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

1 Introduction

Intensive development in high-speed networks, workstations, and accompanying system software has rendered distributed systems, particularly clusters of workstations (COWs), a sufficiently stable environment for use as a platform for running parallel and distributed software. With these cheap and easy-to-build systems at hand, using the many advantages of parallelism becomes a realistic option for many. Still, in order to produce a correctly functioning and efficient distributed application, a developer has to face some very specific difficulties with respect to synchronization, distribution and replication, which she does not encounter in sequential-software development. 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 providing basic message-passing primitives, are of little help here, 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 provides the techniques to systematically develop an application's (communication) structure as well as the sequential behavior of its constituents. Within this framework, 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.

Using a diagram technique called structure models, a designer models her application in terms of a communication graph of processes, hierarchically organized using simple process decomposition features. Our approach to communication modeling decouples communicating processes entirely in the sense that a complete abstraction from internal process behavior can be maintained throughout structure design.

Conversely, using 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 an application's structure model and the behavior models of its processes gives a complete picture of all its communication aspects.

Unlike many other techniques, ADL-d has a solid, formal definition for every symbol used in its models. As a consequence, ADL-d designs are suitable for automated target code generation. More specifically, an ADL-d design can be automatically translated into skeleton code which contains *all*

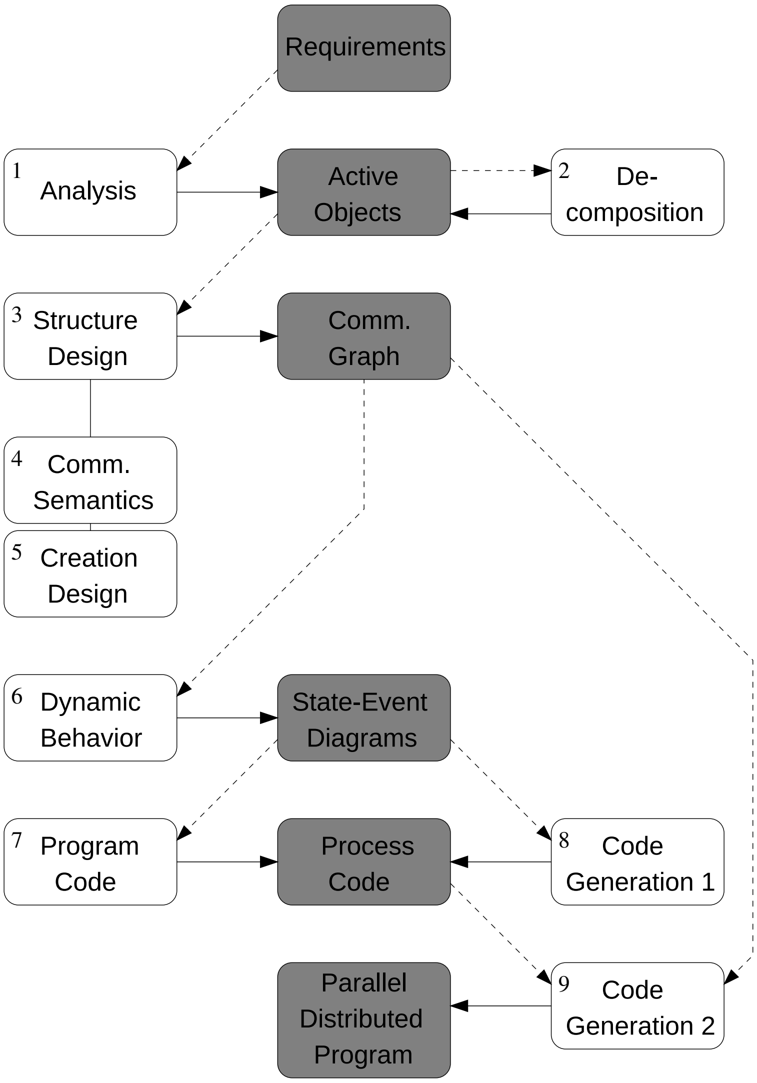


Figure 1: Using ADL-d *in an object-based method*

necessary communication code. A developer need only fill in the details of strictly sequential code, which is then automatically integrated with the skeleton code, resulting in an executable parallel, distributed application.

An important new 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. Surprisingly, this aspect of design for distributed systems seems to get little attention in comparable techniques.

