A Structured Design Technique For Distributed Programs

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

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

Arie de Bruin
Erasmus University, Rotterdam
e-mail: adebruin@few.eur.nl

Abstract

A non-formal motivation and description is given of ADL*-d, a graphical design technique for parallel and distributed software.* ADL*-d allows a developer to construct an application in terms of communicating processes. The technique distinguishes itself from others by its rigid orthogonal approach to communication modeling, which is advantageous in many areas. Without being committed to one particular design method,* ADL*-d as a technique can be used from the early phases of application design through phases that concentrate on algorithmic design, and final implementation on some target platform. In this paper, we discuss and motivate all* ADL*-d components, including recently incorporated features such as support for connection-oriented communication, and support for modeling dynamically changing communication structures.*

1 Introduction

Developers of parallel and distributed applications often face difficulties with respect to synchronization, distribution and replication. It is generally recognized that these problems should be attacked in the early development stages of logical and technical design. Consequently, classical support tools such as monitors and debuggers, and communication libraries are of little help, since these are generally applied much later. Instead, we require specific *design* support, in the form of methods and techniques, which, ideally, takes an application developer from early design all the way to implementation in a seamless way.

ADL-d is a graphical design technique for the development of parallel and distributed software, based on a model of communicating processes. ADL-d's primary focus is on *communication modeling*, exemplified by a set of highly orthogonal communication concepts, jointly covering a wide range of patterns for communication. The orthogonalization of communication modeling can be seen as ADL-d's primary contribution to the field, and we believe that other recognized techniques such as SDL [1] could benefit from this.

Using a diagram technique called structure models, a designer models an application in terms of a communication graph of processes, hierarchically organized using simple process decomposition features. Through a second diagram technique called behavior models, the sequential behavior of each process is modeled in total isolation from its environment. Here, focus is again on *communicative* behavior, postponing implementation details to later development stages. The combination of both models gives a complete picture of the application's communication aspects.

ADL-d's solid, formal definition makes ADL-d designs suitable for automated target code generation. More specifically, an ADL-d design can be automatically translated into skeleton code which contains *all* necessary communication code. Only the details of strictly sequential code need to be explicitly coded.

Another feature of ADL-d is its ability to model dynamically changing communication structures in an application, giving it the possibility to adapt to changing demands in terms of speed, workload and robustness, or new opportunities in available resources during runtime.

2 Using ADL-d

Being a general technique for modeling communication structures, ADL-d is not necessarily committed to one particular design *method*. It goes well with traditional functional decomposition, but also with object-identifying strategies. Also, its simple decomposition techniques make it suitable for bottom-up as well as top-down approaches. For example, when using a top-down design method, a developer starts with a set of process objects which are the result of a requirements analysis (according to some unspecified strategy). From then on, ADL-d's structuring technique is used in the decomposition of these processes, build-

ing a structure design with fully specified communication semantics. Also, parent-child relations are established between processes. The primary notion during this stage is establishing a maximal degree of process decoupling. Here, ADL-d's communication model of *channels*, which explicitly model the medium between communication endpoints (process *gates*), is beneficial. Naturally, the refinement of communication structure may take several iterations to yield the desired result.

When a designer chooses not to decompose a process any further, the next refinement step is specifying its *behavioral semantics*, initially only communication, and later also internal computations. ADL-d provides a special type of state-transition diagrams (STDs) for this, which separate communications from (internal) computations. These STDs also include actions for initiating dynamic creation of process instances.

After all this, the specification of the process' behavior is complete and target code for the process can be automatically generated. All individual process codes together with the structure design suffice, in principle, to automatically generate target code for a working parallel, distributed program.

