

Report EUR-CS-93-16
December 1993

The Hamlet
Application Design Language
Introductory Definition Report

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

Abstract

This report provides an introduction to the definition of the Hamlet Application Design Language (ADL). ADL is a graphical-based language and notation supporting the design of parallel real-time applications. Designs expressed in ADL are based on a model of processes that communicate by message-passing. Communication can either be synchronous or asynchronous, and orthogonally, may be subject to blocking, delaying, or nonblocking timing constraints. The language has been devised in such a way that automated (skeletal) code generation can be supported. To this aim, structural aspects are expressed in a notation somewhat similar to data-flow diagrams, whereas behavioral aspects are expressed as state-transition machines following a syntax similar to that of SDL. Exploitation of parallelism is obtained by annotating a design with process replication specifications.

keywords: design language, parallelism, replication, real-time systems.

Contents

1	Introduction	2
1.1	A brief overview of ADL	2
1.2	Rationale for ADL	3
1.3	Overview of this report	3
2	The structure model	5
2.1	Activities and communication media	5
2.1.1	Activities	5
2.1.2	Communication media	5
2.1.3	Some general remarks	9
2.2	Structural organization	9
2.2.1	Designs	9
2.2.2	Gates	10
2.3	Advanced communication issues	12
2.3.1	Timed communication	12
2.3.2	Timed communication and hierarchical organization	15
2.3.3	Multicasting	16
2.4	Summary	20
3	Behavioral model	21
3.1	Introduction	21
3.2	State-transition machines	22
3.2.1	Basic states	22
3.2.2	Transitions	23
3.2.3	Select states	24
3.3	Tailoring the execution mechanism	26
3.4	Summary	26
4	Process replication	28
4.1	The process model	28
4.2	Process replication	29
4.2.1	Replicators	29
4.2.2	Connection statements	33
4.3	Examples	34
4.4	Summary	36

Introduction

This report introduces the graphical *Hamlet Application Design Language*, a technique to assist the development of real-time parallel applications. The Hamlet Application Design Language, or ADL for short, has been developed as part of the Esprit project *Hamlet* [6] which focuses on the development of, and support for construction of industrial real-time embedded applications, executing on transputer-based systems.

ADL is a graphical language based on a model of Communicating Sequential Processes [9] and Data Flow Diagrams [21], aiming at providing the necessary support for the design phase during application development. The language is still under development, and as such, this report should be seen as a preliminary guide to introduce the concepts we anticipate to be part of the language.

1.1 A brief overview of ADL

An ADL design roughly consists of two types of components: **activities** essentially communicating with each other through **communication media**. Constructing an ADL design of a system requires that the developer decomposes the system into functional components, the activities, and explicitly describe by which means these activities exchange information. An activity can be further decomposed, if necessary, leading to an hierarchical organization.

Data between activities can be communicated either in a synchronized fashion by means synchronous channels, or in an asynchronous fashion through message queues. In addition, mere synchronization of activities, i.e., without exchange of data, is achieved through semaphores.

A salient feature of ADL is its support for *timed communication*. Timed communication enables the specification of the instant *when* communication should have taken place. This is particularly important in real-time application development where it is unacceptable that processes would be waiting indefinitely in the case of some failure in the communication system.

An ADL a distinction is made between a **structure model** and a **process model**. The former deals with organizing a system using functional decomposition; the latter deals with modeling the system in terms of *instances* of communicating processes. The distinction between the two is somewhat artificial, except for the fact that processes can be *replicated* into regular geometrical structures. In this way, ADL also supports *data*

parallelism, although emphasis remains on exploiting parallelism through functional decomposition (also referred to as *task parallelism* [4, 11]).

Behavior of activities is modeled by means of state-transition machines (STMs). However, where STMs normally consist of a single notion of a state, and where transitions between states can only occur as the result of an event, ADL uses an approach which more or less combines STMs and flow charts. In this way, we have also integrated data flow and control flow in a single concept, essentially making our approach object-based. The main emphasis in modeling the behavior of an activity lies in the specification of when and how communication takes place in the course of time. STMs in ADL are strongly biased to model exactly such communication details, and less towards modeling behavioral aspects which are not related to communication.

1.2 Rationale for ADL

In order to support the *design phase* of parallel real-time application development there are roughly two extremes which can be followed: one can choose to devise a complete new method with accompanying techniques, or otherwise simply use existing methods. The first approach not only requires a great deal of research, it can also be expected that at best many years will pass before a new method is accepted in an industrial environment. The second approach has so far been followed by many application developers. In particular, methods based on data flow diagrams such as introduced by Yourdon [21] and specifically extensions thereof to support real-time developments (e.g., Ward [17]) are now often used as common development methods in industry. But none of these traditional methods is actually suitable for dealing with parallelism, although their inventors often claim otherwise. As we see it, this is caused by failing to make a distinction between parallelism as a means for, on the one hand, *modeling* a solution, and on the other hand, as a means for *implementing* it.

The problem that needs to be addressed then is the development of a design method and supporting tools which:

- are familiar to developers of industrial real-time applications,
- are based on methods that have proved to be applicable in an industrial context,
- deal parallelism as a means for modeling and implementation.

Based on these requirements, we have chosen to support parallel real-time application development based on data flow diagrams (DFDs). However, where DFDs are generally used in the analysis phase of a development project, we have adapted them in such a way that they are more suitable for technical design. In particular, emphasis has been put on the support for design of different communication structures, replication of processes for exploitation of parallelism, and integration of data and control transformations.

1.3 Overview of this report

The language constructs of ADL are discussed in the next three chapters. Chapter 2 deals with designing applications in terms of functional units, whereas Chapter 4 concentrates on nonfunctional aspects. In particular, parallelism by process replication

is discussed extensively there. Chapter 3 discusses how behavioral aspects can be modeled in ADL.

In this report we have made an attempt to provide accurate, but often still informal definitions of the various constructs supported by ADL. The reason for doing so is that in this stage of the project we felt that definitions should be provided that are precise enough for users and implementors. The report does not discuss any implementation aspects; these are presented in an accompanying report [16].

The structure model

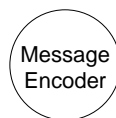
An ADL model consists of a collection of one or more activities that communicate through communication media. Activities are logically organized into so-called designs which reflect the hierarchical relationships between activities based on their functionality. This organization is referred to as the **structure model**. In this chapter we discuss the concepts and notations in ADL that allow a developer to structure an application.

2.1 Activities and communication media

An ADL structure model reflects the structure of an application expressed in terms of a collection of **activities** that communicate by means of **communication media** based on a message-passing paradigm. In this section we take a closer look at both concepts.

2.1.1 Activities

A key concept in ADL is formed by **activities**. Similar to data flow diagrams, an activity is used to model a logical entity capable of transforming incoming data or tokens which can then be passed to other activities. An activity is graphically represented by a large circle, annotated with its name. For example,



represents an activity named MessageEncoder. An important observation is that activities in ADL are treated as self-contained objects. This means that each activity exhibits its own functional behavior, quite similar to the notion of objects in concurrent object-oriented languages [20].

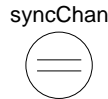
2.1.2 Communication media

In data flow diagrams, the *means* of communication is often left open. Instead, flows between activities merely represent data that is to be communicated, not the communication itself. In ADL we have chosen to follow a different approach by

distinguishing two types of communication media: **data (communication) media** and **token (communication) media**. Data media serve as the primary means for transporting messages between activities. Token media, on the other hand, are used to pass tokens. Tokens differ from data by their absence of contents – as such they can be considered as “null messages” [1]. ADL currently supports two data media: **message queues** and **synchronous channels**. In addition, there is a single token medium available: **semaphores**. These communication media are discussed next.

Synchronous channels

A **synchronous channel** is used to model synchronous message-passing between several activities. In particular, if an activity wants to send a message m via a synchronous channel c , the communication will only take place if there is an activity that can receive m via c . If there is no receiving activity, a sending activity will normally block until communication can take place. A synchronous channel `syncChan` is graphically represented as follows:



A distinction with many other languages that support synchronous message-passing, is that ADL permits multiple senders and receivers to communicate via the same synchronous channel. For example, in languages based on Hoare’s CSP [8, 9], such as OCCAM [12], synchronous channels are strictly modeled as 1:1 relations. Informally, the semantics of synchronous channels are such that when M sending and N receiving activities want to communicate via a synchronous channel, $|M - N|$ (sender,receiver)-pairs are selected nondeterministically and the message is transferred from sender to receiver. This process continues until there are either no more senders or no more receivers. To explain these semantics more accurately, assume that at a specific instant there is a set \mathbf{S} of activities that want to send a message via the synchronous channel c . Furthermore, let \mathbf{R} be a collection of activities that want to receive a message via c . Denote by $\mu(S)$ the message that is to be sent by activity S . Message transfer via synchronous channel c then proceeds according to the following algorithm:

```

let  $K = \min(|\mathbf{R}|, |\mathbf{S}|)$ 
while  $K > 0$  do
  let  $S \in \mathbf{S}, R \in \mathbf{R}$ ;
   $\mathbf{S} \leftarrow \mathbf{S} - \{S\}$ ;
   $\mathbf{R} \leftarrow \mathbf{R} - \{R\}$ ;
  transfer message  $\mu(S)$  from  $S$  to  $R$ ;
   $K \leftarrow K - 1$ ;
endwhile

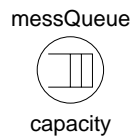
```

When message transfer has taken place, both sending and receiving activity continue. It is thus seen that a total of $\min(|\mathbf{R}|, |\mathbf{S}|)$ (sender,receiver)-pairs are nondeterministically constructed and the message belonging to the sender is transferred to the receiver. Note that this means that if $|\mathbf{S}| > |\mathbf{R}|$ that a total of $|\mathbf{S}| - |\mathbf{R}|$ nondeterministically

selected senders will remain blocked, and likewise, if $|\mathbf{S}| < |\mathbf{R}|$, that a total of $|\mathbf{R}| - |\mathbf{S}|$ nondeterministically selected receivers will remain waiting for a message via c .

