Report EUR-CS-94-2
April 1994

The Hamlet
Design Entry System
An overview of ADL and its environment[1]

Maarten R. van Steen
Erasmus University, Faculty of Economics
Department of Computer Science
P.O. Box 1738, 3000 DR Rotterdam
e-mail: steen@cs.few.eur.nl

Armand ten Dam
Teus Vogel
TNO Institute of Applied Physics
P.O. Box 155, 2600 AD Delft
e-mail: {tendam,tvogel}@tpd.tno.nl

[1]A concise version of this report, entitled "Computer-aided Support for Designing Parallel Real-Time Solutions on Transputer-based Systems," is to be presented at the 3rd IEEE Conference on Control Applications, Glasgow, August 1994.

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. This need illustrates that developing parallel real-time applications is indeed a difficult and tedious task. 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. In the paper we primarily concentrate on the support for application *design*, as we feel that parallelism should essentially be addressed at this level.

Keywords: parallelism, real-time systems, application design, software development experimentation.

Contents

1 Introduction **2**

2 The design phase in a system development cycle **4**

2.1 Traditional design approaches 4

2.2 Experimental development during design 5

3 The Hamlet approach **8**

3.1 System design 9

3.1.1 An ADL structure model 10

3.1.2 Expressing behaviors 12

3.1.3 Process replication 14

3.1.4 Designing the target architecture 17

3.2 System implementation 18

3.3 System execution 20

3.4 System analysis 20

4 Discussion and conclusions **23**

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*. In particular, solutions have been targeted towards increase of scalability of scientific applications [22]. 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. In particular, they do not provide the right means for expressing process replication and resource usage.

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. However, this approach is hardly supported by current design methods used in the realm of parallel real-time application development. In this paper we describe how such an integrated approach can be realized. In particular, 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. In particular, we describe an approach that is currently being implemented in the ESPRIT project *Hamlet*.

The paper is organized as follows. In Chapter 2 we briefly discuss the traditional approaches towards software design, and address the question why they do not adequately support development of parallel applications. In addition, we also present an

outline of our approach which is based on experimentation. The bulk of the paper is presented in Chapter 3 in which we further describe our approach by focusing on design, implementation, execution, and analysis of applications. A main emphasis is put on support for system design as we feel it is here that exploitation of parallelism in a solution should primarily be addressed. We conclude our presentation in Chapter 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 [6]: 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 [5]:

1. *Functional structuring* by which the structure of the system is described in terms of functional interaction of the various components. Typically, data-flow based techniques as introduced by Yourdon [29] and extended for real-time system development by, for example, Ward/Mellor [28] and Hatley/Pirbhai [9] are examples of this approach.
2. *Object structuring* by which the system is decomposed into a collection of communicating objects, where each object is responsible for a specific function. This has become more familiar as object-oriented design, and techniques such as developed by Booch [2] and Rumbaugh et al. [20] are gaining wide-spread popularity.
3. *Data structuring* that puts a main emphasis on the data to be manipulated in a system. Typically, when using a data structuring approach, a developer concentrates on data modeling, and modeling the events that change data. An example of a data structuring method is JSD [13].

But 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 [14], developing industrial 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. The validity of this statement can certainly not be underestimated when it is decided to exploit parallelism as a means for achieving performance demands.

Unfortunately, to our opinion, it is precisely with respect to these three fundamental issues that traditional design methods lack sufficient support, although their inventors often claim otherwise. As 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*.

When addressing parallelism as a modeling vehicle, designers are, of course, confronted with the traditional problems of concurrency: communication and mutual exclusion. This means that effort has to be put in designing solutions that are, for example, deadlock free or free from starvation. However, it is naive to state that from the perspective of performance enhancement, one can then, for example, subsequently map processes onto multiple processors: exploitation of parallelism is more than just a resource allocation problem. Instead, 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. It is here that, to our opinion, traditional methods fail when applied to development of parallel real-time applications.

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. In the next chapter, we shall see how specifications for process replication can address this aspect. 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.

Traditionally, analytic models such as (stochastic) Petri Nets [18] or (closed) queuing networks [19] have been used for performance modeling and evaluation. As examined by Jonkers [17], a problem with this approach is that a trade-off must be made between analytical complexity and expressive power, a reason why combinations of both formalisms are often sought. Another disadvantage, as we see it, is that analytic models generally are only loosely related with the structural aspects of an application

design. Design fundamentals such as abstraction, modularity, or information hiding have no counterpart in such models.