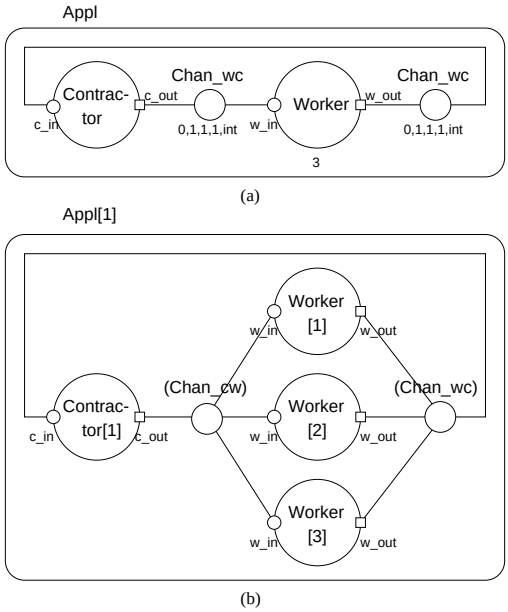

Figure 1: (a) example structure diagram; (b) Instance structure

3 The ADL-d Design Technique

3.1 Basic Components: Process, Channel, Gate

An ADL-d process is a prototype for a self-contained unit of functionality (much like an object class in object-oriented languages). A process' communication interface consists of *gates*, through which it sends or receives data. These, in turn, are attached to *channels* for data transfer between processes. The example structure diagram in Figure 1(a) models the pattern of an application Appl with a Contractor communicating data to Worker processes to perform some job and return the result. To that end, a Contractor's output gate c_out is attached to a channel Chan_cw, in turn attached to a Worker's input gate w_in. Similarly, there is a channel from the Workers to the Contractor. The integer annotation below Worker is the *replication* factor (the default value is one). In Figure 1(b) we see its effect: in an instance of Appl (Appl[1]), we will initially encounter one Contractor instance and three Worker instances. Its channels are instantiated only once.

Channels and gates separate independent communication concepts. Gates take care of the behavior part, which concerns *blocking*. To this end, each gate has a timer which is set by its owner on activation, indicating the time that the owner is willing to block for communication. Consequently, a communicating process now has a simple view on communication through a gate: either it succeeds within the specified time, or it fails.

Channels provide for the transfer of data between activated output and input gates. By attaching a gate to a channel, we indicate that it can communicate over that channel. A channel in complex process P_super is represented by a circle, and annotated with its buffer capacity, its distribution semantics, its receiver synchronization semantics, and the datatype of the messages it transports, in that order. These are all independent communication aspects.

Data Typing The data type annotation contains the name of a data type. The default data type is token, which is used for mere signaling purposes. In Figure 1(a), we see that Chan_cw transports integers.

Buffer Capacity If a channel has zero buffer capacity, communication can take place only if there is a sender and sufficiently many receivers active on it to let communication proceed. The term used for these channels is *synchronous*. If a channel has buffering capacity, communication for output gates can succeed if the buffer is not full, and for input gates if the buffer is not empty. With this, we model *asynchronous* communication. In Figure 1, both Chan_cw and Chan_wc are synchronous.

Distribution Semantics To specify how a channel distributes messages, we provide it with two parameters *min* and *max*, indicating the minimum and maximum number of receivers to which the message should be delivered. Both

numbers can be specified absolutely or as a percentage of the total number of potential receivers. For example, *min* = *max* = 1 means that communication of a message is completed if and only if it could be transferred to one input gate (unicast). Likewise, *min* = 1 and *max* = 100% requires transmission to at least one input gate (a form of multicast). A channel with *min* = *max* = 100% requires transmission to 'all' attached input gates (broadcast). In Figure 1(a), both Chan_cw as Chan_wc are unicast channels.

Receiver Synchronization Semantics The last channel parameter is *stride*. If there is a message *m* available on a channel *c*, which can still be received by *stride* receivers (remember the *max* parameter), and if there are *stride* opened input gates on *c*, which have not yet received *m*, *m* should be transferred to all of them at the same time. Like *min* and *max*, *stride* can be given in absolute and relative values.

Remarks ADL-d channels are nondeterministic in the sense that no assumptions can be made by a designer about the order in which input or output gates are served that are active at any particular moment.

Also, *all* channels are order preserving: all input gates attached to a channel receive messages in the same order. Finally, an ADL-d channel delivers messages in the order of acceptance. Hence, an ADL-d channel acts as a queue, and multicasting is totally ordered.

3.2 Process Decomposition

For hierarchical development, ADL-d includes a process decomposition technique. Decomposition stops with the specification of the dynamic behavior of the lowest level processes. When a process is decomposed, its interface, i.e. its set of gates, is left unaffected. In other words, decomposition proceeds independently of other parts of the design.

Figure 2 illustrates how the Worker process from Figure 1 is decomposed into one Analyzer and one Calculator process, communicating over two channels Chan_ac and Chan_ca. Furthermore, gate a_in_1 of Analyzer is associated with gate w_in of the original Worker process. This means that any data that was originally sent through gate w_in, is now sent through gate a_in_1. Likewise, gate a_out_1 has been associated with gate w_out.

