate more different types models in the overall simulation model together. This includes some types of models which are *incompatible* with the discrete event

3. The simulation of the LISP–like HDL description of the primes16 example has been aborted after three hours, because it would consume too much memory resources: the internal data structures required for this example could not be built at the time of the benchmarking.

paradigm, and other types of models may only differ in subtle details like time representation and time advance mechanism or the representation of signal values. Often, dedicated simulators are required to be able to simulate a model accurately (e.g. a simulator for switched capacitor circuits [Fan83]) or to get a reasonable performance during simulation (e.g. co–simulating software on a microprocessor model [Row94]). It is often not possible to *translate* or *compile* such a model into a model that is compatible with built–in types of models that are based on the discrete event paradigm.

A simulator should be flexible enough to incorporate new types of models into the system or to integrate external simulators in order to be able to simulate complex heterogeneous systems. It is essential that all types of models used to describe a system are simulated in a single simulation run. This implies that a simulation environment should provide the flexibility to include new simulation algorithms or integrate external simulators in a simple and straightforward way. Even then, this is often a quite complicated and time consuming task. In this section, a technique is described that facilitates the integration of new types of simulation models and allows the simulation of these models in a homogeneous way. Seen from the simulator, all models behave like regular discrete event models.

The following terminology is often used for simulators that are able to simulate multiple types of models:

DEFINITION 3.1: A mixed mode simulator is a simulator that is able to simulate both digital and analog models.

DEFINITION 3.2: A mixed level simulator is a simulator that is able to simulate models at various levels of abstraction.

DEFINITION 3.3: Co–simulation is the ability to simulate different types of models together in one simulation environment.

The terms mixed mode simulator and mixed level simulator are often interchanged or appear as synonyms, because the simulator processes the analog parts of a system at the circuit level and the digital parts at a higher level of abstraction. Co–simulation is often referred to in the context of simulating processor models running software or in the context of simulating models written in the popular HDLs like Verilog and VHDL in one environment.

In the past, mixed mode and mixed level simulators have been introduced to be able to simulate a system composed of both analog and digital components at various levels of abstraction. They have primarily been introduced to reduce the simulation time. Simulation at multiple levels of abstraction greatly reduces the computational costs, because large parts of the system are simulated at a high level of abstraction. Only those parts of the system are simu-

lated at a low level of abstraction, that really need to be simulated at that level in order to simulate it accurately. For instance, the analog components of a system are simulated at the circuit level, whereas the digital components are simulated at the register transfer level.

In [All90], three basic types of mixed mode simulators are identified[4]:

- The use of an analog simulator to perform both analog and digital simulation. One method is to model digital components using the same techniques as analog components (e.g. differential equations in a circuit simulator). This results in high computational costs for the overall model. Another method is to provide an interface to include discrete event models in particular analog model representation.
- The use of a digital simulator to perform both analog and digital simulation. Time continuous analog models are transformed into time discrete models. It allows fast and efficient simulation of analog components, but it often proves to be inaccurate.
- The use of both analog and digital simulators that are coupled together. This type allows to combine both performance for the digital components and accuracy for the analog components.

The last type of mixed mode simulation ensures the best accuracy and efficiency, but it strongly depends on the approach that is used to couple the simulators. In [Sal90], three approaches are distinguished for coupling simulators:

- The *manual* approach. This approach is very tedious to use and very inaccurate as well. The designer has to iterate a number of simulation runs with different simulators to simulate the model. The results produced by one simulator have to be used by the other simulator in the next run. However, effects occurring at the interface between two or more models, that are simulated by two different simulators, can hardly be simulated and the final results will be very inaccurate after some iterations. The manual approach is a very time consuming activity for the designer and it is mainly listed in this overview to show the necessity of a real coupling between two or more simulators.
- The *glued* approach. Two or more simulators are coupled using either an inter process communication mechanism or a procedural interface. The latter may be combined with a run–time loading technique. With this approach, each simulator has its own user interface and uses its own input languages. One of the simulators may serve as a master, that controls the

4. The problem of mixed level simulation is closely related to the problem of mixed mode simulation and therefore it is assumed here that mixed level simulation is part of mixed mode simulation.

execution of the other simulators (the slaves). If a simulator is coupled with other simulators, the mapping between signals and the conversion of signal values has to be handled, as well as the time advancement mechanism and the synchronization of all simulators.

- The *fully integrated* approach. Different models are simulated using different algorithms, that use the same time representation, time advancement mechanism (see section 2.3.2), data representation of signal values and the underlying net list. The algorithms used to handle different types of models are especially designed to operate in the mixed mode simulator. In most mixed mode simulators of this type, the resulting simulation engine uses an event driven paradigm.

The main difference between the glued approach and the fully integrated approach is that with the glued approach integration is handled at the simulator level whereas with the fully integrated approach integration is handled at the algorithmic level. Consequently, if a new type of simulation model is to be added using a glued approach, the corresponding simulator needs to be coupled with the other simulators. With a fully integrated approach a dedicated algorithm has to be written that allows simulation of this type of model. The latter is often too expensive if the simulator has already been developed.

Simulation backplanes use a glued approach to connect simulators with each other. The simulation backplane provides all functionality that is required to connect a simulator properly to another one. It provides functions that allow synchronization of the simulation time between various simulators and functions to convert data values to standardized formats that are used to transfer data from one simulator to another one. Because all functionality for interfacing a simulator is provided by the backplane, the effort to couple a simulator to it is greatly reduced and it is restricted to modifying the simulator itself.

In ESCAPE, a different approach is being used. Unlike with a simulation backplane, the built–in discrete event simulator is used as a platform, with which external simulators can be coupled. Events are used as the unifying mechanism to integrate different types of models homogeneously. As a result, models can be simulated by the internal simulator or by external simulators. One of the key differences with other approaches is the role of the built–in simulator. It serves both as a simulation engine for internal models and as a backplane to external models. This improves the efficiency of the simulator considerably.

Simulation algorithms or simulators may be integrated with the built–in discrete event simulator either loosely coupled (glued) or tightly integrated to simulate a new type of model. This depends on the effort put into modifying or implementing the simulation algorithm to be included. The open architec-

ture of the built–in simulator allows integration at various levels of interaction. Depending on the time and effort, one can make a trade–off between implementation time and accuracy desired to integrate a new type of simulation model.

In figure 3.5, the difference between internal and external simulation models is depicted. An internal model is directly executable by the internal discrete event simulator. An external simulation model consists of an external simulator and an interface to the internal simulator. This interface consists of two abstraction layers: the physical interface and the external simulator interface. The interface makes an external simulation model behave like an internal simulation model: the simulation engine activates components, for which events are scheduled. Some of the components are handled internally but others are simulated externally.

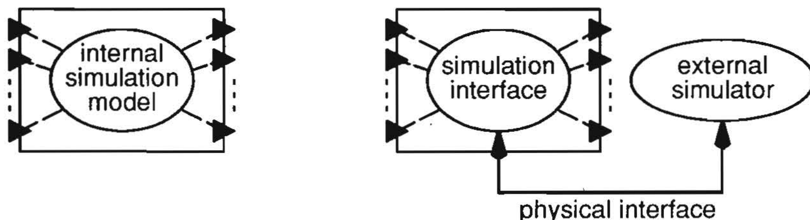


Figure 3.5. Internal and external simulation models

The following definitions are important in the remainder of this section:

DEFINITION 3.4: An *instantiation* of an external simulator is a functionally equivalent *copy* of a simulator. Multiple instantiations of the same simulator do not share resources and may execute different parts of the simulation model independently.

DEFINITION 3.5: A *global* net (signal), in the context of multi–model simulation, is a net (signal) that connects at least one internal simulation model with at least one external simulation model or two or more external simulation models.

DEFINITION 3.6: A *local* net (signal), in the context of multi–model simulation, is a net (signal) that connects either one or more internal simulation models or local submodels of a single external simulation model.

The external simulator interface consists of two abstraction layers. The lower layer represents the physical interface between ESCAPE and the external sim-

ulator. The implementation of this layer is partly operating system dependent. Different approaches have been incorporated in ESCAPE. They can be divided in two categories:

1. The external simulator is dynamically loaded into the address space of ESCAPE. After loading, the functions that control the execution of the external simulator are accessible by the event handler of the built–in simulator. Run–time loading techniques are highly dependent on the operating system that is being used. In general, special attention needs to be paid to an implementation of a simulator to be able to instantiate it multiple times in the same address space. Note that this may require some major modifications in the source code of the external simulator.
2. The external simulator is connected to the central event handler using an inter process communication (IPC) mechanism provided by the UNIX operating system. IPC decreases the performance of simulating the complete simulation model, because it introduces communication overhead. The amount of overhead depends on the IPC mechanism. However, the ability to simulate the complete system often outweighs this disadvantage. Because the external simulator runs independently as another process, synchronisation between the processes is required to orchestrate the proper execution of models in multiple processes. IPC techniques can also be applied to distribute the simulation of different models across various machines in the network. Multiple instantiations of the same type of simulator may be used without any restriction.

The upper layer controls the operation of the external simulator and manages the event flows between the external simulator and the central event handler. This layer is completely embedded in the discrete event simulator. In this thesis, the upper layer is referred to as the external simulator interface.

In figure 3.6, an abstract model of ESCAPE's external simulator interface is depicted [Fle93b]. This model tries to visualize the flow of events between the central event handler of ESCAPE and an external simulator, and, the operations that can be performed onto these events. In practice, all components of the external simulation model, like the event and value conversion modules, are tightly integrated with the internal data structures and the simulation algorithm of the internal simulator. Each component is customized depending on the external simulator to be connected and the model to be simulated by this simulator: this data is stored with the simulation model itself and is loaded during the initialization phase of a simulation run.

At the moment the external simulator is instantiated (coupled), the conversion routines of the interfaces are installed and stored in the proper objects of the internal network data structures. The *synchronization* component is global to all external simulator interfaces. It determines from the flow of events between an external simulator and the internal event driven simula-

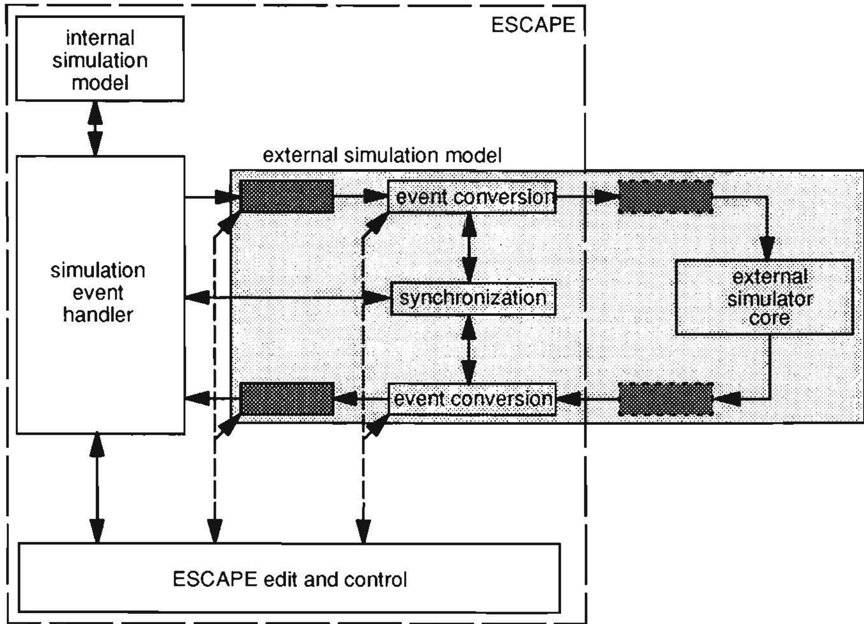


Figure 3.6. Model of the external simulator interface

tor when the simulation clock of the internal simulator can be updated. The synchronization strategy used is important for the overall performance and accuracy of the simulation run. Especially, if an external simulator is coupled through IPC, synchronization strongly influences the overall performance: since it may run completely independent from the internal simulator, it has to be controlled and synchronized by the internal simulator using the appropriate messages or events.

The *event conversion* blocks convert internal events in messages or events that pass signal values of global nets from the internal event handler to an external simulator, and vice versa. These blocks also control and observe the activation and suspension of the execution of an external simulator. This information is used by the synchronization block to update the simulation time of the internal simulator. Event conversion routines are invoked in the first phase of the simulation algorithm, when the contents of each event is evaluated to determine which components need to be simulated in the second phase. Multiple event conversion routines may be defined for a single net. The source of an event (which is actually the port of the model that propagated the event) determines which routines are invoked.

The *value conversion* routines are installed at the ports of the component that is simulated by an external simulator. If such a component is activated, the

signal values at the input ports are converted and propagated to the external simulator. Then the external simulator continues the execution of the model. After suspension of the execution of the model, the signal values at the output ports are converted and propagated to the internal simulator. It is important to note that a proper choice for the value conversion routines reduces the number of events that are communicated across the interface; such a reduction increases the performance of the simulator significantly.

An external simulator interface has to be defined only once for a particular simulator. Such an interface can be instantiated multiple times within a single session or in another session. The only restriction is imposed by the way the external simulator is connected to the central event handler: in case the external simulator is run–time loaded into the same address space, it is only possible to load a single *instantiation* of such a simulator, unless the simulator has been implemented in such a way that the problem of *global* and *static* variables is solved: they will be shared between the various instantiations of a particular simulator. With IPC, there is virtually no limit to the number of instantiations of an external simulator running either on the same machine or another machine in the network.

The effort to integrate an external simulator consists of two tasks:

- Adapting and compiling the source code of the simulator to be integrated. The availability of source code is a prerequisite to be able to create an efficient interface between this simulator and the central event handler of the internal simulator. The main body of the simulator has to be split in different functions. The minimal set of functions to be able to invoke an external simulator is: an initialization function, a *basic simulation step* function and some functions to propagate and to receive signal values. If the external simulator is event driven, the source code to write these functions is often easily identified and isolated.
- Defining the simulation interface in ESCAPE. Currently, this is done by coding a special data structure, which adapts the central event handler to handle synchronization and event conversion. The value conversion routines are stored with the global nets and can be overloaded any time. This allows the simulation of various models using the same external simulation interface. It is also used to adjust the accuracy of the value conversion itself, which strongly influences the overall performance of the simulator.

Although it has been stressed in this section that the availability of source code is a prerequisite, the foreign language interface of an existing simulator may be used to integrate a simulator with ESCAPE as well. The problem that has to be solved is that both simulators are scheduling and processing events continuously and they are both trying to control the simulation. However, the

simulation algorithm of ESCAPE also handles inter process communication. The inter tool protocol (see section) could be used to control the activation of different components by the simulators and to synchronize the simulation time.

3.3.1 Synchronization of simulation time

An external simulator keeps track of its own simulation time. A correct and accurate simulation of the overall model requires careful synchronization of the simulation time of the external simulator(s) and the internal event driven simulator. This is especially true, if an external simulator runs in its own address space communicating with the central event handler through IPC.

Using a single simulator, the paradigm associated with performing a simulation is selecting the event with the smallest time stamp from the event queue and executing the simulation code that is associated with that particular event. The problem using multiple simulators is that in any simulator all events need to be processed in a non–decreasing order with respect to their time stamps. The same problem is solved for parallel discrete event simulators (PDES). Much research has been conducted in this area. In VLSI design, the focus of attention has been the parallel simulation of logic. The results obtained may be applied as well to higher levels of abstraction.

The problem is closely related with maintaining the simulation state as defined in equation (2.11): (T_{sim}, L_V, E_q). In a PDES, the global simulation state is distributed across multiple logical processes (LPs) [Fuj90]. A logical process i maintains the simulation state $(T_{sim}, L_V, E_q)_i$, while interaction between these processes is performed by communicating events. If each LP processes events in a non–decreasing order, it is ensured that no causality errors occur. This is referred to as satisfying the *local causality constraint.*

Most of the time PDES algorithms are classified into two categories: *conservative* and *optimistic*. The definitions of these are:

DEFINITION 3.7: A *conservative* approach always processes events in a strictly non–decreasing order by advancing the local simulation time to the smallest time stamp of an event received from any neighbouring LP. It preserves causality constraints at all times. This implies that an LP has to be halted regulary to wait until other LPs have completed their tasks.

DEFINITION 3.8: An *optimistic* approach processes each event at the time stamp it is received and the local simulation time is advanced to the time stamp of that event. If an event is

received with a time stamp smaller than the local simulation time, the LP has to perform a *rollback* operation to a state with a local simulation time that is smaller than the time stamp of the newly received event.

One of the problems using a conservative approach is the prevention of a deadlock situation. To prevent deadlock, *null* messages are sent that carry a time stamp only but that do not contain data. The Chandy–Misra [Cha81] algorithm is an alternative approach that avoids the use of null messages. After detecting a deadlock situation, it breaks the deadlock using a special algorithm. One of the problems using an optimistic approach is that each LP has to save its state on a regular basis, which may be a time consuming operation. Furthermore, a rollback also requires *anti–events* to be sent to cancel events that have been sent out in the meantime. The first optimistic approach has been called the Time Warp mechanism [Jef85]. Many alternatives have been presented later for both approaches.

In [Rey88], a more detailed classification is described for PDES algorithms. It is interesting to note that one can also distinguish between asynchronous and synchronous approaches. In an asynchronous approach, each LP maintains its own simulation time, whereas in a synchronous only one global simulation time is maintained. More recently, a state of the art overview of parallel logic simulation techniques has been published in [Bai94]. These approaches may also be applied for multi–model or heterogeneous simulation, either in ESCAPE or another mixed–level simulator.

In a fully integrated approach, only one event queue and only one simulation time is maintained: in that case, no synchronizing mechanism is required to orchestrate the multiple simulation processes, but still care has to be taken to integrate the algorithms in such a way that accurate simulation is ensured. Often, one can not afford to use the integrated approach and one has to use a glued approach. In that case, careful synchronization of the simulation times in the simulators is a must. One of the main differences between synchronization of the simulation times in a glued mixed–level simulator and the synchronization of different LPs at the same abstraction level is that one has to deal with different time scales as well. The most important conservative approaches that are applied in mixed–level/mixed–mode simulators are [Sal94]:

- The *lock–step* method. The analog simulator determines the step size, because it uses the smallest time scale. It controls the digital simulator, which has to use the time points determined by the analog simulator.
- The *digital controlled* method. The digital simulator controls the analog simulator. After applying a time step in the digital simulator, the analog

simulator is forced to simulate up to that time point. In this period, no events from the analog simulator are being processed.

Many variants on these two approaches are developed depending on the organization of the event queues implemented in the simulators used in the mixed–level simulator.

In [Ben91] a variant is described, which the authors refer to as a variant of the lock–step method. This variant is used to couple the Mozart and ELDO simulators. The fixed–increment time advance mechanism of Mozart controls the next–event time advance mechanism of ELDO. If ELDO generates an event for Mozart before its stop time, it is scheduled for the next fixed time point in Mozart and control is returned to Mozart. Delta delay events of ELDO are scheduled at the next fixed time point of Mozart to prevent a rollback to a previous time point. Depending on the time scale used in Mozart, this seems to be a decision that may compromise the accuracy.