So, if preliminary performance evaluation is to be incorporated in a design method, its means should be integrated with the traditional structuring approaches mentioned above. In the first place this implies that, at the least, unambiguous *behavioral semantics* should form an integrated part of the design method. Expressing behavior can be decomposed into two constituents:

1. A behavioral model of the application in terms of some abstract execution mechanism, incorporating parallelism, and thus also communication and synchronization.
2. A description of the mapping of this behavioral model onto a (parallel) target machine, such that the execution semantics are completely preserved.

In this paper, we advocate the use of state-transition machines to describe the behavior of an application. In addition, communication between a collection of such machines should adhere to unambiguous semantics.

In the second place, the actual evaluation of the behavior of a design should also form part of the design method. As we have argued, analytic models are, in our view, inappropriate as a general means to this end. (Of course, analytic methods should be used to validate designs against those timing constraints which under no circumstances may be violated.) Instead, 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. This experimental approach towards parallel real-time software development has been adopted in the ESPRIT project *Hamlet*.

In particular, a design in terms of Hamlet 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 on account of actual performance results when executing the implementation on the target

machine. 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 chapter we shall take a closer look at how this experimental approach towards parallel application development is realized in practice.

3

The Hamlet approach

Within Hamlet, we roughly distinguish four different activities as part of the development process: design, implementation, execution, and analysis. These four activities are highly integrated, but each is supported by different tools. In this chapter we consider each of these activities in more detail, thereby putting the most emphasis on the actual support for constructing and evaluating designs. The global architecture of our Application Development System is shown in Figure 3.1. 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) which generates the necessary files to configure a transputer network.
- 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 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 sections we shall a closer look at these components.

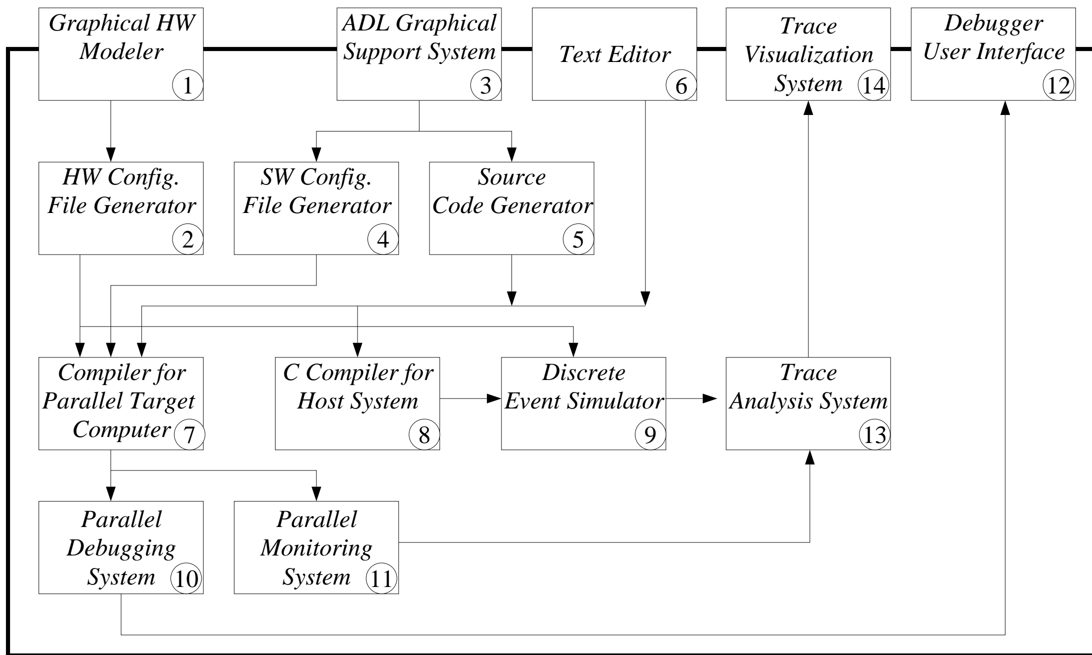

Figure 3.1: The global architecture of the Hamlet Application Development System.

