Computer-aided Support for Designing Parallel Real-Time Solutions on Transputer-based Systems*

Maarten VAN STEEN
Erasmus University (Woudestein), Department of Computer Science
P.O. Box 1738, 3000 DR Rotterdam (steen@cs.few.eur.nl)

Armand TEN DAM, Teus VOGEL
TNO Institute of Applied Physics P.O. Box 155, 2600 AD Delft, ({tendam,tvogel}@tpd.tno.nl)

Abstract

Exploiting parallelism for industrial real-time applications has not received much attention compared to scientific applications. The available real-time design methods do not adequately address the issue of parallelism, resulting still in a strong need for low-level tools such as debuggers and monitors. In this paper we show how problems can be alleviated if an approach is followed that allows for experimentation with designs and implementations. In particular, we discuss a development system that integrates design, implementation, execution, and analysis of real-time applications, putting emphasis on exploitation of parallelism[1]

1 Introduction

In the last decade exploiting parallelism has received considerable attention from the scientific research community. In many cases, research has focussed on exploiting parallelism for solving problems of *increasing size*. However, the reasons for exploiting parallelism in real-time applications originate from the demand to meet harder timing constraints rather than from scalability issues. Exploiting parallelism in these cases leads to more intricate models by which one can analyze the actual behavior of the application. Such an analysis is needed to determine *a priori* if the required timing constraints will be satisfied. This type of analysis is generally obsolete in the case of parallel scientific applications where meeting timing constraints is not a hard requirement. The effect to date is that many tools for development of parallel applications are not adequate when applied to real-time application development. On the other hand, traditional development tools do not support exploitation of parallelism as a design activity.

A solution to this problem can be obtained by first developing analytic models of an application, and later using these models to support the actual design. Unfortunately, this does imply that two separate activities, namely performance modeling and software design, should later be integrated. In practice, this often turns out to be a hard and error-prone process. A more elegant solution is to allow designs to be evaluated directly. In this paper we describe how such an integrated approach is realized in the ESPRIT project *Hamlet*. We argue that adequate means are to be provided to *experiment* with software designs. In this way, it is feasible to devise solutions that adhere to standard criteria of well-engineered software, and at the same time can meet the timing constraints imposed by their problems.

The paper is organized as follows. In Section 2 we discuss the traditional approaches towards software design do not adequately support development of parallel applications. The bulk of the paper is presented in Section 3 in which we further describe our approach by focusing on design, implementation, execution, and analysis of applications. We conclude our presentation in Section 4 by taking a look at related work.

2 The design phase in a system development cycle

System development life cycles are traditionally decomposed into five global phases [5]: analysis, design, implementation, test, and maintenance. In this paper we mainly concentrate on the design and implementation phases. We assume that requirements analysis has resulted in a definition of *what* the system should do, and that a development team has reached a point where it should decide *how* these requirements are to be met.

2.1 Traditional design approaches

When speaking in terms of system design, one can roughly distinguish three different approaches [4]:

1. *Functional structuring* by which the structure of the system is described in terms of functional interaction of the various components.
2. *Object structuring* by which the system is decomposed into a collection of communicating objects, where each object is responsible for a specific function.
3. *Data structuring* that puts a main emphasis on the data to be manipulated in a system.

Regardless of the actual approach taken, emphasis during the design phase in each case is put on abstraction, structuring, information hiding, modularity, and concurrency. Of these concepts, concurrency is extremely important when dealing with (parallel) real-time systems. As mentioned by Jacobson [11], developing real-time systems requires that the three fundamental issues of *processes*, the means of *communication*, and the method of *synchronization* should be taken into account. Unfortunately, to our opinion, it is precisely with respect to these three fundamental issues that traditional design methods lack sufficient support. As

* The work presented here is done within the framework of the ESPRIT program Hamlet and partly funded by the Commission of the European Communities.

[1] An extended version of this paper can also be obtained as technical report [18].

