Designing Distributed Programs with Dynamic Communication Structures

Mark Polman
Erasmus University, Rotterdam
e-mail: polman@cs.few.eur.nl

Maarten van Steen
Vrije Universiteit, Amsterdam
e-mail: steen@cs.vu.nl

Abstract

The growing interest in using clusters of workstations as the target platform for high-performance applications, has again emphasized the need for support tools that can be used during application design. In this paper we present a graphical technique, called ADL-D, that allows a developer to construct an application in terms of communicating processes. The technique distinguishes itself from others by its use of highly orthogonal concepts, and the support for automated code generation. Developers are encouraged to concentrate on designing components in isolation, making the complex design space more manageable than would otherwise be the case. ADL-D can be used from the early phases of application design through phases that concentrate on algorithmic design, and final implementation on some target platform. Rather than presenting details of ADL-D, we use it here as a vehicle for a more general discussion on design level support for parallel and distributed applications. In this discussion, an emphasis is put on the design of dynamic communication structures, i.e. structures that can change during runtime.

1 Introduction

The widespread availability and low cost of clusters of workstations (COWs), together with the extensive support with regard to systems software, has led to a growing interest in using COWs as a platform for parallel, distributed applications. In contrast to the massively parallel distributed architectures that came up during the mid-1980's, COWs now offer a stable and affordable environment for research in medium to coarse-grained parallelism. Hence, many research groups now use them as their implementation vehicle. The development of software libraries for more direct access to the network (i.e. without having to pass through many layers) only adds to this trend [EKO95, vEBBV95].

Unfortunately, the traditional problems of parallel application development are as valid for COWs as they are for any other distributed architecture. The exploitation of parallelism to gain application efficiency still imposes design hazards on a developer, such as complex synchronization issues, process-processor mappings, replication strategies, and so on. Thus, parallel and distributed software engineering remains a difficult, error-prone process.

Traditionally, tools to speed up development were offered that make problem solving with respect to communication easier, such as monitors and debuggers. But rather than focusing on how to *correct* distributed applications, support should be provided to *prevent* poor designs and implementations. This support should be provided during the phases of logical and technical design, and continue in a seamless way down to and including the implementation phase. This requires a clear and simple model of communication as a basis, which is easily mapped to target platforms, such as COWs.

In the next section, we present the basics of such a model, including communication structure and semantics, and dynamic behavior. Also, we discuss the possibility of runtime changes in the communication structure, which requires careful modeling. Finally, we address the subject of automated code generation, an essential tool in software development. In Section 3 we introduce a graphical design technique for parallel and distributed software, called ADL-D. ADL-D is based on the proposed communication model, allowing for easy design of communication structures and automated generation of efficient target code. The notations of the technique are given, with short outlines of their semantics. Details can be found in [vS93, PvS96a]. In Section 4, we discuss ADL-D's features for modeling runtime structure changes. We conclude this paper with the current state of affairs regarding ADL-D and a comparison of our work to that of others.

2 Distributed application design

2.1 Choosing a communication model

Irrespective of the level at which support is provided, the most important requirement for any model is that it provides a clear and unambiguous view on the structure and dynamics of the final program. For parallel, distributed programs, the two basic components of a model are its unit of execution and its communication form. As for the first, practice shows that the unit of sequential execution is almost invariably chosen to that of a *process*, and that parallelism is exploited by having several processes run simultaneously.[1]

Process interaction is determined by the chosen communication model. Here, communication through *shared data* [LKBT92] has demonstrated to be a relatively easy model to work with. Protecting shared data from simultaneous access then becomes the main design issue, but given the proper synchronization abstractions, this need not be hard to accomplish. However, deriving efficient applications from programs designed in this model is hard if the target system does not support it as well. More specifically, in the absence of shared memory, implementing a shared data model efficiently requires

[1] Finer grained parallelism, such as expressed at the level of statements, is often advocated for developing algorithms, which we are not concerned with in this paper.

intricate optimization schemes including replication of data for fast access and broadcasting for consistent updating.

An alternative is to use message-passing, which is supported by many platforms, notably COWs, allowing for straightforward derivation of efficient implementations. This model, however, is at a low level of abstraction, requiring more effort from the designer.

This problem of a lack of abstraction is partly offset in an *object-based* model, where self-contained objects communicate through method invocation, with semantics similar to ordinary procedure/function calls. The object-based approach has been advocated for long by language designers, but how to actually incorporate parallelism into an object-oriented language is still a subject of much debate (see also [Mey93, Pap95]). For one, it can be argued that synchronous method invocation reduces the degree of parallelism. But perhaps more important, is that the object-based approach may easily lead to designs that are not well-suited to be executed on parallel and distributed platforms. More specifically, there can be a clash between the drive to capture data into objects from a logical point of view, and the need to distribute that data to exploit parallelism for efficiency.

Because of the implementation problems with other communication models, we advocate a message-passing model when COWs are to be used as the target platform. The difficulties related to this model can, in our opinion, be tackled at the design level, where an emphasis should be put on designing *communication structures*. Thus, communication becomes the main issue instead of the individual processes that constitute an application.