3.3 Behavior Modeling

ADL-d uses separate state-transition diagrams to model the sequential behavior of nondecomposed (simple) processes. Each instance of a process prototype behaves according to the STD of that prototype. In STDs, emphasis is put on communicative behavior by means of *communication states*,

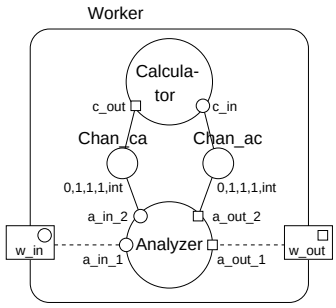

Figure 2: Example decomposition

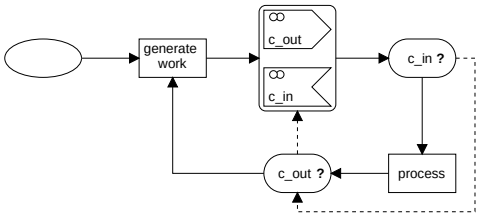

Figure 3: Example behavior model

which correspond directly to one or more of the process' gates. Strictly sequential behavior is aggregated into *computation states*.

Figure 3 contains the STD of the Contractor process. From its initial state (single ellipsis), it enters a computation state generate work (rectangular box). Then it simultaneously opens gates c_out and c_in in a so-called parallel blocking state. When an event occurs on either of the gates (timeout or success), this state is left. The Contractor then tests gate c_in in a test state (rounded box). If communication succeeded for c_in, the solid transition fires to processing the received input and testing c_out. Otherwise, the dashed transition fires to immediately testing c_out. If the generated work was not sent yet (c_out failed), the failure transition fires to a new communication attempt (dashed line). Otherwise, new work is generated. Notice that the timer values for the communication attempts in this example are set to infinity, implying that execution simply blocks until communication over at least one gate has succeeded.

3.4 High-Level Communication

Connection-oriented communication is becoming increasingly important in distributed computing: client-server computing and data streams can be naturally modeled as short and long-lived connections, respectively, between individual processes. Since the basic communication channels of ADL-d offer only unidirectional message transfer/distribution on a per-message basis, ADL-d requires ad-

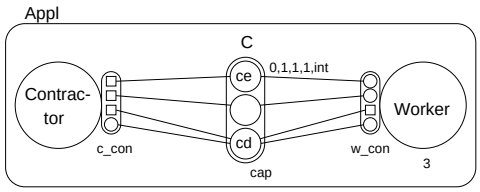

Figure 4: Connection gates attached to a connection channel

ditional modeling notions to capture these connections, which it provides in the form of connection channels and gates.

ADL-d considers connections special cases of group communication: among a number of potential participants in communication, a group is formed, whose members can subsequently engage in communication. With client-server communication for example, we start with a number of potential clients and servers, among which groups of two members (one client and one server) are established for exclusive interaction. Also, ADL-d considers group establishment a side effect of communication. The initiator, which is also the first group member, sends a message, and each participant that receives it becomes another group member. These two observations form the basis for connection modeling.

Representation A connection channel is an aggregation of basic ADL-d channels, denoted as subchannels. The three possible subchannel types are: connection establishment, disconnection, and regular data channels (which we have covered above). In Figure 4, we remodel the communication between a Contractor and a Worker to include a connection channel C, represented by a rounded box with subchannels, at least one of which is a connection establishment channel (ce-channel). All subchannels have the usual five parameters (we show only the ones of the connection establishment channel), whereas the connection channel as a whole has a capacity, indicating the maximum number of connections that can be supported at any given time. Processes are attached to connection channels through connection gates, which are aggregations of regular communication gates.

Semantics The establishment of a connection in ADL-d is based on the dynamic creation of subchannel instances that become dedicated to a group of gates. Because of space limitations, we illustrate connection semantics through a simple example.

If the design is as in Figure 4, then the initial instance structure is as in Figure 5(a), with only the ce-channel instantiated. If the Contractor sends a message over the ce-channel, received by Worker[2], then, as a side effect, the other subchannels of C are instantiated, and some new attachments and detachments are made, as illustrated in Figure 5(b), making the Contractor and Worker[2] the exclusive users of the subchannel instances. From then on until disconnection (see below), they are 'connected'.

Disconnection is triggered by communication over a disconnection subchannel. Through a successful send by the Contractor over the disconnection subchannel, it becomes reattached to the ce-channel (for making new connections) and detached from the others. If subsequently the Worker receives the disconnection message, the situation of Figure 5(a) is restored.