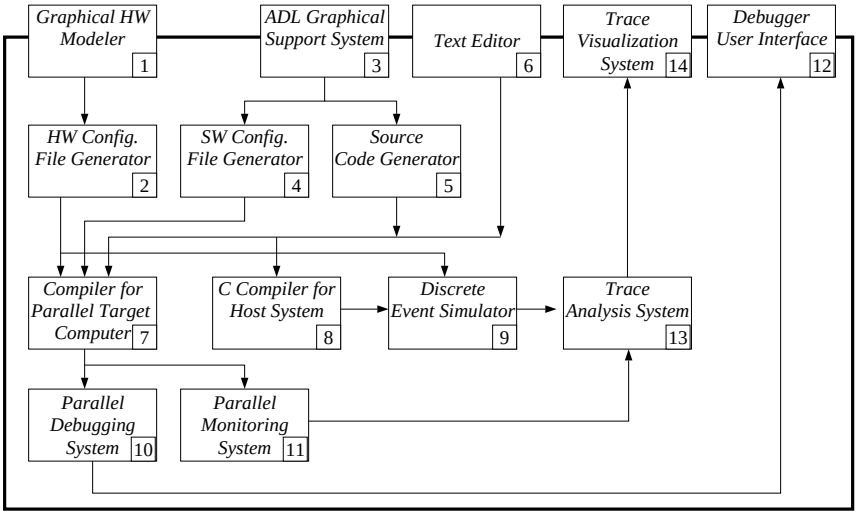

Figure 1: The global architecture of the Hamlet Application Development System.

we see it, this is caused by failing to make a distinction between parallelism (or concurrency for that matter) as, on the one hand, a means for *modeling* a solution, and, on the other hand, as a means for *achieving performance*. In the latter case, additional design issues such as process replication, communication bandwidth, buffering capabilities, etc. need to be addressed as well. And as these issues influence the design, they should be addressed at that level.

2.2 Experimental development during design

So what additional support can we expect for designing parallel real-time applications? In the first place, there should be a means for explicitly expressing exploitation of parallelism. Secondly, and at least as important, there should be a means for obtaining preliminary insight in the *effects* of exploitation of parallelism. In other words, we require a means for preliminary performance evaluation. This is a difficult demand to meet for it requires that we at least have a fairly accurate model of the behavior of the application in conjunction with its mapping onto a target parallel platform.

If preliminary performance evaluation is to be incorporated in a design method, its means should be integrated with the traditional structuring approaches mentioned above. This has two implications. Firstly, unambiguous *behavioral semantics* should form an integrated part of the design method. In this paper, we advocate the use of state-transition machines to describe the behavior of an application, combined with information on how to map them onto processors. Secondly, the actual evaluation of the behavior of a design should also form part of the design method. In particular, we feel that a design method for parallel real-time applications should allow for a high degree of *experimentation* with respect to performance evaluation. This need for experimentation not only originates from the practical intractability of analytic models. It is also motivated by the complexity of deriving *a priori* exact behavior models of the final (parallel) implementation of an application. By conducting experiments, designs can be gradually adapted as insight in this behavior grows.

2.3 The Hamlet approach

A design in our terms consists of a *structure model* combined with a *behavior model*. The structure model describes a design as a collection of hierarchically organized processes communicating by message-passing. The behavior model consists of a state-transition machine for each lowest level process. This integration of a structure model and a behavior model then allows us to *simulate* a design, by taking a model of the parallel target machine into account. The result of preliminary performance evaluation may then possibly lead to (1) an adaptation of the design (either with respect to its structure or behavior model), (2) an adaptation of the hardware model (e.g. by adding more processors), or (3), an adaptation of the mapping of a design onto hardware resources. By experimentation through simulation, a developer eventually arrives at an initially satisfactory design.

This design can then subsequently be used for deriving an implementation. However, in our approach, we anticipate that further adaptations may be necessary. Therefore, it is essential that the transition from design to implementation proceeds as seamless as possible. To this aim, we have tailored our design technique towards one which enables (partially) automated generation of efficiently executable parallel code.