2.2 Designing communication

All aspects of communication in a message-passing model together form quite a complex design space for a developer to handle. This complexity can be reduced by distinguishing orthogonal modeling concepts. Communication structure versus (communicative) behavior of individual processes is one example. Other communication aspects include message transfer semantics, message data types, and dynamic changes in the communication structure. We will discuss the incorporation of these orthogonalities into our model in sequence.

Isolating behavior

An important aspect throughout the design of parallel, distributed applications, is limiting a process' view on communication in such a way that it is isolated from, for example, other processes. This allows a designer to concentrate on a single process without having to take (the behavior of) other components into account. Such an isolation can be realized by means of proper interfacing techniques. To that end, we advocate the use of *communication endpoints*. Communication endpoints are a process' 'doors' to the outside world. Whenever a process wants to communicate, it sends or receives a message through one of its endpoints. Where exactly the message goes after that is irrelevant to the process' behavior. Now we can model its behavior using just two basic components: (internal) computations, and communication through its communication endpoints, all in total isolation from other processes.

Modeling structure

What happens to a message once it is sent through an endpoint of a process, is captured in the communication *structure* model. Here, we model the possible flows of messages between endpoints. In order to capture the full semantics of this communication, *channels* can be introduced. A communication channel connects one group of endpoints to another, and thus provides for the transport of messages between members of both groups.

Message transfer semantics

The outlined structural framework of processes, endpoints and explicit channels now provides the hooks to incorporate the modeling of actual *message transfer semantics*, such as:

- blocking semantics, that describe whether and how processes block during communication;
- multicast semantics, that describe exactly which processes are involved in a communication;
- buffering semantics, that describe whether and how messages are buffered during communication.

Since it is the channels that are responsible for distribution of messages over receivers, they should carry the multicast semantics, resulting in, for example, unicast and broadcast channel forms. Also, by giving each channel a certain capacity, we can model buffer semantics. Finally, the time that a process is prepared to block for communication to succeed is a question that mainly concerns processes. Hence, blocking semantics are carried by the interfaces, resulting in a *timer* per endpoint, that indicates the maximum blocking time for that endpoint.

Message data types

The data types of messages is something that could be modeled as part of an endpoint or as part of a channel (or perhaps both). In any case, data types should be made available to the endpoints, since in the end they must be known to the processes.

Runtime Structure Changes

In distributed applications the communication structure itself can be subject to runtime changes, due to a need to dynamically load functionality, or to replicate components for increased parallelism or reliability. When adopting a model of processes that are connected through channels as explained so far, runtime structure changes can easily be expressed in terms of *creating* and *deleting* processes and channels. This approach has at least one implication: the description of a process is actually the description of a *process prototype*. A process is then to be seen as a prototype instance. A similar reasoning holds for channels.

We distinguish three independent aspects in modeling dynamic creation:

- modeling parent-child relationships between processes;
- modeling creations as a part of process behavior (since processes create other processes);
- modeling communication structures that evolve during runtime.

Parent-child relationships should form part of the structure model. Likewise, if we adopt a model in which runtime changes are initiated by processes, modeling when creation or deletion takes place should be part of the behavior design of processes. The difficulty lies in expressing dynamically changing communication structures, which implies expressing changes in a communication graph. Here, graph rewriting grammars as used by [BC87] are considered by us as too complicated to incorporate in a general design technique. An alternative approach, in which a simple default scheme is used to cover most cases, but which can be customized for special cases, is adopted in ADL-D. This scheme is discussed below.

2.3 Code generation

Design support should not end with providing a designer with a tool to structure an application in terms of, for example, processes, channels and communication endpoints. If that were the case, considerable effort would still be needed for translating a design into an implementation. We advocate that a design technique should enable automated code generation as much as possible. In fact, a design technique should be integrated with implementation techniques in a seamless way.

Of course, we need the generated code to be *efficient*. Here, our message-passing model proves its value. First, it is sufficiently low-level to allow for a straightforward mapping on the platforms it is intended for. Second, the orthogonality of modeling concepts allows us to independently determine the most efficient implementation of each concept.

An important condition for automated code generation is that the technique has a solid semantical basis. Every modeling concept that is introduced must be semantically well-defined (i.e. on top of possessing the properties of orthogonality and intuitive appeal).

In our ADL-D discussion below our main purpose is to explain the essence of the most important modeling concepts. Therefore, we shall omit any formal definitions. However, it should be noted that a formal definition exists for every part of the technique. These formal definitions are the guidelines for translation of ADL-D designs into parallel and distributed code.

More about code generation can be found in the discussion section on ADL-D.

3 ADL-D

In this section, we introduce ADL-D, a graphical design technique, based on the model outlined in the previous section. ADL is an abbreviation of Application Design Language. As a design technique for parallel software, ADL has existed for several years. ADL-D extends ADL by adding higher-level communication and dynamic creation concepts, thus making it more suitable for *distributed* software design. Below, we will describe ADL-D's communication features and its behavior model. Modeling dynamic creations is discussed separately in Section 4. To avoid blurring the point of our discussion, we use mostly semi-formal descriptions. For the formal definitions, we refer to [vS93, PvS96a].