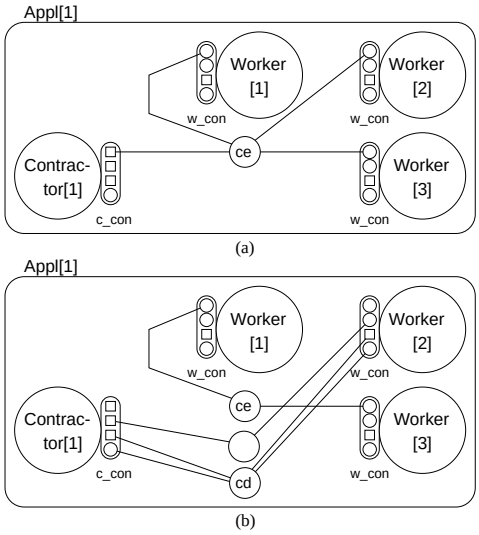

Figure 5: Connection establishment

Remarks A couple of remarks complete the discussion on connection channels. First, after one of the parties in a connection disconnects, the other party is still attached to the data-channel from the connection. This can be useful if, for example, we want all the data in its buffer to be transferred before (passive) disconnection.

Second, the entire scheme of connections and passive and active disconnections is very much reminiscent of the way TCP connections are set up and terminated. This gives us confidence in the efficiency of possible implementations of connections on parallel, distributed systems. Furthermore, RFC1379 proposes a TCP based protocol specially for short-lived request-response connections. These could also be used for implementation.

Third, modeling connection establishment as a side effect of normal communication allows for many different group patterns. For example, giving a connection establishment channel a *stride* of two always gives us an even number of

additional group members (other than the initiator), which can be useful if, for example, we want every member to have another member as backup.

Finally, the connection establishment channel and the disconnection channel both transport normal messages. This means that a short-lived connection between two parties can be modeled using a connection channel with only a connection establishment and a disconnection subchannel.

Behavior Modeling Aspects A major advantage of modeling connections the way we described above, is that no additional notations or semantics are needed in the behavior models of processes involved in such communication. Since the subgates of connection gates are basic ADL-d gates, we can use basic ADL-d blocking and test states to use them.

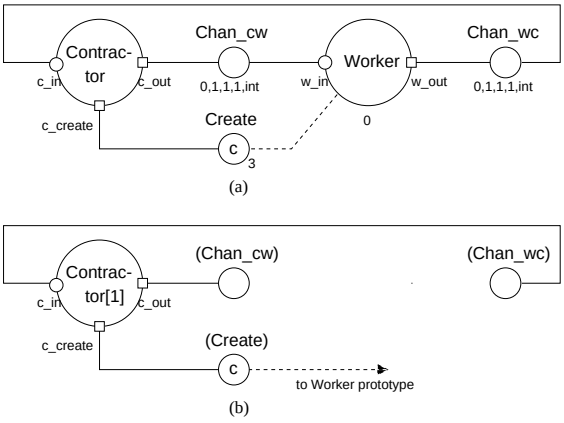

Figure 6: (a) ADL-d creation channel; (b) Situation immediately after startup

4 Dynamic Creation and Replication

An application is generally much more dynamic than reflected by its design. In particular, processes are dynamically created and destroyed, and likewise, communications are dynamically set up and broken down again. In ADL-d, we have chosen for an approach to dynamic creation that renders a clear view on (1) parent-child relations between processes, (2) communication graph evolvement, and (3) process behavior aspects of dynamic creation. On creation of a (complex) process instance, an instantiation of its entire internal structure is recursively created. By default, gates to the outside are then attached to channels exactly conform the modeled structure.

A runtime creation is triggered by a process instance, communicating over an output gate that is attached to a *creation* channel. The channel, which is associated with a (complex) process prototype, will then attempt to create new instances of this prototype, as many as its *num* parameter specifies. Success or failure of this attempt determines success or failure of the communication over the output gate. Consequently, sending a creation message looks to the sender just like a regular communication attempt.

In Figure 6(a), we remodel the communication structure among a Contractor and its Workers to include a creation channel Create with *num* value three. Also, we set the replication factor of Worker to zero. This means, that at application startup, no Worker instance is active (Figure 6(b)). After communication by a Contractor process over Create, three Worker instances are created, rendering the old structure of Figure 1(b).

The standard attachment algorithm for gates of new instances, such as illustrated in Figure 6, does not allow for the modeling of dynamic creation of structures such as pipes, trees and grids. For this reason, we have extended ADL-d with a small Prolog-like construction language, wit instructions for, for example, the creation of a process instance, and the attachment of a gate to a channel. A designer can associate a program in this language with a creation channel in order to override the default. Again, space limitations prohibit a full description of the construction language.

5 Implementation

ADL-d's orthogonal design and formal basis ([2]) allow for easy automated code generation, as was already exemplified by earlier ADL versions. We have recently finished a small, but fairly complete runtime system (RTS), written in C++ for a cluster of workstations. It includes object classes for every ADL-d concept, which can be dynamically instantiated and attached to each other. Communication is implemented using fairly low-level communication primitives to gain efficiency. The dynamic construction module offers a Prolog interface, and communicates to the rest of the RTS through messages.