In the next section we shall take a closer look at how this experimental approach towards parallel application development is realized in practice.

3 The Hamlet Application Development System

Within Hamlet, we roughly distinguish four different activities as part of the development process: design, implementation, execution, and analysis. The global architecture of our Application Development System is shown in Figure 1. The following components are distinguished:

- A graphical hardware modeler (1) that allows a developer to describe the global architecture of the target parallel (transputer) system. The modeler is connected to a configuration file generator (2).
- A graphical software development tool (3) by which so-called ADL designs can be made. Designs expressed in this *application design language*, can be used to automatically generate software configuration files (4) for transputer systems, as well

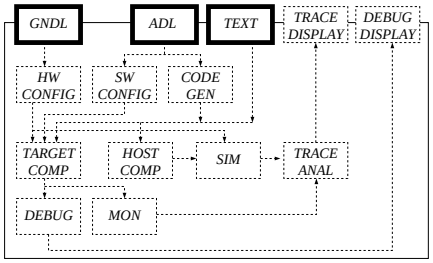

Figure 2: The design components.

as skeleton target code (5). Any additional textual information is provided by means of standard text editor (6).

- Compilers: one for the actual target system (7), and one to compile a design for simulation on the host system (8).
- A runtime support system, consisting of a discrete event simulator (9) by which the behavior of an application can be evaluated before executing it on an actual target machine. In addition, there is also a parallel debugger (10),(12) and a performance monitor (11), both used when the application is running on the target system.
- A trace analysis system (13) and accompanying visualizer (14) for displaying performance measurements during simulation, as well as actual target execution.

In the following subsections we shall a closer look at these components.

3.1 System design

System design in Hamlet consists of two subactivities. The most prominent one is the design of the software components of an application. In addition, attention is paid to the design of the parallel target hardware. The position of the design components in the development system is shown in Figure 2.

Software designs are expressed in the so-named *Hamlet Application Design Language*, or ADL for short [16, 17]. ADL is a graphical-based language that allows a developer to express a design in terms of a collection of communicating processes quite similar to the CSP-model of communication [10]. A distinction is made between a **structure model** and a **behavior model**, as we shall briefly explain next.

3.1.1 An ADL structure model

An ADL structure model reflects the structure of an application expressed in terms of a collection of **processes** that communicate by means of **communication media** based on a message-passing paradigm. A process is used to model a logical entity capable of transforming incoming data which can then be passed to another process. Contrary to, for example, data flow diagrams, we have made the *means* of communication more explicit. For example, **synchronous channels** in ADL are used to model unbuffered, blocking communication between several senders and receivers. In addition, **message queues** in ADL can be used to express buffered communication.

As a simple example, consider a part of a flight simulator which manages the fuel system. In particular, we assume that this subsystem calculates the consumed fuel at regular intervals. To this aim, it receives the current engine speed from a control unit whenever the speed of the aircraft changes. At the same time, it should be able to supply the amount of fuel left to other units: a display, and a unit which periodically calculates the

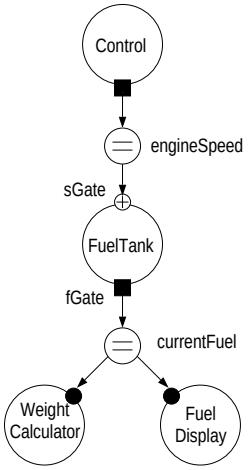

Figure 3: The structure model of a simple fuel tank system.

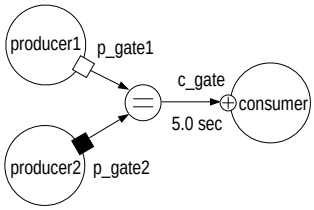

Figure 4: A simple producer-consumer system based on various forms of timed communication.

weight of the aircraft. The structure model of this small system can be constructed as shown in Figure 3. All communication has been expressed in terms of synchronous channels, which are denoted by the symbol "⊜".