2 Using ADL-d

Being a general technique to model 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, such as in [1, 2]. Also, its simple decomposition techniques make it suitable for bottom-up as well as top-down approaches.

In Figure 1, we have illustrated how a designer can use ADL-d in a top-down application development method. We assume that during the problem analysis phase (1), she has extracted some initial set of process objects from the functional requirements. From then on, she uses ADL-d's structuring technique in the decomposition (2) of these processes, building a structure design (3) with fully specified communication semantics (4). Also, she establishes parent-child relations between processes in the design (5). The primary notion during this stage is establishing a maximal degree of decoupling between the processes that are identified. This is achieved by using *channels*, which explicitly model the medium between communication endpoints (*gates*, in ADL-d) of processes.

Naturally, the above process is one of continuous refinement, which may take several iterations to yield the desired result. Here, a designer is not required (at least, not by ADL-d) to take her entire design from one refinement stage into another and onwards. A depth-first descent in the process hierarchy is an alternative, as well as any scheme in between, dependent on the demands of the design method that is followed.

The above items were all related to building the application's *structure*. When a designer chooses not to decompose a process any further, the next refinement step is specifying its *behavioral semantics*, initially only with regard to communication (6). ADL-d provides a special type of state-transition diagrams (STDs) for this, which separate communication and computation states. Our STDs also include actions for initiating dynamic creation of process instances.

The final refinement step in developing a process is the adding of implementation details that do not concern communication (7). More specifically, during this step, the computation states of its STD are filled in. The specification of the process behavior is then complete and target code for the process can be automatically generated (8). All individual process codes together with the structure design suffice, in principle, to automatically generate target code for a working parallel, distributed program (9), including dynamic creation of application components.

3 The ADL-d Design Technique

ADL-d's components are the result of careful analysis of the parallel, distributed design space. By making orthogonal phenomena and other independencies explicit, we have been able to keep our technique concise and manageable. This approach has led to a complete separation of communication structures and component behavior (see also [3]). Likewise, other issues are modeled separately in ADL-d as well, such as blocking semantics, various aspects of synchronization, data typing, and dynamic creation.

Basic Components

For an application's communication structure, ADL-d uses only three basic notions: processes, channels, and gates. A process is a prototype for a self-contained unit of functionality (much like an object class in object-oriented languages), recognized as such by the designer, possibly during problem analysis. A process' communication interface consists of a set of *gates*, through which it sends or receives data. These, in turn, are attached to *channels* which are responsible for data transfer between processes. Figure 2 shows the graphical representations of these concepts in a so-called structure diagram. The structure in Figure 2(a) models the typical communication pattern of 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, which is in turn attached to a Worker's input gate w_in. Similarly, there is a channel from the Workers to the Contractor. Notice that below a process, an integer

