Formalizing A Design Technique For Distributed Programs

Mark Polman
Erasmus University, Rotterdam
polman@few.eur.nl

Maarten van Steen
Vrije Universiteit, Amsterdam
steen@cs.vu.nl

Arie de Bruin
Erasmus University, Rotterdam
adebruin@few.eur.nl

Abstract

ADL-d is a graphical design technique for parallel and distributed software, in which communication modeling plays a central part. Recently, we have used formal methods to define ADL-d's semantics. The original objective was to provide well-defined guidelines for future implementations of ADL-d's communication constructs, but, as it turned out, significant feedback resulted to the notation itself. We give an outline of the ADL-d notation and its intuitive semantics. Also, we introduce the formal semantics, and discuss what impact this formalization has had on the original notation.

1 Introduction

With the increasing popularity of off-the-shelf networked hardware as a platform for distributed computing, a renewed interest can be observed in design support for parallel and distributed software. We have developed ADL-d, a graphical distributed design technique, with ease of usage, elegance, completeness, and suitability for automated code generation as its main objectives.

ADL-d offers notations to model an application in terms of a process hierarchy, in which processes communicate through *gates* over *channels* by message passing. A wide range of communication constructs and facilities for modeling dynamic process creation belong to its features. In Section 2, we discuss the ADL-d technique, avoiding details (which can be found in [11, 9, 10]) to save space and to remain focused on the main theme (see below).

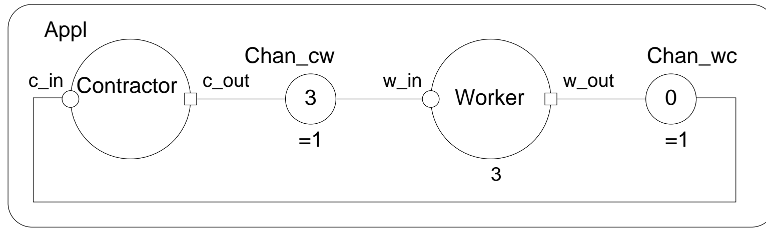
At a fairly advanced stage in ADL-d's development, we started using formal methods in describing the semantics of the ADL-d notations, primarily as a basis for automated code generation. As it turns out, the use of formal methods has helped in meeting all four of the aforementioned objectives. In Section 3, we give a brief introduction to the formal method we have used, an 'operational semantics' specified through a transition system. We then explain how ADL-d diagrams can be fitted into this method. In Section 4, we discuss the formal ADL-d semantics in greater detail, and explain how the use of formal methods has been helpful in further developing ADL-d, revealing unrealistic assumptions, unelegant constructs, and, very importantly, providing a model for an ADL-d runtime system implementation.

Section 5 discusses related work, whereas in Section 6, we briefly state our conclusions and discuss future research directions.

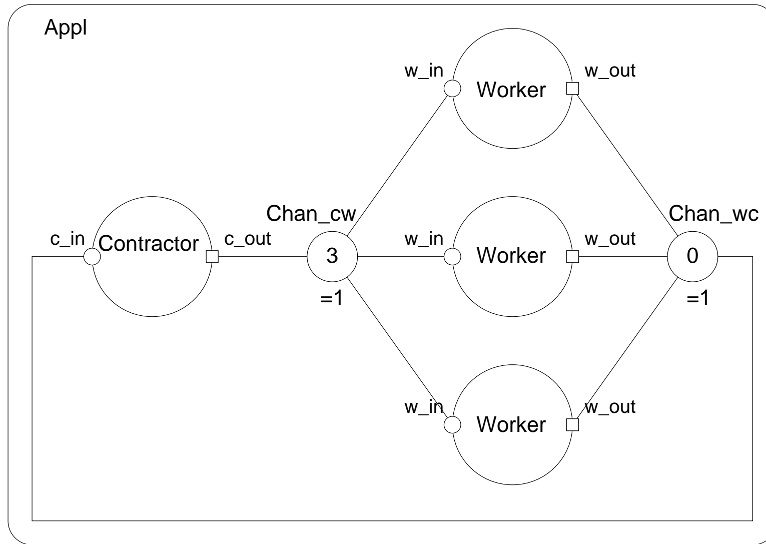
2 ADL-d

In ADL-d, an application is modeled as a process, which can consist of subprocesses, consisting of other subprocesses, etc. For each complex process, the internal structure is modeled through a set of subprocesses, communicating to each other through *gates* over *channels*.

