Design Level Support for Parallel and Distributed Applications

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 a cluster 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.*

1 Introduction

There is a growing interest in clusters of workstations (COWs) as the target platform for parallel and distributed applications. Unfortunately, these networks impose the same problems on application development as parallel distributed memory machines did before. With the low-level communication support through message-passing, the development process remains relatively hard and time-consuming. Traditionally, support tools are offered that make problem analysis 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. We advocate that this support should be provided during the phases of logical and technical design, and that it should continue in a seamless way down to and through the implementation phase. To that end, a clear and simple communication model is needed that can easily be mapped to target platforms.

In this paper we start with presenting the basic requirements for such a model and its implementation. In Section 3 we present a technique, called ADL-D, which is based on such a model. We conclude this paper by briefly comparing our work to that of others.

2 Distributed application design

2.1 The basic model

The two basic elements of a model for parallel, distributed programs are its unit of execution and its communication form. In practice, we observe that the unit of sequential execution is almost invariably chosen to that of a *process*, and parallelism is exploited by having several processes run simultaneously.[1]

Process interaction is determined by the chosen communication model. Communication through *shared data* [5] has demonstrated to be a relatively easy model to work with, given the proper synchronization abstractions. However, deriving efficient applications is hard if the target system does not support this model as well.

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 is partly offset in an *object-based* model, where self-contained objects communicate through method invocation, with semantics similar to ordinary procedure/function calls. This approach has been advocated for long by language designers,

[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.

but how to actually incorporate concurrency into an object-oriented language is still a subject of much debate [7, 8]. The object-oriented model itself seems to be too restrictive when it comes to supporting exploitation of parallelism for the sake of efficiency. We feel this is at least partly caused by the fact that not objects, but the data they encapsulate, should be the starting point for parallelization. Distribution of data may easily conflict with the requirement that it is also to be encapsulated into (non-distributed) objects.

Mainly for these reasons, 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 processes.

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 many 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. Below, we discuss how we can equip a message-passing model with the proper features to implement the above orthogonalities.

In order to clearly separate the behavior of a process from communication between processes, it is necessary to use explicit interfaces. To that end, we advocate the use of *communication endpoints*. They allow us to model process behavior using just two basic components: (internal) computations, and communication through communication endpoints. In particular, behavior models can be constructed on a pure per-process basis, i.e. in isolation from other processes.

Similar reasoning holds for modeling the overall communication structure, which can also be treated separately. Modeling interprocess communication becomes, in fact, nothing else but modeling the communication between communication 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.

The outlined structural framework now provides the hooks to incorporate the modeling of actual *message transfer semantics*, such as blocking, multicast and buffering semantics. 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.

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 they must be known to the processes.

2.3 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 class*. A process is then to be seen as a class instance. A similar reasoning holds for channels.

We distinguish three independent aspects in modeling dynamic creation, namely modeling parent-child relationships between processes, modeling creation as a part of application behavior, and modeling the kind of structures that emerge during runtime. Parent-child relationships can be captured in the application structure model. Likewise, if dynamic creations are initiated by processes, the moment of creation can be captured in the process behavior model. The difficult part is the description of a dynamically changing communication graph. Here, we think that graph rewriting grammars [1] are too complicated to use in a design technique. An alternative approach is discussed below.

2.4 Code generation

Design support should not end with providing a designer with an application structuring tool. Instead,

the designer should be relieved from as much as possible of the burden of translating a design into an implementation, through automated code generation. A condition to be satisfied is that the generated code is *efficient*. Here, we see another reason to maintain a message-passing model. 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 individual concept.

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, its behavior model, and dynamic creation model.

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 above. Gates are the communication endpoints that form the interface between processes and channels. In Figure 1 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.