In ESCAPE, the strategy depends on the external simulator that will be connected. An alternative approach may be used, if all simulators use a fixed–time increment strategy. In that case, the time step of ESCAPE is set the least common multiple of the time steps of all simulators. Note that this requires the delay values of the internal HDL models to be adjusted to exhibit the correct timing behaviour. This is solved internally in the simulator by updating the simulation model.

If a next–event time advance is used by an external simulator, it is controlled by the internal event driven simulator. A similar control strategy is used as in [Ben91]. Delta delay events, which are considered an artifact of the simulator, are not allowed to be scheduled by an external simulator onto a global net. Simulators at a low level of abstraction hardly use delta delay events, whereas in higher level models the use of a delta delay may help to implement the correct behaviour.

Another problem is the scheduling of delta delay events for global nets by the internal simulator: normally, this should result in another invocation of the external simulator at this time point or if the event is discarded, the external simulator is using the previous incorrect value as an input. Therefore it can be specified, if all delta delay events on a global net should be handled first before the external simulator is called. This has a penalty with respect to simulation performance. The scheduling of delta delay events for global nets may also be prohibited.

3.3.2 Example: The PLATO piecewise linear simulator

PLATO is a mixed–level simulator, based on a continuous dynamic piecewise linear (PL) modeling technique. Using this technique, both analog and digital components can be modeled at several levels of accuracy. The homogeneous modeling makes no difference between analog and digital components [Buu93]. The PLATO PL simulator has several advantages compared to the well–known SPICE simulator [Nag75]. Its main advantage is its global convergence property while solving the PL equations. This property is based on the global convergence of the algorithm devised by Van de Panne [Pan74], which is the core of this PL simulator. This algorithm can handle discontinuities without any problems.

To solve the dynamic PL equations during transient analysis, the Van de Panne algorithm is combined with an integration method. The method applied is an implicit multirate method, which implies that the differential equations are solved by assigning different timesteps to subsets of equations (subcircuits) [Eij90]. The circuit is split dynamically into these subcircuits. As a result, most computational effort is put in solving the active parts of the circuit. An event driven technique is used and at each event either the PL equations are solved or a new integration step is determined. In figure 3.7, an example of a few signals and the barcode representing the events, is depicted.

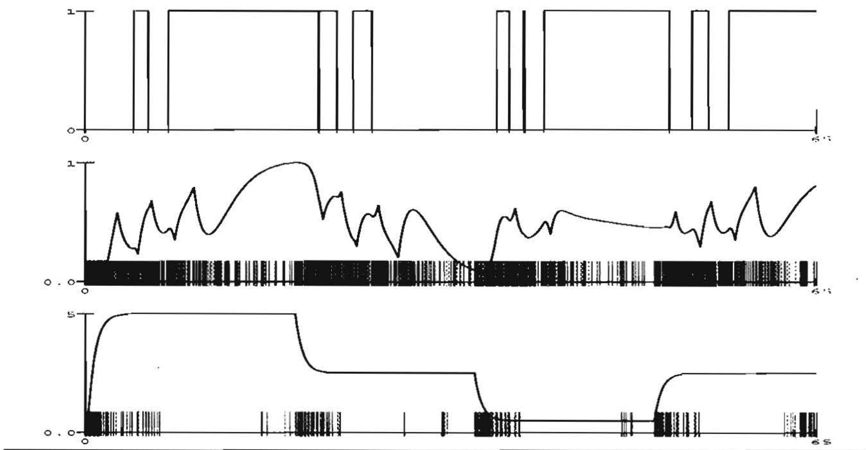


Figure 3.7. Example showing output values and event densities of PLATO signals.

Although PLATO is an event driven simulator, the semantics of the events in PLATO are different from the semantics in ESCAPE, and both simulators use a different timing control mechanism: PLATO is a multirate event–driven simulator, in which events can be scheduled at non–equidistant time points,

whereas ESCAPE uses a fixed increment time advance strategy by default. Typically many PLATO events are generated and processed between two time points in ESCAPE. This depends on the activity of the various subcircuits. The events scheduled for local signals are always handled internally by PLATO. Events scheduled for global signals are forwarded to ESCAPE, if the corresponding value conversion routine detects a possible modification of the signal value.

PLATO has been coupled to ESCAPE using both an IPC technique as well as a run–time loading technique [Fle93a]. The source code of this simulator has been modified in order to provide the following functions:

- the initialization routine
- the simulation step routine
- the global signal mapping routine (to pass the value of the signal).

In algorithm 3.1, the simulation step routine is shown. As many events may be generated between the start time and the stop time, this routine contains an event loop to process local events (lines 5 – 17). Note that in a more integrated approach these events could have been handled by the event handler of ESCAPE. Before entering the simulation loop, the values of the global input signals are stored in the internal data structures of PLATO.

Algorithm 3.1: The simulation step routine

```
1.  procedure execute_plato ( stop–time )
2.  {
3.      update_global_signals ( ) ;
4.      /* simulation loop */
5.      while (time < stop–time) {
6.          event = next_event ( ) ;
7.          ....
8.          time = time ( event ) ;
9.          update_cluster ( cluster ( event ), time ) ;
10.         if ( type ( event ) = pwl )
11.             van_de_panne ( cluster ( event ) )
12.         else
13.             foreach ( leafcell ∈ cluster ( event ) )
                    new_time_step ( leafcell ) ;
14.         ....
15.         foreach ( leafcell ∈ cluster ( event ) )
                new_event ( leafcell ) ;
16.         ....
```

```
17.     }
18.     propagate_global_signals ( ) ;
19.     send_synchronization_event ( ) ;
20. }
```

After leaving the simulation loop, the values of all global output signals are propagated, while the appropriate value conversion routines convert the value of each global signal and determine if the resulting value results in the generation of an event scheduled for the internal event handler. Control is returned to the internal event handler by sending a synchronization event.

In figure 3.8, the events scheduled for a specific signal are depicted as (*time*, *value*) pairs. Let's assume that the value represents a voltage. Note that the figure does not present information on the time points that the events have been generated. The first time axis represents the real time used by PLATO. The second time axis resembles the discrete time points used by the internal simulation algorithm of ESCAPE. The relation between the discrete time point k and the floating point time of the external simulator is $t = \Delta \times k$.

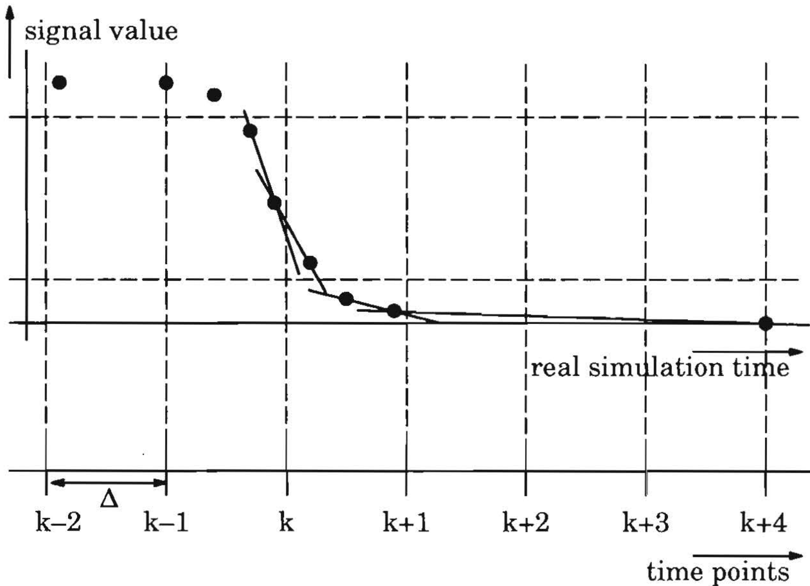


Figure 3.8. Interpolation of signal values at fixed time points.

The signal's value plays an important role to synchronize the time clocks of the simulators, since it is one of the preconditions that determines the discrete time points at which control is returned to the central event handler and

the time points at which execution of the model is continued. Another precondition is of course the activity at the input ports of the model.

Normally, the time stamps of these events do not match with the time stamps of the internal simulator. However, it is essential that the value of the signal is calculated as accurate as possible at these fixed time points. In the PLATO simulator, the signal values used by the integration method are used to calculate the values of the signal at these fixed time points.

3.3.3 Simulation examples and results

Two examples, which are explained in more detail in [Buu93], are presented:

- a switch capacitor filter;
- an analog to digital converter.

A switch capacitor filter is modelled using the piecewise linear modelling technique and is simulated by PLATO. As an input of the filter, a discrete sine generator is used, which is modelled using the LISP HDL. The output of the filter is fed to a 256–point fast Fourier transform (FFT), modelled using the FORTRAN language and called through the programming language interface. For demonstration purposes, the output of the FFT is visualized using a special–purpose view window representing a frequency analyzer. The switch capacitor filter itself contains a number of switches, capacitors and operational amplifiers (see figure 3.9).

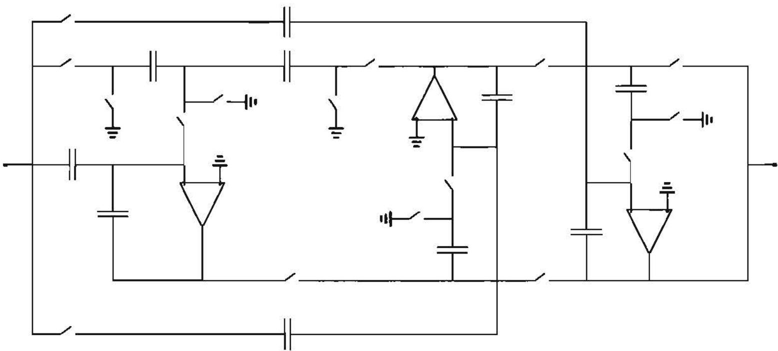


Figure 3.9. Switch capacitor filter

The analog to digital converter circuit consists of the analog–digital converter itself, two analog multiplexers, an analog subtracter and a number of digital

components to control the other components of the circuit. The digital components have been simulated by ESCAPE and the other parts in PLATO.

The simulation times of these examples are presented in table 3.4. A dynamic loading technique has been used to connect PLATO and ESCAPE. The results have been obtained using animated simulation: only the activity on the nets has been visualized.

Table 3.4. Run times of mixed level benchmarks

benchmark	run time [sec]
A/D converter	4.4
switch capacitor filter and FFT	30.0

The complete A/D converter benchmark has been simulated with the piecewise linear simulator PLATO as well. Both the run times and the simulation results have been compared with the mixed level experiment:

- The simulation results of the mixed level simulator are as accurate as the results obtained for the complete benchmark with PLATO.
- The run times of the mixed level simulator are much better than the run times of PLATO: 4.4 versus 18.9 seconds. Note that for this particular example only a few digital components have been simulated in ESCAPE giving already a performance increase by a factor 4.

PLATO has also been connected to ESCAPE using an IPC technique based on sockets. Running PLATO on a different workstation in another network bridged to the workstation ESCAPE was running on, the run times of this experiment are about 17 seconds. Although much communication overhead reduces the simulation performance, it is important to be able to connect to different workstations, for instance to distribute the simulation task.

3.4 Token flow models

In this section, the simulation of token flow models is discussed. Token flow models are often used to model (specific aspects of) systems during the synthesis and verification of protocols and hardware.

The following definitions are very important to describe (and execute) the behaviour of a token flow model:

DEFINITION 3.9: A *token* is an abstract representation of a data item, a condition or an event. A token is non coloured if no data associated it. Otherwise, a token is coloured.

DEFINITION 3.10: A *firing rule* defined for a component defines which conditions have to be satisfied before a component may be executed.

Token flow models are represented as directed graphs. Two types of representation are being used in literature. The first representation is a directed graph $G = (V, E)$, where

- $v \in V$ represents an actor. An actor may fire, if the token distribution at the input edges satisfies the actor's firing rule.
- $e \in E$ represents a queue that holds zero or more tokens.

The second representation of a token flow model is a bipartite directed graph $G = (P, T, E)$, where

- $p \in P$ is a place that holds zero or more tokens,
- $t \in T$ is an actor or transition,
- $e \in E, E \subseteq (P \times T) \cup (T \times P)$ is an edge connecting a place and a transition or vice versa.

There are many application areas, where token flow simulation may help to prototype and validate system behaviour:

- Simulation of complex concurrent system behaviour. At the system level, a process network may be used to define a number of concurrent processes and its communication behaviour. Each process may be modelled using different descriptive means, for instance a hardware description language or a programming language.
- Concurrent simulation of a Petri net and a system model may be used to validate the communication behaviour of the system model at a higher abstraction level.
- Simulation of the behaviour of synchronous data flows for DSP applications.
- Simulation of the behaviour of data flow graphs during the architectural synthesis of digital systems.

First, some types of token flow models will be discussed. Then the simulation of these models is discussed. This is illustrated using the ASCIS data flow graph and a Petri net as examples. Furthermore, it is described how token flow models are simulated in a discrete event simulator and how token flow models are integrated with discrete event models in a homogeneous way.

3.4.1 Petri nets

Petri nets [Pet81] are used to model a great variety of real–life systems, in particular concurrent systems. Examples are computer networks, communicating processes, asynchronous logic. They may be used to model and verify the communication behaviour of complex concurrent processes.

A Petri net is defined as a graph

$$PN = (P, T, F) \tag{3.5}$$

with

- $P = \{ p_1, p_2, ..., p_n \}, n \geq 0$, the set of places.
- $T = \{ t_1, t_2, ..., t_m \}, m \geq 0$, the set of transitions.
- $V = P \cup T$, the set of nodes with $P \cap T = \emptyset$.
- $F \subseteq (P \times T) \cup (T \times P)$, the set of edges. This relation is sometimes referred to as the *flow relation*.

Some types of Petri nets allow multiple edges between a place and a transition. This is accounted for by using a weight function defined on the set of edges, that replaces an equivalent *number of edges* between a transition and a node:

$$W : (P \times T) \cup (T \times P) \rightarrow \mathbb{N} \tag{3.6}$$

This function returns 0, if no edge is present between a transition and a place: $\forall f \notin F : W(f) = 0$.

The set of input places of a transition is

$$I(t) = \{ p \in P \mid (p, t) \in F \} \tag{3.7}$$

The set of output places of a transition is

$$O(t) = \{ p \in P \mid (t, p) \in F \} \tag{3.8}$$

Note that in general the sets $I(t)$ and $O(t)$ are not disjoint. The input and output transitions of a place may be defined in a similar way.

A marking μ of a Petri Net PN is a function

$$\mu : P \rightarrow \mathbb{N} \tag{3.9}$$

This function returns for each place the number of tokens in that place. The marking μ can be defined as a vector as well: $\vec{\mu} = (\mu_1, \mu_2, ..., \mu_n)$ with μ_i the number of tokens in place p_i.

A marked Petri net is a Petri net PN with a marking μ and is denoted as $M = (PN, \mu)$ or $M = (P, T, F, \mu)$. An example is depicted in figure 3.10.

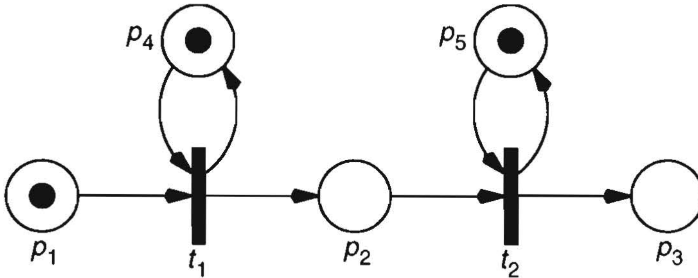


Figure 3.10. Example of a marked Petri Net.

A transition $t_j \in T$ in a marked Petri net $M = (PN, \mu)$ is *enabled* if for all $p_i \in I(t_j)$:

$$\mu(p_i) \geq W((p_i, t_j)) \tag{3.10}$$

A transition $t_j \in T$ in a marked Petri net $M = (PN, \mu)$ may *fire*, whenever it is enabled: a transition fires by removing tokens from its input places and generating new tokens which are added to its output places. Firing a transition $t_j \in T$, results in a new marking μ':

$$\forall\ p_i \in (\ I(t_j) \cup O(t_j)\ ) :$$

$$\mu'(p_i) = \mu(p_i) - W((p_i, t_j)) + W((t_j, p_i)) \tag{3.11}$$

The equations (3.10) and (3.11) describe the *operational semantics* of a marked Petri net. Equation (3.10) is referred to as the firing rule of a marked Petri net.

In the Petri net of figure 3.10, transition t_1 is enabled. After firing this transition, the resulting marking is depicted in figure 3.11. Now, transition t_2 is enabled for firing. Note that t_1 can fire again before t_2, if a new token arrives in p_1 (for instance if this net is part of a larger net).

The *next state* function δ returns the new marking for a marking and a transition that is being fired:

$$\delta(\mu, t) = \begin{cases} \mu', & \text{if } \forall p \in I(t) : \mu(p) \geq W((p, t)) \\ \emptyset, & \text{Otherwise} \end{cases} \tag{3.12}$$

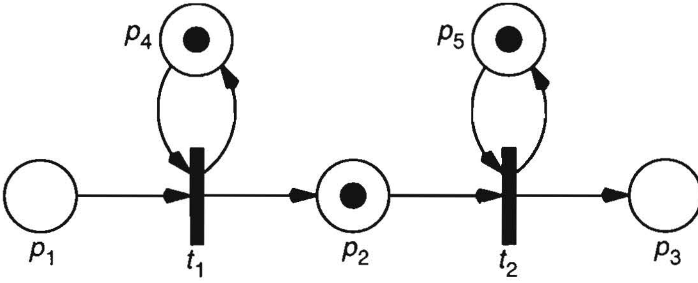


Figure 3.11. The marking of the Petri Net of figure 3.10 after firing transition t_1.

Using equation (3.12), the set of transitions that is enabled in a marking μ is:

$$enabled(\mu) = \{ \; t \in T \mid \delta(\mu, t) \neq \emptyset \; \} \qquad (3.13)$$

Let μ_0 be the initial marking of a marked Petri net M. Firing an enabled transition, $t_{i_0} \in enabled(\mu_0)$, results in a new marking μ_1: $\mu_1 = \delta(\mu_0, \; t_{i_0})$. In the marking μ_1, an enabled transition, $t_{i_1} \in enabled(\mu_1)$, can be fired resulting in marking μ_2: $\mu_2 = \delta(\mu_1, \; t_{i_1})$.

The execution of a Petri net can be described by a sequence of markings:

$$H_\mu = < \mu_0, \; \mu_1, \; \mu_2, \; ... > \qquad (3.14)$$

An alternative notation to describe the execution of a Petri net is a sequence of transitions, that have been fired:

$$H_t = < t_{i_0}, \; t_{i_1}, \; t_{i_2}, \; ... > \qquad (3.15)$$

The sequences H_μ and H_t are related by $\mu_{k+1} = \delta(\mu_k, \; t_{i_k})$, $k \geq 0$. The execution order given by H_μ or H_t normally represents only one of the possible executions of the marked Petri net: one of the enabled transitions is selected for firing in each marking μ_k. This choice determines the resulting new marking: existing enabled transitions may become disabled and new transitions may become enabled.