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 3.2.

Software designs are expressed in the so-named *Hamlet Application Design Language*, or ADL for short [24, 26]. ADL is a graphical-based language that allows a developer to express a design in terms of a collection of communicating processes

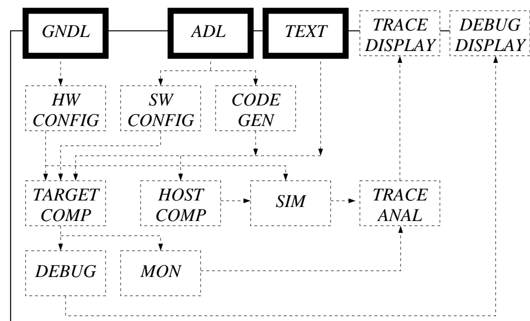

Figure 3.2: The design components.

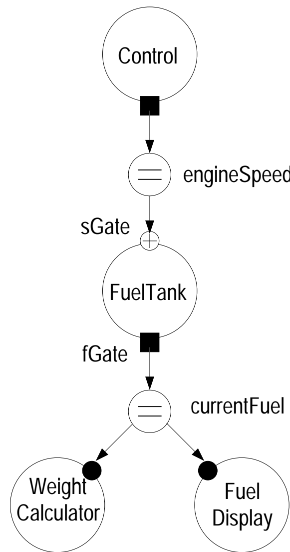

Figure 3.3: The structure model of a simple fuel tank system.

quite similar to the CSP-model of communication [12]. 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. Similar to data flow diagrams, a process is used to model a logical entity capable of transforming incoming data or tokens which can then be passed to another process. Contrary to data flow diagrams, however, 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, in particular a display and a unit which periodically calculates the weight of the aircraft. The structure model of this small system can be constructed as shown in Figure 3.3. All communication has been expressed in terms of synchronous channels, which are denoted by the symbol "⊜".

Formally, diagrams such as shown in Figure 3.3 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 [21], the interface of

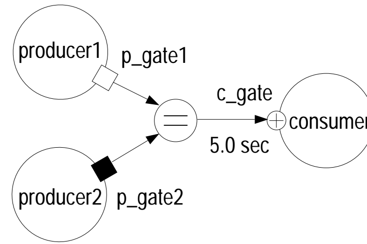

Figure 3.4: A simple producer-consumer system based on various forms of timed communication.

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 or token transfer through g has actually taken place. (Blocked communication is represented by the symbols "●" and "■", respectively.)
- 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. (Non-blocked communication is represented by the symbols "○" and "□", respectively.)
- Finally, **delayed communication** indicates that a process is willing to wait for communication until a specified amount of time has elapsed. (Delayed communication is represented by the symbols "⊕" and "⊞", respectively.)

It is important to note that these forms of communication relate to the moment when communication should take place *as required by the communicator*, and if this requirement could not be met communication is cancelled all together. This is different from (a)synchronous communication which involves *all communicating parties*, and which is, in principle, never cancelled.

To illustrate the semantics of timed communication, consider a simple producer-consumer system as shown in Figure 3.4. 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, whereas the timing at gate c_gate is subject to a 5 second delayed communication. 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

all together. It should now be clear that producer2 will always wait until it can send its data to the consumer.

ADL further supports various forms of group communication by means of multicast constructs for each of the available communication media. Again, we shall not go into any further details here, but refer to [24].

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. State-transition machines normally consist of a single notion of a state, and transitions between states can only occur as the result of an event. In ADL, however, 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 used for exclusively modeling data transformations. This means that while a process is residing in a such a state no communication with other processes takes place. The question that comes to mind is how one can describe the actual behavior that takes place within a processing state. As state-transition machines in ADL are primarily intended to support design of communication, further specification of data transformations is not possible in our language. Instead, a developer may directly attach source code to a processing state. (We note that we are currently investigating how adding data transformations in this way can be adequately supported.) As we shall see, this code will later be inserted directly when generating an implementation (either for simulation or target execution purposes) from a design.

Communication states.

Communication states describe the situation in which a process is involved in communicating data or tokens 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. *How* communication proceeds is entirely dependent on the communication medium to which the gate is connected. *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. However, *which* transition will be made can only be partially specified in our language. In particular, processing states may have multiple outgoing transitions and the one actually selected can be dependent on the data transformations within such a state. As we have mentioned above, these transformations cannot be modeled at the level of state-transition machines. We shall return to this issue further below.