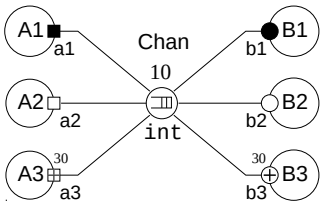

Figure 1: ADL *example communication graph*

The contents of the gate symbols indicate their maximum blocking time: indefinitely for A1 and B1, no time at all for A2 and B2, and 30 seconds for B3 and C3. An annotation to a channel symbol reveals its buffer capacity (synchronous channels have capacity 0). A full buffer implies that additional senders get blocked according to the timing of their output gates. From each channel, multicast variants exist as further described in [11]. An ADL-D channel is unidirectional, stateless and non-deterministic. That means that for each message from a sender, a channel makes a new decision about the receiver. This decision can, in principle, not be influenced by the designer.

3.2 Process decomposition

ADL-D provides a hierarchical view on an application through process decomposition. For example, process B2 from Figure 1 could be decomposed into C1, C2 and C3 shown in Figure 2. Here, gate b2 of B2 returns as an *external* gate in the decomposition diagram of B2.

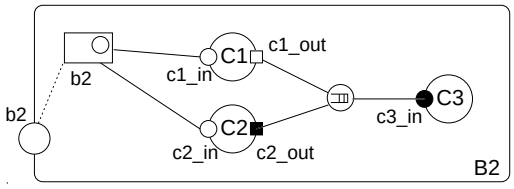

Figure 2: Decomposition of process B2.

3.3 The Behavior Model

Sequential behavior modeling per process in ADL is done through state-transition diagrams (STDs) of simple, i.e. non-decomposed processes. Emphasis is put on communicative behavior by supporting separate *communication states*. An example STD is given in Figure 3 which shows the dynamic behavior of process C_2.

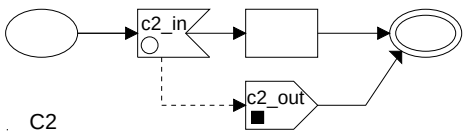

Figure 3: State-transition diagram of process C2 *from Figure 2.*

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 is followed, leading to the *output* state c2_out in which it communicates through output gate c2_out. After that, the process terminates. Notice that no direct interaction with channels or other processes occurs: everything is modeled in terms of gates.

3.4 High-level Communication

A consequence of the use of unidirectional, stateless and non-deterministic channels is that modeling the sending of multiple messages to the same receiver and request-reply modeling are hard to realize. ADL-D solves these problems by introducing the *connection* channel and the *two-way* channel.

The two-way channel, shown in Figure 4(a), is a combination of two unidirectional channels to which processes are connected by two-way gates. A two-way gate is nothing but a combination of one input and one output gate. The semantics of the channels and gates that underlie the two-way channel are exactly as explained above.

A connection channel (see Figure 4(b)) consists of three channels: connect (the '=' symbol denotes a synchronous channel) for establishing a connection between two processes, data for the actual data transmission, and 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. Disconnecting is done by communication over disconnect. A precise description of these semantics can be found in [9]. ADL-D also supports multiplexed connection channels, which can maintain several connections simultaneously.

By making data in the figure an ordinary message queue, the sending of multiple messages to the same receiver can be easily modeled. By making data a two-way channel, request-reply behavior can be captured. The advantage of modeling connections using three different channels is that we hardly need to add new semantics to ADL-D as it is. In our STDs we do not even need new notations: ordinary input and output states can be used to model communication over (dis)connection gates.

As a final remark, we note that for relatively simple forms of two-way communication traffic, such as RPC, using the full syntax for connection channels and gates can be rather cumbersome. To overcome this problem, we propose the use of *macros*: abbreviated notations for design situations that occur frequently. 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 [9] for more details. Note that macros are purely a *notational* issue. Nothing is added that could not already be modeled in ADL-D as it was.

3.5 Dynamic Creation

Support for modeling the dynamic creation of communication structures is a difficult issue. In practice, the number of instances of each component is often fixed at design time. ADL-D tackles the problem by the use of *creation channels*, which determine the parent-child relations in an application structure. Figure 5(a) shows how instances of process A can create instances of the (decomposed) process B. A *creation message* is sent through an output gate of A, but it is not explicitly received through an input gate. Communication through the output gate fails if the creation fails, leading to a normal failure transition in A's STD.