Let R be a function, which returns the set of markings that are *immediately reachable* from a marking μ:

$$R(M) = \{ \; \mu' \mid \exists t \in T : \mu' = \delta(\mu, \; t) \; \} \qquad (3.16)$$

The reachable set of a marked petri net M is defined as the reflexive transitive closure of (3.16):

$$R^*(M) = \bigcup_{n \geq 0} R_n(M) \tag{3.17}$$

with

$$R_0(M) = \mu$$
$$R_{k+1}(M) = \{ \mu' \mid \exists \mu \in R_k(M) : (\exists t \in T : \mu' = \delta(\mu, t)) \}$$

The following properties are defined for a marked Petri net $M = (P, T, F, \mu_0)$ with reachable set $R^*(M)$:

- M is *k–bounded,* iff $\forall \mu \in R^*(M) : (\forall p \in P : \mu(p) \leq k)$, i.e. the number of tokens in any place may never exceed k. If $k = 1$, then M is called *safe*. Note that the property of boundedness can be defined for a single place or a subset of the places of the net. Boundedness is a useful property to verify the number of resources that are required to execute the net.
- M is *strictly conservative,* iff

$$\sum_{i=1}^{n} \mu(p_i) = \sum_{i=1}^{n} \mu_0(p_i) \tag{3.18}$$

The reachability set of a marked Petri net can be very large. The state space associated with a marked Petri net consisting of n places, is k^n, if the net is *k–bounded*. Therefore the codomain of the reachability set is $P(k^n)$. The reachability set is often presented as a tree called the *reachability tree*. The root node of this tree is the initial marking of the marked Petri net.

In [Sub86], an interactive tool for modeling and analyzing Petri nets is described. With this tool, a Petri net is executed graphically and the tool can test for properties like safeness, boundedness, conservation, reversability, and deadlock. Many other analysis and verification tools are developed for various types of Petri nets [Fel92]. They often include a net editor and an interactive simulator. The execution or simulation of a Petri net is not only to show one of the possible behaviours of the net, but also to execute a path in the reachability graph that leads to a specific condition in the net. In the latter case, simulation may help to locate and correct the errors found with an analysis tool.

A simple algorithm to execute a Petri net is depicted in algorithm 3.2. The function *get_enabled_transition* selects one out of more enabled transitions and therefore determines the path in the reachibility graph. The function *fire_transition* removes the appropriate number tokens from its input places and produces tokens that are put into its output places. This is followed by adding and deleting transitions from the set of enabled transactions. These actions are equivalent to an update of the marking shown in equation (3.11).

Algorithm 3.2: Executing a Petri net

```
1. initialize_tokens () ;
2. update_enabled_transitions () ;
3. /* marking is valid */
4. while ( t = get_enabled_transition () ) {
5.     fire_transition ( t ) ; /* updates token distribution */
6.     update_enabled_transitions () ;
7.     /* marking is valid */
8. }
```

In (2.11), the simulation state of a model is defined as the 3–tuple (T_{sim}, L_V, E_q). For a Petri net:

- $T_{sim} = 0$. No delays are normally associated with the firing of a transition unless a timed Petri net is simulated.
- $L_V = \mu$. The marking of the Petri net is the state of the simulation model.
- E_q are the transitions which are enabled.

Similar to the simulation of a discrete event model, the simulation of a Petri net proceeds until $E_q = \emptyset$. Note that an arbitrary element from E_q may be selected as the transition to be E_q fired. The firing of a transition may remove or add new elements to E_q.

If a Petri net is used as a submodel of a larger simulation model, it is executed at the current simulation time using delta delays. Its simulation state contributes to the global simulation state as described in section 2.4.1. During simulation, the simulation time of the Petri net always equals the global simulation time. In section 3.4.3, it is discussed how a token flow model is simulated by a discrete event simulator.

3.4.2 Data flow graphs

Data flow graphs are often used as an representation of behaviour in various synthesis problems. The synchronous data flow graph is used to represent the behaviour of a DSP algorithm. In architectural synthesis, various types of data flow graphs are used as an intermediate representation of behaviour [Gaj92].

A data flow graph (DFG) is defined as a directed graph:

$$G = (V, E) \tag{3.19}$$

with

- $V = \{ v_1, v_2, ..., v_n \}, n \geq 0$, the set of nodes. A node represents an operation.
- $E = \{ e_1, e_2, ..., e_m \}, E \subseteq V x V, m \geq 0$, the set of directed edges. Each edge, $e_k = (v_i, v_j)$ represents a FIFO (First In First Out) queue. In general, the firing of the node v_i puts a new token in the queue of e_k and the firing of node v_j removes a token from the queue of e_k.

A theory of data flow graphs is described in [Jon93]. A more detailed definition is used in that thesis. It also adds ports to the definitions of the nodes in a data flow graph. Although the ports of a node are required to describe the firing rules of specific nodes, the definition described in (3.19) is sufficient. In the implementation, the ports are important to be able to execute the behaviour of a node properly.

The set of input edges of a node $v_i \in V$ is

$$I(v_i) = \{ (v_j, v_i) \in E \} \quad (3.20)$$

The set of output edges of a node $v_i \in V$ is

$$O(v_i) = \{ (v_i, v_j) \in E \} \quad (3.21)$$

The firing rules of the nodes in a data flow graph depend on the type of node. In [Kav86], the firing rules of various nodes are divided into 5 different classes: conjunctive, disjunctive, collective, selective and distributive. In [Lee95], each node may have a set of firing rules. A node may fire if and only if one of its firing rules is satisfied.

By analogy with Petri nets, the marking μ of a data flow graph G is a function

$$\mu : E \rightarrow D^* \quad (3.22)$$

where D is a value domain and D^* is the domain of sequences with elements from D. With an edge e_i, an ordered stream of n data values or tokens is associated:

$$\mu(e_i) = < d_{i_1}, d_{i_2}, ..., d_{i_n} >, n \geq 0 \quad (3.23)$$

As an example, the firing rule of an operator node is described. An operation v_i may fire if

$$\forall e_k = (v_j, v_i) : |\mu(e_k)| > 0 \quad (3.24)$$

Firing the operator node results in a new token distribution:

$$\forall e_k = (v_j,\ v_i)\ :\mu(e_k) = < d_{k_2}, d_{k_3}, ..., d_{k_n} > \tag{3.25}$$

$$\forall e_k = (v_i,\ v_j)\ :\mu(e_k) = < d_{k_1}, d_{k_2}, ..., d_{k_n}, d_{\otimes} > \tag{3.26}$$

where $d_{\otimes}$ is the result of applying the operator to the data values of the tokens on the input edges.

A delay is associated with each node in the data flow graph. The actual delay of a node is dependent on the module type that actually implements the behaviour of that node. The function δ is defined with the following domain and co–domain:

$$\delta : V \rightarrow \mathrm{N} \tag{3.27}$$

In analogy with a Petri net, the simulation state of a data flow graph is represented by $(T_{sim},\ L_V,\ E_q)$ with

- T_{sim} the current simulation time. This may remain 0 during simulation, if only the precedence relations are used to simulate the flow graph.
- $L_V = \mu$. The marking of the data flow graph as defined in (3.22).
- E_q are the nodes that may be fired at this time point.

In this section, the ASCIS data flow graph [Eij91] is used as an example. This DFG serves as an intermediate representation used in the architectural synthesis toolbox NEAT. Various (subsets of) hardware description languages, like VHDL and Hardware C [Ku88], and other languages can be translated into a DFG.

In figure 3.12, the ASCIS data flow graph of the calculation of the faculty of a number is depicted. This representation is compiled from the following description in Hardware C:

```
process fac ( input, output )
    in input;
    out output;
{
    output = 1;
    while ( input > 0 ) {
        output = output * input ;
        input = input - 1 ;
    }
}
```

In the figure, the entry and exit nodes are easily identified. These nodes are used to build a loop construct. The output of the operation node > is connected

to the control input of all entry and exit nodes. During initialization, all entry nodes require a token at their control input, which allows external data to be passed from outside to inside the loop construct. After termination of the iteration in the loop construct, a token is left at the control inputs of each entry node. The corresponding tokens have been removed from the exit nodes to be able to pass data from inside to outside the loop construct.

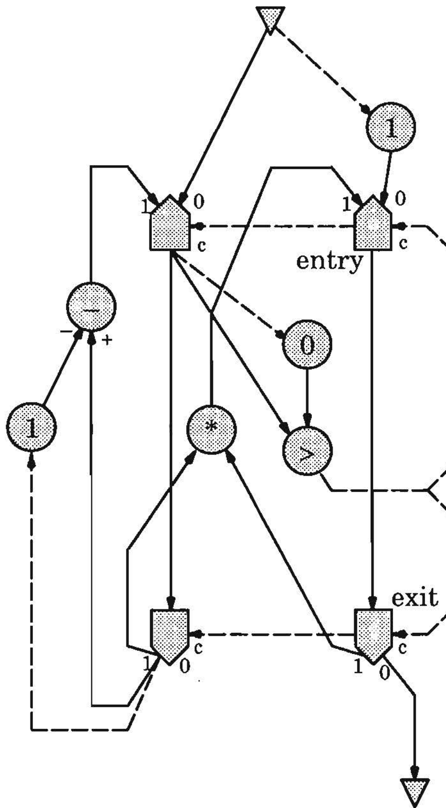


Figure 3.12. A DFG example of the calculation of the faculty of a number.

3.4.3 Simulation of data flow graphs

In section 3.4.1, the simulation of a Petri net has briefly been discussed. In this section, the simulation of token flow models is discussed in more detail. Throughout this section, the ASCIS data flow graph and the Petri net used as examples of a token flow model. The same concepts may be applied to other variants of a token flow model, for instance a coloured Petri net [Jen92] or a

synchronous data flow graph as used in describing signal processing applications [Lee87].

The simulation of data flow graphs may be handled in many different ways. Examples are

- a purely functional simulation satisfying all precedence constraints but without taking timing information into account;
- a simulation taking timing information into account (delay values of nodes);
- a simulation in which the firing of nodes is controlled by a previously derived schedule;
- a simulation in which a control flow graph controls the execution of a data flow graph.

In [Geu93], the development of an interactive simulator for data flow graphs is described: in particular, the ASCIS data flow graph is simulated numerically, symbolically (with some restrictions) or a combination of these two. This simulator operates independently of the event driven simulator and it is embedded into ESCAPE in a similar way as the event driven simulator (see also figure 1.2): it directly accesses the internal representation (network data structure) of the graph, which are kept up–to–date incrementally during an edit session.

This simulator does not take timing information into account. It basically uses a delta delay to simulate the data flow graph. Two event lists are used to orchestrate a proper execution of the graph.

The problem of simulating a data flow graph taking timing information into account is directly related to the problem of scheduling a data flow graph in architectural synthesis. A schedule of a data flow graph specifies at which time point a node should be executed or fired relative to the time point it started execution. A data flow graph $G = (V, E)$ represents a partial order $\prec$ on its nodes $v \in V$ [Sto92].

Scheduling assigns to each node in the data flow graph a start time such that all precedence constraints are preserved and the execution of the complete graph is terminated before a pre–defined time T_{max}:

$$T_{sched} : V \rightarrow \mathbb{N} \quad (3.28)$$

such that

$$v_i \prec v_j \Rightarrow T_{sched}(v_i) < T_{sched}(v_j) \quad (3.29)$$

The *as soon as possible* (ASAP) schedule of a DFG executes the DFG in a minimum time preserving all precedence constraints. The ASAP schedule of a data flow graph $G = (V, E)$ is a function that returns the start time for each node v. It is defined as

$$ASAP(v) = \begin{cases} 0, & \text{if } pred(v) = \emptyset \\ \max\limits_{w \in pred(v)} (ASAP(w) + \delta(w)), & \text{if } pred(v) \neq \emptyset \end{cases} \tag{3.30}$$

For each node v, $ASAP(v)$ returns its start time. Its execution terminates at $ASAP(v) + \delta(v)$. This is called the finish time of a node. An ASAP schedule represents the fastest possible execution time of a data flow graph, that preserves all precedence constraints.

It is easy to verify, that if a token is put on each input node of a data flow graph, each node will be fired at the time point determined by this ASAP schedule after starting its execution.

Another schedule that preserves all precedence constraints, while mininimizing the execution time of a data flow graph is the *as late as possible* schedule. It requires a maximal time T_{max}, in which the data flow graph should be executed. Of course, T_{max} should be equal to or larger than the largest finish time of any node in the data flow graph:

$$\forall v \in V : ASAP(v) + \delta(v) \leq T_{max} \tag{3.31}$$

The ALAP schedule of a data flow graph is a function that returns the finish time for each node v. It is defined as

$$ALAP(v) = \begin{cases} T_{max}, & \text{if } succ(v) = \emptyset \\ \min\limits_{w \in succ(v)} (ALAP(w) - \delta(w)), & \text{if } succ(v) \neq \emptyset \end{cases} \tag{3.32}$$

If a token flow graph is executed by a discrete event simulator, the following tasks need to be handled in the simulator:

- firing the nodes that satisfy the precedence constraints,
- managing the enabling and disabling of nodes,
- mapping the schedule using events such that each node is executed at the proper time point. If no timing information is taken into account, this condition is irrelevant. If no schedule information is used, the data flow graph will execute an ASAP schedule.

Disabling of previously enabled nodes should be possible during token flow simulation. This is illustrated using the Petri net depicted in figure 3.13. It

shows a conflict: the enabled transition t_2 will be disabled because of the firing of transition t_1.

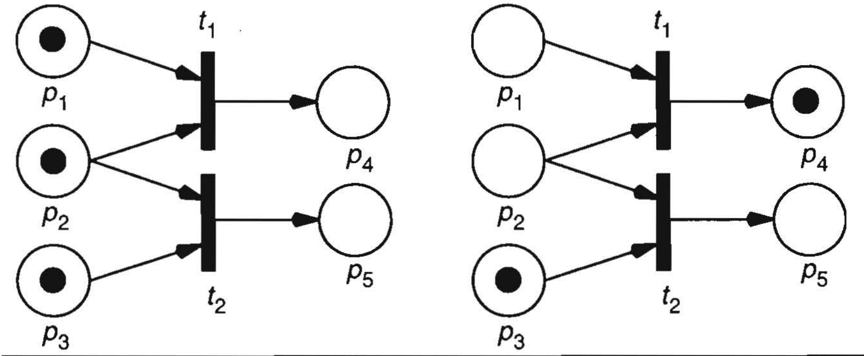


Figure 3.13. A conflict because the firing of transition t_1 disables the firing of transition t_2.

Simulating a token flow graph, an event is defined as:

DEFINITION 3.11: An event indicates a future firing of a token flow node.

This means that the enabling and disabling of nodes should be handled by scheduling events and cancelling events in the discrete event simulator.

In figure 3.14, an embedding technique is applied to use a data flow graph as a discrete event model. The execution of the data flow graph is initialized by changing the value on port $p_{c,in}$. The termination of the execution of the data flow graph is passed to the system by scheduling an event on port $p_{c,out}$. Normally, some protocol is involved in the communication with the controller. For instance, one could also use 4 control ports using a four phase handshake protocol.

The input ports of the data flow graph itself are insensitive to events. An event on a net connected with such an input port never triggers the execution of the DFG. Only an event on the input of the controller may trigger the execution of the DFG. Once, the execution of the DFG has finished, the output value is scheduled as an event on port $p_{d,out}$, but as a refrain event: it does not trigger the execution of other modules connected to this net. The termination of the execution of the DFG is passed to the environment by scheduling an event on the output port of the controller.

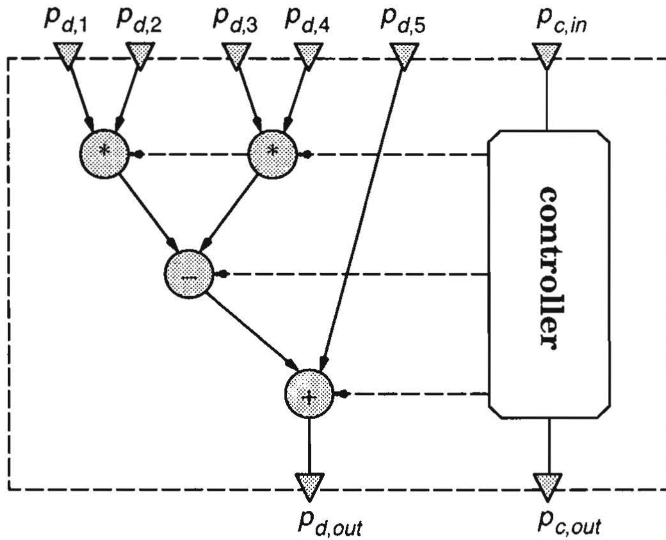


Figure 3.14. A DFG instantiated as a discrete event model

3.5 Run–time configuration of the simulator

In this chapter, it has been described how different models of computation can be included into an event driven driven simulation algorithm. In the previous chapter, various techniques have been described that allow easy prototyping and debugging of simulation models. Summarizing, the following aspects have to be dealt with:

- different event types;
- different timing models;
- different data types and conversion between these types;
- synchronization between slave simulators running independently;
- discrete event automata;
- complex nets and busses;
- animated simulation.

Many of the techniques are dealt with in the simulation algorithm, in particular the event handler. This unnecessarily decreases simulation performance in those runs, in which a particular feature is not used. Therefore, the possibility to adapt or modify the simulation algorithm is very useful.

In figure 3.15, it is depicted how the simulator can be adapted to particular simulation needs. The simulation kernel is generated from a set of options,

that resemble the requirements to simulate a particular model, and the template files. The template files contain constructs that allow to generate code for a particular simulation feature or algorithm. The code is generated either to create a stand–alone batch simulator or to create a module, that can be loaded run–time in the current application. In the latter case, the existing simulation kernel is replaced by a fine–tuned new one.

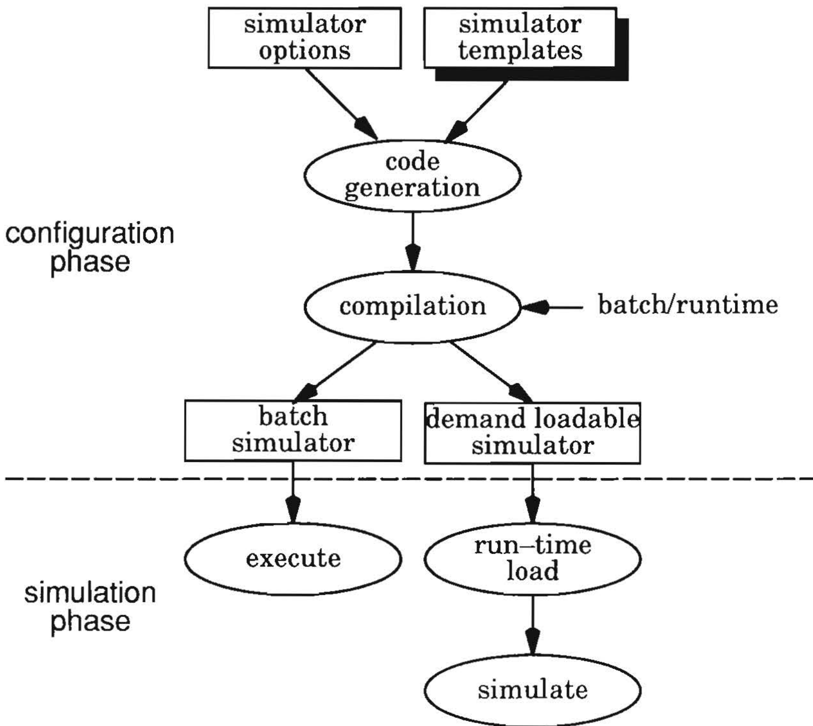


Figure 3.15. Flow to configure a simulator.