Message queues

ADL also provides support for modeling asynchronous communication by means of **message queues**. As their name suggests, message queues provide a buffering capability for messages that preserves the order in which messages are sent. In other words, an activity will receive messages in the same order as they were sent by a sending activity. Message queues may either have an infinite or finite capacity. The graphical representation of a message queue named `messQueue` with capacity `capacity` is as follows:



As with synchronous channels, message queues can be used to model an $M : N$ relationship between activities. When several activities attempt to receive a message from a message queue, one of the receivers is selected nondeterministically and the message at the front of the queue is removed and passed to that activity. This process continues until there are either no more receiving activities, or the queue has become empty. If there are no messages pending in the queue, a receiving activity will be delayed. Similarly, if a number of activities want to send a message via a message queue, one of the senders is selected nondeterministically and its message is appended to the queue. This process continues until there are no more sending activities, or because the queue becomes full. When no messages can be appended to the queue, a sending activity will be delayed.

We can express these semantics more accurately as follows. Again, assume that at a specific instant there is a set \mathbf{S} of activities that want to send a message via the message queue q . Furthermore, let \mathbf{R} be a collection of activities that want to receive a message via q . Denote by $len(q)$ the length of q , by $cap(q)$ its capacity, and by $first(q)$ the message at the front of the queue, if any. Communication via message queue q then proceeds according to the following algorithm:

```

let  $K = \min(|\mathbf{S}|, |\mathbf{R}| - \text{len}(q) + \text{cap}(q));$ 
let  $C_q = \text{cap}(q);$ 
let  $\text{cap}(q) = \infty;$ 
while  $K > 0$  do
  let  $S \in \mathbf{S};$ 
   $\mathbf{S} \leftarrow \mathbf{S} - S;$ 
   $q \leftarrow q + \mu(S);$ 
   $K \leftarrow K - 1;$ 
endwhile;

let  $K = \min(|\mathbf{R}|, \text{len}(q));$ 
while  $K > 0$  do
  let  $R \in \mathbf{R};$ 
   $\mathbf{R} \leftarrow \mathbf{R} - R;$ 
  transfer  $\text{first}(q)$  to  $R$ 
   $q \leftarrow q - \text{first}(q);$ 
   $K \leftarrow K - 1;$ 
endwhile;
let  $\text{cap}(q) = C_q;$ 

```

Only when a message $\mu(S)$ has been appended to the queue, will the sending activity S continue. Likewise, only after transferring a message from the queue to a receiving activity, will the latter continue. A few remarks concerning these semantics are in order. In the first place, note that the capacity of the queue is temporarily set to infinity. This has merely been done to express our algorithm, by letting a maximum number of senders start with appending their message to the queue. Secondly, note that at any instant, a maximum of $|\mathbf{R}| - \text{len}(q) + \text{cap}(q)$ messages from sending activities can be processed. If \mathbf{S} exceeds this number then $|\mathbf{S}| - (|\mathbf{R}| - \text{len}(q) + \text{cap}(q))$ nondeterministically selected senders will remain blocked. Likewise, if there are not enough messages in the queue, a total of $|\mathbf{R}| - \text{len}(q)$ nondeterministically selected receiving activities will block until new messages are appended to the queue. Note further that the semantics of message queues are defined in such a way that also queues with a capacity $\text{cap}(q) = 0$ are well-defined. In that case, the algorithm above is seen to reduce to that of synchronous channels.

Semaphores

Where data media are used for actual *communication* of data, token media are primarily used to model *synchronization* of activities. Only a single type of token media is currently supported in ADL: **semaphores**. The semantics of semaphores are conventionally defined as ([7]):

wait(s)	while $s \leq 0$ do od $s \leftarrow s - 1;$ return
signal(s)	$s \leftarrow s + 1;$ return

In ADL semaphores semantically correspond with message queues with infinite capacity with the exception that tokens instead of data are queued. In particular, the *length* of the message queue now corresponds with the traditional *value* of a semaphore. This means

that sending a token to a semaphore is equivalent to performing a signal-operation, whereas retrieving a token from a semaphore is the same as a wait-operation. As usual, a semaphore in ADL may have an initial value, or, likewise, an initial number of tokens. This leads to the following graphical representation for a semaphore named `sema` with initial value `initValue`:



Semaphores are generally used to protect critical regions in shared-variable environments. Shared variables are not supported in ADL, implying that semaphores will generally be used for mutual exclusive access to other resources.

2.1.3 Some general remarks

So far, we have only mentioned in a rather abstract manner that data and tokens can be communicated between activities. We have said nothing about the *type* of the communication. For example, at the level of programming languages it is generally required that messages are typed. The minimalist approach is followed by communication libraries that demand that at least the size of the message is provided. In the current version of ADL, there is no means to specify data or token types. Instead, we merely assume that data is represented by a sequence of bytes with a specified length, and that tokens are nothing more than byte sequences of length 0. This implies that a developer using ADL is fully responsible for conversion of typed data and token objects to byte sequences and *vice versa*. The lack of strong typing facilities in ADL is to be considered as an immature feature of the language that will be corrected in the future.

2.2 Structural organization

So far, we have discussed the basic components of an ADL structure model: activities and communication media. In this section we present a means for organizing communicating activities as modular designs.

2.2.1 Designs

Activities and communication media are organized in ADL into so-called **designs**, reflecting their logical grouping based on functionality. Each design consists of at least one activity, and, similarly, each activity is contained in precisely one design. Graphically representing the fact that an activity communicates via a specific communication medium is done by means of arcs. In particular, if an activity sender sends information to a communication medium `cm`, that is subsequently to be read by an activity reader, this is graphically represented as shown in Figure 2.1.

The symbols “●” and “■” represent so-called gates. They are discussed below. Because activities can *only* communicate by means of communication media, it is clear that it would syntactically be an error to draw an arc between two activities. In a

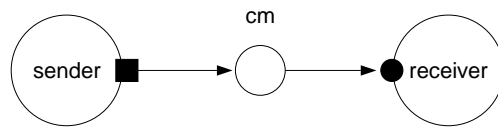


Figure 2.1: The basic graphical representation of an ADL design.

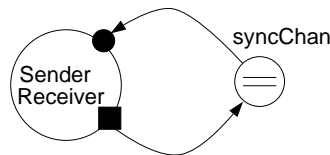


Figure 2.2: A syntactically correct ADL design in which a communication medium has at least one identifiable sender, and one identifiable receiver.

similar fashion, we consider a design to be erroneous when, for a given communication medium, there are no identifiable (potentially) sending or receiving activities. Syntactically, this implies that there should always be at least one arc *from* a communication medium *to* an activity, and *vice versa*. We do not forbid that the sending and receiving activity are the same. For example, the design shown in Figure 2.2 is syntactically perfectly in order.

Activities can be decomposed into smaller units. In particular, these so-called **complex activities** are activities that have been decomposed into a design, thus allowing for a hierarchical organization. Activities which have not been decomposed are called **simple activities**. Given this, it should now be clear that Figure 2.2 may even represent a semantically sensible design. Instead that an apparent deadlock occurs, the activity SenderReceiver may be decomposed into a number of parallel acting subactivities.

This organization of activities into designs leads to a **structure model**, which, by definition, is a tree as shown in Figure 2.3. Only the activities contained in the leaves of this tree are simple ones; the other activities are complex. An alternative approach would be to construct the hierarchy as a directed acyclic graph (DAG). (In fact, this is done *during* the development itself.) However, the structure model reflects the organization of an application at a certain point of completion. Then, each activity can be only have been refined to a *unique* subdesign, if any, which obviously leads to a tree structure.

There is no separate syntactical representation for a design, other than the grouping of activities and communication media according to their syntax. On the other hand, we do assume that an implementation of ADL supports the notion of a design.

2.2.2 Gates

In data flow diagrams, the interface of an activity is not made explicit – a situation which we feel is inappropriate when technical design issues are to be dealt with. In particular, if a technique is to support the design of *parallel* applications, support for communication structure design becomes apparent. In our view, an essential

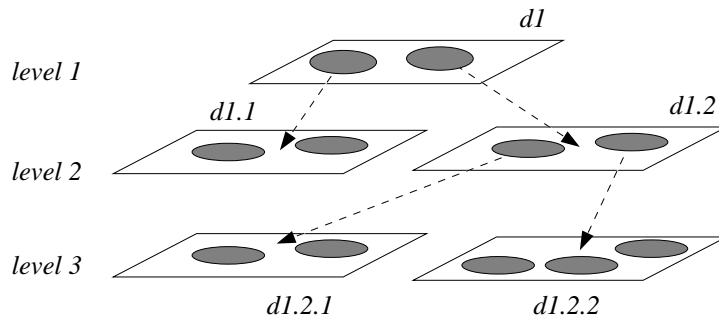


Figure 2.3: A structure model represented as a tree of designs.