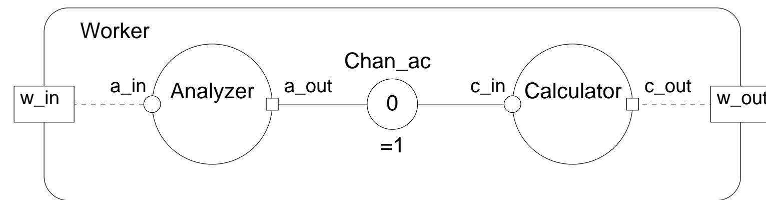
Whenever a process wants to send a message, it places the message in one of its *output* gates. The output gate is attached to a communication channel, which, according to some algorithm (see below), allows attached output gates to communicate to attached *input* gates. An input gate is opened by its owner, whenever the owner wants to receive a message through it. Notice that with opening an input (output) gate, nothing is specified about the intended sender (receiver). Channels are responsible for the distribution of messages. In Figure 1(a), the communication structure is modeled of an application Appl, describing a popular communication pattern. One process, the Contractor, is supposed to generate work and distribute it over Workers, which process the work and send the results back. To this end, Contractor communicates through output gate c_out (small square) over channel Chan_cw. On the receiving end of Chan_cw are three Worker processes, with input gates w_in.



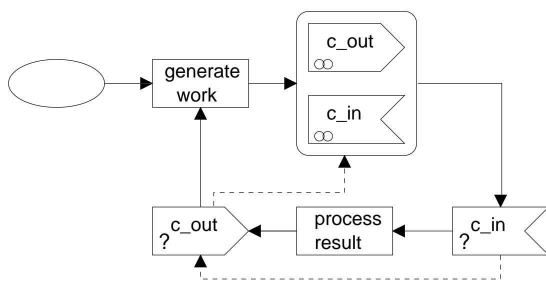
(a)



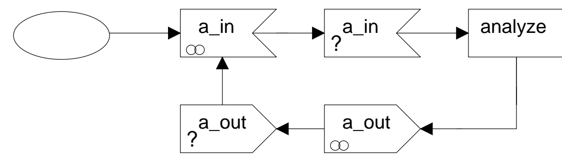
(b)



(c)



(d)



(e)

Figure 1. (a) Example ADL-d application structure; (b) Semantical equivalent of (a) with expansion; (c) Internal structure of Worker process; (d) State-transition diagram for Contractor process; (e) State-transition-diagram for Analyzer process

From within a Worker process, output can be sent back to Contractor through w_out over Chan_wc.

The integer below the Worker process indicates its initial multiplicity. An equivalent diagram is given in Figure 1(b).

Decomposition

In Figure 1(c), the internal structure of a Worker process is given. Apparently, Workers consist of an Analyzer and a Calculator, communicating over Chan_ac. The association with Figure 1(a) lies in the Worker gates: while on a higher abstraction level, only w_in and w_out are seen to be attached to Chan_cw and Chan_wc, respectively, we see that, in fact, a_in of Analyzer is attached to Chan_cw and c_out of Calculator is attached to Chan_wc.

Dynamic Behavior

On the lowest level of the decomposition hierarchy are the simple (i.e. nondecomposed) processes, that actually display dynamic behavior. In ADL-d, this behavior is modeled in state-transition diagrams (STDs) per simple process. The notations used in these diagrams are derived from those used in SDL [2], but have different semantics.

ADL-d distinguishes between *processing states* and *gate actions* in its STDs. In Figure 1(d), the dynamic behavior of the Contractor is given. Roughly, the Contractor executes a loop of generating work, sending it away and receiving the results back. From its initial state (ellipsis), it proceeds by entering the processing state generate work. After that, it *opens* c_out for sending the work and c_in for receiving the results, and *blocks* for an event on one of the two gates. If this has happened, it can check what it was by means of a *test*, which has two possible outcomes, success or timeout. First, c_in is tested. In case communication over c_in was successful, the solid transition is taken, and the message is processed in process result. Otherwise, the dashed transition fires. Finally, c_out is tested. If the message on c_out could actually be sent, the process starts all over. Otherwise, no new work is generated; instead, a new attempt is made to send the old work.

Figure 1(e) models the dynamic behavior of an Analyzer process, in which it executes a loop of receiving work through a_in, analyzing it, and forwarding it through a_out. Without actually giving the STD, we assume that a Calculator process performs a similar loop.

Intuitive Communication Semantics

The annotations to channels in Figure 1(a), (b) and (c), and to blocking actions in Figure 1(d) and (e) are used to establish the semantics of communication, which becomes increasingly important as a design moves on from logical to more technical phases.