This approach is not only useful to adapt the event–driven simulator to specific designer's needs. This may increase overall simulation performance. The simulation algorithm (and thus the event handler) itself can be adapted as well for specific design flows and design implementations. This is illustrated best by the following examples:

- If a detailed timing simulation is performed using floating point values as delays, the next–event approach is better suited as overall timing advance approach. The main loop of the event driven simulator is replaced to accomplish this.

- If the execution order of the modules in the toplevel description of the system can be determined, this order can be compiled to improve simulation performance. In this case, the event driven simulator is replaced by a compiled routine of the execution order of all modules.

Dedicated simulation algorithms may be stored with particular examples for fast and efficient simulation. Such an algorithm is run–time loaded on the demand together with the simulation models of all components. Various options may be chosen to make a trade–off between accuracy and performance. As a result, the simulation environment offers flexibility and customization at various levels of detail:

- the main algorithm of the discrete event simulator can be chosen, for instance oblivious simulation or cycle–based simulation;
- the addition of a new type of simulation models (models of computation), or, the integration of a specific simulation algorithm or simulator, for instance switch level simulation;
- the addition of a new model (of a specific type), for instance a 2–input nand gate at the switch level.

The flexibility and openness of a simulator is a very important aspect during the implementation of the simulator. It is very hard to adapt an existing simulator in order to get a more open architecture. Especially, commercial simulators lack the mechanism to be able to use them with another simulator. They merely provide a foreign language interface to allow the simulation of models written in C. More attention should be paid to provide that functionality in a simulator, that allows easy integration with other simulators. That would give other companies the opportunity to integrate those simulators together that suits their needs for a specific design flow.

Chapter

4 Simulation examples

4.1 Introduction

In the previous chapters, some results have been presented that illustrate the performance and the costs of specific features of ESCAPE. In this chapter, some examples are presented to illustrate the flexibility and simulation capabilities of ESCAPE. The examples presented in sections 4.4 and 4.5 clearly illustrate the advantages of using animation to present simulation results. Both examples rely on a handshake protocol, that is used to move the objects along the road or railroad respectively. The railroad example is based on the traffic example.

4.2 The inner product calculation chip

The inner product $z = \vec{x}.\vec{y}$ of two n–tuple vectors, $\vec{x} = (x_1, x_2, ..., x_n)$ and $\vec{y} = (y_1, y_2, ..., y_n)$, is defined as

$$z = \sum_{i=1}^{n} x_i * y_i \qquad (4.1)$$

The computation of the inner product requires n multiplications and $n - 1$ additions. During this computation $2n - 1$ rounding errors may occur, which can have a great influence on the final result. This problem can be solved by storing each intermediate result using a long accumulator. A long accumulator is a shift register with a full adder connected to it. The length of the shift register is determined by the length of the mantissa, the range of the exponents and an additional number to account for overflows. Note that the addition of a multiplication results can be done serially, which may be too slow, or in parallel, which requires a huge full adder. Therefore, other architectures have been investigated.

In figure 4.1, one architecture is depicted (see also [Fle91] and [Tan92] for more details) consisting of two circular shift registers (rings). Both rings store the intermediate results (summands) of the inner product calculation: one ring stores all positive results and the other ring all negative results. The I/O module calculates the final result from the results stored in both rings.

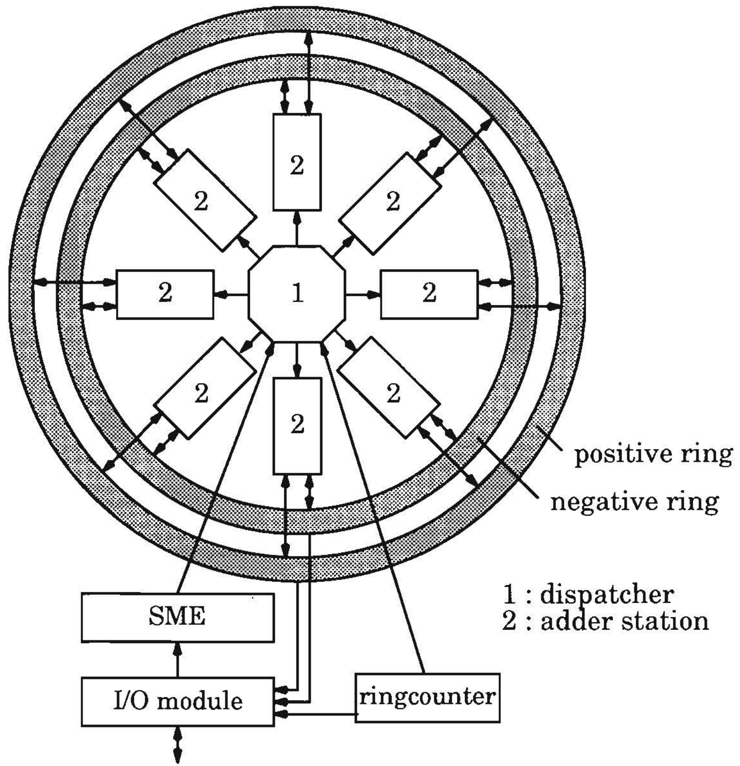


Figure 4.1. Architecture of the inner product calculation chip.

In the rings, data is shifted each clock cycle: in the next clock cycle, the value of the bit shifted in a specific flip flop is doubled (see figure 4.2). The following equation describes the value of the flip flop at position i at clock cycle k:

$$v_i(k) = b_i(k)2^{(i+k+m) \bmod N} \tag{4.2}$$

with $m \in \mathbb{N}$ an arbitrary constant, that determines the LSB/MSB boundary at $t = 0$ and b_i the state of the flip flop i in the ring. Note that $b_{(i+1) \bmod N}(k+1) = b_{i \bmod N}(k)$. At each time point, a specific exponent is associated with each bit in the ring.

The Sign Mantissa Exponent (SME) module takes the ith element from each input vector and calculates the sign, mantissa and exponent of the multiplication of both elements. A number of adder stations are attached to both rings at equidistant places. The mantissa of a new multiplication result is inserted

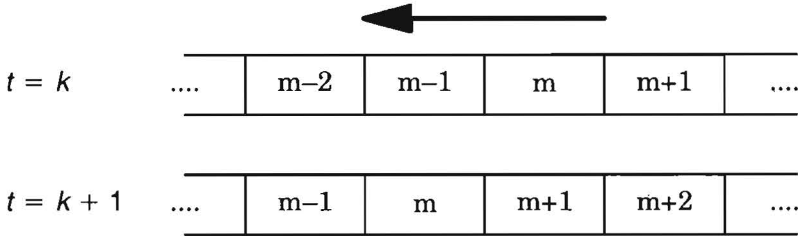


Figure 4.2. Shifting bits with different weight in a ring

by a particular station if the exponent of the result is equal to the exponent of the data at the insertion point in the rings. Depending on the sign of the result, the mantissa is added to either ring. The time between the acceptance of the mantissa and the insertion into the ring is called the hold time.

To be able to add numbers in parallel to a ring, each adder station has a local carry. This also explains why two rings are being used: if a two's complement representation of a number in a single ring is used, all the leading bits of a number have to be toggled if the sign of an intermediate result changes due to an addition. In that case the carry has the ripple through all bits up to the MSB to change the sign of the number blocking the insertion of a number at any station. A sign–magnitude representation can not be used in combination with multiple adder stations, because at all times the sign must be known to each station to decide if a number should be added or subtracted.

The dispatcher is used to determine which station is most appropriate. It calculates the hold times of all stations and selects the station with the minimal hold time, which is not busy yet. To calculate these times, it uses the exponent of a multiplication result and the current position of the LSB in the ring. This position is kept in the ring counter. Once an adder station has been selected, both the hold time and the mantissa are sent to the selected station.

The optimal number of adder stations depends on the desired throughput of data from multiplier to the adder stations and the ring length. The optimal number has been calculated using a C program. The result of this calculation was eight. However, different simulations in ESCAPE showed that this architecture is extremely sensitive to congestion, if an adder station is selected that is already busy. In that case, the next adder station will be selected which increases the hold time. This leads to a decrease in performance. The results of the HDL simulations in ESCAPE and some additional performance analysis have resulted in a new and better architecture for this chip: this architecture guarantees a constant throughput independent of the data.

In table 4.1, some statistics of the models of the various architectures are listed. Kulisch 1 is the original architecture and Kulisch 2 is the improved architecture.

Table 4.1. Statistics on the size of the Kulisch examples.

name	instances	nets	graphics obj.	size [MB]
Kulisch 1	679	843	3512	5
Kulisch 2	706	910	3952	5

In table 4.2, the simulation statistics of these examples are listed. The results have been obtained by executing about 100 runs of 500 simulation cycles. The table contains both the simulation runs with and without animation. This is indicated by the + and – symbols following the name of the benchmark. The following sequences are used:

- +/+ : all objects are updated during simulation;
- –/+ : only instances and wires are updated during simulation;
- –/– : no objects are updated during simulation.

Table 4.2. Simulation statistics and performance of the Kulisch examples.

name	av. # events [* 1000]	av. # evaluations [* 1000]	av. # draw requests [* 1000]	av. elapsed cpu time [sec]
Kulisch1 +/+	68.7	39.1	293.0	9.4
Kulisch1 –/+	68.9	39.1	156.2	8.3
Kulisch1 –/–	68.3	39.0	0.0	5.0
Kulisch2 +/+	131.6	64.1	712.9	14.3
Kulisch2 –/+	131.6	64.1	476.4	10.8
Kulisch2 –/–	131.6	64.1	0.0	6.3

These examples generate an extremely high number of draw requests during simulation, which strongly influences the simulation performance.

4.3 The bit blitter

A bit blitter (block image transferrer) is a graphics processor that is designed to handle blocks of bitplane data. It improves on the performance of a general purpose processor in manipulating blocks of bitplane data. A simulation model of this processor has been developed using the LISP HDL described in appendix A, this as a part of the test case described in [Phi88].

Some of the features of the blitter are:

- it can use a bi–directional addressing scheme,
- it can perform a logic operation on data before transferring of that data,
- it can copy data,
- it can shift two of its data sources,
- it can perform an area fill.

In figure 4.3, the simulation model of the bit blitter is depicted. It consists of two parts: the blitter itself including some additional models for simulation control and a model of part of a display.

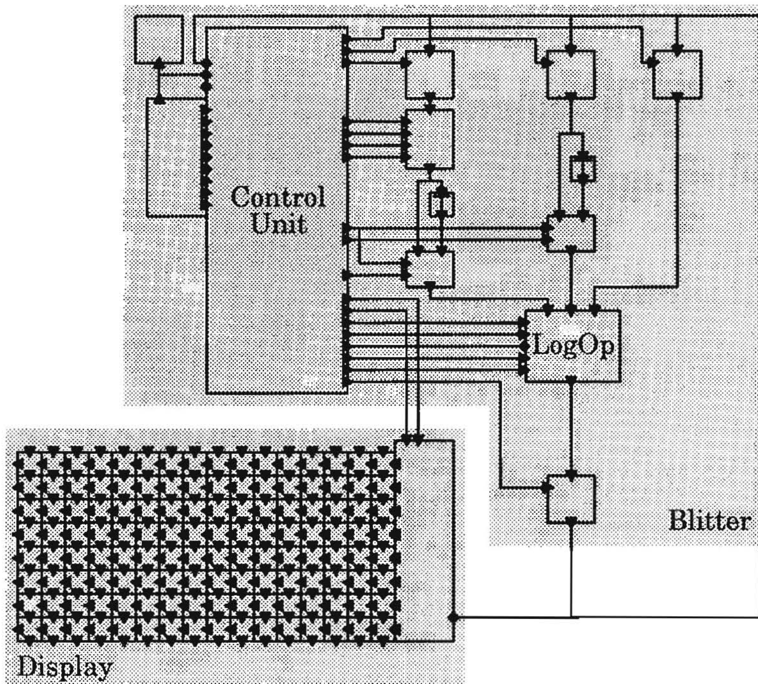


Figure 4.3. Simulation model of the blitter

In appendix B, some snapshots of a simulation of the bit blitter are depicted. Some of the features of the blitter are visualized in these snapshots.

4.4 Simulation of traffic on a road

In this example, the road is composed of different modules with 4 ports, that are used to communicate data, which controls the behaviour of the cars. Each module represents a road segment, which has a notion of the presence of a car. Each car has a random speed, which is modeled using a random delay for propagation to the next road segment (module). They maintain a minimum

distance of one road segment, even if cars have to stop for a red traffic light. The behaviour of a road segment is:

```
(behaviour road_seg0
    (term in)
    (term out)
    (term ack_in)
    (term ack_out)
    (state car 0)
    (state sched 1)
    ;; car drives into this segment
    (if (and in (not car)
        (progn
        (setq car 1)))
    ;; request to move forward
    (if car
        (progn
            (color 1)
            (delay 0 ack_out 1)))
; schedule car to move
    (if (and sched car (not ack_in))
        (progn
            (delay (+ 1 (random 5)) out 1)
            (setq sched 0)))
    ; carmoves to next segment
    (if ack_in
        (progn
            (color 0)
            (delay 0 out 0)
            (delay 0 ack_out 0)
            (setq sched 1)
            (setq car 0)))
)
```

A simulation snapshot of this model is presented in figure 4.4. It clearly shows the cars that are moving in one lane and the single car that is moving in the other lane. The cars in the first lane pass a point in the road, which will possible generate a new car after each pass of another car after a random delay. The actual generation only occurs, if the associated road segment does not already contain another car. This results in an increasing number of cars in this lane to the point in time, where no new car can be generated anymore. Once the lane contains many cars, the random speed of the cars clearly shows the conditions and overall behaviour that occurs in a traffic jam.

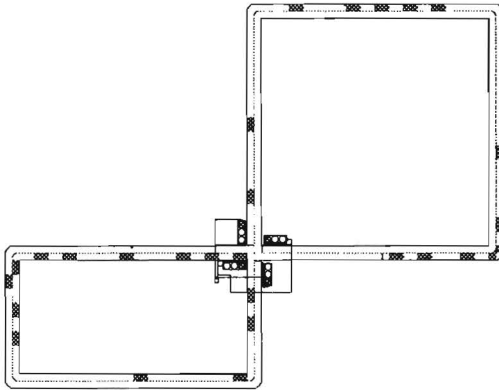


Figure 4.4. Snapshot of the simulation of road traffic

The simulation performance is directly related to the number of events that need to be processed every clock cycle. This number is dependent on the number of cars that are moving. Because new cars are generated during simulation, the number of events and computational time is increasing as simulation time passes. To give an impression, a number of benchmarks have been performed with a constant number of cars. This is shown in table 4.3.

Table 4.3. Traffic benchmarks with animation

# cars	Av. # events	Av. # draw calls	Av. cpu time
2	2900	20100	0.81
4	4800	38100	1.31
6	6800	56200	1.55
8	8600	73400	2.11
10	10600	91800	2.78

In table 4.4, some simulation data is listed without using animation during simulation. Instead of the number of draw calls, the average number of evaluations is listed.

Table 4.5. Traffic benchmarks without animation

# cars	Av. # events	Av. # evals	Av. cpu time
2	2900	1900	0.51
4	4800	3100	0.67
6	6700	4300	0.73
8	8600	5500	0.82
10	10500	6700	1.00

4.5 Model of a railroad and block control system

This example shows the advantage of animated simulation: the correct behaviour of the overall simulation model is shown by highlighting the modules, if the internal state indicates the presence of a train.

The building blocks of this example are described in more detail in [Nie91] and are composed of a number of primitive modules. This example contains models on two levels: the track level and the control level. The track level is used to simulate trains. which move along the tracks. The control level handles the signals that guard each block and the switches that determine the route of the trains.

In figure 4.5, three components of a track are depicted. Each component of the track has a state variable, which indicates the presence of a train. The ports of these components are connected using abutment. Each component has four ports, that are used by the handshake protocol that *moves* the trains along the track. The length of a train is initialized during the initialization of the model: a module will execute the handshake protocol to move the train *number of train's length* times. The number of trains and the length of each train in the model therefore determine the simulation performance. Two ports for communicating the presence of a train to the control level (i.e. in_L_occ and out_R_occ).

The largest example, which has been composed, is an abstract map of the London Underground consisting of more than half a million graphics objects (see figure 4.6). In this example, hierarchy is used to enable crossings of different Underground lines; the objects modelling these parts of the railroad lines can easily be identified: the parallelograms and rectangles. The network view of these modules models the railroad, which are simulated at a lower level in the hierarchy. The trains moving on these parts of the tracks can be visualized in a separate window. The visualization of lower levels in the hierarchy may be selected during the simulation. This example clearly shows that very large examples can be captured and simulated by ESCAPE.

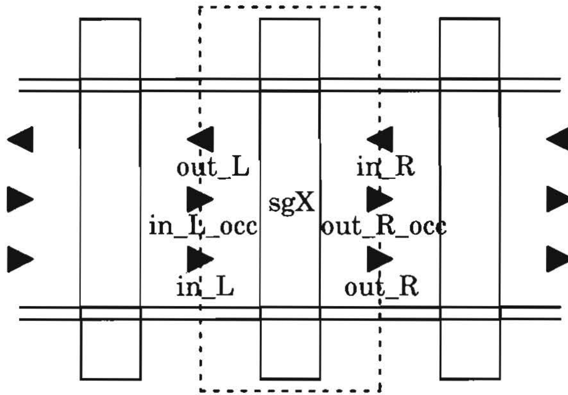


Figure 4.5. Modelling of tracks and trains moving along it.

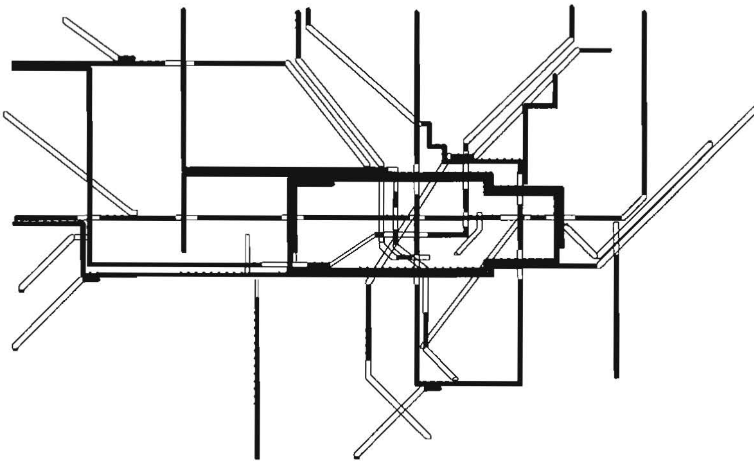


Figure 4.6. Map of London Underground

In figure 4.7, a snapshot is depicted that has been taken during the simulation of this example. One can easily identify five trains, that are moving along the tracks and the signals that show the presence of a train in a particular track. All signals are set correctly depending on the speed and direction of the trains. As a result, a train will wait for free tracks to prevent collisions. The speed of a train and the direction, if more than one choice is possible, are chosen randomly during simulation. Despite this, the control level manages to control the trains without deadlock and collisions.