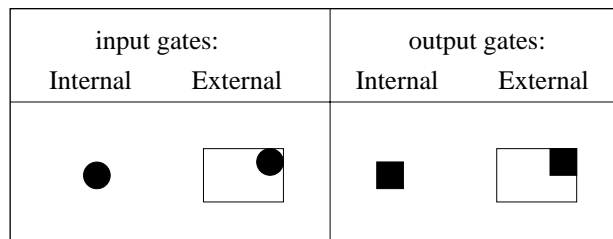


Figure 2.4: Notations for internal and external gates.

aspect of designing communication structures for parallel applications is *localization*. This means that a developer can concentrate on a relatively small part of the communication independent of other parts that constitute the overall communication structure. In ADL, we have made the interface of an activity explicit through so-called **(input and output) gates**. Gates in ADL provide a means to specify *where* communication with an activity takes place, i.e. they serve as explicit *access points* to communicate data or tokens with an activity. This is similar to the approach followed in Mascot [3, 14]. A further distinction is made between gates that are either **internal** or **external** to a design, leading to the graphical notations as shown in Figure 2.4.

Gates that are internal to a design (also called internal gates), represent *where* activities and communication media are connected. Each internal gate is always connected to exactly one communication medium by means of an arc. Gates that are external to a design (called external gates) represent the interface of a single design.

An external gate always belongs to precisely one design. At the same time, it should also be related to at least one internal gate of an activity contained in the same design as the external gate. This relationship between an external gate and one or more internal gates is referred to as a **horizontal gate association**. It reflects the relationship between external and internal gates *in a single design*. Likewise, each internal gate of a complex activity, i.e., an activity which has been decomposed into a number of designs, should be associated by means of a so-called **vertical gate association** with precisely one external gate of its child design. A vertical gate association thus reflects the relationship between an internal and an external gate in *different designs*. These notions of horizontal and vertical gate associations are illustrated in Figure 2.5.

To explain and illustrate the semantics of these associations, suppose that a complex

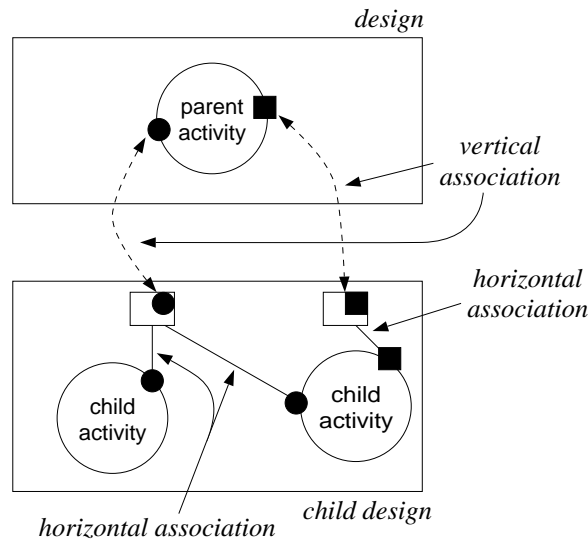


Figure 2.5: The hierarchical organization of activities as designs.

activity A has an internal gate g_{in} which is connected to a communication medium c . Furthermore, let A be decomposed into a design \tilde{D} consisting of subactivities $\tilde{A}_1, \dots, \tilde{A}_n$. Firstly, the internal gate g_{in} attached to A should be vertically associated with exactly one external gate \tilde{g}_{ext} of \tilde{D} . This external gate should be horizontally associated with at least one internal gate of \tilde{D} . Gates that are internal to \tilde{D} can, in turn, be horizontally associated with either zero or one external gate of \tilde{D} . Assume that gate \tilde{g}_{in} is an internal gate of \tilde{D} , and is horizontally associated with the external gate \tilde{g}_{ext} . Then, semantically, this association implies that all communication through gate \tilde{g}_{in} also passes through gate \tilde{g}_{ext} of the parent activity A . Consequently, the internal gate \tilde{g}_{in} is thus indirectly connected to the communication medium c . This is also illustrated in Figure 2.6.

2.3 Advanced communication issues

The concepts introduced so far are based on what could be referred to as traditional communication features. In this section we introduce two more advanced features: timed communication and multicasting operators.

2.3.1 Timed communication

Besides their use as a means to specify *where* communication is to take place, gates are also used to show *when* communication should take place from the moment an activity is willing to communicate. To this aim, gates are subject to so-called **timed communication**. Three types of timed communication are available in ADL:

- In the case of **blocked communication** at a gate g , an activity which is waiting for communication via g , will not proceed until data or token transfer through g has actually taken place.

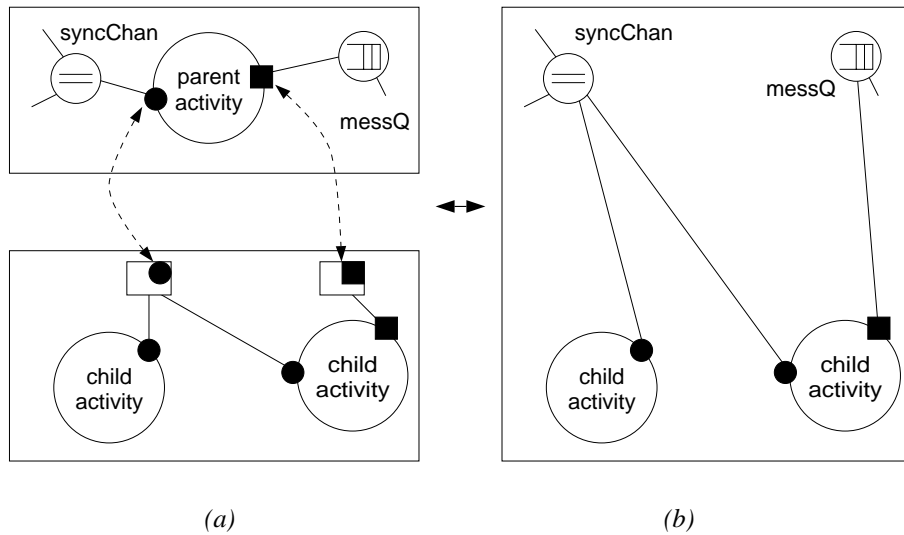


Figure 2.6: The relationship between internal and external gates with respect to communication media. Figure (b) is semantically equivalent to (a).

- In the case of **non-blocked communication** at gate g , an activity will never wait until communication through g takes place, unless it can take place immediately.
- Finally, **delayed communication** indicates that an activity is willing to wait for communication until a specified amount of time has elapsed.

The graphical notations for the various gates in ADL is shown in Figure 2.7. Blocked and non-blocked communication are in fact special cases of delayed communication. If Δt denotes the specified time an activity is willing to wait before communication can take place, then clearly the case $\Delta t = 0$ corresponds to non-blocked communication whereas the case $\Delta t = \infty$ is the same as blocked communication. For practical reasons, we have chosen to incorporate all three communication types. Also 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.

Anticipating our discussion on timed communication and hierarchical organization, we note here that the timing shown at a gate is generally *variable*. Depending on the refinement of an activity into constituents (and as we shall see later, also into state-transition machines), the timing may vary with the actual communication that an activity wants to perform at a certain instant in time. We shall return to this issue further below and in the following chapter.

Timed communication affects the semantics of communication via data or token media in a number of ways. In the first place, note that the semantics for synchronous channels, message queues, and semaphores have been formulated in terms of blocked communication. In other words, whenever communication could not take place (for whatever reason), a communicating party will be delayed. Using non-blocked com-

	input gates:		output gates:	
	Internal	External	Internal	External
blocked	●	◻●	■	◻■
delayed	⊕	◻⊕	⊕	◻⊕
nonblocked	○	◻○	◻	◻◻

Figure 2.7: Notations for ADL gates.

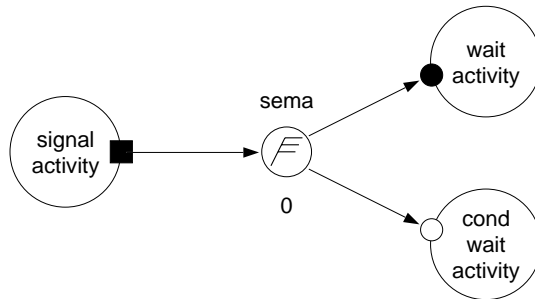


Figure 2.8: An example of a design using conditional semaphores.

munication, however, we can model other situations as well.

Example 2.3.1. Conditional semaphores are like general semaphores but support a so-called conditional wait-operation [2] having the following semantics:

<i>condwait(s)</i>	if $s \leq 0$ return false else $s \leftarrow s - 1$; return true
--------------------	--

In other words, rather than delaying a caller until the semaphore's value can be decremented, control is passed back with notification of the failure. A conditional wait operation can easily be modeled in ADL using timed communication by means of a non-blocked input gate connected to a semaphore. Such a design is for example shown in Figure 2.8.

□

Note, as a matter of fact, that using delayed or non-blocked communication on an *output gate* connected to a semaphore (corresponding to a signal-operation) has the same effect as blocked communication because communication will always succeed. A similar case holds for output gates connected to message queues having an infinite capacity.

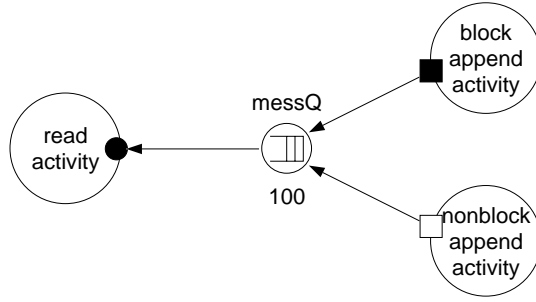


Figure 2.9: An example of a design using blocked and non-blocked appending of messages to a queue with finite capacity.