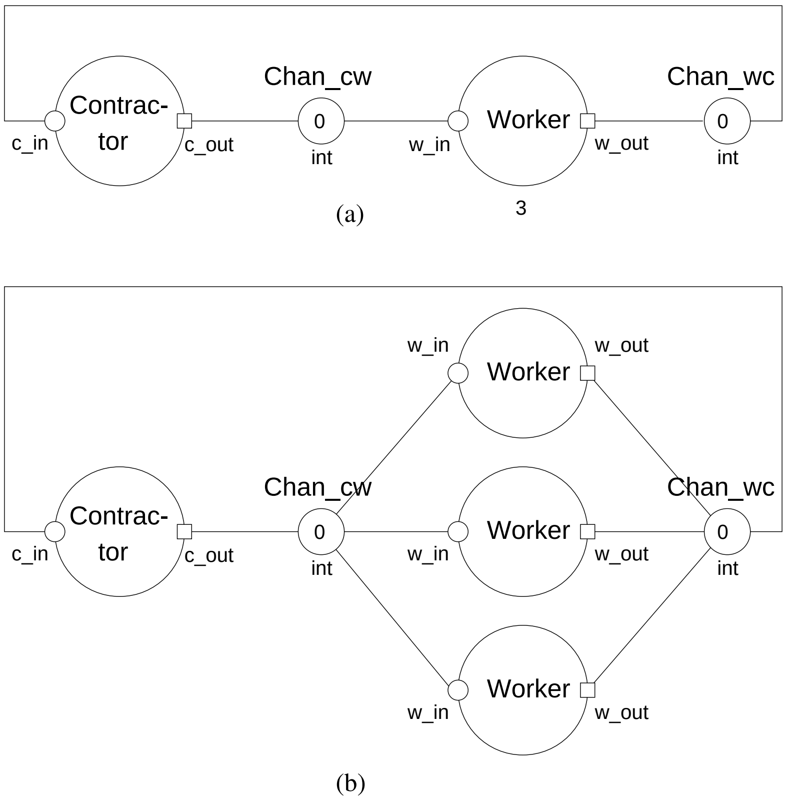


Figure 2: (a) example structure diagram; (b) effect of replication factor

can be given, representing the *replication* factor, which indicates how many times a process is initially replicated in an actual instance of the application. The default value is one. In Figure 2(b) we see the effect of such a replication factor. In a running application, we will initially encounter one Contractor instance and three Worker instances.

By using channels and gates, we can separate independent communication concepts. First of all, gates take care of the process behavior part of communication, which concerns *blocking*. To this end, each gate has a timer which is set when it is activated by its owner, indicating the amount of time that the owner is willing to block for communication to take place. The timer can be set to any non-negative value, including infinity. Consequently, a communicating process now has a simple view on communication through a gate: either it succeeds within the specified time, or it fails.

Data transfer semantics is wholly captured by channels. Each channel has an associated data type that prescribes the type of messages it can transfer. The capacity of a channel specifies how many messages it can buffer before delivery. In the case of a capacity of zero, we have fully synchronous communication. A channel accepts a message from an active output gate only if its capacity is not exceeded, or if it can immediately deliver the message. With multicast channels, the specified multicast type determines whether a message is delivered simultaneously to all processes that are attached to the channel, or one-by-one, as soon as a process is prepared to accept the message. These types are called *postmedium* and *premedium* multicast, respectively. An ADL-d channel always delivers messages in the order that they were accepted, and a next message is delivered only after the delivery of the previous one has completed. In other words, a channel acts as queue, and multicasting is totally ordered [4]. It should be noted that ADL-d channels are non-deterministic in the sense that if multiple senders are active on the same channel, we can make no assumptions as to who will be served first. A similar reasoning goes for receivers.

Process Decomposition

Hierarchical development allows a designer to look at her design at different levels of abstraction. For this, ADL-d includes a simple decomposition technique. A process can be recursively decomposed into a collection of communicating processes. 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, a developer decomposes a process completely independent of other parts of the design.

Figure 3 illustrates ADL-d decomposition. In Figure 3(a), we have taken the Worker process from Figure 2 and decomposed it into one Analyzer and one Calculator process, communicating over two channels Chan_ac and Chan_ca. Furthermore, we have associated gate a_in_1 of Analyzer 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. From a semantical point of view, Figure 3(b) shows a design that is equivalent to that of Figure 2, but now taking the decomposition and replication of the Worker process into account.

Behavior Modeling

As a final refinement step, ADL-d uses state-transition diagrams to model the sequential behavior of simple, i.e. non-decomposed processes. In a running application, each instance of a process prototype behaves according to the behavior model of that prototype. In the behavior model, the emphasis is put on communicative behavior by means of *communication states*, which correspond directly to one or more of the process' gates. Strictly sequential behavior is aggregated into one or more *computation states*. We illustrate the dynamic behavior of an Analyzer process in Figure 4. From its initial state (single ellipsis), it proceeds by blocking (the timer value for the gate is set to ∞) for input through gate a_in_1. It then analyzes incoming data and sends the result through a_out_2. In case of failure, the Analyzer terminates (double ellipsis). In case of success, it blocks for input data through gate a_in_2 which it then forwards through a_out_1. After that, the Analyzer returns to blocking for input through a_in_1.

High-Level Communication

Connection-oriented communication is becoming increasingly important in distributed computing [5]. For example, client–server computing is essentially based on a short-lived connection between two processes where the requesting client holds the connection until a reply has been sent back. Likewise, data streams are naturally modeled as an iteration of data transfers over a single connection. The basic communication channels of ADL-d offer only unidirectional communication on a per-message basis, without, as was explained before, designer control over the direction of messages in the case of multiple senders and receivers. As a consequence, ADL-d requires additional modeling notions to capture connections, which it provides in the form of connection channels, and connection gates.