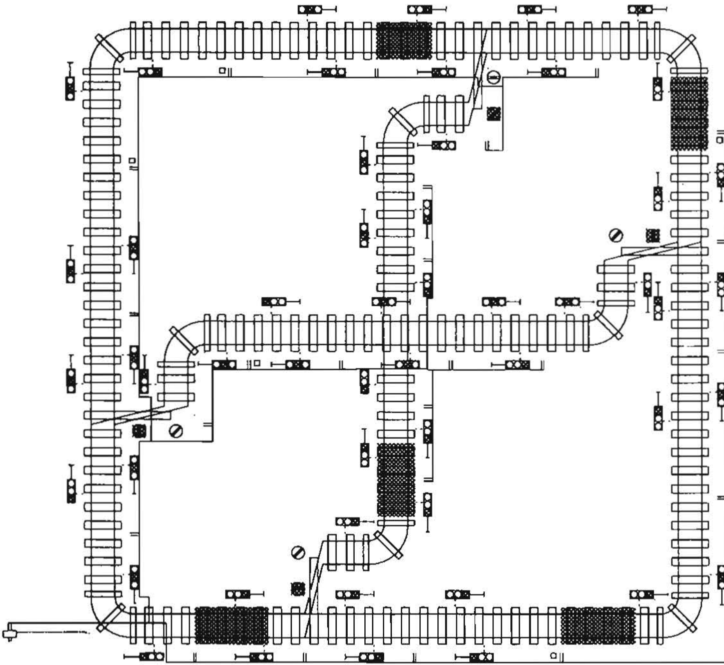


Figure 4.7. Simulation snapshot of railroad example.

The simulation performance of the model is related to the number of events per time step. This depends on:

- the number of control blocks;
- the number of trains;
- the length of the trains;
- the average length of the blocks used in the example;
- the average speed of the trains.

The speed of the trains is modeled using different delay times. These times are chosen randomly, but depend on the signals along the track. The speed averages a constant value for the different benchmarks.

In figure 4.9, the average number of events as a function of the number of trains is depicted for three different number of blocks.

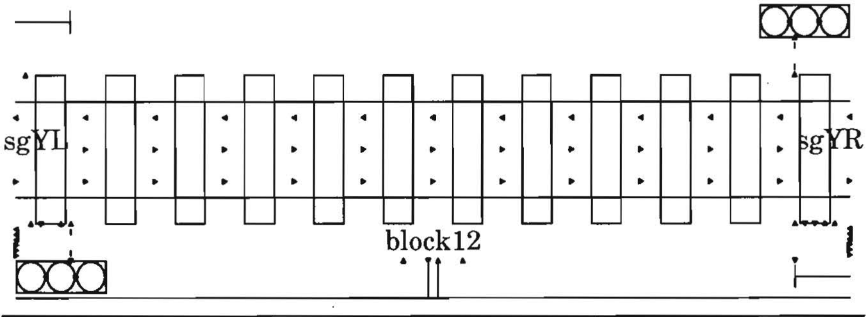


Figure 4.8. A block consisting of the track components, a control block and two signals

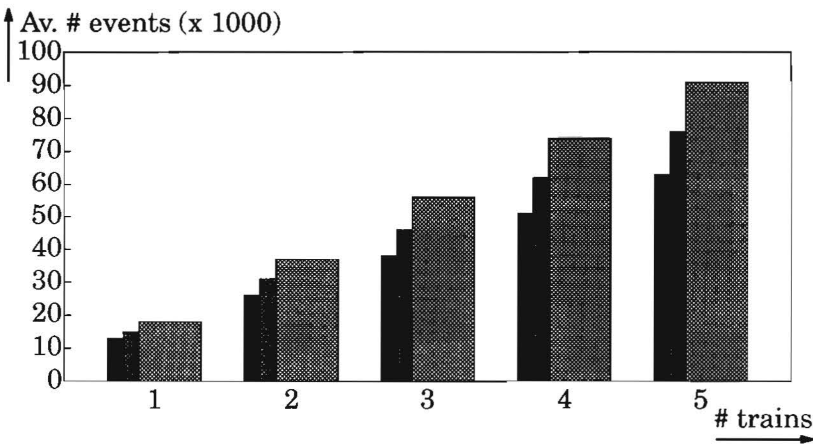


Figure 4.9. Average number of events for various number of blocks and number of trains.

In figure 4.10, the average number of events as a function of the length of the train is depicted: the linear increase is expected, because an active track module is performing the handshake protocol *length of train* times.

In figure 4.11, the average number of events is shown for a constant track length composed of blocks with different size. It clearly shows that it is more efficient to build tracks using large blocks.

The simulation performance of each *railroad* example is determined by the size of the model and the number of trains. Various examples that vary considerably in size and complexity, have been edited and simulated using ES-

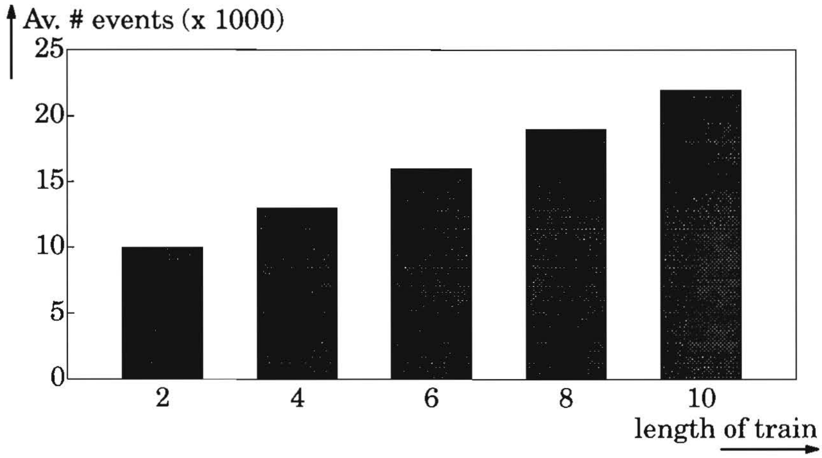


Figure 4.10. Average number of events for various lengths of the trains

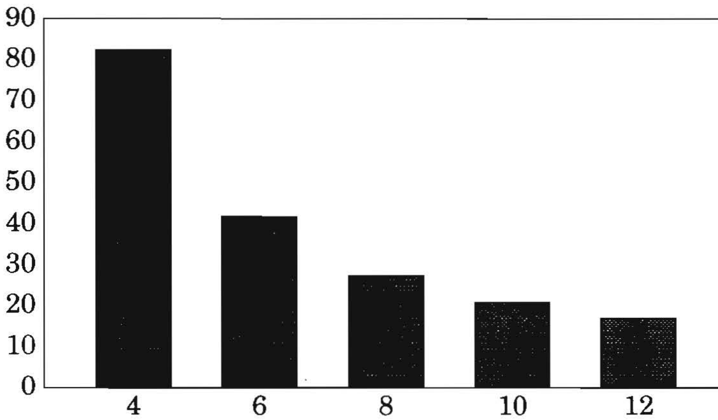


Figure 4.11. Average number of events for various block lengths

CAPE. Table 4.6 lists some statistics of various examples. Note that the number of instances, nets and graphics objects are the totals for the model itself and they do not include the designs the model is composed of. The size of the model includes all models that are required to build the simulation model and also includes the event list and the maximum number of events used during simulation.

Table 4.6. Statistics on the size of some railroad examples.

name	instances	nets	graphics obj.	size [MB]
small	483	1546	5164	7

small2	848	2657	8954	8
large	8114	22879	80253	31
London1	23719	70951	245586	87
London2	24894	71400	247071	86
London3	54170	154201	536222	229

In table 4.7, some statistics of the simulation runs and the run times of the various railroad examples are listed. As some parameters of these examples are modified randomly during simulation, these numbers have been for a number of consecutive runs of intervals of 5000 clock cycles (the largest delay value used in the model is 100). The number of runs, which have been used to average the overall results, have been at least 100.

Table 4.7. Simulation statistics and performance of the railroad examples.

name	av. # events [* 1000]	av. # evaluations [* 1000]	av. # draw requests [* 1000]	av. elapsed cpu time [sec]
small	14.3	2.4	14.3	1.5
small2	24.8	3.9	25.0	2.0
large	16.0	2.9	9.0	3.7
London1	62.2	14.0	23.7	8.5
London2	195.6	41.8	85.8	18.2
London3	50.5	9.9	19.0	7.5

Chapter

5 Design process integration

5.1 Introduction

In the previous chapters, different aspects of the simulation of heterogeneous systems have been described. However, the simulation (validation) of systems is only one of the activities in the design flow. To be of real use, a tool like ESCAPE needs to be tightly integrated with other tools in the design flow. To achieve this, ESCAPE is extended with some functionality for easy customization and tool integration in different application areas. Some other advantages of an open architecture are:

- the user interface is configurable to different needs;
- the built–in functionality is easy accessible by other tools;
- the internal data is accessible by other tools;
- the process of integrating a simulator or another design tool may be simplified.

Related work in this area are the extension and customization languages, that are provided with some commercial CAD systems. An example of such a language is SKILL [Bar90], that is provided with Cadence Design Framework II. SKILL is a fully featured programming language based on the LISP language [Ste84]. Besides many of the function found in LISP, SKILL also includes functions to access the underlying CAD framework and tools, and functions to build user interface components. It has proven to be a very useful language; although it was originally intended for customization purposes and small additions of functionality only, it has been used to develop large programs as well.

In this chapter a number of features are described, which allow customization and extension of ESCAPE itself as well as integration with other applications or tools. For instance, ESCAPE has been used in combination with the following tools:

- a term rewrite program to simplify the expressions generated during the **symbolic** simulation of data flow graphs;

- some formal verification tools to visualize output results by annotation onto a schematic (network view);
- the tools for place and route of layout data;
- the architectural synthesis tools of the NEAT system (see section 5.6).

5.2 A programmable graph view

Graphs are used to capture different aspects of a design at all abstraction levels (from the layout to the system level) in the design trajectory. At the higher abstraction levels, they are used to capture system structure and behaviour. Examples are data flow graphs, control flow graphs, Petri nets, finite state machines and state charts. Each graph formalism differs with respect to its structural representation and its graphical representation. This is referred to as the type (definition) of a graph. Therefore, the graph view, unlike the other views in ESCAPE, is customizable and even programmable. As a result, ESCAPE may be used as a graph editor that serves as a user interface to many other application areas as well.

The graph definition language is a language using the syntax of the LISP language. It is used to specify the structural and graphical attributes of a specific type of graph, its nodes and its edges. This information is used to build the graphics and the network data structure of a graph of this type.

As an example, a simple graph definition of a Petri net is as follows:

```
(graph-definition "petri"
   (scale 1000)
   (directed)

   ;; nodes
   (node "transition"
      (fill-color "black")
      (bounding-object (poly (-10 -1) (-10 1) (10 1) (10 -1) (-10 -1)))
   )
   (node "place"
      (fill-color "black")
      (bounding-object (circle 0 0 8))
   )

   ;; edges are not explicitly specified
)
```

In this graph definition file, two node types have been defined: a transition and a place. In particular, the graphical representations of the nodes are specified. It is also specified that a Petri net is a directed graph.

Another definition of a node is the *branch* in the data flow graph definition file (see figure 5.1).

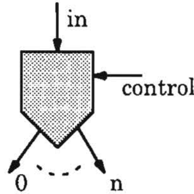


Figure 5.1. Branch node of a data flow graph

```
....
(node "branch"
   ....
   (ports
      (input "control")
      (input "in")
      (output "selected" (props (prop ("variable"))))
   )
   (function "dfgsim_branch")
)
....
```

In the data flow graph definition file, the ports of each node need to be specified as well. The *branch* node may have a variable number of output ports. This depends on the number of branches that may be selected using the *control* input port. In addition, a function is specified for each node type: this is the function that is called during simulation of a data flow graph.

The flow for visualizing and manipulating different types of graphs is depicted in figure 5.2. The graph definition files are read by ESCAPE to be able to visualize all nodes and edges (graphics data) and to pass properties for the internal data structures. Even specific functions may be passed through a definition file. The structure of a graph is read by a graph placement tool and the data resulting from this placement is sent to ESCAPE using the inter tool protocol (see section 5.3). The information stored in the graph definition file could be used by the placement algorithm to find a better placement. Therefore, some placement programs will read this data as well.

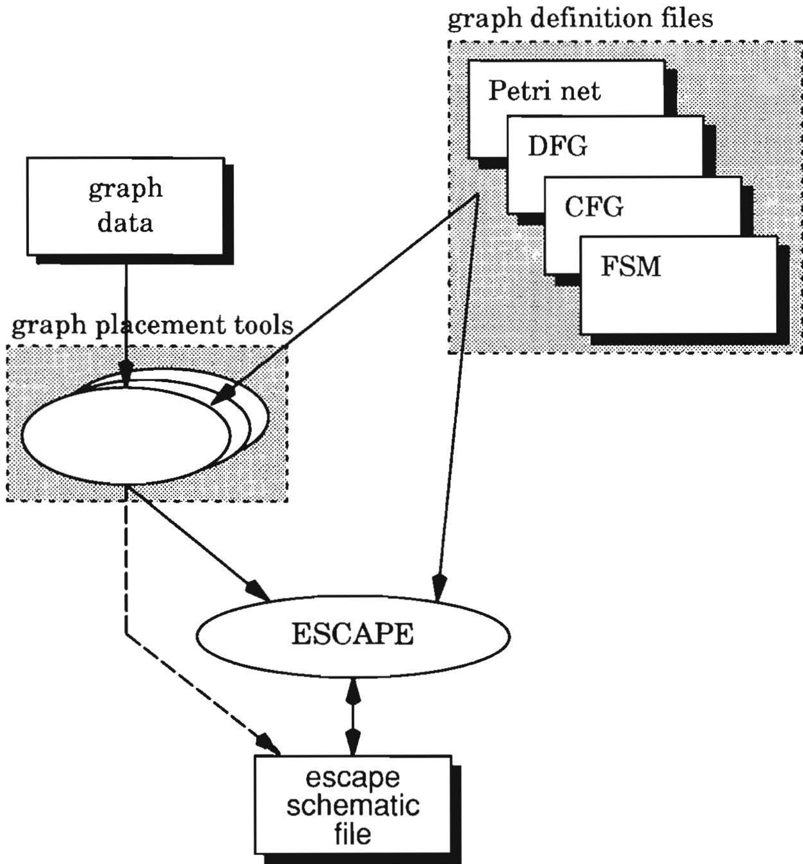


Figure 5.2. Customizing the graph view

Different placement algorithms could be used depending on the structure of the graph. Therefore, it is anticipated that many placement programs will be provided. This is one of the reasons to implement these placement tools as stand–alone tools. In principle, the designer can select one of the available graph placement tools for the placement of a specific graph. Many algorithms have been developed for the placement of different types of graphs. An overview is given in [Ead89]. Two algorithms have been implemented for the placement of the ASCIS data flow graph:

- a dedicated placement algorithm that exploits specific properties of the structure of a data flow graph;
- an algorithm based on an algorithm published in [Sug81], which minimizes the number of edge–crossings in the graph.

After placement, the coordinates of all nodes are sent to ESCAPE, which is then able to draw the graph using both the information from the placement as well as the graphical representations of the nodes and edges as specified in the graph definition file. Of course, the resulting layout but also its structure (connectivity) may be manipulated afterwards; a graph may also be captured interactively using the graphics editor.

Related work in this area is the interface description language described in [New88], which is developed for an extendible directed graph editor called EDGE.

5.3 Inter process communication and the inter tool protocol

An inter process communication (IPC) module is build into ESCAPE to be able to embed it in an existing CAD environment (e.g. a framework) or to connect external tools with it. This IPC module allows the communication between ESCAPE and other applications running on the same machine and other machines in the network as well as the communication between various ESCAPE programs running on possibly different machines. The processing of incoming communication messages is tightly integrated with the processing of graphical events. This even allows the processing of communication messages during a simulation run.

In figure 5.3, the interaction between ESCAPE and other tools on various hosts is depicted. ESCAPE is able to set up communication with other tools on the same machine directly or on any other machine running the daemon program. Note that the daemon is only involved in setting up the communication between an external tool and escape. If this succeeds, the tool communicates directly with ESCAPE.

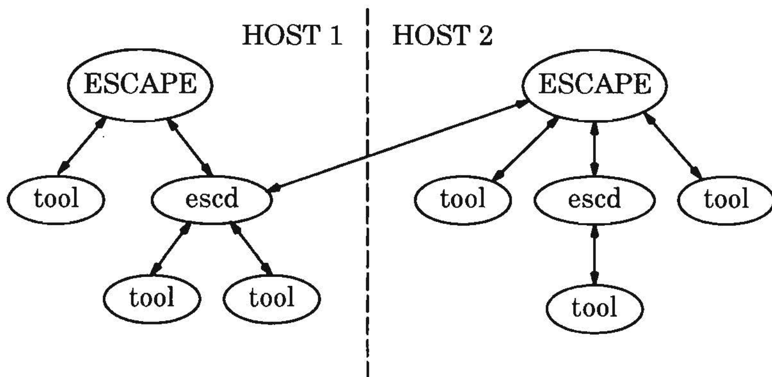


Figure 5.3. Inter process communication with ESCAPE.

ESCAPE can be used in two major modes in combination with other applications:

- *Slave* mode: another application invokes ESCAPE and controls its execution. In this mode, ESCAPE serves as a graphical backend for other applications.
- *Master* mode: the execution of one or more applications is controlled by ESCAPE. Both batch and interactive tools are controllable from ESCAPE using either a synchronous or an asynchronous communication strategy.

An ASCII protocol called the inter tool protocol (ITP) provides access to the application procedural interface (API) of ESCAPE: it provides access to internal functions and variables. A textual description has been chosen to serve as an interface to the API, because it is less sensitive to errors. If a procedural interface is used, an error may result in program termination. The line oriented ITP allows easy recovery from errors in syntax and arguments. Another advantage of an ASCII protocol is that an existing tool is easily adapted for reading and writing messages to ESCAPE.

A summary of the ITP is given in appendix C. The ITP includes different kinds of commands, for instance:

- user interface and control commands built in ESCAPE,
- commands to set up and to control the simulator,
- commands to manipulate views and objects in these views
- commands to visualize data associated with objects or the objects themselves,
- commands to provide user interaction (e.g. selection lists, messages)
- and even run–time specification of graph definition files.

The concept of a *context* is introduced that determine if a command may be executed and which data it should affect. Each time a new context is opened it is pushed on a stack. Only commands that may be executed within that context are executed and all other commands are discarded. If the current context is closed, it is popped from the stack and the previous context is restored. This concept makes it also possible to re–use commands for different purposes. For instance, the *color* command may be used in different context to set the current colour for drawing purposes.