Example 2.3.2. Message queues are generally implemented with a finite capacity. A question that needs to be addressed then is what the semantics should be in case the capacity is exceeded. Depending on the underlying runtime system, senders will either be blocked, or notified that communication failed. In ADL, it is the designer who specifies the semantics, rather than this being implementation dependent. Using non-blocked communication on an output gate connected to a message queue, the developer specifies that failure of message transfer should be notified. Otherwise, in the case of blocked communication, the developer specifies that the sending activity should block until message transfer can take place. Syntactically, these situations are represented as shown in Figure 2.9.

□

2.3.2 Timed communication and hierarchical organization

Now, what can we say about the timing at an internal gate g_{in} of a complex activity A ? According to what has been said above, gate g_{in} should be vertically associated with an external gate \tilde{g}_{ext} of its child design (which we refer to as \tilde{D}). Gate \tilde{g}_{ext} , in turn, should be horizontally associated with one or more gates that are internal to \tilde{D} . Denote these internal gates as $\tilde{g}_{in}^1, \dots, \tilde{g}_{in}^m$, and let Δt_i denote the specified timing at gate \tilde{g}_{in}^i , i.e., $\Delta t_i \in [0, \infty) \cup \{\infty\}$. Now, first of all, note that the specified timing behavior at an internal gate expresses how long an activity is willing to wait before communication should take place. When a number of gates are horizontally associated with an external gate then clearly the *ensemble of activities* connected to that gate is willing to wait at least Δt_{min} and at most Δt_{max} time units where

$$\Delta t_{min} = \min_{i=1..m} \{\Delta t_i\} \text{ and } \Delta t_{max} = \max_{i=1..m} \{\Delta t_i\}.$$

Consequently, the time that the parent activity A is willing to wait is equal to the maximum waiting time of its children, i.e. Δt_{max} . This puts an upperbound on the timing shown at gate g_{in} . Likewise, Δt_{min} is a lowerbound for the timing at this gate. To reflect the situation that timing may vary, we introduce a separate notation for those cases that $\Delta t_{min} < \Delta t_{max}$ as shown in Figure 2.10.

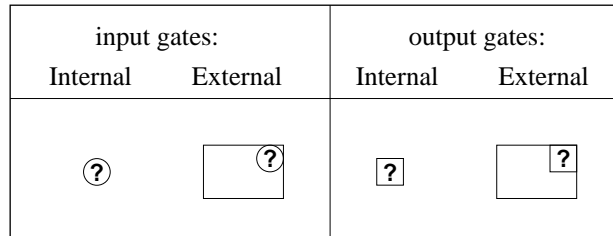


Figure 2.10: The graphical notations for gates with variable timing.

2.3.3 Multicasting

The communication media in ADL can all be used to model $M : N$ relationships between communicating activities. However, sending a message always implies that precisely *one* activity will receive that message (assuming that there are activities actually prepared to communicate with the sender). An alternative communication scheme is **multicasting** which is also supported in ADL. Informally, multicasting a message yields that a message is sent to a preselected *group* of activities.

In ADL, any communication media (i.e. synchronous channel, message queue, or semaphore) can be used as the basis for multicasting. In particular, multicasting a message m via a communication medium c yields that m is replicated and sent to all activities that have been modeled as receivers. A further distinction is made between **post-medium** and **pre-medium** multicasting.

- In the case of **post-medium** multicasting, a message can only be transferred to the receiving activities if this can be done simultaneously to all of them.
- In the case of **pre-medium** multicasting, the message is sent to each receiving activity individually. This implies that the message may be received at different times by the receivers.

In essence, post-medium multicasting yields that a message is replicated *after* the semantics of the communication medium have been taken into account. In the case of pre-medium multicasting, messages are *first* replicated after which the semantics of the communication are followed per receiving activity. Multicasting in practice generally follows the semantics of our notion of pre-medium multicasting. Post-medium multicasting resembles atomic multicasting [15]. Figure 2.11 shows the graphical notation for distinguishing respectively post-medium and pre-medium multicasting. We shall consider both forms of multicasting for each available communication medium.

Synchronous channel

Post-medium multicasting via a synchronous channel implies that communication can only take place if all receivers are willing to accept a message from a sender. In particular, *all* communicating parties, i.e. sender and receivers, synchronize on communication. To illustrate, consider the two situations shown in Figure 2.12.

If we assume in Figure 2.12(a) that sender and receiver1 are ready for communication, they will both have to wait until receiver2 is also ready to accept the message. At



Figure 2.11: The notations for post-medium multicasting (a) and pre-medium multicasting (b).

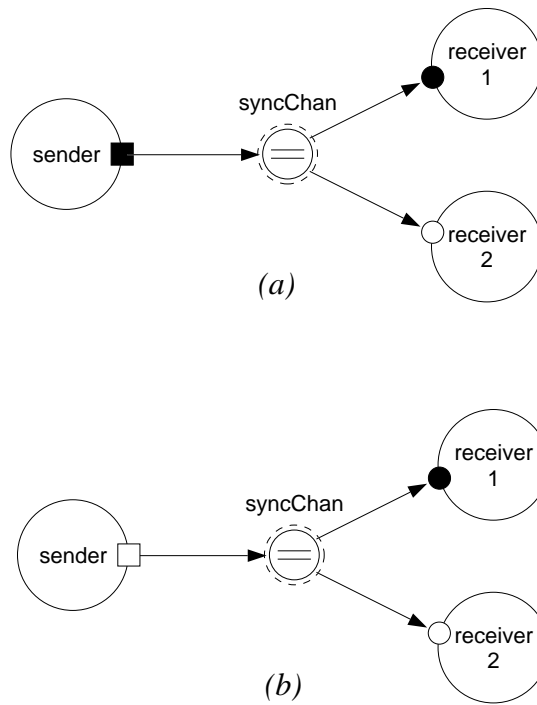


Figure 2.12: An example of a postmedium multicasting via a synchronous channel which is in order (a), and one which is rather hazardous (b).

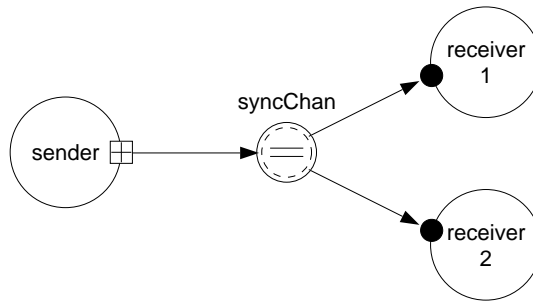


Figure 2.13: An example of a premedium multicasting via a synchronous channel using delayed communication.

that instant, message transfer takes place. Clearly, then, the situation in Figure 2.12(b) reflects a rather poor design as it can be expected that once receiver1 is ready to accept a message via syncChan, the possibility that this will ever happen is indeed very small. Communication can, namely, take only place at the very instant that both sender and receiver are willing to communicate.

In the case of pre-medium multicasting, the receiving entities will synchronize with the sender, but not with each other. In particular, this means that a receiver may continue the instant it has received the message. Pre-medium multicasting does lead to the situation that *partial* failure of communication may occur. This differs from post-medium multicasting in which communication succeeds entirely, or otherwise fails completely. To illustrate, consider the situation shown in Figure 2.13.

Now suppose that at the instant sender wants to send a message to both receiver1 and receiver2, that only receiver1 is waiting for such a message. Denote by Δt the specified timing at the output gate of sender. By definition of pre-medium multicasting, the message is replicated and sent to receiver1, regardless if receiver2 is willing to accept the message as well. However, if receiver2 is not prepared to receive any message from sender before Δt time units have elapsed, sender will withdraw from further communication via syncChan. Consequently, the multicast will only have succeeded partially. As we shall see in the next chapter, this partial failure cannot be directly reported to sender, who instead, will receive an event indicating a complete failure.

Message queue

When considering message queues, post-medium multicasting yields that a message is removed from the front of the queue the instant that all receiving activities are prepared for communication. The message is then transferred to all receivers simultaneously. A few observations are important to note. In the first place, just as with post-medium multicasting via synchronous channels, all receiving activities synchronize. Secondly, observe that if, respectively, two messages m_1 and m_2 are sent in that order, that no receiver will ever read message m_2 before *all* receivers have read message m_1 . In other words, message receipt within the group of receiving activities is never skewed.

Pre-medium multicasting via message queues is commonly applied in practice. In this case, one can imagine that instead of a single message queue each receiving

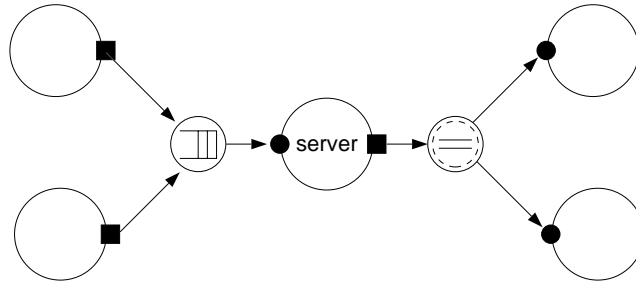


Figure 2.14: Combining post- and pre-medium multicasting.

activity as its own message queue. Multicasting a message yields that the message is replicated, and appended to each individual queue. Consequently, receiving activities do not synchronize as was the case with post-medium multicasting. Also, although the order of message transfer is still preserved between a sender and a receiver, it is now possible that message receipt within the group of receiving activities is completely skewed. In other words, if messages m_1 and m_2 are sent after each other, it is possible that a receiver R_1 will have already read message m_2 before receiver R_2 has read message m_1 .

Note that by nature of message queues, pre-medium and post-medium multicasting are two extremes. An alternative form would be to allow multicasting in a such a way that no skewing of message receipt between receivers occurs (a feature of post-medium multicasting), but that, on the other hand, receivers need not synchronize (a feature of pre-medium multicasting). This form of multicasting is not supported in ADL, but instead should be modeled as a combination of normal communication via a message queue, and pre-medium multicasting via a synchronous channel. This is shown in Figure 2.14. In this case, the activity `server` simply removes each message from the front of the queue and transfers it to the synchronous channel where it is multicasted to the receivers. Clearly, the receivers no longer synchronize on message receipt, but never is a message read before all other receivers have read its predecessor.