3.1 The Communication Graph

ADL-D's communication graph notation consists of symbols for *processes*, *gates*, and *channels*. Processes and channels are as described in the previous section. Gates are the communication endpoints that form the interface between processes and channels. In Figure 1(a) we see how processes A1, A2, and A3 each have an *output* gate on channel Chan, whereas processes B1, B2, and B3 have *input* gates on Chan.

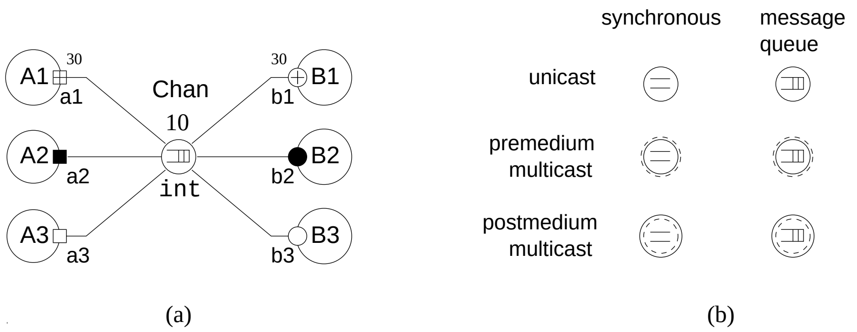

Figure 1: (a) ADL-D *example communication graph; (b)* ADL-D *channels*

Channels

A channel is represented by a named circle with two annotations: the data type and the buffer capacity. Chan in the figure is a *unicast* channel that transports integers, and has a buffer capacity of 10 messages. Chan only accepts messages from the output gates connected to it (a1, a2 and a3), if its buffer is not full. If an input gate (b1, b2 or b3) is ready to receive and there is a message in Chan, the longest pending message is transferred to the input gate.

Sending

Output gates appear as small squares on the edge of a process. They have one annotation: their timer. The semantics on the senders' side of the figure are as follows. Suppose A1 wants to send a message over a1. A1 then blocks on a1 until:

- the message is accepted by Chan (see below), or
- a1's timer (set to 30 seconds) expires.

Special cases are timers of ∞ and 0 seconds, indicating fully blocking and non-blocking gates, respectively. These are represented by filled and open gate symbols (a2 and a3, respectively).

Receiving

Input gates are represented by small circles. They, too, have timers. To illustrate, suppose B1 wants to receive over its input gate, b1. Then B1 blocks on b1 until:

- b1 receives a message from Chan's buffer, or
- its timer expires.

Chan is a unicast channel, which implies that a message can only be received through one input gate. Chan is also stateless and non-deterministic. This means that if several input gates are ready to receive, then no assumptions can be made beforehand as to which one will be the actual receiver of the message.

Other channels

For a message queue, a buffer capacity of 0 messages is a possibility. For this special case, ADL-D has a special symbol ('='). Such a message queue is also known as a *synchronous* channel, which implies that communication only succeeds if there are, at the same time, both a sending process and a receiving process willing to communicate through the channel.

For both synchronous and asynchronous channels, multicast variants exist, which, in principle, transfer messages to all receivers. Here, the predicate *postmedium* indicates that a message is not transferred before *all* receivers are ready. With a *premedium* multicast every receiver gets the message as soon as possible. Multicasting is further described in [vS93].

3.2 Process decomposition

As was stated, it is essential that a designer can model a process' behavior in isolation from other processes in the application. In ADL-D, it is also possible to model a certain part of the communication structure in isolation from other parts. This is done through process decomposition, rendering a process hierarchy.

As an example, consider again Figure 1(a). Here, process B2 could be composed of three other processes (C1, C2, and C3), as shown in Figure 2. To model that the processes in Figure 2 form a refinement of process B2, we have to associate every gate of B2 with gates in its decomposition. To that end, we let gate b2 of B2 return as an *external* gate in the decomposition of B2. This is called a *vertical* association.

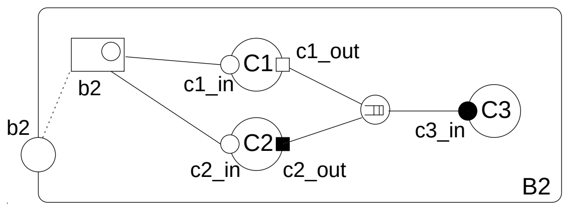

Figure 2: Decomposition of process B2

The external gate, in turn, has a *horizontal* association to gates c1_in and c2_in. In Figure 3, we have drawn the semantical equivalent of the combination of Figure 1(a) and 2.