The following text shows an excerpt from the ITP communication between an external tool and ESCAPE to send graph definition data and the graph itself:

```
graph sample dynamic
  ....
```

```
  # node definition
  node–definition mynode
    bounding–object box 0 0 8 4
    add–box 0 0 8 4
    add–text 1 1
      text name = <name>
      align center
    end–add–text
    color fill green
    property myvar
  end–node–definition
  ....
  # node data
  add–node 1 1 n1 mynode
  add–node 3 3 n2 mynode
  add–node 8 1 n3 mynode
  ....
  add–hyper–edge someedge net
    nodes name n1 n2 n3
    add–line 1 1 3 3
    add–line 1 1 8 1
  end–add–hyper–edge
  ....
end–graph
set–graph <none> dynamic
....
view–link myset
  link graph sample type dynamic node n1
  link graph sample type dynamic edge someedge
end–view–link
```

In figure 5.4, the organization of the inter tool communication is depicted. An external tool writes messages to the IPC channel, which are read by the ITP parser built in ESCAPE. Once a complete command is parsed, the appropriate function in the API is called. This function may access the internal data structures. To communicate with an external tool, ESCAPE formats a specific messages and sends it via the appropriate IPC channel.

5.4 Configuration of the user interface

The open architecture of ESCAPE can only be exploited, if the user interface is customizable as well. A customization language has been developed, that may be used to adapt and to extend the user interface to specific needs. The language also serves as a tool encapsulation language. Using this language, ESCAPE is easily interfaced with other CAD tools or environments.

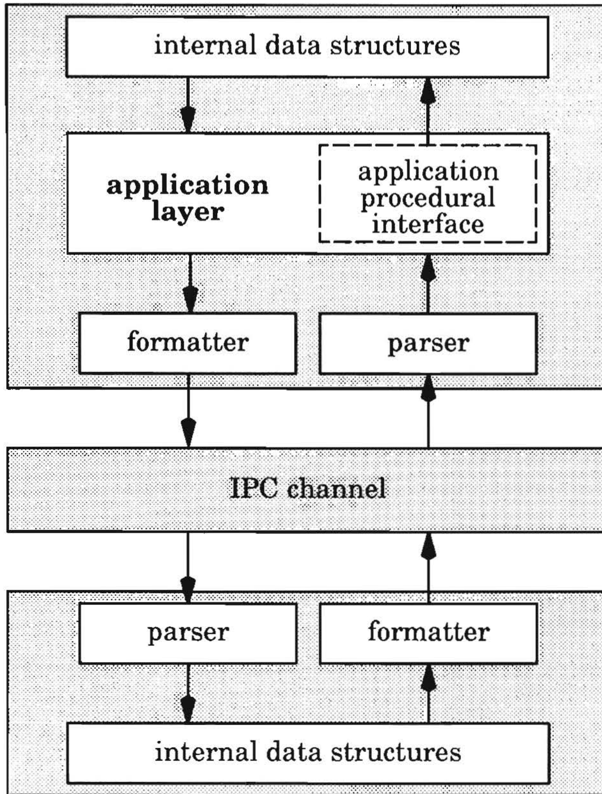


Figure 5.4. Inter tool communication with ESCAPE

The language to customize the user interface provides the following features:

- variables and some programming constructs,
- interactive editing of commands and queries,
- access to internal functions,
- invocation of external programs using various communication schemes,
- synchronization mechanisms.

In figure 5.5, an excerpt from a definition of the menu to interface with the NEAT system is shown. The *set* command is used to set the value of a variable. A few variables are already defined in ESCAPE. The value of a variable can also be set interactively. This is shown in the *actions* part of the definition of a push button.

In the strings specifying commands, variables are specified between < and >. The parser replaces the variable during the execution of the command by its actual value.

```
;; variable declarations
(set interactive false)
(set neatbin "./neat/bin")
(set designpath ".")
....
;; the pulldown menu
(pulldown "neat)
   (resources
      (label-string "Neat")
      (mnemonic "N")
   )
   ....
   ;; a push button
   (push "hc2dfg"
      (actions
         (set prompt (variable interactive))
         (extern    "<neatbin>/hc2asd -i <designpath>/<basename>.hc "
                    "-o <designpath>/<basename>.asd"
         )
         (set prompt false)
         (extern    "<dfgread> -v -d -g <graphdefs> "
                    "<designpath>/<basename.asd"
                    (io-mode read)
         )
      )
      (resources
         (label-string "hc -> dfg")
         (mnemonic "h")
      )
   )
)
```

Figure 5.5. Part of a menu definition

5.5 The interactive data access language

An interactive data languages has been developed that provides programming capabilities and read access to the internal data structures of ESCAPE [Luk92]. This interactive data access language may be used for the following purposes:

- to add new functionality to a program in an interactive and easy way;
- to safely prototype new algorithms that use the existing data structure of ESCAPE;
- to inspect data stored in the internal data structures of ESCAPE;
- to generate complex structural descriptions; in that case, a combination of both procedural and graphical constructs (the graphics editor) can be used to describe structure. An example of such a system is described in [Ebe89].

Although the LISP interpreter has been developed to be used with ESCAPE, it may be used with any other C program as well. The interpreter provides a general interface to access to the data structures of a C program. To enable this, the interpreter needs to be linked with the C program. The following steps have to be performed to use the interpreter and to access the data structures and variables of the C program:

- The C program explicitly *publishes* the types, functions and variables that the user wants to access using the interpreter. In the LISP interpreter, these C objects have a special type and are treated differently from other types.
- The interpreter is initialized.
- The LISP objects that embed the C objects are created and bound to user–defined names. All functions, that have been published, are added to the interpreter as well and may be used as any regular LISP function.

Once this process is completed, the user may use the interpreter to develop programs in the LISP language or to load and execute functions that have been developed previously. Within the interpreter, full read access is provided to the data structures and variables of the C program that have been published.

5.6 An example: the integration of NEAT and ESCAPE

NEAT (New Eindhoven Architectural synthesis Tool–box) is an open interface for architectural synthesis [Hei94]. NEAT uses three graph representations to capture all data during the various synthesis steps:

- The ASCIS data flow graph is used to describe the behaviour of a design. Usually, a data flow graph is the intermediate representation of the behaviour of a design generated from a hardware description language like Silage, VHDL or Hardware C.
- The control (flow) graph is a representation of a finite state machine. Each node represents a state, whereas the edges specify the threads of control (state transitions).

- The network graph is used to describe the structure of a design: it is a representation of the design at the register transfer level.

Furthermore, *inter–* and *intra–relations* are used to represent the relationships between different objects in the various graphs. These relations are generated by the various synthesis tools.

The following actions have been performed to enable the interaction between NEAT and ESCAPE:

- The definition of the three graph representations used in the NEAT environment: the data flow graph, the control graph and the network graph.
- The development of a dedicated placement tool for data flow graphs, which places control flow graphs and network graphs as well. This tool reads the ASCII data format used in the NEAT system and generates ITP messages that are read by ESCAPE.

This allows to store the graphs in ESCAPE and to present them graphically. Not only the graphs are stored but also the inter– and intra–domain relations between these graphs and their objects.

ESCAPE has been used both in master as in slave mode to interact with the tools of the NEAT system. In master mode, tools in the NEAT system are invoked from the user interface of ESCAPE. The user interface to control NEAT has been defined using the UI configuration language. Once, the tool has terminated successfully, the results are read by the graph placement program, which sends the graphs and their relations to ESCAPE. Then the designer can inspect the results on the screen.

In slave mode, ESCAPE is invoked from another tool and its standard input/output streams may be used to send a series of ITP commands to ESCAPE, which transfers all design data from the synthesis tool to ESCAPE.

The experiments to integrate NEAT and ESCAPE have shown that is very useful to have a graphics backend for synthesis tools. The backend is able to visualize all synthesis data. A designer can inspect all data and interactively validate that the results are correct. Besides tool control and visualization, such a backend may also be used to animate the behaviour of a tool onto the data during synthesis. For instance, the progress of the scheduler may be visualized. It is even possible to adapt the scheduler program to allow user control during scheduling.

Chapter 6 Concluding remarks

In this thesis, some concepts and techniques applied to a discrete event simulation tool are described that

- reduce the design cycle time to facilitate prototyping, debugging and design space exploration,
- allow to compose a homogeneous discrete event simulation model from different types of simulation models.

A prototype of a flexible and interactive simulation environment, called ESCAPE, has been developed, in which these concepts and techniques have been incorporated. Many experiments have shown its usefulness for modelling and simulating a great variety of examples. It can easily be adapted to fulfil different simulation needs not only for CAD applications but for other real world applications as well. The prototype is not only used as a modelling and simulation tool but as a generic frontend to other CAD tools as well, because of its flexibility and openness.

The reduction of simulation time and overall design time has been tackled by tightly integrating the various phases of design and simulation. To be more specific, the following features have been provided:

- Incremental updating of and direct access to internal data structures.
- Support of higly interactive interaction models between the simulator and the editor.
- Animation/visualization of system behaviour during simulation.
- Support of discrete event monitors to detect erroneous behaviour at higher levels of abstraction.

The integration of various types of models such that is possible to simulate heterogeneous systems, is based on the idea to map different simulation algorithms onto the event driven simulation paradigm. Different types of events are used to model different delay models and interconnect behaviour, to control and to synchronize multiple simulators, and to execute user–defined functions at specific time points. Besides events, an external simulator inter-

face and a model encapsulation technique are used to simulate different types of models homogeneously.

The open architecture of the simulator allows for extension at three different levels of detail:

- new models of a specific type,
- new types of models,
- the simulation algorithm itself.

Note that most simulators claim to be flexible if they just support the addition of new models of a specific type (as opposed to using only predefined components from a number of libraries).

Besides its use as a modelling and simulation tool, ESCAPE may be used as a generic user frontend for other CAD tools. The flexibility of the user interface and its graphics capabilities are used to control CAD tools and to visualize output results of these tools. Especially, its use in combination with the architectural synthesis toolbox NEAT has proven to be very successful.

Suggestions for future work

Many suggestions can be made to enhance the prototype. In the remainder of this section, a few important ideas are proposed that need to be investigated to be able to decide on the usefulness of such a tool as a product for real life design projects. Using such a project as an example may help to identify potential deficits. It may also be worthwhile to apply the concepts described in this thesis to a commercially available simulator. The prerequisites for such an approach are described here as well.

First, a few inadequacies of the current prototype are described that need some additional research.

In ESCAPE, events are used for many purposes and the efficient management of these events is important to limit the decrease in performance. Flexibility and openness in general leads to a performance penalty in simulation speed. Events are managed in different data structures to fulfil different tasks during simulation, for instance:

- the master event queue to process events in a non–decreasing order,
- the storage of events and state variables to perform incremental simulation,
- the storage of events and state variables to be able to restore a previous simulation state.

It is very important to reduce the number of data that has to be managed during simulation, as it has costs in both performance and memory usage. This is a topic that may be investigated more thorough in the near future.

ESCAPE lacks support to manage design data. This manifests itself in setting up simulation experiments involving hierarchy, alternatives and multiple types of simulation models. A real topic for future research is how the management of designs may be combined with an incremental update approach of the simulation model as currently used in ESCAPE. This support is required to be able to use such a tool in real life design practices. This problem has not been addressed yet, because it should be dealt by CAD frameworks.

As stated before, many of the concepts and techniques described in this thesis may be applied to an existing commercial (event driven) simulator as well. A commercial simulator may serve as a basis on top of which software modules are developed that allow the simulation of different models of computation that interface with the underlying simulation paradigm. Most simulators today include a foreign language interface, but the openness and flexibility of such an interface are very limited. It basically allows to include C models in a very primitive manner. It could however be investigated, if this is sufficient to integrate more complex models or even other simulators.

Of course, it would even be better if commercial vendors would put simulators on the market that have an open architecture instead of a foreign language interface. Such an architecture should allow to include models simulated on that simulator as a submodel of another simulator. Although such simulators are not on the market today, it is possible to integrate a commercial simulator with ESCAPE using its foreign language interface. However, this approach seems to be restricted to a single simulator to avoid deadlock and synchronization problems. It may be interesting to investigate this problem in more detail, since re–use of existing simulators to build a heterogeneous simulators is very cost effective.

The last suggestion concerns the use of ESCAPE as a generic user frontend. In the prototype, many of the features to customize ESCAPE are developed in isolation. A good alternative would be to provide a programming language that offers all these features including full access to the data structures and functionality of the tool.

References

[All90] ALLEN, P.E., B.P. LUM SHUE CHAN and W.M. ZUBEREK, "Comparison of Mixed Analog–Digital Simulators," in *Proceedings 1990 IEEE International Symposium on Circuits and Systems*, pp. 101–104, New Orleans, Louisiana, May 1–3, 1990.

[Aug90] AUGUSTIN, L.M., D.C. LUCKHAM, B.A. GENNART, Y. HUH and A.G. STANCULESCU, "Hardware Design and Simulation in VAL/VHDL," Kluwer Academic Publishers, Dordrecht, 1990.

[Bai94] BAILEY, M.L., J.V. BRINER JR. and R.D. CHAMBERLAIN, "Parallel Logic Simulation of VLSI Systems," in *ACM Computing Surveys*, vol. 26, no. 3, pp. 255–294, September 1994.

[Bar87] BARZILAI, Z., J.L. CARTER, B.K. ROSEN and J.D. RUTLEDGE, "HSS – A High–Speed Simulator," in *IEEE Transactions on Computer–Aided Design*, vol. 6, no. 4, pp. 601–616, July 1987.

[Bar90] BARNES, T.J., "SKILL™: A CAD System Extension Language," in *Proceedings 27th ACM / IEEE Design Automation Conference*, pp. 266–271, Orlando, Florida, June 24–28, 1990.

[Bee90] BEECE, D.K., R. DAMIANO, G. PAPP and R. SCHOEN, "The EVE Companion Simulator," in *Proceedings European Conference on Design Automation*, pp. 290–295, Glasgow, Scotland, March 12–15, 1990.

[Bel93] BELHADJ, M., R. MCCONNELL and P.L. GUERNIC, "A Framework for Macro– and Micro–time to Model VHDL Attributes," in *Proceedings European Design Automation Conference with EURO–VHDL*, pp. 520–525, Hamburg, Germany, September 20–24, 1993.

[Ben91] BENKOSKI, J., J. DESNARD, S. GAI, M. MAGNI and E. PROFUMO, "Mozart–MM: A Mixed–Mode and Multi–Level Simulation System," in *Proceedings 1991 IEEE International Symposium on Circuits and Systems*, pp. 2387–2390, Singapore, 1991.

[Ber88] BERKEL, C.H. VAN, M. REM and R.W.J.J. SAEIJS, "VLSI Programming," in *1988 IEEE International Conference on Computer Design: VLSI in Computers & Processors*, pp. 152–156, Rye Brook, New York, October 3–5, 1988.

[Ber92] BERKEL, K. VAN, "Handshake Circuits: An Intermediary between Communicating Processes and VLSI," Ph.D. thesis, Eindhoven University of Technology, The Netherlands, 1992.

[Ber93] BERKELAAR, M.R.C.M., "Area–Power–Delay Trade–off in Logic Synthesis," Ph.D. Thesis, Eindhoven University of Technology, The Netherlands, 1993.

[Bin94] BINGLEY, P. and W. VAN DER LINDEN, "Application of Framework Technology in System Simulation Environments," in *Proceedings of the Seminar Database Systems and Applications for the Nineties*, Delft University of Technology, The Netherlands, October 11–12, 1994.

[Bla84] BLANK, T., "A Survey of Hardware Accelerators in Computer–Aided Design," in *IEEE Design & Test of Computers,* vol. 1, no. 3, pp. 21–39, August 1984.

[Bor92] BORIELLO, G., "Formalized Timing Diagrams," in *Proc. European Conference on Design Automation*, pp. 372–377, 1992.

[Bre76] BREUER, M.A. and A.D. FRIEDMAN, "Diagnosis & Reliable Design of Digital Systems," Computer Science Press, Woodland Hills, California, 1976.

[Brg85] BRGLEZ, F. and H. FUJIWARA, "A Neutral Netlist of 10 Combinational Benchmark Circuits and a Target Translator in Fortran," in *1985 IEEE International Symposium on Circuits and Systems*, pp. 695–698, Kyoto, Japan, June 5–7, 1985.

[Bry84] BRYANT, R.E., "A Switch–Level Model and Simulator for MOS Digital Systems," in *IEEE Transactions on Computers*, vol. 33, no. 2, pp. 160–177, 1984.

[Bry87] BRYANT, R.E., D. BEATY, K. BRACE, K. CHO and T. SCHEFFLER, "COSMOS: A Compiled Simulator for MOS Circuits," in *Proceedings 24th Design Automation Conference*, pp. 9–16, Miami Beach, Florida, June 28 –July 1, 1987.

[Buc94] BUCK, J.T., S. HA, E.A. LEE and D.G. MESSERSCHMITT, "Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems," in *International Journal of Computer Simulation: Special Issue on "Simulation Software Development"*, vol. 4, pp. 155–182, April 1994.

[Bus89] BUSCHKE, R. and K. LAGEMANN, "An Approach to Understanding Evaluation of Simulation Results as an Integrated Task," in *Proceedings IEEE Custom Integrated Circuits Conference*, pp. 13.6.1–13.6.4, 1989.

[Buu93] BUURMAN, H.W., "From Circuit to Signal: Development of a Piecewise Linear Simulator," Ph.D. Thesis, Eindhoven University of Technology, The Netherlands, 1993.

[Cae93] CAERTS, C., R. LAUWEREINS and J.A. PEPERSTRAETE, "PDG: A Process–Level Debugger for Concurrent Programs in the GRAPE Rapid Prototyping Environment," in *Proceedings Fourth International Workshop on Rapid System Prototyping*, pp. 17–30, Research Triangle Park, North Carolina, June 28–30, 1993.

[CFI94] CAD FRAMEWORK INITIATIVE, INC., "Simulation Backplane Programming Interface Specification," Working Group Draft Proposal, Version 0.6.0, January 1994.

[Cha81] CHANDY, K.M. and J. MISRA, "Asynchronous Distributed Simulation via a Sequence of Parallel Computations," in *Communications of the ACM*, vol. 24, no. 4, pp. 198–206, April 1981.

[Cho88] CHOI, K., S.Y. HWANG and T. BLANK, "Incremental–in–Time Algorithm for Digital Simulation," in *Proceedings 25th ACM / IEEE Design Automation Conference*, pp. 501–505, Anaheim, California, June 12–15, 1988.

[Cho89] CHOI, K., "Incremental Approach to Digital Simulation," Ph.D. Thesis, Department of Electrical Engineering, Stanford University, June 1989.

[Ead89] EADES, P. and R. TAMASSIA, "Algorithms for Drawing Graphs: An Annotated Bibliography," Technical Report No. CS–89–09, Department of Computer Science, Brown University, Providence, Rhode Island, October 1989.

[Ebe89] EBELING, C. and Z. WU, "WireLisp: Combining Graphics and Procedures in a Circuit Specification Language," in *Proceedings of the ICCAD–89*, pp. 322–325, Santa Clara, California, November 5–9, 1989.