A connection channel is in essence an aggregation of basic ADL-d channels, which we denote as subchannels. Two subchannels are used to control the connection, i.e. its setup and release. The others are used for the actual message transfer. A process can use a connection channel only by passing data through a connection gate that is attached to that channel. A connection gate is an aggregation of basic

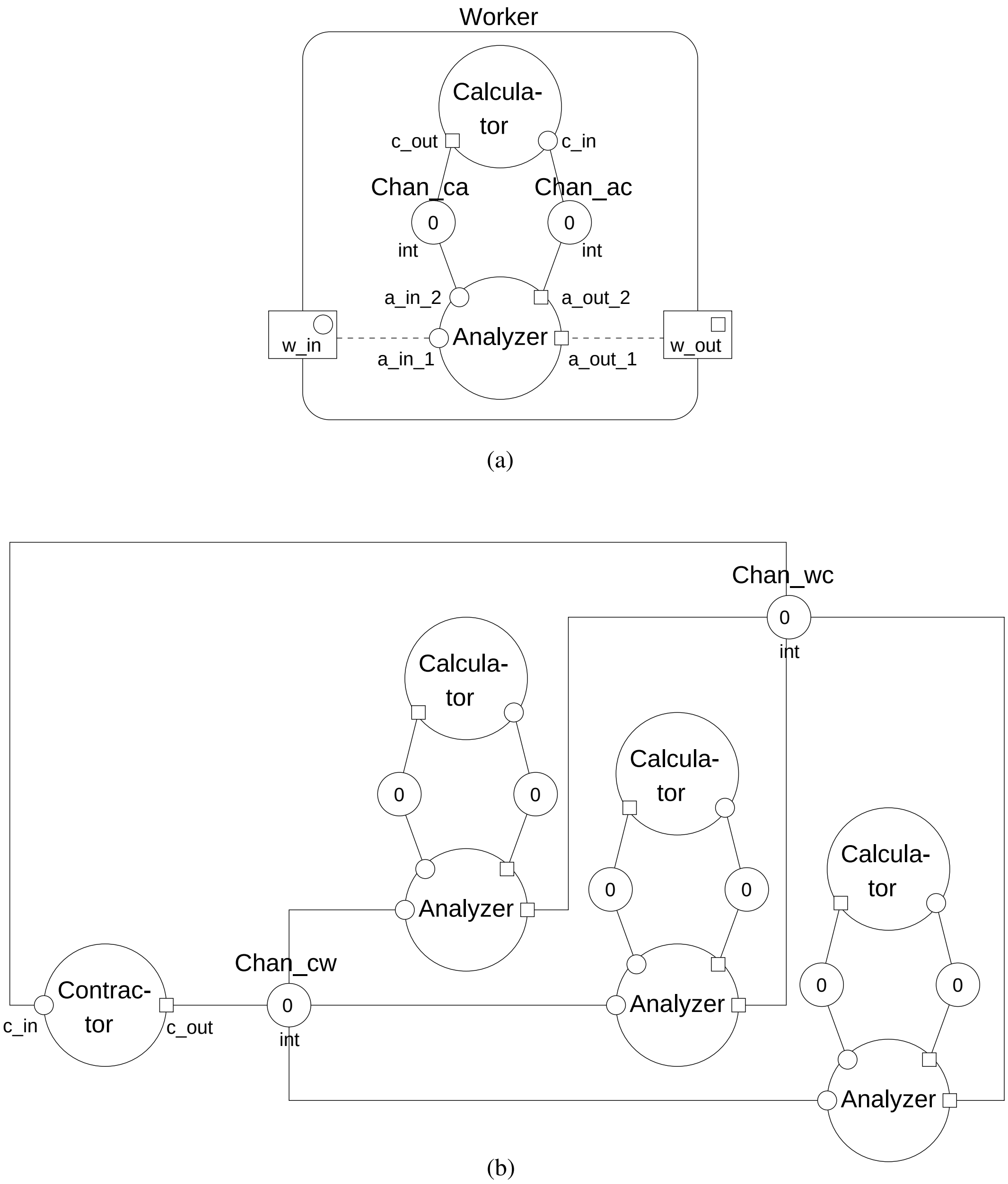


Figure 3: (a) Example decomposition, and its expanded, semantical equivalent (b).