Semaphore

By viewing semaphores as token-bearing message queues, the semantics of post- and pre-medium multicasting via a semaphore should now be clear. In the case of post-medium multicasting, a number of receiving activities will synchronize on their individual *wait*-operation. In other words, signaling a post-medium multicasted semaphore is to be interpreted that a collection of receivers should synchronize on a single event. On the other hand, a pre-medium multicasted semaphore is to be interpreted as a medium that allows a sender to indicate that each activity from a group of receivers is to react on a repeated event, as often as the event as occurred. No synchronization between the receiving activities will take place.

2.4 Summary

A structure model is a hierarchical organization of activities that communicate with each other by explicitly passing data or tokens through communication media. The interface of an activity, or a group of activities (referred to as a design) is modeled explicitly in the form of respectively internal and external gates.

There is no notion of shared data, and the semantics of communication between activities strongly follows that of general message-passing programming models. However, an explicit distinction is made between *how* and *when* communication or synchronization takes place. The former is expressed through synchronous channels, message queues, and semaphores. When communication should have started is modeled by means timed communication at gates, which can either be blocked, non-blocked, or delayed.

In addition, multicasting is supported for each communication medium. Two forms are available. Post-medium multicasting synchronizes the collection of receiving activities in the sense that a message is only transferred to the receivers if all of them are prepared for receipt. Pre-medium multicasting replicates a message, after which communication follows the semantics of the medium via which communication is to take place.

Behavioral model

Activities form the units of behavior in ADL and obviously there should be a means for supporting the description of behavioral aspects. In ADL this is done by means of state-transition machines (STMs) which collectively form the *behavioral model* of an activity. They are the subject of this chapter.

3.1 Introduction

In most methods based on data flow diagrams, there is a strict distinction between data activities and control activities. Data activities are used for modeling data transformations, whereas control activities describe the system's flow of control. In order to describe data transformations, pseudo-code, or sometimes even a high-level procedural language is used. Control flow is described by means of state-transition machines. When separating control and data transformations, the process of integrating them (which is required for an implementation) may turn out to be less straightforward than one would initially expect.

An alternative approach that has been adopted by object-oriented development techniques, is to encapsulate control and data functionality into a single entity. And indeed, practice indicates that this approach is quite elegant when it comes to designing applications. In ADL, there is also no explicit distinction between control and data functionality: both are encapsulated into activities. In this sense, ADL can be considered as an *object-based* design language (see also [10, 19]). Continuing our line of thought, it seems natural to devise a technique for designing the behavior of an application on a *per activity* basis. In other words, given a structural model, a developer should be able to design the behavior of each individual activity. To this aim, ADL supports a notion of **state-transition machines**, of which there will be precisely one for each *simple* activity. The behavior of complex activities is assumed to be captured by the composite behavior of its sub-activities.

The ADL state-transition machines have been partly inspired by those used in Jacobson's approach [10], which in turn are based on similar concept defined in CCITT's Specification and Description Language [5, 13].

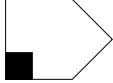
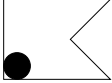
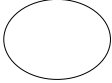
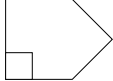
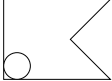

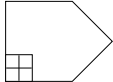
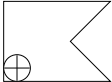
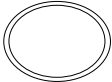
Communication states			Processing states	
	output states	input states		
blocked			initial	
non-blocked			computation	
delayed			final	

Figure 3.1: The notations for states in state-transition machines in ADL.

3.2 State-transition machines

In ADL each simple activity has precisely one associated state-transition machine (STM). The behavior of complex activities is always assumed to be specified by the collective behavior of their constituents. An STM consists of a finite number of states and an execution mechanism by which state-transitions are made. At each time instant, the activity to which the STM is associated will always reside in exactly one of the states specified by the STM. This state is also denoted as the **current state**. In the following sections we shall mainly concentrate on the explanation of the execution mechanism as this describes the (operational) semantics of our state-transition machines, and thus the behavior of activities.

3.2.1 Basic states

The set of states in an STM is essentially partitioned into two subsets: the set of so-called **processing states**, and the set of **communication states**. Communication states come in two kinds: input states and output states, and are further divided according to the three types of timed communication. The syntactical notations for states is shown in Figure 3.1.

Processing states

Processing states are used to model those behavioral aspects which are not related to communication. Three different types of processing states are distinguished: a single **initial state**, **computational states**, and **final states**.

The initial state indicates where an activity starts for the first time. It is typically used to model the initialization section of an activity's behavior. Processing states serve to model a series of pure sequential statements in which *no* communication with other activities takes place. Final states, of which there may be several in a single

STM, are typically used for modeling a finalization section of an activity's behavior. Contrary to an initial state, an activity need not have a final state as part of its STM.

Communication states

A communication state reflects the situation that an activity is currently involved in communicating data or tokens with some other activity. Two types of communication states are distinguished: input states representing the situation that data or tokens are received, and output states representing the transfer of data or tokens to other activities. Each input or output state is always associated with exactly one internal input or output gate, respectively, of the simple activity associated with the STM. The converse need not be true: an internal gate can have several associated communication states, reflecting the situation that communication through that gate may occur at different moments in time. The communication state determines the *actual* timing at the gate it is associated with.

3.2.2 Transitions

A transition between states is formally provided by means of a **transition specification** and consists of precisely one source state and one target state. The transition is always made from the source to the target state. With respect to the structural aspects of a state-transition machine, we demand that each state is reachable from the initial state. This can be expressed more accurately as follows. Denote by \mathbf{S} the set of states, and by $\mathbf{T} \subseteq \mathbf{S} \times \mathbf{S}$ the collection of transition specifications. For any state $s \in \mathbf{S}$ define the set $R(s)$ of reachable states from s recursively as follows:

1. $s \in R(s)$
2. $u \in R(s), (u, v) \in \mathbf{T} \Rightarrow v \in R(s)$

We then demand that if s_{init} is the initial state, that $R(s_{\text{init}}) = \mathbf{S}$. We now take a closer look at transitions with respect to processing and communication states.

Processing states

In the case of processing states, the following rules with respect to transitions apply. The initial state can never be specified as the target state in a transition specification. In addition, precisely one transition specification should include the initial state as source state. Likewise, a final state can never be the source state of a transition specification, but, on the other hand, may be specified as target state of several transition specifications.

Processing states may occur as the source state of at least one, but possibly also more transition specifications. Selection of a transition originating in a processing state is nondeterministic from the point of view of the execution mechanism underlying state-transition machines. However, as we shall discuss further below, the execution mechanism itself can be refined in such a way that nondeterministic selections can be avoided.

Transition specifications originating in a processing state are graphically represented by a solid arc from the processing state to a target state.

Communication states

Transitions originating in communication states are related to so-called **events**. An event is always raised by a gate. Two types of events are distinguished in ADL:

- a **transfer event** is raised by an input or output gate whenever a data or token passes through it,
- a **timeout event** is raised by either an input gate or an output gate whenever the timing constraints cannot be met.

Whenever an activity 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 communication states there are two types of transition specifications. A **transfer transition specification** specifies the state to which a transition is made whenever a transfer event is raised. It is graphically shown as solid arc from the communication state (the source state) to the next state (the target state). Likewise, a **timeout transition specification** specifies the state to which a transition will be made on account of a timeout event. It is graphically notated as dashed arc from the source to the target state. For each communication state precisely one transfer transition specification must be provided. In addition, if a communication state is subject to either delayed or non-blocked communication, a timeout transition specification should also be given.

3.2.3 Select states

ADL supports two forms of composite states: single select select states and total select states. They are collectively referred to as **select states**. A select state always consists of two or more communication states. Input and output states may be jointly used to compose a select state. No two communication states within a select state can be associated with the same gate. The collection of gates associated with a select state is simply the union of gates to which its constituents are associated.

None of the communication states in a select state can ever occur as the target state of a transition specification. The select state itself, however should be reachable from the initial state. The definition of reachability is extended in such a way that all constituents of a select state are reachable from the initial state if the select state itself is reachable from the initial state. Specifying select states and its constituents as source of transition specifications is dependent on the type of select states. This is discussed below.

Single select state

A collection of communication states can be aggregated into a so-called **single select state**. A single select state is used to model those situations in which one of several possible communications may occur. The graphical notation for a single select statement is shown in Figure 3.2(a).

The constituents of a single select state adhere to the rules of normal communication states with respect to their specification as source states of transition specifications. This means that for each communication state precisely one transfer transition specification should be provided, and in addition, for each communication state that is subject to

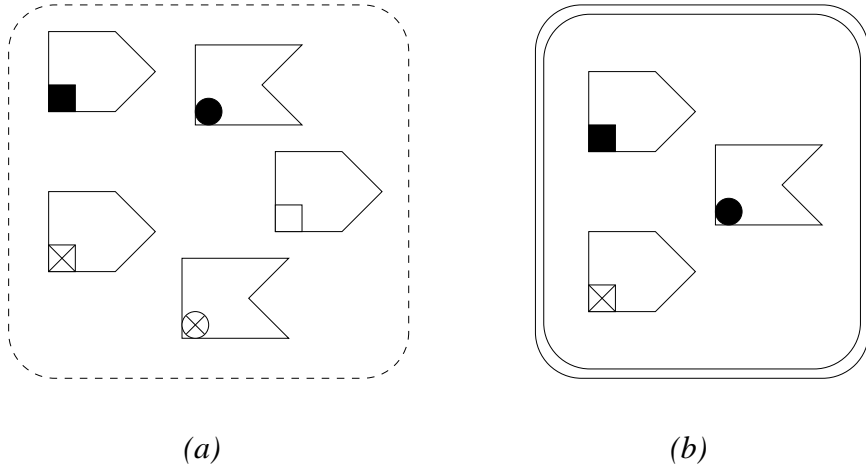


Figure 3.2: The notation for a single select state (a) and a total select state (b) in ADL.

delayed or non-blocked communication, there should also be one timeout transition specification. The first event that is raised after entering the select state determines the selection of the transition to the next state.