Formally, diagrams such as shown in Figure 3 are referred to as **designs**. A process in a design can be decomposed into a collection of constituent processes, yielding a subdesign. Similar to the approach followed in Mascot [13], the interface of a process is made explicit by means of so-called **input** and **output gates**. They allow for specifying *when* communication through that gate should commence. Three types of communication at gates are distinguished in ADL:

- In the case of **blocked communication** at a gate *g*, a process which is waiting for communication via *g* will not proceed until data transfer through *g* has actually taken place.
- In the case of **non-blocked communication** at gate *g*, a process will never wait until communication through *g* takes place, unless it can take place immediately.
- Finally, **delayed communication** indicates that a process is willing to wait for communication until a specified amount of time has elapsed.

To illustrate the semantics of timed communication, consider a simple producer-consumer system as shown in Figure 4. In this case, we have modeled two producers and a single consumer that communicate by means of a synchronous channel. The timing at gate p_gate1 is subject to non-blocked communication (indicated by "□"), whereas the timing at gate c_gate is subject to a 5 second delayed communication (indicated by "⊕"). From the perspective of the consumer, communication proceeds as follows. If one of the producers is prepared to send data within 5 seconds from

the moment that the consumer is prepared to receive data, communication succeeds. Otherwise, after 5 seconds have elapsed, the consumer withdraws its willingness to communicate, so that no data exchange can take place. Similarly, producer1 will only send its data if the consumer is waiting for the data the instant that producer1 wants to communicate. Otherwise, communication is cancelled altogether. Process producer2 will always wait until it can send its data to the consumer, which is indicated by the symbol "■"

ADL further supports various forms of group communication by means of multicast constructs for each of the available communication media. Specification of replicated structures is also supported by the language. Again, we shall not go into any further details here, but refer to [16].

3.1.2 Expressing behaviors

Modeling the behavior of an activity in ADL is done by means of state-transition machines. Each process which is not further decomposed has precisely one associated state-transition machine. In ADL we have chosen to use a form by which a developer can focus on *communication* entirely. This means that we are not initially interested in control flow and data transformations that do not immediately relate to parallelism. In the following we shall briefly describe our notion of state-transition machines, and conclude with a simple example.

Processing states. Processing states (drawn as rectangles in ADL) are exclusively used for modeling data transformations. This means that while a process is residing in a such a state no communication with other processes takes place. In order to describe the actual behavior that takes place within a processing state, a developer may directly attach source code to a processing state.

Communication states. Communication states describe the situation in which a process is involved in communicating data through one of its gates. Each communication state is associated with exactly one of the gates attached to the process. This leads to a further distinction between **input states** (designated as "⬠") and **output states** (designated as "⬠"). While a process is residing in a communication state, it attempts to send or receive information via the gate to which the state is associated. When communication takes place is determined by the timing of the state, which, just as in the case of gates, can either be blocked, non-blocked, or delayed.

Transitions. Possible transitions between states are shown as arcs in state-transition machines. Processing states may have multiple outgoing transitions and the one actually selected can be dependent on the data transformations within such a state. In the case of communication states there are only two possibilities for making a state transition depending on whether the communication could take place or not. To this aim, a gate can raise two types of events: (1) a **transfer event** is raised whenever data has passed through that gate, or (2) a **timeout event** is raised whenever communication cannot take place because the timing constraints cannot be met. Whenever a process is residing in a communication state it can only make a transition to a next state on the occurrence of an event. In the case of a timeout event, the next state is designated according to a dashed arc; otherwise, in the case of a transfer event, the transition represented by a solid arc is made.