The situation is different in the case of communication states. Here, there are only two possibilities for making a state transition depending on whether the communication could take place or not. To this aim, gates can raise a so-called **event**:

- a **transfer event** is raised by a gate whenever data or a token has passed through that gate,
- a **timeout event** is raised whenever communication through the gate cannot take place because the timing constraints cannot be met.

Now, 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. Which transition is made is specified by means of an arc from that communication state to the next state. 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 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.

Alternatively, **total select states** are used to describe the behavior of a process waiting for a number of communications to take place. The difference with a single select state is that a next transition will only be made as soon as *all* communications have succeeded. The order, however, in which communication takes place, is irrelevant.

To illustrate our notion of state-transition machines, consider the behavior description of our fuel tank system which is shown in Figure 3.5. By modeling the communication with the outside world as a single select state select, we now have the following situation:

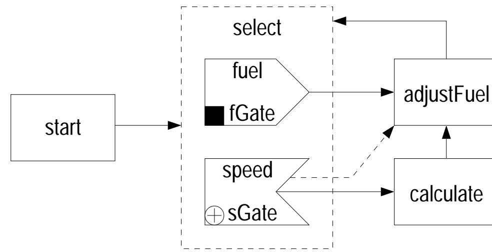

Figure 3.5: A state-transition machine for the process FuelTank of example system.

- 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 datum or token sent *from* the requesting activity *to* the fuel tank.

3.1.3 Process replication

Although a structure and a behavior model reflect inherent, functional parallelism [15] it would be erroneous to state that this parallelism is targeted towards achieving performance. In this sense, the modeling support presented so far captures the traditional support that is provided by most design methods used today. And as we have argued in Section 2.1, we need more when dealing with the design of applications that exploit parallelism for the sake of meeting performance demands. To this aim, ADL provides a means for specifying process replication. Process replication in ADL is entirely constructed by means of model annotations. These annotations are not considered as part of the language core, but instead, are merely abbreviations to concisely express repetitive structures. These structures are expressed by using so-called **replicators** and **connection statements**.

A replicator consists of a *replication factor* which is a positive integer, say N, and a *replication variable* which is an integer variable taking values from the set $\{0,\ldots,N-1\}$. A replicator with replication factor N and replication variable j is graphically denoted as

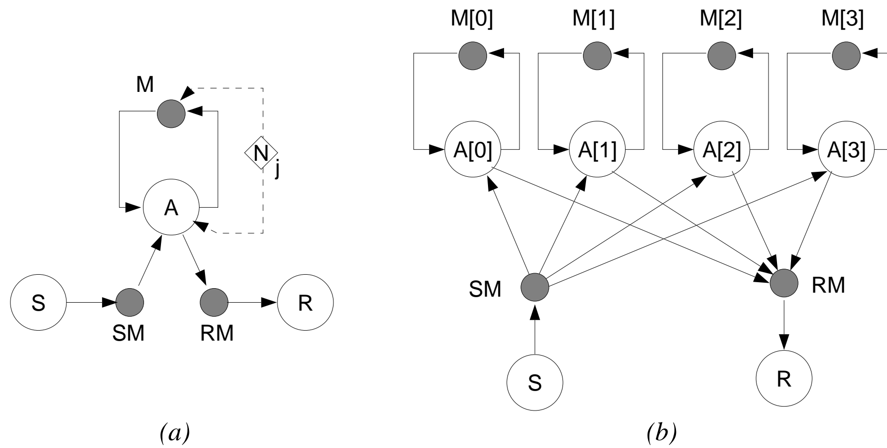

Figure 3.6: The result of applying a replicator with replication factor 4.

A replicator can be applied within a design by selecting a number of processes and communication media. Such a set is identified by drawing a dashed arc from the replicator to each object that takes part in the replication. The only restriction we impose is that a communication medium can only be selected if all processes that are connected to it, are selected for replication as well (we shall return to this later). The effect of replication on a collection of processes and communication media effectively boils down to replicating the complete substructure. In addition, each communication medium that is not part of the replication but that was connected to a process P subject to replication, is subsequently connected to each replica P_i.