[Eij90] EIJNDHOVEN, J.T.J. VAN, M.T. VAN STIPHOUT and H.W. BUURMAN, "Multirate Integration in a Direct Simulation Method," in *Proceedings*

of the European Design Automation Conference, pp. 306–309, Glasgow, Scotland, March 12–15, 1990.

[Eij91] EIJNDHOVEN, J.T.J. VAN, G.G. DE JONG and L. STOK, "The ASCIS Data Flow Graph: Semantics and Textual Format," EUT Report 91–E–251, Eindhoven University of Technology, The Netherlands, June 1991.

[Fan83] FANG, S.C., Y.P. TSIVIDIS and O. WING, "SWITCAP: A Switched Capacitor Network Analysis Program," in *IEEE Circuits and System Magazine*, vol. 5, no. 3, pp. 4–10, September 1983.

[Fel92] FELDBRUGGE, F., "Petri Net Tool Overview 1992," in *Petri Net Newsletter*, no. 41, April 1992.

[Fle91] FLEURKENS, J.W.G. and R.J.W.T. TANGELDER, "The High Level Design of a Chip for Scientific Computation," in *Proceedings of the CompEuro 91*, pp. 811–815, Bologna, Italy, May 13–16, 1991.

[Fle93] FLEURKENS, H., "Interactive System Design in ESCAPE," in *Proceedings of the IEEE International Workshop on Rapid System Prototyping*, pp. 108–113, Research Triangle Park, North Carolina, June 28–30, 1993.

[Fle93a] FLEURKENS, H. and P. BUURMAN, "Flexible Mixed–Mode and Mixed–Level Simulation," in *Proceedings of the 1993 Internation Symposium on Circuits and Systems*, pp. 2137–2140, Chicago, Illinois, May 3–6, 1993.

[Fle93b] FLEURKENS, H. and J. JESS, "ESCAPE: A Flexible Design and Simulation Environment," in *Proceedings SASIMI '93 Workshop*, pp. 277–288, Nara, Japan., October 20–22, 1993.

[Fle95] FLEURKENS, J.W.G., C.A.J. VAN EIJK and J.A.G. JESS, "Run–time Consistency Checking in Discrete Simulation Models," in *Proceedings European Design and Test Conference ED&TC 1995*, pp. 223–227, Paris, France, March 6–9, 1995.

[Fon87] FONTAYNE, Y.D. and R.J. BOWMAN, "The Multiple Storage Radix Hash Tree: An Improved Region Query Data Structure," in *Proceedings of the International Conference on Computer–Aided Design*, pp. 302–305, Santa Clara, California, November 9–12, 1987.

[Fuj90] FUJIMOTO, R.M., "Parallel Discrete Event Simulation," in *Communications of the ACM*, vol. 33, no. 10, pp. 30–53, October 1990.

[Gaj92] GAJSKI, D.D., N.D. DUTT, A.C.-H. WU and S.Y.-L. LIN, "High-Level Synthesis: Introduction to Chip and System Design," Kluwer Academic Publishers, Dordrecht, 1992.

[Gar70] GARDNER, M., "Mathematical Games," in *Scientific American*, pp. 120–123, October 1970.

[Gar71] GARDNER, M., "Mathematical Games," in *Scientific American*, pp. 114–117, April 1971.

[Gen86] GENDEREN, A.J. VAN and A.C. DE GRAAF, "SLS: A Switch-level Timing Simulator," in *The Integrated Circuit Design Book*, pp. 2.93–2.112, 1986.

[Gen92] GENNART, B.A. and D.C. LUCKHAM, "Validating Discrete Event Simulations using Event Pattern Mappings," in *Proceedings 29th ACM / IEEE Design Automation Conference*, pp. 414–419, 1992.

[Geu93] GEURTS, L.J.F., "Graphical Simulation of Data Flow Graphs," Master Thesis, Eindhoven University of Technology, The Netherlands, 1993.

[Gho89] GHOSH, S. and M.-L. YU, "A Preemptive Scheduling Mechanism for Accurate Behavioral Simulation of Digital Designs," in *IEEE Transactions on Computers*, vol. 38, no. 11, pp. 1595–1600, November 1989.

[Gho95] GHOSH, S. and M.-L. YU, "An Asynchronous Distributed Approach for the Simulation of Behavior–Level Models on Parallel Processors," in *IEEE Transactions on Parallel and Distributed Systems*, vol. 6, no. 6, pp. 639–652, June 1995.

[Gol93] GOLIN, E.J., A.C. FENG, L. HUANG and E. HUGHES, "A Visual Design Environment," in *Proceedings IEEE / ACM International Conference on CAD-93*, pp. 364–367, Santa Clara, California, November 7–11, 1993.

[Har87] HAREL, D., "Statecharts: A Visual Formalism for Complex Systems," in *Science of Computer Programming*, vol. 8, pp. 231–274, 1987.

[Har90] HAREL, D., H. LACHOVER, A. NAAMAD, A. PNUELI, M. POLITI, R. SHERMAN, A. SHTULL-TRAURING and M. TRAKHTENBROT, "STATEMATE: A Working Environment for the Development of Complex Reactive Systems," in *IEEE Transactions on Software Engineering*, vol. 16, no. 4, pp. 403–414, April 1990.

[Hei94] HEIJLIGERS, M.J.M., H.A. HILDERINK, A.H. TIMMER and J.A.G. JESS, "NEAT: an Object Oriented High-Level Synthesis Interface," in *Proceedings 1994 IEEE International Symposium on Circuits and Systems*, pp. 233–236, London, May 30 – June 2, 1994.

[Hen85] HENNION, B., P. SENN and D. COQUELLE, "A New Algorithm for Third Generation Circuit Simulators: The One-Step Relaxation Method," Proceedings of the 22nd Design Automation Conference, pp. 137–143, Las Vegas, Nevada, 1985.

[Hey88] HEYDEMANN, M., A. PAIGNAUD and D. DURE, "The Architecture of a Highly Integrated Simulation System," in *Proceedings 25th ACM / IEEE Design Automation Conference*, pp. 617–621, Anaheim, California, June 12–15, 1988.

[Hoa78] HOARE, C.A.R., "Communicating Sequential Processes," in *Communications ACM*, vol. 21, no. 8, pp. 666–677, August 1978.

[Hoe92] HOEVEN, A. VAN DE, "Concepts and Implementation of a Design System for Digital Signal Processor Arrays," Ph.D. Thesis, Delft University of Technology, The Netherlands, October 1992.

[Hou93] HOUF, H., "VHDL Simulation in ESCAPE," Master Thesis, Eindhoven University of Technology, The Netherlands, December 1993.

[Hwa87] HWANG, S.Y., T. BLANK and K. CHOI, "Incremental Functional Simulation of Digital Circuits," in *Proceedings of the ICCAD-87*, pp. 392–395, Santa Clara, California, November 9–12, 1987.

[Hwa88] HWANG, S.Y., T. BLANK and K. CHOI, "Fast Functional Simulation: An Incremental Approach," in *IEEE Transactions on Computer-Aided Design*, vol. 7, no. 7, pp. 765–774, July 1988.

[IEE88] IEEE, "IEEE Standard VHDL Language Reference Manual," IEEE Std 1076–1987, New York, 1988.

[Jan89] JANSSEN, G.L.M.J., "Circuit Modelling and Animated Interactive Simulation in ESCHER," in *Proceedings SCS European Simulation Multiconference: Simulation applied to manufactoring, energy and environmental studies and electronics and computer engineering*, pp. 265–270, Rome, Italy, June 1989.

[Jef85] JEFFERSON, D.R., "Virtual Time," in *ACM Transactions on Programming Languages and Systems*, vol. 7, no. 3, pp. 404–425, July 1985.

[Jen92] JENSEN, K., "Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use," vol. 1, Springer–Verlag, Berlin Heidelberg, 1992.

[Jon89] JONES, L.G., "Fast Online/Offline Netlist Compilation of Hierarchical Schematics," in *Proc. of the 27th ACM / IEEE Design Automation Conference*, pp. 822–825, Las Vegas, Nevada, June 25–29, 1989.

[Jon89a] JONES, L.G., "Fast Incremental Netlist Compilation of Hierarchical Schematics," in *Proceedings of the ICCAD–89*, pp. 326–329, Santa Clara, California, November 5–9, 1989.

[Jon92] JONES, L.G., "An Incremental Zero/Integer Delay Switch–Level Simulation Environment," in *IEEE Transactions on Computer–Aided Design*, vol. 11, no. 9, pp. 1131–1139, September 1992.

[Jon93] JONG, G.G. DE, "Generalized data flow graphs: theory and applications," Ph.D. Thesis, Eindhoven University of Technology, The Netherlands, 1993.

[Kal92] KALAVADE, A. and E.A. LEE, "Hardware/Software Co–Design Using Ptolemy – A Case Study," in *Proceedings of the IFIP International Workshop on Hardware / Software Co–Design*, Grassau, Germany, May 19–21, 1992.

[Kav86] KAVI, K.M., B.P. BUCKLES and U.N. BHAT, "A Formal Definition of Data Flow Graphs Models," in *IEEE Transactions on Computers*, vol. C–35, no. 11, pp. 940–948, November 1986.

[Ker88] KERNIGHAN, B.W. and D.M. RITCHIE, "The C Programming Language: Second Edition," Prentice–Hall, Englewood Cliffs, 1988.

[Kle84] KLECKNER, J.E., "Advanced mixed–mode simulation techniques," Ph.D. Thesis, University of California, Berkeley, May 1984.

[Koe87] KOENIG, P.M., "SEE–SAW – A graphical Interface for System Level Design," Research Report No. CMUCAD–87–49, Department of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, Pennsylvania, December 1987.

[Ku88] KU, D. and G.D. MICHELI, "Hardware C – A Language for Hardware Design," Technical Report CSL–TR–90–419, Stanford University, 1988.

[Lee87] LEE, E.A. and D.G. MESSERSCHMITT, "Synchronous Data Flow," in *IEEE Proceedings*, vol. 75, no. 9, pp. 1235–1245, September 1987.

[Lee95] LEE, E.A. and T.M. PARKS, "Dataflow Process Networks," in *IEEE Proceedings,* vol. 83, no. 5, pp. 773–799, May 1995.

[Lod86] LODDER, A., M.T. VAN STIPHOUT and J.T.J. VAN EIJNDHOVEN, "ESCHER: Eindhoven SCHematic EditoR reference manual," EUT Report 86–E–157, Eindhoven University of Technology, The Netherlands, February 1986.

[Luk92] LUKASSEN, R.J.P.B., "Using LISP as an Interactive Database Access Language," Master Thesis, Eindhoven University of Technology, The Netherlands, June 1992.

[Mat92] MATSUMOTO, Y. and K. TAKI, "Parallel Logic Simulation on a Distributed Memory Machine," in *Proceedings European Conference on Design Automation*, pp. 76–80, Brussels, Belgium, March 16–19, 1992.

[Mau92] MAURER, P.M., "Two New Techniques for Unit–Delay Compiled Simulation," in *IEEE Transactions on Computer–Aided Design,* vol. 11, no. 9, pp. 1120–1130, September 1992.

[McM94] MCMILLAN, K.L., "Fitting Formal Methods into the Design Cycle," in *Proceedings 31st Design Automation Conference*, pp. 314–319, San Diego, California, June 6–10, 1994.

[Nag75] NAGEL, L.W., "SPICE2: A Computer Program to Simulate Semiconductor Circuits," Memorandum No. ERL–M520, University of California, Berkeley, 1975.

[New88] NEWBERY, F.J., "An Interface Description Language for Graph Editors," in *Proceedings IEEE Workshop on Visual Languages*, pp. 144–149, Pittsburgh, Pennsylvania, 1988.

[Nie91] NIESSEN, A.J., "Simulatie en verificatie van een modulair blokbeveiligingssysteem," Training Report, Eindhoven University of Technology, The Netherlands, November 1991 (in Dutch).

[Nie92] NIEMEYER, M., "Multi–Simulator Coupling," in *Proceedings of the Synthesis and Simulation Meeting and International Interchange SASIMI '92*, pp. 233–242, April 6–8, 1992.

[Ocz91] OCZKO, A. and C. OCZKO, "Putting Different Simulation Models Together – The Simulation Configuration Language VHDL/S," in

Proceedings Computer Hardware Description Languages and their Applications, pp. 115–129, Marseille, France, April 22–24, 1991.

[Pan74] PANNE, C. VAN DE, "A Complementary Variant of Lemke's Method for the Linear Complementary," in *Mathematical Programming*, vol. 7, pp. 283–310, 1974.

[Pet81] PETERSON, J.L., "Petri Net Theory and the Modeling of Systems," Prentice–Hall, Englewood Cliffs, 1981.

[Phi88] PHILIPSEN, W.J.M., "The Blitter Project. A Test Case," Master Thesis, Eindhoven University of Technology, The Netherlands, 1988.

[Ram92] RAMMIG, F.J., "Synthesis Related Aspects of Simulation," in *The Synthesis Approach to Digital System Design*, vol. 170, pp. 303–333, Kluwer Academic Publishers, Dordrecht, The Netherlands, 1992.

[Rey88] REYNOLDS JR., P.F., "A Spectrum of Options for Parallel Simulation," in *Proceedings of the 1988 Winter Simulation Conference*, pp. 325–332, San Diego, California, December 12–14, 1988.

[Ros85] ROSENBERG, J.B., "Geographical Data Structures Compared: A Study of Data Structures Supporting Region Queries," in *IEEE Transactions on Computer–Aided Design,* vol. 4, no. 1, pp. 53–67, January 1985.

[Rou89] ROUMELIOTIS, M. and J.R. ARMSTRONG, "HDL Modeling for Process Oriented Simulation," in *Proc. of the IFIP WG 10.2 Eight Int. Conf on Computer Hardware description Languages and their Applications*, pp. 1–8, Amsterdam, The Netherlands, April 27–29, 1989.

[Row94] ROWSON, J.A., "Hardware/Software Co–Simulation," in *Proceedings 31st Design Automation Conference*, pp. 439–440, San Diego, California, June 6–10, 1994.

[Sal90] SALEH, R.A. and A.R. NEWTON, "Mixed–Mode Simulation," Kluwer Academic Publishers, Dordrecht, 1990.

[Sal94] SALEH, R.A., S. JOU and A.R. NEWTON, "Mixed–Mode Simulation and Analog Multilevel Simulation," Kluwer Academic Publishers, Dordrecht, 1994.

[Sas83] SASAKI, T., N. KOIKE, K. OHMORI and K. TOMITA, "HAL; A Block Level Hardware Logic Simulator," in *Proceedings 20th ACM / IEEE De-*

sign Automation Conference, pp. 150–156, Miami Beach, Florida, June 27–29, 1983.

[She90] SHERER, A.D., B.S. STANOJEVICH and R.J. BOWMAN, "SMALS: A Novel Database for Two–Dimensional Object Location," in *IEEE Transactions on Computer–Aided Design*, vol. 9, no. 1, pp. 57–65, January 1990.

[Smi87] SMITH, S.P., M.R. MERCER and B. BROCK, "Demand Driven Simulation: BACKSIM," in *Proceedings 24th ACM / IEEE Design Automation Conference*, pp. 181–187, Miami Beach, Florida, June 28 – July 1, 1987.

[Sri92] SRIVASTAVA, M.B., "Rapid–Prototyping of Hardware and Software in a Unified Framework," Ph.D. Thesis, Department of EECS, U.C. Berkeley, California, 1992.

[Ste84] STEELE, G.L., "Common LISP: The Language," Digital Equipment Corporation, 1984.

[Sto92] STOK, L., "Architectural Synthesis and Optimization of Digital Systems," Ph.D. Thesis, Eindhoven University of Technology, The Netherlands, July 1992.

[Sub86] SUBODH-KUMAR, M.P. and Y.N. SRIKANT, "Graphical Simulation of Petri Nets," in *Computers and Graphics*, vol. 20, no. 3, pp. 225–228, 1986.

[Sug81] SUGIYAMA, K., S. TAGAWA and M. TODA, "Methods for Visual Understanding of Hierarchical Systems," in *IEEE Transactions on Systems, Man, and Cybernetics*, vol. SMC–11, no. 2, pp. 109–125, February 1981.

[Tah93] TAHAWY, H.E., D. RODRIGUEZ, S. GARCIA-SABIRO and J.-J. MAYOL, "VHD$_e$LDO: A New Mixed Mode Simulation," in *Proceedings European Design Automation Conference with EURO–VHDL*, pp. 546–551, Hamburg, Germany, September 20–24, 1993.

[Tak90] TAKASAKI, S., N. NOMIZU, Y. HIRABAYASHI, H. ISHIKURA, M. KURASHITA, N. KOIKE and T. NUKATA, "HAL III: Function Level Hardware Logic Simulation System," in *1990 IEEE Int. Conf. on Computer Design: VLSI in Computers & Processors*, pp. 167–174, Cambridge, Massachussetts, September 17–19, 1990.

[Tan92] TANGELDER, R.J.W.T., "The Design of Chip Architectures for Accurate Inner Product Computation," Ph.D. Thesis, Eindhoven University of Technology, The Netherlands, 1992.

[Tho91] THOMAS, D.E. and P.R. MOORBY, "The Verilog Hardware Description Language," Kluwer Academic Publishers, Boston, 1991.

[Ulr69] ULRICH, E.G., "Exclusive Simulation of Activity in Digital Networks," in *Communications of the ACM,* vol. 12, no. 2, pp. 102–110, February 1969.

[Vah91] VAHID, F., S. NARAYAN and D.D. GAJSKI, "SpecCharts: A Language for System Level Synthesis," in *Proceedings Computer Hardware Description Languages and their Applications*, pp. 165–174, Marseille, France, April 22–24, 1991.

[Vel92] VELLANDI, B. AND M. LIGHTNER, "Parallelism Extraction ad Program Restructuring of VHDL for Parallel Simulation," in *Proc. European Conference on Design Automation*, pp. 81–87, Brussels, Belgium, March 16–19, 1992.

[Wan90] WANG, Z. and P.M. MAURER, "LECSIM: A Levilized Event–Driven Compiled Logic Simulator," in *Proc. of 27th ACM / IEEE Design Automation Conference*, pp. 491–496, Orlando, Florida, June 24–28, 1990.

[Wol93] WOLF, P. VAN DER, "Architecture of an Open and Efficient CAD Framework," Ph.D. Thesis, Delft University of Technology, The Netherlands, June 1993.

Appendix

A The LISP–like HDL

A.1. Introduction

In this appendix, an overview is given of the LISP–like HDL. This language can be used to describe the behaviour of primitive components in ESCAPE. The language contains primitive functions to interface with the event driven simulator and to visualize simulation results.

A.2. The language

The HDL supports different types of variables. The following types can be used in the declaration part:

- *Ports,* which are used to communicate values between a module and its environment. Ports have also to be defined in the symbol view of the corresponding design.
- *State variables,* which are used to model memory elements in a behavioural description. A state variable holds a value between consecutive evaluations of a description and it is initialized before the beginning of a simulation experiment.
- *Local variables,* which are used to simplify expressions. A local variable is initialized before each evaluation of a behavioural description.