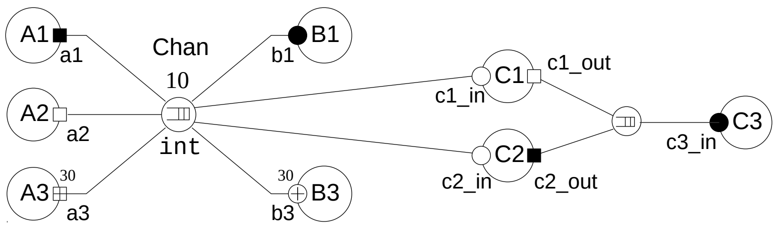

Figure 3: Combination of Figures 1(a) and 2

3.3 The Behavior Model

Sequential behavior modeling per process in ADL-D is done using state-transition diagrams (STDs) of simple, i.e. non-decomposed processes. Here, emphasis is put on communicative behavior by supporting separate *communication states*. An example STD is given in Figure 4 which shows the dynamic behavior of process C2.

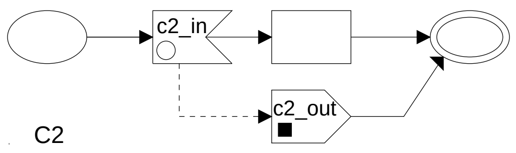

Figure 4: State-transition diagram of process C2 *from Figure 2*

From its initial state (left), it proceeds to receive over gate c2_in in a so-called *input* state (also named c2_in). If communication succeeds, a transition occurs along the solid arc to a *computation* state, after which the process terminates. Otherwise, in the case of communication failure, the transition along the dotted line (we call this a *time-out* transition) is followed, leading to the *output* state c2_out. Here, it communicates through output gate c2_out. After that, the process terminates. Notice that no direct interaction with channels or other processes occurs: all communication is modeled in terms of gates, as was emphasized in Section 2.

3.4 High-level Communication

A consequence of the use of unidirectional, stateless and non-deterministic channels in ADL-D is, that modeling the communication of a series of messages to the same receiver, and modeling request-reply behavior, are hard to realize. Still, these are very natural communication forms, and hence we search for a way to incorporate them into ADL-D, without sacrificing ADL-D's valuable property of inter-process anonymity. The solution lies in two new concepts: the *two-way* channel and the *connection* channel.

The two-way channel

The two-way channel, shown in Figure 5(a), is a combination of two unidirectional channels to which a process can be attached (only) by a so-called *two-way gate*. A two-way gate is a combination of one input and one output gate that only jointly can be assigned to a process. The semantics of the individual channels and gates that underlie the two-way channel are exactly as explained above.

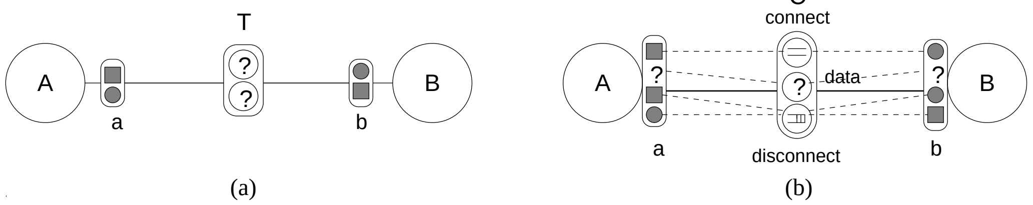

Figure 5: (a) Two-way channel; (b) Connection channel

The connection channel

A connection channel, as the term indicates, is for establishing a connection between processes over a channel, making directed communication possible. A connection channel, as shown in Figure 5(b), consists of three separate channels: a synchronous channel (shown as connect) for establishing a connection, a designer-chosen channel (data) for the actual data transmission (this can be a two-way channel), and a message queue of capacity one (disconnect) for disconnecting the communicating parties. The semantics are, that once A and B have communicated over connect, data will behave to them as if they were the only processes connected to it. This means that every message sent by A can only be received by B, and vice versa. Disconnecting is done by communication over disconnect.

Similar to two-way channels and gates, connection channels require *connection gates*. These are combinations of four (not three, as might be expected) gates: one for connection establishment, two for disconnection and one for actual data transfer (which, again, can be two-way). A precise description of the semantics of connection channels and gates can be found in [PvS96a]. There, we also explain *multiplexed* connection channels, which can maintain several connections between process pairs simultaneously.

Notice, that by making data in Figure 5(b) a message queue, the sending of multiple messages to the same receiver can be easily modeled; a situation that resembles a datastream over a TCP-connection. By making data a two-way channel, request-reply behavior can be captured. In a situation with one request and one reply, we move towards the semantics of RPC over a reliable (e.g. TCP) connection. These apparent similarities are further proof of the fact that ADL-D designs are easy to implement on COWs.

Finally, notice that ADL-D's scheme of connection-oriented communication does not violate the principle of inter-process anonymity. After all, processes still only communicate through their gates. This does imply that a connection request can never be addressed to one specific process. In other words, the requesting process does not know (or care) to whom it gets connected, as long as there can be directed communication once the connection is made.

Behavior

An advantage of modeling connections through a scheme of multiple channels and gates is that we can model their use in the STDs by standard communication states with the usual success and timeout transitions. Thus, no extra notations or new formal semantics are needed in the ADL-D STDs to handle two-way and/or connection-oriented communication.