At present, most channel implementations are centralized, but in [3] we have shown that decentralization is feasible. Also, we are investigating optimization strategies. Translation of ADL-d designs to skeleton code that interfaces to the RTS has shown to be almost trivial.

6 Related Work

Any design technique should provide a clear and unambiguous view on the structure and dynamics of the final program. For parallel and distributed programs, the two basic components of a model are its unit of execution and its communication model. 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. As for the communication model, basically there are two paradigms: shared data and message passing. The shared data approach has proven to be relatively easy to work with. Unfortunately, implementing the model on distributed-memory systems is not easy. Despite much research in this area [4], it seems that efficiency can only be achieved if we relax the memory coherence model as, for example, in TreadMarks [5].

The advantage of message passing is that it is directly supported by distributed-memory platforms such as networked workstations. However, a remaining drawback is the low level of abstraction of the message-passing model, requiring more effort from program developers. Problems seem to be alleviated if we use an object-based model, as objects naturally hide message-passing communication through method invocation. This alternative has been advocated for long by language designers, but how to actually incorporate parallelism and distribution into an object-based language is still a subject of much debate. For one, it can be argued that the synchronous method invocation reduces the degree of parallelism. But perhaps more important, is that traditional method invocation, being on the level of *instances*, may easily lead to an undesirably high degree of coupling between the application's components, and obfuscate its structure (see also [6]). Nevertheless, the feasibility of the approach has been demonstrated, for example, by Orca [7].

A model that allows for dynamic binding, such as advocated in the ODP standard ([8]), solves the problem of object coupling. The proposed strategy is to use an active binding object to bind objects that are ready to communicate, which allows object behavior to be modeled without any structural dependencies. In essence, ADL-d offers an abstraction of active binders in the form of channels, which become the carriers of most communication semantics.

In line with our approach to communication are the models used by some other design techniques, most notably SDL [1] and Darwin/Regis [9]. ADL-d distinguishes itself from the others by its rigid orthogonal approach, which is advantageous to many aspects, such as the modeling of dynamic creation, connection-oriented communication, and dynamic behavior.

7 Conclusion

In the complex design space of parallel and distributed software, three areas of major importance are communication structure, component behavior and structure dynamics. For a technique to significantly contribute to the development process, we advocate that it should provide support in all three areas *at least* to a level where all the specific problems of parallel and distributed software are solved. Furthermore, we advocate that, through separate techniques and notations, it should explicitly recognize the orthogonality of these three areas, such that they can be conquered separately. ADL-d has these characteristics, while maintaining a concise set of easy-to-use notations. Also, its communication model is devised to achieve versatility in the sense that ADL-d can be used in a broad spectrum of design *methods*.

References

[1] CCITT Z.100, "Specification and Description Language SDL," Recommendation Z.100, Mar. 1993.

[2] M. Polman, M.R. van van Steen, and A. de Bruin, "Formalizing a Design Technique for Distributed Programs," in *2nd International Workshop on Software Engineering for Parallel and Distributed Systems*. 1997, pp. 150–159, IEEE Computer Society Press, Los Alamitos CA.

[3] M.R. van Steen and M. Polman, "A Distributed Implementation of Many-to-Many Synchronous Channels," in *PCAT-94*, Wollongong, Australia, November 1994, pp. 218–227, IOS Press.

[4] J. Protić, M. Tomašević, and V. Milutinović, "Distributed Shared Memory: Concepts and Systems," *IEEE Parallel and Distributed Technology*, vol. 4, no. 2, pp. 63–79, Summer 1996.

[5] C. Amza, A.L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel, "TreadMarks: Shared Memory Computing on Networks of Workstations," *Computer*, vol. 29, no. 2, Feb. 1996.

[6] J. Kramer, J. MacGee, M. Sloman, N. Dulay, S. Cheung, S. Crane, and K. Twidle, "An Introduction to Distributed Programming in REX," in ESPRIT *'91*, November 1991, pp. 207–221.

[7] G.V. Wilson and H. Bal, "Using the Cowichan Problems to Assess the Usability of Orca," *IEEE Parallel and Distributed Technology*, vol. 4, no. 3, pp. 36–44, Fall 1996.

[8] ISO, "Open Distributed Processing Reference Model - Part 3: Architecture," International Standard ISO/IEC IS 10746-3, 1995.

[9] J. Magee, N. Dulay, and J. Kramer, "A Constructive Development Environment for Parallel and Distributed Programs," *IEE/IOP/BCS Distributed Systems Engineering*, vol. 1, no. 5, pp. 304–312, September 1994.