In addition, *parameters* can be used in the body of a behavioural description. The value of a parameter can be modified using the user interface at any time in a simulation experiment. After modification, its new value will be used immediately. Parameters are objects stored in the network view, whereas the other types are embedded in the behavioural description language. Note that a parameter is bound to an instance of a design not to a design itself.

A.3. Summary of functions

A summary of the most important functions of the LISP–like HDL is given below. In this HDL, the number 0 is equivalent with the logical value false. Any number not equal to 0 is equivalent with the logical value true.

(**+** { <number> }*)
Returns sum of the <number>'s. When no arguments are supplied, it returns the integer 0.

(**–** <number–1> { <number> }*)
With one argument negates <number–1>. Otherwise, successively subtracts the <number>'s from <number–1>.

(* { <number> }*)
Returns product of the <number>'s. When no arguments are supplied, it returns the integer 1.

(/ <number–1> <number–2> { <number> }*)
Returns <number–1> divided by product of <number–2> and rest of <number>'s. Uses integer division. The result when attempting to divide–by–zero is undefined.

(**mod** <number> <divisor>)
Returns remainder of <number> divided by <divisor> (also a number).Uses C–language % operator.

(**<** <number> { <number> }*)
Returns true if <number>'s are monotonically increasing.

(**<=** <number> { <number> }*)
Returns true if <number>'s are monotonically non–decreasing.

(**=** <number> { <number> }*)
Returns true if all <number>'s are equal.

(**>=** <number> { <number> }*)
Returns true if <number>'s are monotonically non–increasing.

(**>** <number> { <number> }*)
Returns true if <number>'s are monotonically decreasing.

(**random** [<range> [<init–seed–p>]])
Returns a pseudo–random integer. If the first argument is supplied then an integer in the range [0..<range>–1] is returned. If <init–seed–p> is true the random number seed is set based on the current time and pid prior to calling the generator.

(**min** <number> { <number> }*)
Returns smallest of all the <number>'s.

(**max** <number> { <number> }*)
Returns largest of all the <number>'s.

(**and** { <form> }*)
Evaluates the <form>'s in order from left to right until one of them yields false, then false is returned. Any remaining <form>'s are not evaluated at all. In case of no argument returns true; if all arguments evaluate to true, the value of the last <form> is returned.

(**or** { <form> }*)
Evaluates the <form>'s in order from left to right until one of them yields true, then this value is returned. Any remaining <form>'s are not evaluated at all. In case of no argument or all arguments evaluate to false, false is returned.

(**eqv** <arg> { <arg> }*)
Returns true if all its arguments are logically equivalent, i.e. either all true or all false.

(**not** <arg>)
Returns the complement of the truth value of the argument.

(**setq** <var> <form>)
Sets (assigns) to <var> the value of <form>. The <var> argument (symbols) is not evaluated, the <form> argument is. Returns the value assigned.

(**delay** <time> { <net> | (<bus–net> <offset>) } <value> [<event–type>])
Causes an event of type <event–type> to be scheduled for the <net> (or the wire of a bus specified by <bus–net> and <offset>) to occur at a time <time> after the current simulation clock value. At that time in the future the net will change to the value <value>.
Allowed event–type values are:
0 normal, conditional evaluation of modules in net's fanout list
1 refrain, no evaluation of modules
2 trigger, forced evaluation of modules
3 cancelled, no effect
4 interrupt, interrupts simulation
Returns <value>.

(**if** <test> <then–form> [<else–form>])
If <test> yields true evaluates <then–form>, else <else–form>. Returns the value of the last form evaluated. If <test> evaluates to false and there is no <else–form> present, false is returned.

(**case** <keyform> {({ ({ <key> }*) | <key> }* { <form> }*)}*)
Evaluates <keyform>, which must result in an integer value. Then treats

clause by clause in order until one is found where <keyform> matches the <key>. If so, evaluates the <form>'s in the clause as an implicit progn and returns result of last. If no key matches, it returns 0.

(**while** <test> { <form> }*)
While <test> yields true execute the <form>'s. Always returns false.

(**for** (<init> <test> <step>) { <form> }*)
C–like for statement. This statement is equivalent with:

```
(progn
   <init>
   (while <test>
      { <form> }*
      <step>
   )
)
```

(**progn** { <form> }*)
Evaluates the body <form>'s in order from left to right, and returns the result of evaluating the last one. When it is said that a function evaluates its forms or body as an implicit progn it is understood that the evaluation takes place as if (progn ...) was actually typed around those forms.

(**write** <value>)
Writes integer <value> in bounding box of instance, returns this value.

(**write** <control–string> [<arg1> [<arg2> [<arg3>]]])
Formats a string out of a <control–string> and <arg>s. <arg> may be string or integer value. Formatting and substituting the arguments into the control–string is done similar to the C language (s)printf routine. The result is a string which is written in the bounding box of instance. The control–string may contain %s, %d, or %c to substitute successive following arguments. To include % itself you must write it twice: %%. Returns 0.

(**simtime**)
Returns simulation time value.

(**color** <value>)
Colors instance box when <value> is true.

(**fillcolor** <r> <g> <b> [<index>])
Sets color used by function "color" using the arguments that specify the values of the red green and blue components, which must be in the range [0..100]. <index> specifies the color index in the table reserved for animation. If <index> is omitted, the color index 0 is assumed.

(**make–bitvector** <length>)
Dynamically allocates bit vector of that length, i.e. number of bits. Returns the vector initialised to all bits zero. Since there is as yet no automatic garbage collection of used space, use this function with care. Best is to call only once, for instance at start of simulation.

(**bit** <vector> <index>)
Returns value of bit vector element at index. Indices count from zero. Value returned is in fact integer, either 0 for false, 1 for true.

(**setbit** <vector> <index> <value>)
Sets bit in bit vector at index to value. Value is integer, 0 for false, any other value for true. Returns value.

(**<<** <vector> <shifts> [<fill–in>])
Shifts bits in bit vector left over number of shifts places. Left means towards higher indices (corresponds to usual model of bitvector to represent binary numbers). Bits shifted out are lost. Bits with value fill–in are shifted in at the right, by default fill–in bits are false. Returns modified bit vector.

(**>>** <vector> <shifts> [<fill–in>])
Shifts bits in bit vector right over number of shifts places. Right means towards lower indices (corresponds to usual model of bitvector to represent binary numbers). Bits shifted out are lost. Bits with value fill–in are shifted in at the left, by default fill–in bits are false. Returns modified bit vector.

(**interrupt** <time>)
Schedules an interrupt type of event. Equivalent to (delay <time> 0 0 4). Returns 0.

Appendix

B Simulating the bit blitter

In this appendix, some snapshots of the animated simulation of the bit blitter are depicted. The first snapshot (see figure B.1) shows the bit blitter after initialization of an image in the left half of the display. The figures B.2 and B.3 show the execution of the first operation: the image in the left half of the display is copied to a specific position in the right half of the display. At t = 105, this operation is completed.

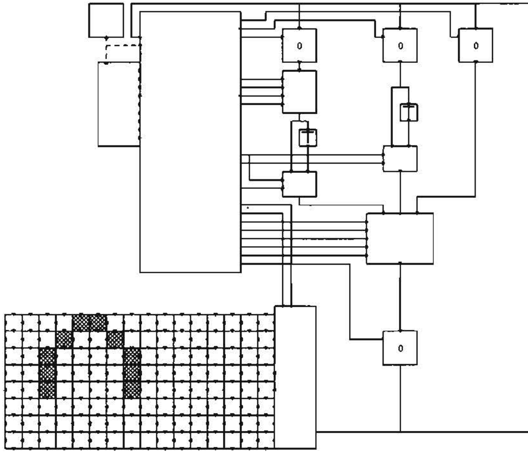


Figure B.1. Simulation snapshot at t = 0.

In figure B.4, the first picture shows the bit blitter after completion of a move of the image in the right half of the display. The second picture is taken after completion of a fill operation of the image in the right half of the display. The next figure shows the bit blitter after completion of an invert operation of pixels in a specific area of the right half of the display. The last picture shows the display after translating the whole contents of the display.

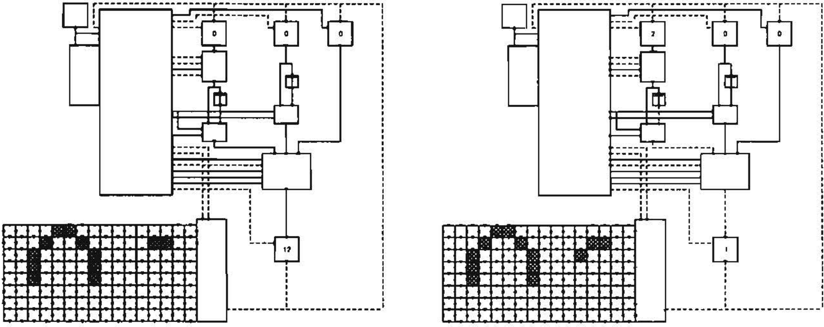


Figure B.2. Simulation snapshots at t = 21 and t = 37.

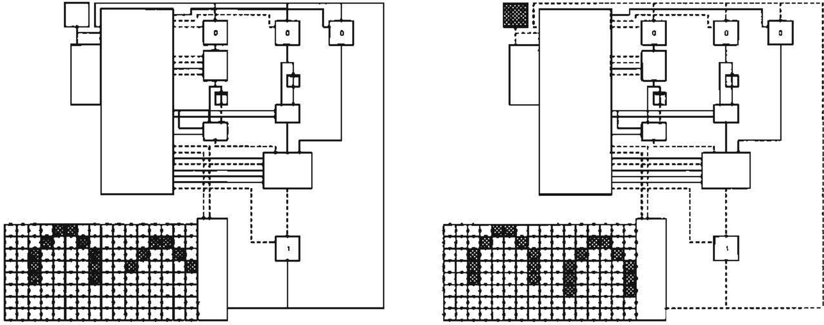


Figure B.3. Simulation snapshots at t = 65 and t = 105.

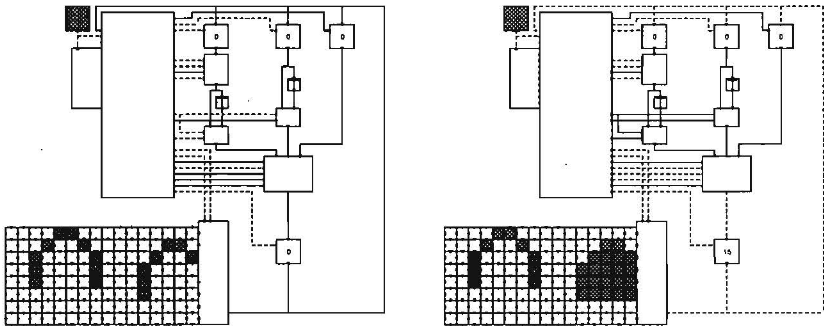


Figure B.4. Simulation snapshots at t = 305 and t = 525.

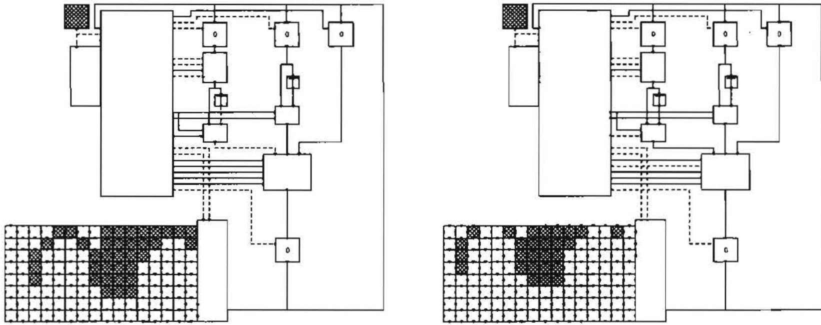


Figure B.5. Simulation snapshots at t = 725 and t = 1059.

Appendix C The inter tool protocol

Table C.1 gives a summary of the inter tool protocol (ITP). The following symbols are used in this table:

- □ : Opens the context associated with the entry in the table. The previous context is saved by pushing it on a context stack (First In Last Out mechanism).
- ■ : Closes the current context and the previous context is restored by popping it from the context stack, if the stack is not empty.
- ● : Command may be applied in this context.
- ○ : Command may be applied in this context, but it is not recommended.

An example is the *add–edge* command. This command is processed in the *none, design* and *graph* contexts. Otherwise, it is ignored. The *add–edge* command opens the *add–edge* context, which is closed by the *end–add–edge* command. Note that commands listed in the *none context* column may be applied at any time in a inter tool session.

Table C.1. Summary of the inter tool protocol.

Command \ **Context**	none	design	graph	network	symbol	add–edge	add–instance	add–net	add–node	add–symbol	add–terminal	add–text	add–wire	select–list	text–view	view–link
add–arc	●	●	●	●	●					□						
add–arrow	●	●	●	●	●	●				□						
add–box	●	●	●	●	●					□						
add–circle	●	●	●	●	●					□						
add–circle–arc	●	●	●	●	●					□						
add–edge	●	●	●			□										
add–instance	●	●		●			□									
add–line	●	●	●	●	●	●				□						
add–net	●	●		●				□								
add–node	●	●	●						□							

Command \ Context	none	design	graph	network	symbol	add–edge	add–instance	add–net	add–node	add–symbol	add–terminal	add–text	add–wire	select–list	text–view	view–link
add–oval	●	●	●	●	●					□						
add–terminal	●	●		●	●						□					
add–text	●	●	●	●	●							□				
add–wire	●	●		●									□			
base–length									●							
clear–textview															●	
close–view–window	●															
delete–all–view–links	●															
delete–graph	●															
design	●	□														
dest–port									●							
end–add–edge						■										
end–add–instance							■									
end–add–net								■								
end–add–node									■							
end–add–symbol										■						
end–add–terminal											■					
end–add–text												■				
end–add–wire													■			
end–design		■														
end–graph			■													
end–network				■												
end–select–list														■		
end–symbol					■											
end–text–view															■	
end–view–link																■
error	●	●	●	●	●	●			●							●
graph	●	●														
item														●		
link																●
load–design	●															
message	●	●	●	●	●	●			●							●
network	●			□												
open–view–window	●															

Command \ Context	none	design	graph	network	symbol	add–edge	add–instance	add–net	add–node	add–symbol	add–terminal	add–text	add–wire	select–list	text–view	view–link
orig–port									●							
prompt–dialog	●															
property									●							
quit	●	○	○	○	○	○			○					○	○	○
reset–sim	●															
select–list	●													□		
set–design	●															
set–editor–mode	●															
set–graph	●															
set–view–window	●															
show–text–view															●	
show–world	●	○	○	○	○	○			○					○	○	○
start–sim	●															
stop–sim	●															
symbol	●				□											
text												●			●	
text–view	●														□	
view–link	●															□
warning	●	●	●	●	●	●			●					●	●	●
zoom–in	●	○	○	○	○	○			○					○	○	○
zoom–out	●	○	○	○	○	○			○					○	○	○

Notation

$a \in A$	a is an element of A.
$A \cup B$	The union of the sets A and B.
$A \cap B$	The intersection of the sets A and B.
$A \setminus B$	The difference of the sets A and B.
$A \subseteq B$	The set A is a subset of the set B.
$P(A)$	The powerset of set A.
$A \times B$	The Cartesian product of the sets A and B.
$A \rightarrow B$	The class of functions defined on $A \times B$ with domain A and codomain B.
$f : A \rightarrow B$	The function f defined on $A \times B$ with domain A and codomain B. Note that $f \subseteq A \times B$.

Biography

Hans Fleurkens was born on February 17th, 1965 in Venray, the Netherlands. He studied Electrical Engineering at Eindhoven University of Technology, the Netherlands, where he graduated from in August 1988. After receiving his degree, he has worked for half a year on artificial intelligence techniques in control engineering at Delft University of Technology, the Netherlands.

From April 1989, he has been working towards a Ph.D. degree in the Design Automation Section of the Department of Electrical Engineering at Eindhoven University of Technology. He expects to receive this degree based on the work described in this thesis on March 26, 1996.

Hans Fleurkens has worked in the Mathematical Sciences Department of IBM Thomas J. Watson Research Center, Yorktown Heights, New York from October 1990 till May 1991.

From November 1994, he is working at Philips Research in Eindhoven, the Netherlands.

Stellingen

behorende bij het proefschrift
Interactive Modelling and Simulation of Heterogeneous Systems
van Hans Fleurkens

1. Er zijn een aantal factoren die invloed hebben op de snelheid waarmee een schakeling gesimuleerd kan worden. De gedragsbeschrijving van het systeem zelf is een factor waarvan de invloed vaak onderschat wordt.

   [Dit proefschrift]

2. De standaardisatie van de *interface* van commercieel verkrijgbare simulatoren is zinvoller dan de standaardisatie van de *interface* voor een zogenaamd *simulation backplane.*

   [Dit proefschrift]

3. Het gebruiken van een interactief ontwerp– en/of simulatiesysteem hoeft niet ten koste te gaan van de prestatie van de simulator zelf.

   [Dit proefschrift]

4. De enige manier om de snelheid van simuleren orde groottes te verbeteren is het verhogen van het abstractieniveau van de systeembeschrijving. Andere technieken die eigenschappen van een systeembeschrijving gebruiken om efficiënter te kunnen simuleren hebben veel kleinere snelheidsverbeteringen tot gevolg.

   [SALEH, R.A. and A.R. NEWTON, "Mixed–Mode Simulation," Kluwer Academic Publishers, Dordrecht, 1990]

5. Het opbouwen van een complexe datastructuur door middel van het toevoegen van nieuwe elementen leidt in het algemeen tot minder fouten dan het afbreken van die datastructuur door het verwijderen van elementen.

6. *Hardware software co–design* zou voor vele ontwerpers de alternatieve betekenis kunnen hebben van het omzeilen van de fouten in de software van de programmatuur, waarmee de hardware ontworpen wordt.

7. De keuze van een passende ontwerptool of –omgeving voor een bepaald ontwerpprobleem is net zo moeilijk als het kiezen van een passende wijn bij een bepaald gerecht.

8. In een onderlinge vergelijking van de kwaliteiten van verschillende faculteiten zou de toegankelijkheid van de gebouwen voor studenten en medewerkers eveneens een belangrijk aspect moeten zijn.

9. Het heffen van belasting op de feitelijke commerciële waarde van gratis verkrijgbare software in bepaalde Amerikaanse staten is een maatregel die aangeeft dat overheden erg vindingrijk zijn in het bedenken van onzinnige oplossingen voor hun begrotingstekort.

10. De bedragen die geboden worden voor het uitzenden van betaald voetbal zijn een maat voor de hoeveelheid geld die beleidsmakers van diverse zendgemachtigden voor hun eigen bestaansrecht en een plaats op alle kabelnetwerken over hebben.

11. Een van de gevolgen van privatisering zou een afgeslankte minder geld kostende overheid moeten zijn.

12. De snelle groei van het gebruik van het Internet na de introduktie van het World Wide Web en de bijbehorende eenvoudig te gebruiken interactieve toegangsprogrammatuur toont aan dat het ontwerp van een goede *user interface* zichzelf terugverdient.

13. Met gratis software kan door derden veel geld verdiend worden.