To illustrate, consider the replication specification shown in Figure 3.6(a), consisting of three processes S, R, and A. Also, there are three communication media SM, RM, and M. (For the sake of clarity, we have omitted further specification of the type of the respective communication media, and have also not specified any gates.) The replicator (N, j) is applied to A and M, which results in the altered design shown in Figure 3.6(b) in the case that $N = 4$.

The condition that a communication medium can only be selected for replication if all processes connected to it are selected as well, may seem rather restrictive. What it establishes is that no *additional* connections will ever be made to a process on account of replication. Consequently, the interface of a process is left intact.

Replication introduced so far is not sufficient to be used as a means for specifying regular communication structures. Examples of such structures are pipelines, meshes, hypercubes, etc. To that aim, we need a means of specifying how the actual connections between processes and communication media should be provided. In ADL, this can be done by annotating connections with so-called **connection statements**. Connection statements come in two forms: redirection statements and attachment constraints.

A redirection statement can be attached to a connection from a source object S to a target object T, where both S and T should be subject to replication. Such a statement

takes the general form

$$i \rightarrow E(i), \text{ with } 0 \leq i < N$$

where we assume that N is the replication factor. $E(i)$ is an integer function denoting the target object *after* replication. To illustrate, assume there was a connection from process P to a communication medium c, and both P and c are subject to replication. Assuming that we annotated this connection with the redirection statement "$i \rightarrow i + 1$", then, after replication, the connection will be directed from P_i to c_{i+1}, instead of the default connection from P_i to c_i. In those cases that $E(i) < 0$ or $E(i) \geq N$, the connection is discarded all together.

An attachment constraint can be attached to a connection from a source object S to a target object T, where either S or T should be subject to replication, but not both. A constraint takes the general form

$$i \textbf{ relop } C(i), \text{ with } 0 \leq N$$

where, again, N is assumed to be the replication factor. In this case, **relop** denotes a standard mathematical relationship operator, and $C(i)$ is an integer expression. To illustrate, assume that process P is connected to a communication medium c, and that only P is subject to replication. The attachment constraint "$i = (i \div 2) \times 2$" implies that after replication, there will only be a connection from those replica's P_i to c for which the index is even. All other connections from P_i to c are discarded. In those cases that not a single connection can be made, the constraint is considered to be at fault.

In both cases, i.e. when using redirection statements or attachment constraints, communication media are also discarded if replication leads to incorrect designs.

To illustrate, reconsider the replication specification shown in Figure 3.6(a). In order to specify a pipeline of processes, we attach connection statements as shown in Figure 3.7(a). Now assume in this case that the replication factor $N = 5$. Then, because no redirection statement has been attached to the connection from A to M, there will be a connection from each replicated process A[j] to M[j], where $0 \leq j \leq 4$. The redirection statement

$$j \rightarrow j + 1$$

attached to the connection from M to A, specifies that, after replication, for each j, with $0 \leq j \leq 4$, M[j] should be connected to A[j+1]. However, because M[4] cannot be connected to A[5] for the simple reason that A[5] does not exist, this connection is discarded. This would imply that only the connection A[4] to M[4] would exist which violates the syntax rules of ADL. Consequently, M[4] is discarded all together. This single redirection statement then leads to the pipeline of processes A[0] … A[4] shown in Figure 3.7(b).

Now consider the two connection constraints. In the first place, the constraint "$j = 0$" attached to the connection from SM to A, specifies that the only connection

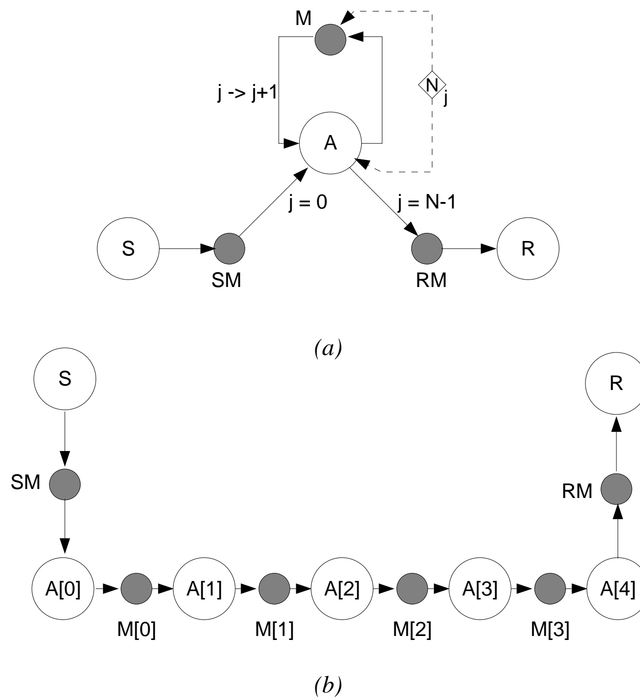

Figure 3.7: The specification of a pipeline of processes using replication.