The annotations within the channel symbols refer to the channel's buffering capacity. For example, the symbol 3 in Chan_cw indicates a buffer size of three messages. Roughly, a channel accepts a message from an open output gate, only if the buffer is not full, and copies a message from its buffer to an open input gate, only if its buffer is not empty. For the open gate, communication then succeeds. If a channel has zero buffering capacity (e.g. Chan_wc), it gets the semantics of a *synchronous* channel: communication succeeds only if input and output gates are open simultaneously.

Annotations below channel symbols refer to the *distribution* semantics of the channel, i.e. how many receivers each sent message must reach. In the figure, all channels have '$= 1$' as an annotation, meaning that communication is completed if and only if a message has been delivered to one receiver. However, other annotations are possible as well. In Figure 2 a few of them are listed. In case each message is

criterion	meaning
=1	exactly one receiver must be reached
>1	at least one receiver must be reached
=50%	fifty procent of all potential receivers must be reached
=100%	all potential receivers must be reached

Figure 2. Example success criteria

possibly intended for multiple receivers, a distinction can be made between *postmedium* and *premedium* replication semantics. In case of postmedium semantics, a sufficiently large number of attached input gates must be open simultaneously, before a message is transferred to all of them. With premedium semantics, open input gates can get the message one by one, until a sufficiently large number of them received it. The symbols for pre- and postmedium are given in Figure 3. Finally, in Figure 1(d) and (e), annotations are provided with blocking actions. These indicate the maximum amount of time that a process is prepared to block for communication through the corresponding gate to succeed (∞ in our example). When an event (success or timeout) occurs on at least one gate in a blocking state, the rest of the gates are retimed to zero, in order to end communication as soon as possible (i.e. to make them timeout). This way, after leaving a blocking state, every gate involved has either succeeded or failed.

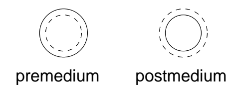


Figure 3. Pre- and postmedium

Other Features

ADL-d has a number of additional features, which are important from a software engineering point of view, but which are slightly beyond the scope of this paper (see also Section 3). They include the ability to model dynamic creation of both complex and simple processes, which is done by means of *creation* channels, and the ability to model connection-oriented communication, using which request-reply and other forms of directed communication can be captured. For this latter feature, so-called *connection* channels and gates are used. Details with regard to these concepts can be found in [11].

3 Operational Semantics

As was stated in the introduction, the use of formal methods to describe the semantics of ADL-d was primarily motivated by the need for rigid implementation guidelines. This implied that the formal description had to be on a level of abstraction that:

- hides implementation details, but
- is still easy to derive implementations from.

A first consequence of this is the choice for an 'operational semantics' as opposed to denotational semantics [1]. Using an operational semantics has the advantage over an actual implementation, that we are free to abstract from many implementation details that are of less importance, or that we simply do not want to fill in yet. Still, we can choose a level of abstraction that is still sufficiently close to 'reality' to reveal problems as well as opportunities for distributed implementation.

3.1 Basic Concepts

In our operational semantics, a distributed application is modeled as a set of objects, some of which execute a sequence of statements in some language. An object is described by a tuple (p, s_p, σ_p), where p is a unique object identifier, s_p (optional) is the list of statements still to be executed by p, and σ_p is a function of p's variables to values in some domain, i.e. σ_p defines p's *state*.

> As an example, a simple language for objects could be defined as follows:
>
> $$s \quad ::= \quad v := e \mid \textsf{skip} \mid \textsf{if } e \textsf{ then } s \textsf{ else } s \textsf{ fi} \mid \textsf{while } e \textsf{ do } s \textsf{ od} \mid (s; s) \mid E$$
>
> implying that a statement can be an assignment to a variable (v denotes a variable name and e denotes an expression), a skip, a conditional statement, a conditional loop, a sequence of statements, or the empty statement E.

A parallel, distributed application at some point in its execution can now be described as a set of tuples according to the above sketch. How the application executes from there is determined by a set of transition rules, which fire dependent on the state of the application (or a subset thereof), thereby bringing one or more objects into a different state and possibly a step further in the execution of their statements.

> As an example, consider the following transition rule:
>
> $$\{(p, \textsf{ if } e \textsf{ then } s_1 \textsf{ else } s_2 \textsf{ fi}, \sigma)\} \rightarrow \{(p, s_1, \sigma)\}$$
> $$\text{if } \mathcal{V}(e)(\sigma) = tt$$
>
> which tells us that if there is an object in the application set willing to execute a statement s_1, conditional on an expression e, and if, given the state of p, e evaluates to tt (=true), then this rule can fire, causing p to move on to executing s_1.