Select states. In many cases, a process in a parallel application may reach a point in which it should select from a set of alternative communications. These situations can be modeled

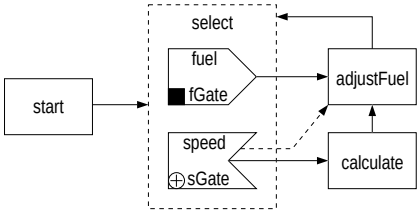

Figure 5: A state-transition machine for the process FuelTank of example system.

in ADL by means of a **single select state**. A single select state consists of two or more communication states, represented by enclosing them in a dashed box. Upon the occurrence of an event associated with one of these constituent states, a transition will be made from that state to the specified next state. In this sense, single select states in ADL resemble the select statement in Ada, or the alt statement in OCCAM. The main difference lies in the fact that select states in ADL may contain input as well as output states. In Ada and OCCAM it is only possible to select from a number of *incoming* messages.

To illustrate our notion of state-transition machines, consider the behavior description of our fuel tank system which is shown in Figure 5. By modeling the communication with the outside world as a single select state select, we now have the following situation:

- Due to the fact that the select state contains a delayed input state, FuelTank will never indefinitely wait until communication with outside world takes place. In particular, if no other activity wants to communicate, FuelTank will occasionally update the fuel level by making a transition to state adjustFuel.
- Issuing a request for the current fuel level is modeled by means of unidirectional blocked synchronous communication: if FuelTank can pass on the current fuel level via gate fGate while residing in the select state, it will do so. Otherwise, it will eventually respond to communication via gate sGate, or a timeout at that gate. In other words, we need not explicitly model a request for the current fuel level by means of data sent *from* the requesting activity *to* the fuel tank.

3.1.3 Designing the target architecture

So far, we have concentrated on designing the software components of a system. In the case of parallel application development, it is also necessary to pay attention to designing the target hardware. To this aim, Hamlet provides a graphical form of the INMOS Network and Description Language [8], referred to as GNDL. For example, a simple transputer network consisting of six T805 transputers can be described as shown in Figure 6. Detailed information on the actual performance of a transputer platform is supplied by our system. In particular, we have made detailed models of various transputer configurations, and use these models in order to properly simulate a design.

3.2 System implementation

Implementation in Hamlet is primarily supported by means of automated code generation. The place of the corresponding tools within the development system is shown in Figure 7.

Using GNDL we immediately generate the hardware configuration files necessary to configure a target transputer platform.

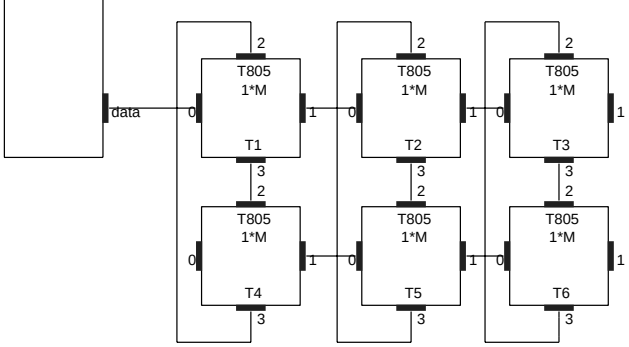

Figure 6: An example of a simple transputer network expressed in GNDL.

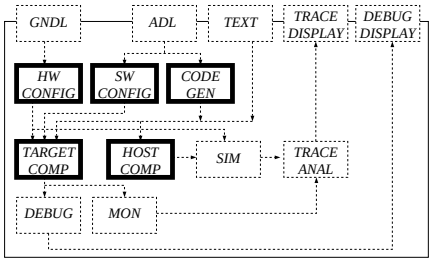

Figure 7: The implementation components.

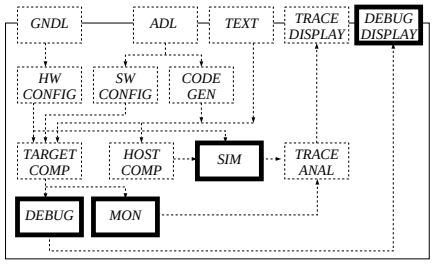

Figure 8: The execution components.