that is to be made after replication, is the one from SM to A[0]. Similarly, “$j = N - 1$” implies that there will only be a connection from A[4] to RM.

Complex replications can also be formulated in ADL, but are not considered here. For such replications, as well as formal definitions of replications in general, we again refer to [24].

3.1.4 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. In particular, it is necessary to describe the parallel machine in terms of processor characteristics, topology, etc., as these aspects will generally influence the performance of the application. To this aim, Hamlet provides a graphical form of the INMOS Network and Description Language [10], referred to as GNDL. For example, a simple transputer network consisting of six T805 transputers can be described as shown in Figure 3.8.

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. The developer need merely provide the topology and some global characteristics such as transputer type, memory size, etc.

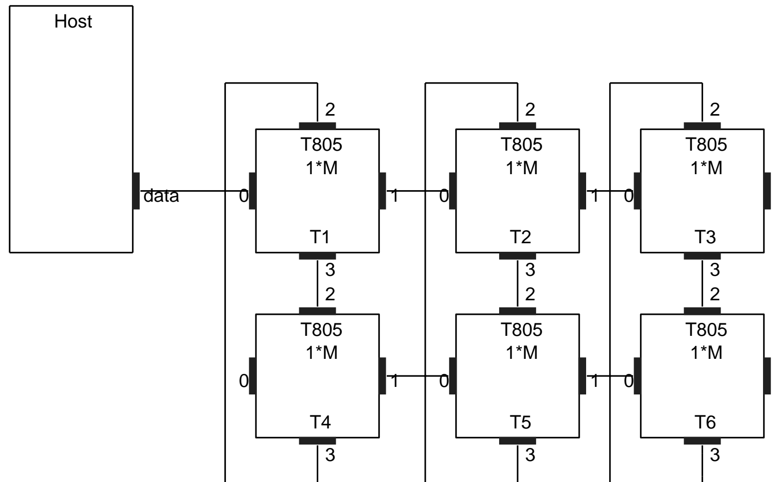

Figure 3.8: An example of a simple transputer network expressed in GNDL.

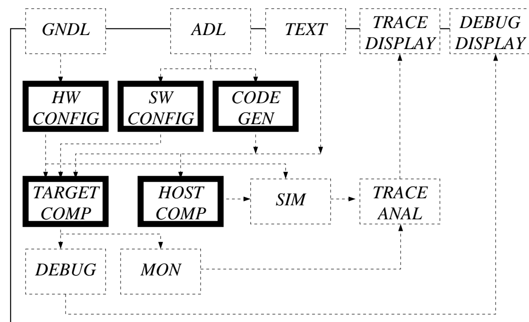

Figure 3.9: The implementation components.

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 3.9.

Our system design languages ADL and GNDL have been purposefully designed in such a way that automated code generation can be supported. To start with, using GNDL we immediately generate the hardware configuration files necessary to configure a target transputer platform. It is thus no longer necessary to construct these files by hand, which is generally considered as a harsh and error-prone process. 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. In a future version of our support environment, we shall provide heuristic algorithms that generate a (sub)optimal mapping automatically. Apart from the mapping information, all other configuration information such as process

descriptions, process connections, etc., are generated automatically from a structure model.

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. Furthermore, due to the fact that communication structures can be directly derived from a structure model, we are also capable of generating the necessary interface to communication libraries. At present, we are using C as our target language, augmented with calls to the RTSM real-time communication library [4].

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. Let us take a closer look at these two issues.

Data-dependent control flow.

To solve the first problem, we explicitly need to consider the state-transition machines that constitute a behavior model. It should be clear that we can generate the necessary target code for data-independent control flow structures on a per process basis by simply considering a state-transition machine. 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, possibly directed towards a specific implementation through annotations provided by the developer.