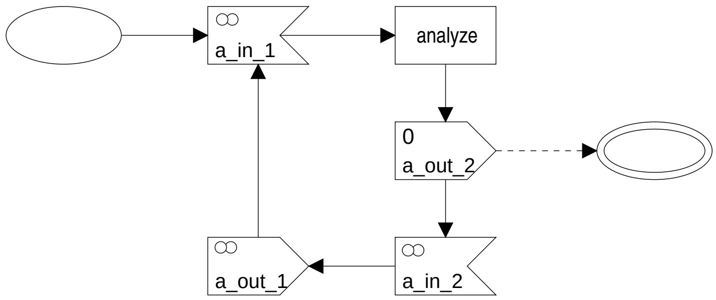


Figure 4: Example behavior model

gates. One gate is used to send or receive a connection request; two gates are used for tearing down the connection; and the others are used for transferring data during a connection. These concepts are illustrated in Figure 5.

We have modeled the situation that a Contractor process can connect to one of the three replicated Worker processes for bidirectional communication. To this end, the connection channel Chan_cw includes two subchannels data_1 and data_2, shown in Figure 5(b). By sending a request through the connect subchannel of Chan_cw, the Contractor issues a connection request. As soon as one of the Worker processes has accepted, a connection between the two processes is set up, involving a data_1 channel, a data_2 channel, and the appropriate subgates of c_con and w_con, as illustrated in Figure 5(c). From this moment on until disconnecting, bidirectional communication is possible.

Releasing the connection is done by sending a request through the fourth subchannel, named disconnect. Either of the two processes involved in an active connection can request the release of that connection. In contrast, setting up a connection is supported in an asymmetrical client–server fashion. This is to avoid complicated setup semantics in the case of simultaneous connection requests.

The integer annotation to a connection channel indicates how many connections between instances on one side, and instances on the other can be maintained simultaneously.

Note that our approach of using basic ADL-d gates for connection establishment and release implies that no extra notations are needed in the ADL-d STDs where connections are concerned. The old communication states can be used here. Note also, that although we now have support for connections, we have still maintained a clean separation between structure and behavior modeling.

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. Traditional design techniques hardly provide any support for modeling this kind of dynamic behavior. We advocate that a design technique for parallel and distributed applications cannot lack such support.

In ADL-d, we have chosen for an approach to dynamic creation that renders a clear view on:

- parent-child relations between processes
- communication graph evolvement
- process behavior aspects of dynamic creation