Total select state

Besides the notion of a single select state, ADL state-transition machines also support so-called **total select states**. Again, total select states consist of a collection of communication states. Contrary to a single select state, however, total select states are used to model those situations in which several communications need to be performed (or perhaps timed out) before continuation. The order in which communication takes place is considered irrelevant. Figure 3.2(b) shows the graphical notation for a total select state.

The constituents of a total select state can never be specified as the source of a transition specification. Instead, there should be precisely one transfer transition specification in which the select state is specified as the source. In addition, if any of the communication states in the select state is subject to delayed or non-blocked communication, there should also be precisely one timeout transition specification originating in the select state.

The transfer transition is selected as soon as all communication has taken place, where each constituent state has been visited precisely once. The timeout transition is selected on the first timeout event raised by any of the gates associated with the select state. It is important to note that such an event indicates that communication did not fully succeed and that it can never fully succeed. However, it cannot be determined if any communication took place while residing in the select state before the timeout event was raised.

These semantics can be expressed more accurately as follows. Denote by $\mathbf{S} = \{S_1, \dots, S_n\}$ the collection of communication states in the total select state, and let g_i denote the gate to which state S_i is associated. A transfer event raised by gate g_i is denoted as τ_i , whereas a timeout event is denoted as ϵ_i . The events that can be handled

at a specific time by a collection of states \mathbf{R} is denoted as $E(\mathbf{R})$. Finally, denote by $\mu(S_i)$ the message that is to be transferred via gate g_i on account of the semantics of state S_i . The execution mechanism in a total select state then proceeds according to the following operational semantics.

```

let  $\mathbf{R} = \mathbf{S}$ ;
while  $\mathbf{R} \neq \emptyset$  do
  waituntil
     $\exists \tau_i \in E(\mathbf{R}) \Rightarrow \text{transfer } \mu(S_i); \mathbf{R} \leftarrow \mathbf{R} - S_i$ ;
  or
     $\exists \epsilon_j \in E(\mathbf{R}) \Rightarrow \text{select timeout transition}$ ;
  endwait
endwhile
select transfer transition;
```

As soon as a transition is selected the execution mechanism immediately continues according to the semantics of the selected next state.

3.3 Tailoring the execution mechanism

As we have mentioned, it is possible to further refine the execution mechanism underlying an STM. To this aim, we anticipate that processing states are further refined by attaching a series of sequentially executable statements from a **guest** language. At present, ADL is targeted to support “C” as its only guest language. To this aim, we demand that an implementation of ADL supports either the identification of each transition originating in a processing state, or otherwise the identification of a next selectable state. Specifying a transition is then to be supported by a simple library routine, such as e.g., `transit(transition)` or `select(next_state)` of which the execution yields an immediate state-transition.

3.4 Summary

Where the structural model of an application reflects the static aspects of an application design, the behavioral model describes the behavior of activities. The behavioral model is constructed through state-transition machines (STMs) for which there is one per simple activity. An STM is constructed as a collection of states for which transition specifications are provided. Two types of states are essentially distinguished: processing and communication states.

A processing states represents a part of the behavior of an activity which is characterized by absence of communication. A communication state models the sending or receipt of messages, and as such is always associated with exactly one internal gate of the simple activity associated with the STM. State transitions originating from a processing state are chosen nondeterministically. Those originating in a communication state are selected on the basis of the occurrence of either a transfer or timeout event which are raised by the gate to which the communication state is associated.

Communication states can be grouped into either a single select state or a total select state. A single select state reflects the situation that alternative communications

are possible at a specific time. The first communication that takes place, or fails due a timeout, determines the behavior of the single select state. Total select states are used to model those situations in which a number of communications should take place, but in which the order of communication is irrelevant. The first timeout event that is raised by a gate to which one of the constituent states in a total select state is associated, causes a transition from the select state.

Process replication

Activities reflect *logical* communicating entities. As such they form the means to model the *functionality* of a system. ADL has been developed to capture part of the intricacies related to parallel application development. To that aim, it also supports the notion of **processes**. A process is quite similar to an activity, but distinguishes itself from the latter by the fact that it is explicitly used for modeling *sequential executions*. As such, processes deal with modeling nonfunctional aspects. In this chapter we discuss how processes can be used to exploit parallelism in an application.

4.1 The process model

As the unit of sequential behavior in ADL is formed by a simple activity, such an activity also forms the basic means to construct processes. By default, each simple activity is considered to be a process. A process, then, is an *active* entity that can be executed on a processor and as such will behave according to the specifications given by the state-transition machine of the associated simple activity. In order to hierarchically organize a collection of processes and communication media, the structure model of an application is first “flattened” by considering only the simple activities and all communication media. The hierarchical organization of the structure model is discarded by removing the complex activities, and recursively replacing vertical and horizontal associations by direct connections between communication media and activities, as explained in Section 2.2.2 (see also Figure 2.6). This flattened structure model results in a directed bipartite graph where the nodes are now formed by the set of processes and communication media, and the links as the connections between them. The model itself is denoted as the **root task**.

The root task can be decomposed by repeatedly grouping a number of processes and communication into subtasks. If \mathbf{P} denotes an arbitrary set of processes, and \mathbf{C} a set of communication media such that if process P is connected to a communication medium $c \in \mathbf{C}$, then also $P \in \mathbf{P}$, then a subtask T can be constructed by the graph induced by the set $\mathbf{P} \cup \mathbf{C}$, i.e., it consists of all elements from $\mathbf{P} \cup \mathbf{C}$, including the connections between them. Such a subtask is denoted as a **complex process** if $|\mathbf{P}| > 1$. This grouping of processes and communication media is to be repeated until each process is associated to precisely one subtask, and each communication medium is associated to *at most* one subtask. The result is that the root task will have been

decomposed into a number (complex) processes that communicate by means of the communication media that have not been associated to any subtask. Decomposition may then proceed by repeating this procedure of grouping for any complex process. This, eventually, leads to a so-called **process model**.

Introducing two very similar, but distinct entities (designs and tasks, activities and processes), in principle allows us to develop two possibly radically different hierarchies: one for capturing functional aspects (the structure model), and one for capturing nonfunctional aspects (the process model). However, the distinction between a design and a task is, to a certain extent, artificial: in ADL a design and a task are treated as a single entity. Our assumption is that at a certain stage in the design process of an application, *the complete organization of an application into logical units is the same as the organization into executable units*. In terms of Ward and Mellor [18] we are thus assuming that essential modeling and processor modeling lead to the same application model.

The relationship between a process model and a structure model is similar to the relationship between a variable and its type in imperative programming languages, or between an instance and its class in the world of object-orientation. The process model thus reflects the structure of a specific implementation. As such, it can be *annotated*, meaning that additional information is supplied which does not affect the functionality, but is purely directed towards deriving a “better” implementation. In the case of ADL, one sort of annotation that is currently supported, is that of **process replication**. These annotations are targeted towards performance enhancements by replicating processes, and thus enhancing parallelism in an implementation. A second form of annotation which is anticipated, but not yet supported, is that of mapping objects in a process model to the hardware resources of a target machine.

4.2 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.

4.2.1 Replicators

Substructures in an ADL task can be replicated by applying a so-called **replicator**. A replicator consists of a **replication factor** which is a positive integer, say N , and a **replication index** which is an integer variable taking values from the set $\{0, \dots, N - 1\}$. A replicator with replication factor N and replication index j is graphically denoted as



A replicator can be applied to a substructure of a task by selecting a number of processes and communication media. These sets are denoted as the **replication process set** and **replication media set** respectively and are graphically identified by drawing a dashed arc from the replicator to each element that takes part in the replication. A communication media can only be selected if all processes that are connected to it are

selected for replication as well. The effect of applying a replicator can be formulated more accurately as follows. Denote by (N, j) a replicator, applied to a substructure consisting of a set of processes \mathbf{P} and a set of communication media \mathbf{C} . Let T denote the task before replication, and T^* the task after applying replication.

1. For each communication medium $c \in \mathbf{C}$ we have that if a process P is connected to c , then it should also hold that $P \in \mathbf{P}$.
2. For each process P of T , with $P \notin \mathbf{P}$, then P will also be part of T^* . Similarly, for each communication medium c of T with $c \notin \mathbf{C}$, c will also be part of T^* .
3. For each process $P \in \mathbf{P}$, P is replicated N times yielding the processes P_1, \dots, P_N in T^* . Similarly, each communication medium $c \in \mathbf{C}$ is replicated N times resulting in the communication media c_1, \dots, c_N in T^* .
4. Let P denote a process in T , and c a communication medium. Assume that P is connected to c in T . Then, the resulting connections in T^* are as follows:
 - (a) $P \notin \mathbf{P}, c \notin \mathbf{C}$: P is connected to c like it was in T .
 - (b) $P \in \mathbf{P}, c \notin \mathbf{C}$: c is connected to each process P_k in T^* , as it was connected to P in T .
 - (c) $P \notin \mathbf{P}, c \in \mathbf{C}$: this situation cannot occur.
 - (d) $P \in \mathbf{P}, c \in \mathbf{C}$: then for each $j \in \{0, \dots, N-1\}$ we have that P_j is connected to c_j in T^* just as P was connected to c in T .