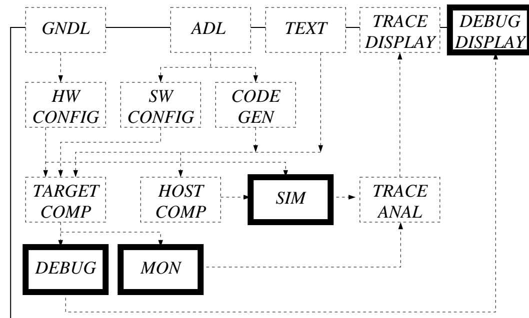

Figure 3.10: The execution components.

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 3.10.

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). The simulator assumes a configuration file, generated from a GNDL model. 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 which is discussed next.

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 3.11. 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 [11]. 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

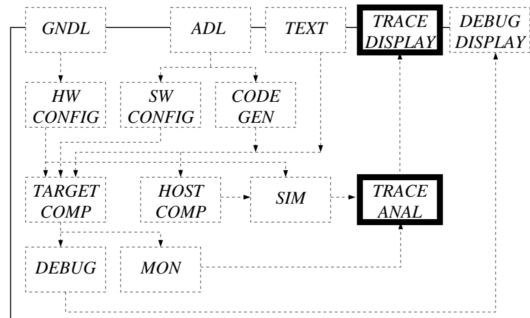

Figure 3.11: The analysis components.

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.

The analysis system is organized into three separate layers: a *universal layer*, a *control layer*, and a *display layer*.

The universal layer.

The lowest layer is concerned with mapping the events associated with logical measurements (such as e.g. processor utilization or lengths of system queues) to actual physical measurements that have to be provided by the software monitor for each processor. Typically, the universal layer provides a number of trace buffers which contain the actual data generated by the software monitors.

The control layer.

The control layer lies at the heart of the analysis system and is responsible for interpreting the traces generated through the universal layer. In this sense, the control layer is responsible for transforming logical events into actual measures that can subsequently be displayed in a later stage. For example, the control layer may relate changes in the lengths of various ready queues to a global ordering in time, so that weighted averages of queue lengths in time-intervals can actually be measured.

The display layer.

The display layer provides a variety of diagrams for displaying measurements. Displays include multicurve diagrams to relate multiple time-dependent measurements, histograms, distribution graphs, and Gantt charts. In addition, matrix diagrams are used to reflect “hot spots” per measurement/per processor by using value-related colors for each element in the matrix.

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. [16, 23]). Unique in our design approach is the fact that we support development of parallel *real-time* applications.

In order to allow for flexible experimentation, we have recognized that at least two demands should be met:

1. preliminary evaluation of designs, implying that incomplete implementations should be amenable to performance analysis.
2. seamless integration of design and implementation in order to avoid that the (usually manual) transformations of a design into an implementation lead to target code that later has to be considerably modified.

The first requirement is supported in our approach by applying discrete simulation techniques. This idea is, of course, not new. The important aspect, however, is that the models we simulate are exactly those that are used during the actual process of system design. To this aim, hardware models are provided by a developer and are used as additional input for the simulator.

The second requirement is supported by automated code generation. To this aim, we needed to carefully construct our design technique to include unambiguous behavioral semantics, and which could be used to generate efficiently executable parallel code. To a certain extent, this approach has also been successfully applied in the STATEMATE environment [8]. STATEMATE uses the concept of statecharts [7], a form of finite state machines which have been used to generate prototype Ada programs. And

although STATEMATE does address the problem of real-time application development, exploitation of parallelism is not explicitly supported.

Our experimentation approach is further completed by, more or less traditional support for monitoring, debugging, and analysis of executions.

A prototype version of our Application Development System has been installed at the developer's sites for evaluation. The prototype system consists of a graphical support system for (a restricted version of) ADL and GNDL, including generation of hardware and software configuration files.

The feedback we have received so far indicates that our approach may indeed be successfully applicable for parallel real-time application development. 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. 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. Based on this feedback, we are now concentrating on the following items:

- Further development of ADL, in particular its support for automated code generation.
- Development of realistic hardware models for various transputer systems, and incorporation of these models into the simulation system.
- Fine-tuning of the analysis system, in particular by extending its functionality with respect to displaying information.
- Full integration of the tools by means of a *Broadcast Message Server*, comparable to Hewlett-Packard's Softbench [3].