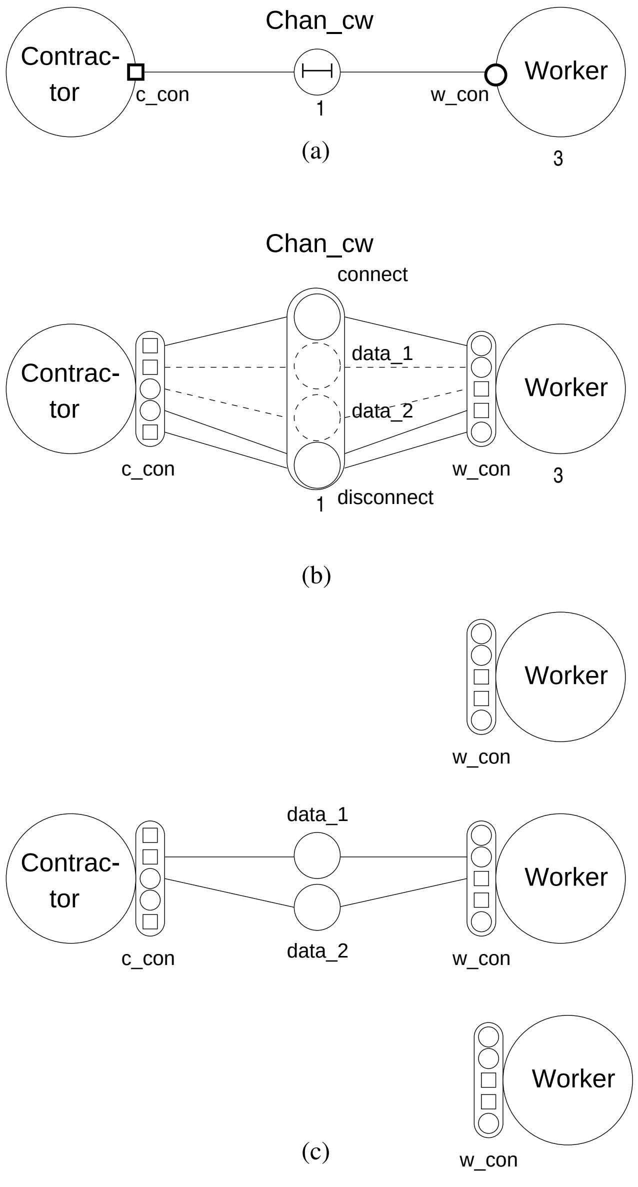


Figure 5: (a) ADL*-d connection channel; (b) Internal structure with individual connections; (c) Example runtime situation*

First of all, ADL-d supports dynamic creation of simple as well as complex processes. On creation of a 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 (cf. step three in Figure 1). In this way, the evolvement of the communication graph during runtime remains comprehensible to the designer.

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 a new instance of this prototype. 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. Also, we set the replication factor of Worker to zero. This means, that at application startup, no Worker instance is active (Figure 6(b)). Only after communication by a Contractor process over Create, will a Worker instance be created, and attached according to this structure model (Figure 6(c)). Notice that again, from the creator's point of view, communication is modeled by a normal output gate. Hence, we do not need any new concepts in our behavior model notation to incorporate dynamic creation.

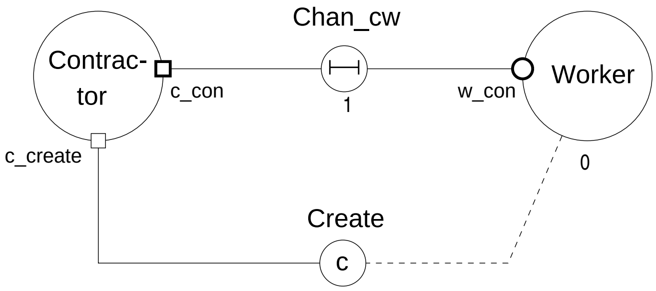
The standard attachment algorithm for gates of new instances, such as illustrated in Figure 6, always follows the gate attachments as modeled for their prototypes. This is well-suited for a communication pattern with Contractors and Workers, but it does not allow for the modeling of dynamic creation of complex communication structures such as pipes, trees and grids, which are other frequently encountered patterns in distributed computing. Incorporating a graph rewrite grammar into ADL-d would solve this problem, were it not for the fact that these grammars need significant skills to use them, which is not in line with the ADL-d spirit. For this reason, we have extended ADL-d with a small construction language, in which instance structures can be built in a series of simple steps, e.g. the creation of a channel or 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 attachment scheme, e.g. to create a pipe of process instances and channels. A full description of the language is beyond the scope of this article.

5 Implementation

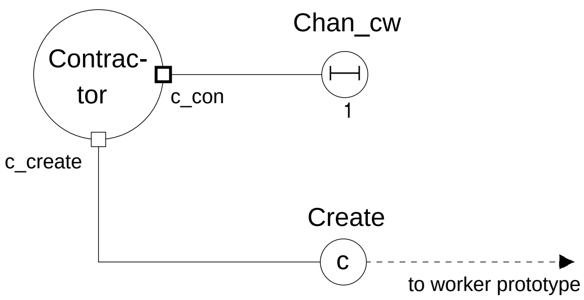
Automated code generation has been an important objective throughout the development of ADL-d. To achieve this, we have introduced orthogonality as much as possible and made sure that every ADL-d concept is formally defined (the formal semantics has not been further discussed in this article). As a result, code generation showed to be easy to accomplish, as was already exemplified by implementations of the first version of ADL. We have recently finished a small, but complete runtime system (RTS) that again indicates the feasibility of our approach. The RTS is written in C++, and runs on 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 module available to a designer to express the dynamic creation of processes and channels, offers a Prolog interface. The actual dynamic creation is established by passing messages to the rest of the RTS.