Macro

A weakness might be the slightly cumbersome notation. However, for simple cases of connection-oriented communication, we introduce what we call a *macro*, an abbreviated notation. At the moment, the only macro that exists in ADL-D is a simpler notation for RPC communication, but others are possible as well. Again, we refer to [PvS96a] for more details. It should be noted that macros are purely a *notational* issue. Nothing is added that could not already be modeled in ADL-D as it was.

4 Dynamic Creation

Adequate support for modeling the dynamic creation of communication structures is a difficult and often neglected issue. In practice, the number of instances of modeled software components (processes, objects, etc.) can only be set at design time. In Section 2, we stated our goals concerning instantiation modeling: parent-child relations, dynamic behavior aspects, and communication structure evolvement over time. Below, we will discuss the ADL-D solutions for them in sequence.

But before doing so, it is important to make a clear disctinction between processes and channels as we have discussed so far, and those that result from dynamic creation. Up until now, processes and channels always corresponded to their actual instances. In other words, a process as appeared in a structure diagram, corresponded to its physical counterpart during runtime. This relationship no longer holds when processes and channels are created during runtime. Instead, a (composite) process or channel in a structure diagram acts, in fact, as a *prototype* of its runtime counterpart. We refer to the latter as *instances* (of a prototype).

4.1 Parent-child relations

Parent-child relations in ADL-D are modeled in structure diagrams. To this end, we introduce a new channel type, the *creation* channel. In Figure 6(a), we see the decomposition of a composite process A_composite, containing a creation channel named Create. Subprocess A has an output gate (a_create) attached to Create. This implies that instances of A can create instances of some other process, in our example B. Here, B is a composite process, consisting of subprocesses C and D, as illustrated in Figure 6(b).

A creation channel is never attached to an input gate. This makes sense, since there is no logical candidate to be the recipient of a "creation message." There is, however, an attachment to the process of which instances are to be created (B in our example). This is indicated by the dashed line between the creation channel Create and process B.

We demand that a creation channel can only refer to creation of instances of processes that are at the same level and directly contained in the same composite process as the creation channel itself.

4.2 Creator semantics and behavior aspects

The semantics on the creator's side are as follows. If A wants to create a new instance of B, it sends a message through a_create on Create. Then, A will be blocked on a_create until:

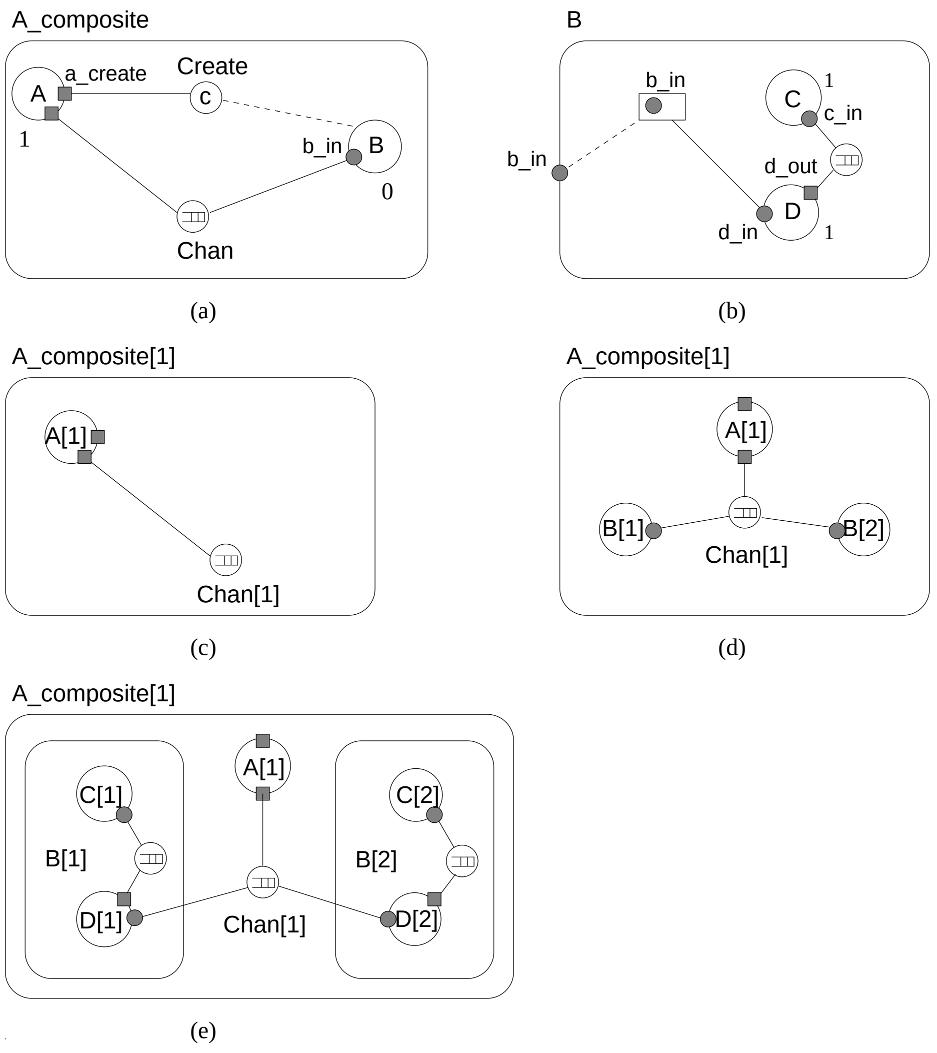

Figure 6: The creation channel and its effects

- the creation succeeded, or
- a_create's timer expires.