Concurrency in an application is, in fact, modeled through the possibility of multiple transitions (concerning different objects) being enabled at the same time.

> The following rule formalizes this:
>
> $$\frac{r_1 \rightarrow r_2}{r_1 \cup r \rightarrow r_2 \cup r}$$
>
> Translated into English this rule tells us, that if any subset (r_1) of objects in a running application matches the left side of a transition rule, resulting in r_2, then this rule can fire, leaving the rest of the objects (r) unaffected.

Roughly, a run of a parallel and distributed application, modeled in the above way, is determined by repeatedly selecting nondeterministically one of the enabled transitions. The set of all possible runs, generated in this way, can be seen as 'the semantics' of the application.

Obviously, a formalism as introduced above is most interesting when the objects *interact*, in other words, when there are transition rules involving tuples of more than one object. This is discussed later.

3.2 Translating Diagrams Into Object Structures

From Diagram Objects to Formal Objects

If we are going to translate ADL-d diagrams into objects in the formal notation introduced above, we have to ask ourselves which concepts in the ADL-d notation domain should return as concurrent objects (tuples) in the translation. A simple OMT-like ([12]) analysis of the ADL-d domain gives us the diagram of Figure 4. Obvious candidates for concurrent objects are channels and processes. However, from the informal descriptions of ADL-d, it appears that gates have a

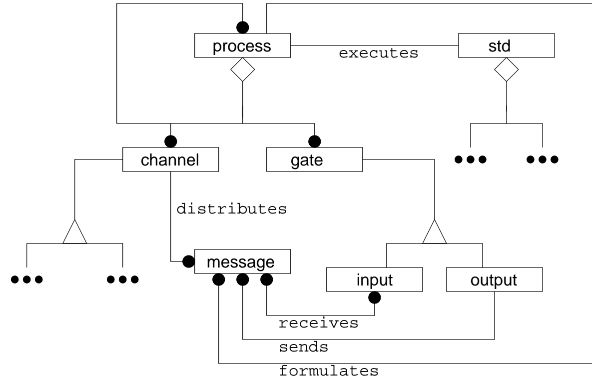


Figure 4. Simple OMT diagram of ADL-d domain

state as well (open, closed, succeeded, etc.), which can be modified by both processes and channels. For this reason, we model gates as tuples as well, despite the fact that they seem to be so closely linked to processes. Finally, we have also chosen to model messages as tuples. However, this was primarily driven by a search for elegance.

From STDs to Statements

The next thing to do is to provide the ADL-d process, channel, and gate tuples with statements, and after that with variables and state functions.

Obviously, the statement s_p in a process tuple (p, s_p, σ_p) should be a textual representation of p's STD. To this end, we adopt the language definition from the previous section for representing what can be done in processing states, and make extensions to represent gate actions:

$$
\begin{array}{lll}
\mathsf{Gset} & ::= & \epsilon \mid (g,t) \mid (g,e,t) \mid \\
 & & (g,t), \mathsf{Gset} \mid (g,e,t), \mathsf{Gset} \\
\mathsf{Block} & ::= & \mathsf{block}(\mathsf{Gset}) \\
\mathsf{Test} & ::= & \mathsf{test}(g,v)\ \mathsf{succ}\ s\ \mathsf{fail}\ s \mid \\
 & & \mathsf{test}(g)\ \mathsf{succ}\ s\ \mathsf{fail}\ s \\
s & ::= & v := e \mid \mathsf{skip} \mid \mathsf{if}\ e\ \mathsf{then}\ s\ \mathsf{else}\ s\ \mathsf{fi} \mid \\
 & & \mathsf{while}\ e\ \mathsf{do}\ s\ \mathsf{od} \mid (s;s) \mid \mathsf{Block} \mid \mathsf{Test}
\end{array}
$$

As can be seen, we extended the definition of s with block and test statements. The block statement has a sequence of tuples as its arguments, where each tuple consists of a gate identifier and a timer value, and, in case of an output gate, the expression to be sent. A test has a gate identifier and, in case of an input gate, a variable name as its arguments (for storing a received value). The success and timeout transitions from the STDs are given their textual representations as well.

For channels and gates (and also messages), we do not give statements in some language to express their behavior. The reason for this is that doing so would come too close to providing an actual implementation, which, as we stated above, was something we wanted to avoid. Instead, the semantics of communication are expressed by transition rules that transform the states of (sets of) gates and channels. This, in fact, resembles a declarative way of specifying things: for a transition rule to fire, channels and gates must be in state X; afterwards, they will be in state Y.