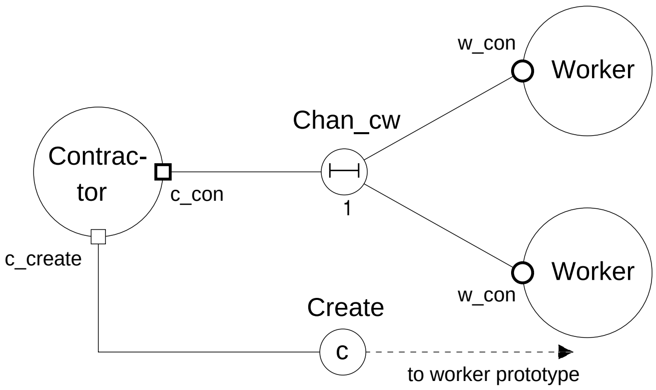
Our implementation experiences so far have indicated that we are on the right track. However, more research is needed, especially concerning the implementation of channel semantics. At present,



(a)



(b)



(c)

Figure 6: (a) ADL-*d creation channel; (b) Situation immediately after startup; (c) Situation after two creations by Contractor*

most solutions use a centralized component, but in [6] we have shown that fully decentralized alternatives are feasible as well. In addition, we intend to optimize solutions that cover the most common cases, and provide only general solutions to cover the remaining cases. In this way, we aim to improve the efficiency of the current RTS. 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.

The communication model prescribes how processes interact. Basically, there are two paradigms: communication is based either on shared data, or on message passing. The shared data approach is attractive because the model has proven to be relatively easy to work with. Unfortunately, implementing the model on distributed-memory systems is not easy. Much research has been conducted in the field of distributed shared memory systems [7], but it seems that efficiency can only be achieved if we relax the memory coherence model. This latter approach has been followed, for example, by TreadMarks [8] and CRL [9]. Using weak coherence models puts an additional burden on the programmer. In effect, we are relaxing the semantics of the shared memory model in favor of efficiency.

An alternative is to use message passing. The advantage of this model 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. In other words, compatibility is attained at the cost of development effort.

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 (see also [10, 11]). 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 [12]), which is generally considered bad software engineering practice. Nevertheless, the feasibility of the approach has been demonstrated, for example, by Mentat [13] and Orca [14].

A model that allows for dynamic binding, such as advocated in the ODP standard ([15]), 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 Parse [16], but also Regis [17], and PAR-SDL [18]. The fact that these techniques were developed independently is an indication that we are gradually reaching a consensus about communication modeling in designs for parallel and distributed programs. 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. It also allows us to automatically generate efficient implementations from a design.

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] G. Booch, *Object-Oriented Design with Applications*, Benjamin/Cummings, Redwood City, Calif., 1991.

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

[3] J. Kramer, "Distributed Software Engineering," in *16th IEEE International Conf. Software Engineering*. 1994, pp. 253–263, IEEE Computer Society Press, Los Alamitos CA.

[4] L. Liang, S.T. Chanson, and G.W. Neufeld, "Process Groups and Group Communication: Classification and Requirements," *Computer*, vol. 23, no. 2, pp. 56–68, Feb. 1990.

[5] J.A. Stankovic, "Distributed Computing," in *Readings in Distributed Computing Systems*, T.L. Casavant and M. Singhal, Eds., pp. 6–30. IEEE Computer Society Press, Los Alamitos, CA., 1994.

[6] 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.

[7] 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.

[8] 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.

[9] K.L. Johnson, M.F. Kaashoek, and D.A. Wallach, "CRL: High-Performance All-Software Distributed Shared Memory," in *15th Symposium on Operating System Principles*, Copper Mountain, Colorado, Dec. 1995, ACM.

[10] B. Meyer, "Systematic Concurrent Object-Oriented Programming," *Communications of the ACM*, vol. 36, no. 9, pp. 56–80, September 1993.

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

[12] 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.

[13] A.S. Grimshaw, "Easy-to-use Object-Oriented Parallel Processing with Mentat," *Computer*, vol. 26, no. 5, pp. 39–51, May 1993.

[14] 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.

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

[16] I. Gorton, J. Gray, and I. Jelly, "Object-Based Modeling of Parallel Programs," *IEEE Parallel and Distributed Technology*, vol. 3, no. 2, pp. 52–63, Summer 1995.

[17] 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.

[18] C. Steigner, R. Joostema, and C. Groove, "PAR-SDL: Software Design and Implementation for Transputer Systems," in *Transputer Applications and Systems*, R. Grebe, J. Hektor, S. Hilton, M.R. Jane, and P.H. Welch, Eds., vol. 2. IOS Press, Amsterdam, 1993.