The first condition may seem as a rather restrictive one. What it effectively establishes is that no *additional* connections will ever be made to an activity on account of replication. Consequently, the interface of an activity is left intact. When dealing with an implementation environment based on CSP, this restriction may lead to serious problems. In these environments, communication between processes is always strictly 1 : 1, i.e. point-to-point, and special measures have to be taken to relax this constraint. However, from a design point of view, these restrictions are viewed as sources of overspecification, and should be dealt with solely during the implementation phase.

Let us illustrate the notion of a replicator by an example.

Example 4.2.1. Consider Figure 4.1(a) which shows a simple task 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 task shown in Figure 4.1(b) in the case that $N = 4$.

□

If a complex process is replicated, then the complete task into which it has been decomposed is replicated as well. Also, a process or communication medium may be subject to several replications, referred to as **replication composition**. Assume that an object O is subject to two replicators R_1 and R_2 . Denote by \mathbf{P}_k the replication process set of $R_k, k = 1, 2$. Furthermore, let $P \in \mathbf{P}_1 \cap \mathbf{P}_2$, and assume that R_1 is applied first

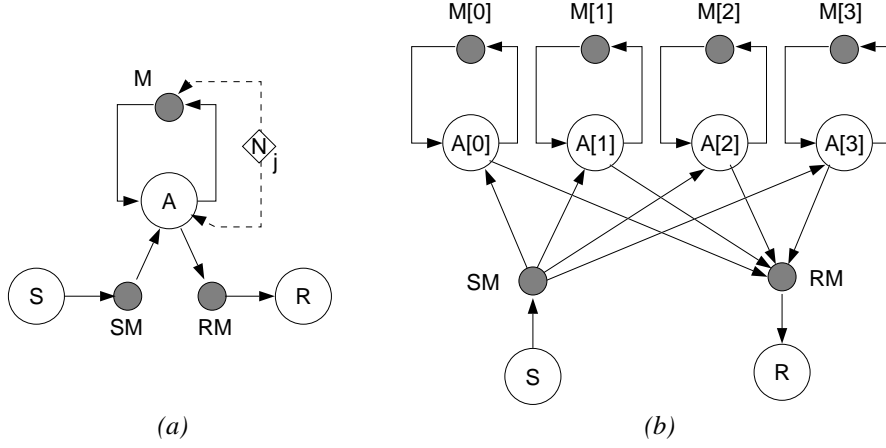


Figure 4.1: The result of applying a replicator with replication factor 4.

resulting in the replication of P into N copies P_1^1, \dots, P_N^1 . In that case, the set \mathbf{P}_2 is to be replaced by the set

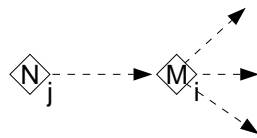
$$\mathbf{P}_2 \leftarrow \mathbf{P}_2 - P + \{P_1^1, \dots, P_N^1\}.$$

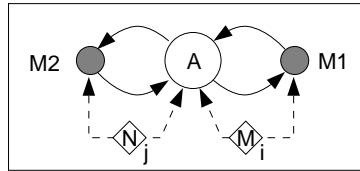
In other words, all replicated processes P_i^1 are to be subject to application of replicator R_2 . An important observation is that it can be readily shown that repetitive application of replicators is commutative. In other words, if $R_2 \circ R_1$ denotes the fact that replication R_2 is applied after application of replicator R_1 , we have that

$$\forall R_1, R_2 :: R_2 \circ R_1 = R_1 \circ R_2.$$

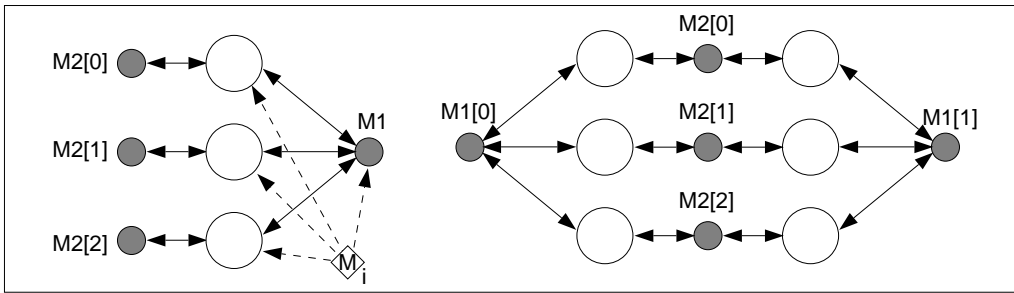
Example 4.2.2. To illustrate, consider the replication specification shown in Figure 4.2(a), and consider the case that $N = 3$ and $M = 2$. If we first apply replicator $R_1 = (N, j)$ and then $R_2 = (M, i)$ we obtain the situation shown in Figure 4.2(b). Note how after application of R_1 , we need to replace the set of processes to which replication of R_2 should be applied, by removing A , and adding its replicated counterparts. Analogously, Figure 4.2(c) shows the application composition $R_1 \circ R_2$. It is not difficult to see we indeed have that $R_2 \circ R_1 = R_1 \circ R_2$. □

If \mathbf{P}_k and \mathbf{C}_k denote the replication process set and replication media set, respectively, of a replicator R_k , we use the following abbreviated notation for a replication composition $R_j \circ R_i$ in the case that $\mathbf{P}_i = \mathbf{P}_j$ and $\mathbf{C}_i = \mathbf{C}_j$:

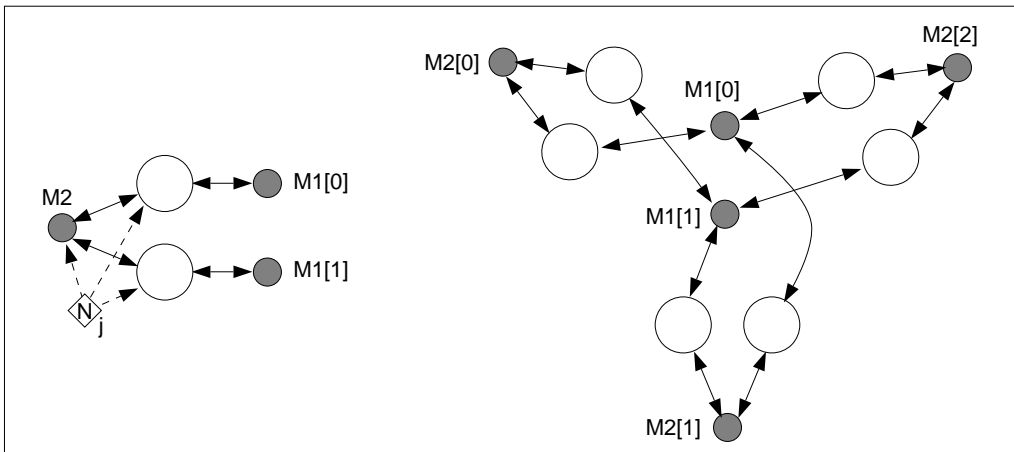




(a)



(b)



(c)

Figure 4.2: An example of replication composition where a process is subject to two replicators.

4.2.2 Connection statements

Replication introduced so far is not sufficient to be used as a means for specifying regular communication structures. Examples of such structures are pipelines, arrays, 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**. We shall first discuss both concepts and then illustrate their applicability.

Redirection statements

Redirection statements are annotations to connections between objects to which the same replicator is to be applied. They take the form:

$$\textit{source index expression} \leftarrow \textit{target index expression}$$

A source index identifies the source object of the connection after replication has taken place. Similarly, the target index expression indicates a single target object after replication. Note that, by definition the source and target object are always distinct: a connection can only be made between a process and a communication medium. Now suppose there was a connection before replication from an object source to an object target. A redirection statement

$$i \leftarrow \textit{expr}(i)$$

will then connect source[i] to target[expr(i)], instead of the *default* connection from source[i] to target[i]. The target index expression expr(i) corresponds to an integer function $E(i)$ on i where $i \in \{0, \dots, N - 1\}$, and N is the replication factor. In those cases that $E(i) \notin \{0, \dots, N - 1\}$, the expression is said to be *out of range*, yielding that the connection is discarded all together. If this also implies that if source[i] no longer satisfies the syntax rules of ADL, then source[i] and its connections are discarded as well. It is important to note that due to this rule, less objects may be replicated than specified by the replication factor.

Attachment constraints

Attachment constraints are annotations to connections between two objects of which only one was subject to replication. An attachment constraint takes the form

$$\textit{index} \mathbf{relop} \textit{index expression}$$

where **relop** is a usual mathematical relationship operator. The variable **index** refers to the object that is subject to replication, and after replication has taken place. After replication, connections for which the attachment constraint does not hold are discarded. If the attachment constraint fails to hold for any connection, it is considered to be at fault.

4.3 Examples

To illustrate the applicability of connection statements, we consider two examples. Both examples are based on the process model shown in Figure 4.1(a).