Notice that these semantics are similar to those of gates attached to other ADL-D channels. This implies that we can use ordinary output states and failure/success transitions when modeling the behavior of a creating process. In other words, from a creator's point of view, a creation attempt is not different in any way from an attempt to communicate.

4.3 Communication graph evolvement: the default case

The real challenge in modeling dynamic creation lies in modeling the changes in the runtime communication structure that occur due to creations of process instances. Here, it should be noted that ADL-D supports a process hierarchy that may be many levels deep. Consequently, instantiation of ADL-D processes (and channels!) must be done recursively, all the way down to the simple processes. In this recursive instantiation, it must be very clear and well-defined,

- how many instances of a process or channel are created, and
- (if we decide to create several instances of the same process) which process instance gets attached to which channel instance.

We start with describing a simple default scheme for instantiation, which suffices for most cases. In doing so, we use the example communication structure of Figure 6(a).

The number of instances

Suppose that A_composite is instantiated, rendering instance A_composite[1]. How many instances of A and B should be created in the recursive creation process? A simple rule would be: one of each. However, this might not always be desirable. Perhaps we would rather have no B instances at all, unless explicitly created by an A instance. Hence, ADL-D makes the number of initial instances variable. This is done by means of annotations to processes (not previously shown in our examples) in the form of integers.

In Figure 6(a), A is annotated with 1 and B with 0. Thus, instantiation of A_composite renders the instance diagram of Figure6(c). Chan is instantiated exactly once. In ADL-D this is the general rule for channels (see [PvS96a] for a motivation of this).

Attachments

At this point, we have instances of A and Chan from Figure 6(a). However, for actual runtime communication to take place, channel instances have to be connected to gates of process instances. In Figure 6(c), we see that A[1]'s output gate gets attached to Chan[1]. This illustrates the general ADL-D rule for attachment: *any process instance, part of a composite instance, say* A_composite[i], *only gets attached to channel instances within* A_composite[i].

Another example

As another example, suppose that, in Figure 6(c), A[1] sends two creation messages. This results in the creation of two new B instances (B[1] and B[2]), recursively instantiated as described above. Now, since B[1] and B[2] are created within A_composite[1], they both get attached to Chan[1], as illustrated in Figure 6(d). The decomposed equivalent of Figure 6(d) is given in Figure 6(e).

4.4 Non-default Attachment

Unfortunately, the relatively simple default scheme of instance attachment that we outlined above does not suffice if we want to model the runtime creation of more complex communication structures such as pipes, grids and trees. It seems that we need a possibility to model our own custom attachment schemes for special creations. Here, we could model, for example, that a creation message results in a pipe of identical process instances, instead of just one instance.

A configuration language

But as was stated before, doing this in a mere graphical way would require graph rewriting grammars on ADL-D communication structures. Such grammars are not easy to use and may needlessly complicate the design process. Hence, we offer, as an alternative, a simple mechanism in the form of a runtime-executable configuration language. In this language a designer can create instance structures the way he likes, simply by writing a small configuration program for each non-default creation. This program is then associated with a creation channel and invoked each time a message is put on this channel. Special input to the program can be captured in the message itself (which implies that creation channels can also be data typed, just like other ADL-D channels).

Example

An example of a non-default instantiation is given in Figure 7, where we want creations of B to result in pipes of variable length. To this end, a configuration program is associated with channel Pipe, taking the length of the resulting pipe as a parameter (an integer, hence the data type int). During runtime, each time an integer is sent over Pipe, the associated program is invoked. Figure 7(b) shows the situation on an instance level after default instantiation of A_composite, and a sending by A[1] of two creation messages, one containing 4, and one containing 2.

The language

The question now is what the language should offer to provide a designer with what he needs. We identified the following requirements:

- First an interface is needed to the designed communication structure. That way, a designer can let his configuration program gather information about the structure that would normally (i.e. by default) be created.
- Second, there must be an interface to query the actual (i.e. runtime) communication structure, so that the place for newly created processes and channels can be properly determined.