Dynamic creation can be applied to both simple and decomposed processes. If a process is instantiated, an instance is created of its whole internal structure (if present), after which it is placed in the communication graph, exactly as specified in the structure model. In Figure 5(b), we see that after creation of two B instances, both instances of D get connected to channel Chan.

Unfortunately, using the default insertion scheme, we cannot model the creation of more complex communication structures such as pipes, grids and trees. For this, we need special notations. As we stated before, a graph rewriting grammar is too complex for our purposes. Instead, we propose the use of a language in which we can describe the building of a communication structure as a series of steps, similar to many configuration languages. A difference lies in the fact that our configuration language should be interpretable during the execution of the application. Any special input to the configuration program is provided by a process, and transported over a creation channel. This also implies that creation channels can be annotated with data types as well.

The question is what this language should look like. In our opinion, it should look upon a communication structure as a relational database of processes, gates, and channels. By querying and manipulating this database, new processes can be inserted into a communication structure easily. As it turns out, Prolog can be used for this purpose (see [9] for further details).

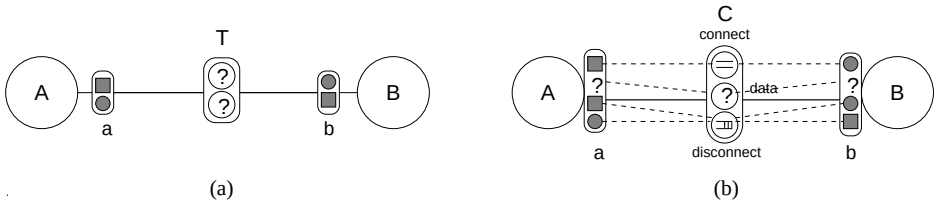

Figure 4: (a) Two-way channel; (b) Connection channel

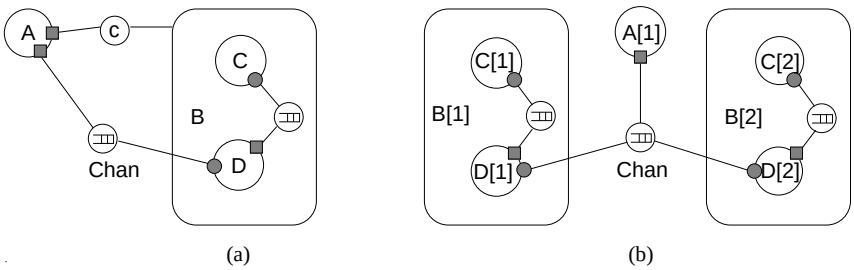

Figure 5: Creation channel and semantics

4 Discussion

An initial version of ADL-D has been operational for some time now [12], 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 [2] and CODE [3]. 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 [4], 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 [10]. 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 [6], 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

[1] 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, (ed.), *PARLE: Parallel Architectures and Languages Europe*, volume 2 of *Lecture Notes in Computer Science 259*, pp. 112–123. Springer-Verlag, Berlin, 1987.

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

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

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

[5] 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.

[6] 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.

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

[8] M. Papathomas. "Concurrency in Object-Oriented Programming Languages". In O. Nierstrasz and D. Tsichritzis, (eds.), *Object-Oriented Software Composition*, pp. 31–68. Prentice Hall, Englewood Cliffs, N.J., 1995.

[9] M. Polman and M. van Steen. "ADL-D: A Technique for Developing Parallel, Distributed Software". Technical Report, Erasmus University Rotterdam, Department of Computer Science, 1995. *to appear*.

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

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

[12] M.R. van Steen, A. ten Dam, and T. Vogel. "The Hamlet Design Entry System: An Overview of ADL and its Environment". Hamlet Technical Report EUR-CS-94-02, Department of Computer Science, Erasmus University Rotterdam, April 1994.