At the moment of this writing, we have installed the second version of our design entry system at the application developer's sites. Work with respect to the aforementioned issues is ongoing. For example, ADL is currently being augmented with data typing facilities [27], and (distributed) algorithms for supporting its implementation are under way [25].

Bibliography

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

[2] G. Booch. "Object-Oriented Development." *IEEE Transactions on Software Engineering*, SE-12(2):211–221, 1986.

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

[4] Hamlet Consortitium. "RTSM Description and Preliminary User Manual." Hamlet Technical Report, Parsytec Industriesysteme, Aachen, Germany, January 1993.

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

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

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

[8] 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 Transactions on Software Engineering*, 16(4):403–414, 1990.

[9] D.J. Hatley and I.A. Pirbhai. *Strategies for Real-Time Specifications.* Dorset House, New York, 1987.

[10] N. Haydock. "NDL Hardware Configuration Language Reference Manual." Internal Technical Report SW-0308-10, INMOS Limited, June 1992.

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

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

[13] M. Jackson. *System Development.* Addison-Wesley, 1983. in Dutch.

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

[15] L.H. Jamieson. *Characterizing Parallel Algorithms*, In L.H. Jamieson and D.B. Gannon and R.J. Douglass, (ed.), *The Characteristics of Parallel Algorithms*, chapter 3, pp. 65–100. MIT Press, Cambridge, Mass., 1987.

[16] I.E. Jelly, I. Gorton, and J. Gray. "Using PARSE for Transputer Software Development." In R. Grebe and J. Hektor and S. Hilton and M.R. Jane and P.H. Welch, (ed.), *Transputer Applications and Systems*, volume 2. IOS Press, Amsterdam, 1993.

[17] H. Jonkers. "Introduction to Probabalistic Performance Modelling of Parallel Applications." Technical Report 1-68340-44(1993)04, Faculty of Electrical Engineering, Delft University of Technology, April 1993.

[18] T. Murata. "Petri Nets: Properties Analysis and Applications." In *Proceedings of the IEEE*, volume 77, pp. 541–580. IEEE, 1989.

[19] R.O. Onvural. "Survey of Closed Queuing Networks with Blocking." *Computing Surveys*, 22(2):83–122, June 1990.

[20] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design.* Prentice-Hall, Englewood Cliffs, N.J., 1991.

[21] H.R. Simpson. "The Mascot Method." *IEE Software Engineering Journal*, 1(3):103–120, May 1986.

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

[23] C. Steigner, R. Joostema, and C. Groove. "PAR-SDL: Software Design and Implementation for Transputer Systems." In R. Grebe and J. Hektor and S. Hilton and M.R. Jane and P.H. Welch, (ed.), *Transputer Applications and Systems*, volume 2. IOS Press, Amsterdam, 1993.

[24] M.R. van Steen. "The Hamlet Application Design Language: Introductory Definition Report." Hamlet Technical Report EUR-CS-93-16, Erasmus University Rotterdam, Department of Computer Science, December 1993.

[25] M.R. van Steen. "The Hamlet Application Design Language: On the Implementation of Synchronous Channels." Technical Report, Department of Computer Science, Erasmus University Rotterdam, 1994. to appear.

[26] M.R. van Steen, T. Vogel, and A. ten Dam. "ADL: A Graphical Design Language for Parallel Real-Time Applications." In R. Grebe and J. Hektor and S. Hilton and M.R. Jane and P.H. Welch, (ed.), *Transputer Applications and Systems*, volume 1. IOS Press, Amsterdam, 1993.

[27] M. van Wijgaarden. "Designing Parallel Applications with Strongly Typed Communication Using ADL." Master's thesis, Department of Computer Science, Erasmus University Rotterdam, August 1994.

[28] P.T. Ward and S.J. Mellor. *Structured Development for Real-Time Systems*, volume I, II & III of *Yourdon Computing Series*. Yourdon Press, Englewood Cliffs, N.J., 1985.

[29] E. Yourdon and L.L. Constatine. *Structured Design: Fundamentals of a Discipline of Computer Program and System Design*. Prentice-Hall, Englewood Cliffs, N.J., 1979.