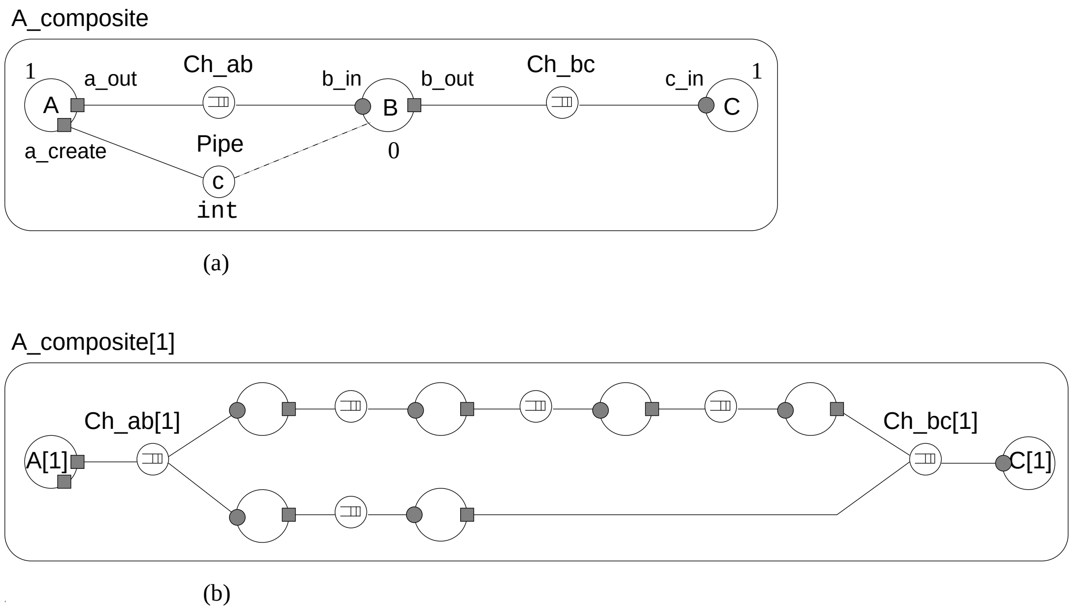

Figure 7: (a) Structure diagram with non-default creation channel; (b) Instance after two pipe creations

- Third, we need primitives to create new process and channel instances, and primitives for their attachment.

Prolog

As it turns out, Prolog is an excellent candidate to serve as a basis for a our configuration interface. To illustrate this, we return to Figure 7. If a designer wants to write a pipe-constructing Prolog program for creations sent over Pipe, he first of all needs information about the design itself. To this end, a Prolog interface to the design is offered to him in the form of a set of Prolog atoms, like

```
d_process('A').                          /* there is a process A */
d_channel('Pipe').                       /* there is a channel Pipe */
d_has_type('Pipe', creation, int).       /* Pipe is a creation channel for integers */
d_gate('a_create', 'A').                 /* A has a gate a_create */
d_has_type('a_create', out).             /* a_create is an output gate */
d_connected('Pipe', 'a_create').         /* a_create is attached to Pipe */
```

So, a designer could incorporate in his program a statement like: d_connected('Pipe', Gate). This would succeed and have Gate unified with 'a_create'.

During runtime, a second, similar database is maintained, describing the application on an instance level. New tuples are inserted as the application is being instantiated or new instances are created by creation messages. The designer can incorporate queries on this database in a program to find out where a new process/channel instance should be inserted. After that predicates like:

```
instantiate(Process).          /* instantiate Process */
connect(Channel, Gate).        /* connect Channel to Gate */
```

can be used to perform the actual structure update. In the case of a pipe of length 4, this means 4 instantiations of B, 3 creations of new message queues, and the necessary connects.

A more comprehensive description of these concepts, and an introduction to the Prolog interface can be found in [PvS96b].

Summary

With the Prolog interface described above, we have three techniques which can be used together to design a parallel, distributed application:

- A graphical communication structuring technique in terms of processes, gates and channels, rendering a process hierarchy by means of process decomposition. Here, we can also specify parent-child relations between processes.
- A graphical technique for the modeling of dynamic behavior per simple (non-decomposable) process, in terms of communication and computation states and transitions between these.
- A language to design complex instantiation schemes associated with creation channels.

5 Discussion

An initial version of ADL-D has been operational for some time now [vStDV94], and the current version which is being targeted towards COWs is still being improved. We are presently completing the implementation of a runtime system to support network applications derived from ADL-D. Our next major step concerns research into efficient code generation for COWs. So far, code generation has only been supported for parallel distributed memory machines such as those based on transputers and the PowerPC.

When placing our work in the context of support for parallel and distributed application development, it is surprising to see that relatively speaking, not much research has been conducted in the area of supporting the design of applications. Instead, most research and development effort has concentrated on the algorithmic level, often in the form of language design and accompanying tools. Nevertheless, there are a number of comparable projects.

Most notable is perhaps the work on modeling applications through functional dependency graphs, exemplified by HENCE [BDGS93] and CODE [BAS89]. In this approach, an application is viewed as a collection of functional and indivisible units, connected to each other by pure input/output relations. Each unit can be executed on a separate node, and parallelism is achieved by scheduling independent units simultaneously on different nodes. Although attractive for its simplicity, this model has a number of serious restrictions making it unsuitable as a general model for distributed applications. The most serious restriction is that units are indivisible. This implies that no communication is assumed during the execution of a unit. Consequently, a unit has to be decomposed whenever decisions based on computations affect communication, possibly resulting in intricate task graphs which need to be explicitly managed by the designer.