By providing an easy-to-use graphical interface, a developer is encouraged to experiment with various configurations. In a similar fashion, we use an ADL structure model to generate the software configuration files. To this aim, a developer must specify to which transputer each ADL process is to be mapped. Apart from the mapping information, all other configuration information is generated automatically.

The structure model also provides all necessary information to generate skeleton target code containing *static* information. Roughly speaking, this means that we have all the information available to generate declarations of variables and function signatures, as well as the initialization and finalization sections of executable code. At present, we are using C as our target language, augmented with calls to the RTSM real-time communication library [3].

But problems start to arise when code needs to be generated for either (1) data-dependent control flow structures, or (2) communication structures that do not have an equivalent construct in our communication library.

Data-dependent control flow. To solve the first problem, we explicitly need to consider the state-transition machines that constitute a behavior model. In order to generate fully executable code, we currently only permit a developer to attach statistical information on expected computational delays and branching in processing states. This information will then allow us to generate executable *simulation code* for an application. At present, we are working towards an adequate means for attaching minimal source code in order generate a complete implementation.

Advanced communication structures. ADL supports a number of communication structures that have no direct equivalent in any communication library. Amongst these are multiple senders and receivers communicating via a synchronous channel, network-wide message queues, multicasting constructs, etc. The question is, if each communication media can be adequately supported by an efficient implementation. Also, select states in state-transition machines which contain output states, generally have no counterpart in implementation languages. Because we are dealing with real-time applications, we shall not impose *any* implementation that could possibly have an adverse effect on performance. Consequently, our system will not generate code for those communication structures that cannot be supported by an *application-independent* efficient implementation. In these cases, we leave it up to the developer to provide an implementation. In all other cases, however, our system will generate target code.

3.3 System execution

As we have already indicated, the Hamlet approach is based on two execution schemes: one by which a design is simulated, and one by which an implementation is executed on a target machine. The position of the supporting tools are shown in Figure 8.

Simulation proceeds by simply compiling and linking an application to a library with approximately the same interface as the actual communication library. Additional functions merely address simulation aspects (such as delays and distribution functions). During simulation, experiments can be conducted such as interactively changing the network model or the mapping of processes onto processors. In the end, the simulator will generate a trace file for further analysis.

The actual execution on a target platform is supported by a debugging and monitoring system which is based on the INMOS Inquest Toolset. During actual execution of an application, the same information is gathered as can be done by means of simulation. Consequently, the simulator and monitoring system produce (post-mortem) trace files that have exactly the same format. These trace files can subsequently be processed by an analysis system.

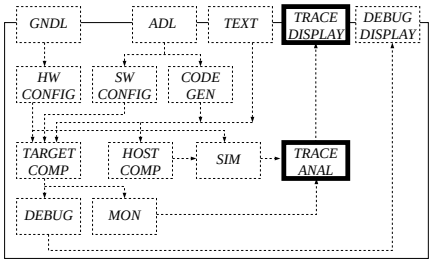

Figure 9: The analysis components.

3.4 System analysis

A prominent component that supports our experimental design approach, is formed by an advanced trace analysis system, as shown in Figure 9. This subsystem, named TATOO is based on the analysis and visualization tools PATOP and VISTOP developed as part of the TOPSYS environment [1].

The functionality of the analysis system is quite comparable to other behavior analysis tools such as, for example, ParaGraph [9]. By installing a software monitor on each processor of the target machine, measurement figures can be provided for active times of processes or processors; delays with respect to ready queues, communication calls, and I/O; as well as information on communication bandwidth. Monitoring need not be done globally. Instead, restricted monitoring in time (e.g. functions and marked areas) or in space (e.g. for a number of processes or processors) is supported.

Clearly, the analysis tool is the primary means to provide feedback to a developer on the actual performance of the application. An important aspect is that the analysis tool provides exactly the same interface for both execution modes supported in Hamlet (i.e. simulation or target execution).