Example 4.3.1. In order to specify a pipeline of processes, we attach connection statements as shown in Figure 4.3(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 $0 \leq j \leq 4$. This is in accordance to the default replication rules described in Section 4.2.1. The redirection statement

$$j \leftarrow j + 1$$

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 $A[5]$ will 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 $A[0] \dots A[4]$ shown in Figure 4.3(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 that is to be made after replication, is the one from SM to $A[0]$. Similarly,

$$j = N-1$$

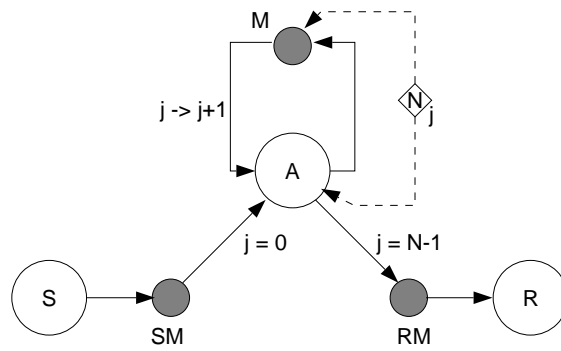
implies that there will only be a connection from $A[4]$ to RM.

□

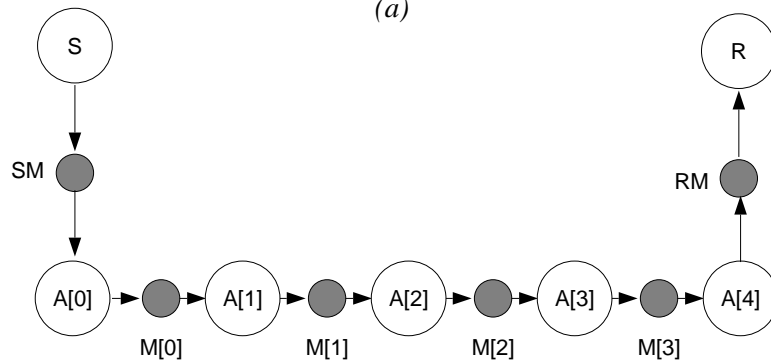
Example 4.3.2. A more intricate example is the specification of an array of processes. Consider the specification shown in Figure 4.4(a). In the first place, we now distinguish two communication media M1 and M2 which are to be replicated. Moreover, these communication media, together with A are subject to two replications: $R_1 = (N, j)$ and $R_2 = (M, i)$. Initially, this will lead to a collection of $N \times M$ objects $A[i][j]$, $M1[i][j]$, and $M2[i][j]$, respectively. The default connections are as follows:

$$\begin{array}{ll} M1[i][j] & \longrightarrow A[i][j] \\ A[i][j] & \longrightarrow M1[i][j] \\ M2[i][j] & \longrightarrow A[i][j] \\ A[i][j] & \longrightarrow M2[i][j] \\ SM & \longrightarrow A[i][j] \text{ for each } i, j \\ A[i][j] & \longrightarrow RM \text{ for each } i, j \end{array}$$

By adding redirection statements and connection constraints these defaults can now be overruled as follows. Assume that $N = 4$ and $M = 3$. First, completely analogous to



(a)



(b)

Figure 4.3: The specification of a pipeline of processes using replication.

the previous example, we construct a *series* of pipelines by attaching the redirection statement

$$j \leftarrow j+1$$

to the connection from M1 to A. Again, all replicated communication media M1[i][4] are discarded for the same reason that M[4] was discarded in Example 4.3.1. The redirection statement

$$i \leftarrow i+1$$

attached to the connection from A to M2 establishes the connection between the pipelines as shown in Figure 4.3(b). Again, the communication media M2[3][j] are discarded as this would lead to a violation of the ADL syntax rules. The connection statements should now be obvious.

□

4.4 Summary

In order to facilitate the exploitation of parallelism in a design, an ADL structure model can be converted into a process model that can subsequently be annotated. A process model describes the application as a collection of parallel processes. By using annotations, complete substructures can be replicated into a series of isomorphic substructures yielding a new process model. By attaching redirection statements and connection constraints, the replicated substructures can be reformed to regular structures such as pipelines and arrays.

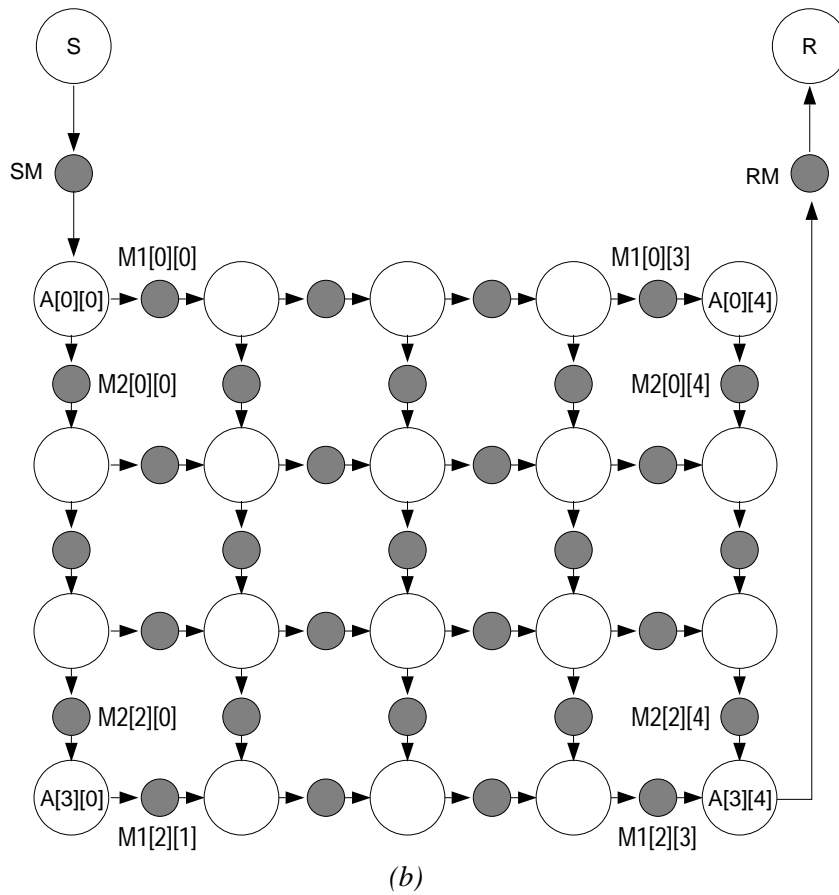
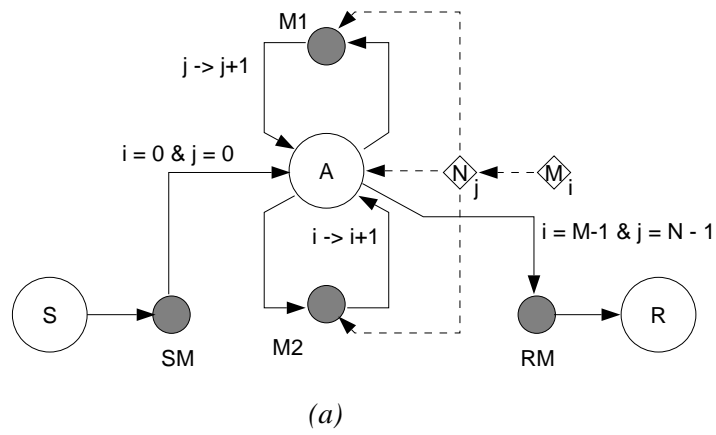


Figure 4.4: The specification of a grid of processes using replication.

Bibliography

- [1] G.R. Andrews. *Concurrent Programming: Principles and Practice*. Benjamin/Cummings, 1991.
- [2] M.J. Bach and S.J. Buroff. "Multiprocessor UNIX Operating Systems". *AT&T Technical Journal*, 63(8, part 2):1733–1749, October 1984.
- [3] D.G. Bate. "Mascot 3: An Informal Introductory Tutorial". *IEE Software Engineering Journal*, 1(3):95–102, 1986.
- [4] N. Carriero and D. Gelernter. "How to Write Parallel Programs: A Guide to the Perplexed". *Computing Surveys*, 21(3):323–358, 1989.
- [5] CCITT. "Functional Specification and Description Language (SDL)". In *The CCITT Red Book*, volume VI, chapter 10. CCITT, Geneva, 1985.
- [6] Hamlet Consortitium. "Application Requirements". Hamlet Technical Report, AEG Electrocom, Konstanz, Germany, September 1992.
- [7] E.W. Dijkstra. "Cooperating Sequential Processes". In F. Genuys, (ed.), *Programming Languages*. Academic Press, 1968.
- [8] C.A.R. Hoare. "Communicating Sequential Processes". *Communications of the ACM*, 21(8):666–677, August 1978.
- [9] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.
- [10] I. Jacobson. *Object-Oriented Software Engineering, A Use Case Driven Approach*. Addison-Wesley, 1992.
- [11] 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.
- [12] G. Jones and M. Goldsmith. *Programming in Occam*. Prentice-Hall, 1988.
- [13] R. Saracco, J.R.W. Smith, and R. Reed. *Telecommunications Systems Engineering using SDL*. North-Holland, Amsterdam, 1989.
- [14] H.R. Simpson. "The Mascot Method". *IEE Software Engineering Journal*, 1(3):103–120, May 1986.

- [15] A.S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, Englewood Cliffs, N.J., 1992.
- [16] M.R. van Steen. "The Hamlet Application Design Language (Version 1.0), On Automated Code Generation". Technical Report, Department of Computer Science, Erasmus University Rotterdam, 1994. In preparation.
- [17] P.T. Ward. "The Transformation Schema: An Extension of the Data Flow Diagram to Represent Control and Timing". *IEEE Transactions on Software Engineering*, SE-12(2):198–210, 1986.
- [18] 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.
- [19] P. Wegner. "Concepts and Paradigms of Object-Oriented Programming". *OOPS Messenger*, 1(1):7–87, 1990.
- [20] A. Yonezawa and M. Tokoro. *Object-Oriented Concurrent Programming*. MIT Press, Cambridge, Mass., 1987.
- [21] 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.