Opposed to functional decomposition are approaches based on message-passing. In PARSE [GGJ95], an application is structured in terms of processes and communication arcs between them. However, PARSE does not provide explicit interfaces between processes and communication, and neither does it provide support for runtime adaptations. Process behavior should be expressed through a separate language, or Petri nets. Automated code generation from a design is not supported at all. In a sense, PARSE is more a *specification* mechanism than a technique to support the design of parallel, distributed applications.

More in line with our work is PAR-SDL [SJG93]. It is also a graphical technique for designing parallel systems and has many similarities with ADL-D. The most important distinction is that PAR-SDL builds on state-transition machines and communication between them. In contrast, ADL-D uses the more abstract concept of processes as its starting-point, which are later refined with respect to their internal behavior. A strict separation between communication and computation is much harder to maintain in PAR-SDL. In fact, many semantical aspects of communication are completely left for the designer to cope with. Another drawback of taking communicating state-transition machines as a starting-point is that replication is much more difficult to incorporate.

Many of the requirements mentioned in Section 2 are met by the work on Regis [MDK94], in which distributed applications are developed through *configuration* of *components*. A component may be hierarchically constructed from other components and corresponds to our notion of a process. Explicit interfaces are also supported, making it possible to develop components in isolation. Communication between components is handled through user-definable communication objects that are placed at components. This is a major distinction from our work: in ADL-D interprocess communication is captured through channels, which are independent of the processes that use them. In effect, Regis supports only point-to-point communication between processes, and indeed, multicast facilities are only provided in a rudimentary form through events. We feel that our approach to modeling blocking, multicast, and buffering semantics through the use of gates and channels is more elegant. It also makes the semantics of dynamic creation of process instances easier to model and understand.

References

[BAS89] J.C. Browne, M. Azam, and S. Sobek. CODE: A Unified Approach to Parallel Programming. *IEEE Software*, pages 10–18, July 1989.

[BC87] D.A. Bailey and J.E. Cuny. An Approach to Programming Process Interconnnection Structures: Aggregate Rewriting Graph Grammars. In J.W. de Bakker and A.J. Nijman and P.C. Treleaven, editor, *PARLE: Parallel Architectures and Languages Europe*, volume 2 of *Lecture Notes in Computer Science 259*, pages 112–123. Springer-Verlag, Berlin, 1987.

[BDGS93] A. Beguelin, J.J. Dongarra, G.A. Geist, and V.S. Sunderam. Visualization and Debugging in a Heterogeneous Environment. *Computer*, 26(6), June 1993.

[EKO95] D.R. Engler, M.F. Kaashoek, and J. O'Toole. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *15th Symposium on Operating Systems Principles*, pages 17–32, Copper Mountain, Colorado, December 1995. ACM.

[GGJ95] I. Gorton, J. Gray, and I. Jelly. Object-Based Modelling of Parallel Programs. *IEEE Parallel and Distributed Technology*, (2):52–63, July 1995.

[LKBT92] W.G. Levelt, M.F. Kaashoek, H.E. Bal, and A.S. Tanenbaum. A Comparison of Two Paradigms for Distributed Shared Memory. *Software - Parctice and Experience*, 22(11):985–1010, November 1992.

[MDK94] J. Magee, N. Dulay, and J. Kramer. A Constructive Development Environment for Parallel and Distributed Programs. *IEE/IOP/BCS Distributed Systems Engineering*, 1(5):304–312, September 1994.

[Mey93] B. Meyer. Systematic Concurrent Object-Oriented Programming. *Communications of the ACM*, 36(9):56–80, September 1993.

[Pap95] M. Papathomas. Concurrency in Object-Oriented Programming Languages. In O. Nierstrasz and D. Tsichritzis, editors, *Object-Oriented Software Composition*, pages 31–68. Prentice Hall, Englewood Cliffs, N.J., 1995.

[PvS96a] M. Polman and M.R. van Steen. ADL-D: A Technique for Developing Parallel, Distributed Software. Technical report, Erasmus University Rotterdam, Department of Computer Science, 1996. *to appear*.

[PvS96b] M. Polman and M.R. van Steen. Dynamic creation in ADL-D. Technical report, Erasmus University Rotterdam, Department of Computer Science, 1996. *to appear*.

[SJG93] 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, editor, *Transputer Applications and Systems*, volume 2. IOS Press, Amsterdam, 1993.

[vEBBV95] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In *15th Symposium on Operating Systems Principles*, pages 303–316, Copper Mountain, Colorado, December 1995. ACM.

[vS93] M.R. van Steen. The Hamlet Application Design Language, Introductory Definition Report. Hamlet EUR-CS-93-16, Erasmus University Rotterdam, Department of Computer Science, December 1993.

[vStDV94] M.R. van Steen, A. ten Dam, and T. Vogel. Computer-Aided Support for Designing Parallel Real-time Solutions on Transputer-based Systems. In *IEEE Conference on Control Applications*, pages 811–816, Glasgow, August 1994.