4 Discussion and conclusions

The distinctive feature of the Hamlet approach towards computer-aided support for developing parallel real-time applications is that an integrated *experimentation* environment is provided. Unlike many other approaches, our goal is to support experimentation starting at the level of system design, as we believe that it is here that exploitation of parallelism manifests itself for the first time as a development criterion. To this aim, we have developed a new design method that takes exploitation of parallelism explicitly into account. To our knowledge, only a very few design methods currently exist that share this feature, and most of them, just as ours, are still in a research phase (see e.g. [12, 15]).

A protoype version of our Application Development System has been installed at the developer's sites. Yet, much work still needs to be done. In particular, automated generation of industrial-quality parallel target code and support for accurate simulations needs further attention (see also [19]). On the other hand, the generation of configuration files from ADL and GNDL designs have proven to be extremely useful. The more traditional tools (i.e. for debugging and analysis) have also shown to suffice quite well.

References

[1] T. Bemmerl. "TOPSYS for Programming Distributed Multiprocessor Computing Environments." In *Proc. Computer Systems and Software Engineering (CompEuro)*, pp. 175–180, The Hague, May 1992.

[2] M.R. Cagan. "The HP Softbench Environment: An Architecture for a New Generation of Software Tools." *Hewlett-Packard Journal*, pp. 36–68, June 1990.

[3] Hamlet Consortitium. "RTSM Description and Preliminary User Manual." Techn. Rep., Parsytec Industriesysteme, Aachen, Germany, January 1993.

[4] J.E. Cooling. *Software Design for Real-time Systems*. Chapman & Hall, London, 1991.

[5] R.E. Fairley. *Software Engineering Concepts*. McGraw-Hill, 1985.

[6] D. Harel. "STATECHARTS: A Visual Formalism for Complex Systems." *Science of Computer Programmming*, 8(3):231–274, 1987.

[7] D. Harel, 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." *IEEE Trans. on Softw. Eng.*, 16(4):403–414, 1990.

[8] N. Haydock. "NDL Hardware Configuration Language Reference Manual." Techn. Rep. SW-0308-10, INMOS Limited, June 1992.

[9] M.T. Heath and J.A. Etheridge. "Visualizing the Performance of Parallel Programs." *IEEE Software*, 8(5):29–39, September 1991.

[10] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[11] I. Jacobson. *Object-Oriented Software Engineering, A Use Case Driven Approach.* Addison-Wesley, 1992.

[12] I.E. Jelly, I. Gorton, and J. Gray. "Using PARSE for Transputer Software Development." In R. Grebe et. al, (ed.), *Transputer Applications and Systems*, IOS Press, Amsterdam, 1993.

[13] H.R. Simpson. "The Mascot Method." *IEE Softw. Eng. J.*, 1(3):103–120, May 1986.

[14] J.P. Singh, J.L. Hennessy, and A. Gupta. "Scaling Parallel Programs for Multiprocessors: Methodology and Examples." *Computer*, 26(7):42–50, July 1993.

[15] C. Steigner, R. Joostema, and C. Groove. "PAR-SDL: Software Design and Implementation for Transputer Systems." In R. Grebe et. al, (ed.), *Transputer Applications and Systems*, IOS Press, Amsterdam, 1993.

[16] M.R. van Steen. "The Hamlet Application Design Language, Introductory Definition Report." Techn. Rep. EUR-CS-93-16, Erasmus University Rotterdam, December 1993.

[17] M.R. van Steen, T. Vogel, and A. ten Dam. "ADL: A Graphical Design Language for Parallel Real-Time Applications." In R. Grebe et. al, (ed.), *Transputer Applications and Systems*, IOS Press, Amsterdam, 1993.

[18] M.R. van Steen. "The Hamlet Design Entry System, An Overview of ADL and its Environment." Techn. Rep. EUR-CS-94-02, Erasmus University Rotterdam, April 1994.

[19] M.R. van Steen. "The Hamlet Application Design Language, On the Implementation of Synchronous Channels." Techn. Rep. EUR-CS-94-03, Erasmus University Rotterdam, May 1994.