From Attachments and Annotations to Variables and States

The third entry in an object tuple is its state function, assigning a value to each of its variables. We distinguish between two types of variables: the ones that refer to other objects, and the ones that do not. The first group is used to define the structure of the application (the attachements from gates to channels, from gates to processes, etc.). For example, every gate has a variable $proc$, which, through the gate's state function, refers to the identifier of the process object that the gate belongs to. Likewise, it has a $chan$ variable pointing to the channel that it is attached to [1].

These and the other variables are given in Figure 5, where a process, a channel, a gate and a message are given as object classes with attributes. Notice that all annotations used for implementing the communication semantics, as discussed in the previous section, are represented here. For example, channels have a capacity variable, and gates have a timer (which is set in blocking actions). Notice also the $stat$ attribute of gates. The possible values represent $open$, $closed$, $succeeded$, and $failed$ for input gates and output gates, respectively.

[1] Actually, since ADL-d supports dynamic creation of processes, the system is more complicated than suggested here. More specifically, in real ADL-d, processes in ADL-d diagrams are process *prototypes*, which, through their structure variables, describe an application's structure on a meta level, such that every prototype and the structure between them can be *instantiated*. Here, we have abstracted from this to simplify matters.

PROCESS	
var	type
g1...gn	gate ID
blockset	set of gate ID
other vars	any

CHANNEL	
var	type
buf	list of mess ID
type	[post, pre]
cap	integer
crit	[=x, >x, =x%, >x%]

GATE	
var	type
mess	message ID
proc	process ID
chan	channel ID
stat	[○ ● ⊗ ⊖ □ ■ ⊠ ⊟]
timer	integer

MESSAGE	
var	type
sender	gate ID
recvrs	list of gate ID
val	any

Figure 5. Conceptual ADL-d objects

4 Transition Rules and Results

Now that we have a formal tuple representation of ADL-d diagrams, both structure models and STDs, we can describe ADL-d semantics through a set of transition rules as discussed above. In order to keep the system elegant and easy to derive implementations from, we strive to:

- keep the number of objects involved in any single transition rule as low as possible, and
- keep the total number of transitions as low as possible.

Keeping the number of different object tuples per transition rule low has the advantage of simplicity. But more important is that it is a success test for the modularity of the approach. The fewer objects per transition rule, the less information on various objects is needed for it to fire. This is clearly beneficial for a straightforward distributed implementation. Conversely, if we can find no such elegant way of describing the intended 'intuitive' semantics of a concept, e.g. postmedium broadcast, then this might be a reason to reconsider the intuitive semantics themselves. We will look at an example of this later.

Below, we first state and explain some of the transition rules, and use these to illustrate the subsequent discussion on the benefits of our formal method.

4.1 Gate Actions

Rule 1: Blocking on a Gate

We start off simple, with the opening of and blocking for an output gate:

$$\{(p,\mathsf{block}((g_{var},e,t),\mathsf{Gset}),\sigma_p),(g,\sigma_g)\} \rightarrow \{(p,\mathsf{block}(\mathsf{Gset}),\sigma'_p),(g,\sigma'_g),(m,\sigma_m)\}$$

if $\sigma_p(g_{var}) = g$ (1)
$\sigma_g(proc) = p$ (2)
$\sigma_g(stat) \in \{\blacksquare, \boxtimes, \boxminus\}$ (3)
where $\sigma'_p = \sigma_p\{(\sigma_p(blockset) \cup \{g\})/blockset\}$ (4)
$\sigma'_g = \sigma_g\{\square/stat,\ m/mess,\ \mathcal{V}(t)(\sigma_p)/timer\}$ (5)
$\sigma_m(val) = \alpha$ (6)
$\alpha = \mathcal{V}(e)(\sigma_p)$ (7)
$\sigma_m(sender) = g$ (8)

This rule tells us that if p blocks on one of its gate variables pointing to (1,2) output (3) gate g, trying to send α (7), then g is added to p's $blockset$ (4), and changes state to opened (5). Also, a new message object is created with val α (6), and $sender$ g (8).

Condition (4) should be read as follows: σ'_p equals the old σ, with the exception of the $blockset$ variable, which gets g added to it. A similar reading goes for condition (5).

The rule for blocking on an input gate is very similar, only with the value of the input gate's variable $stat$ changing to ○, and without generating a new message object. Also, reopening a succeeded gate has no effect. First, the delivered message has to be processed (see below).

Rule 2: Unblocking on a Gate

When an event has occurred on at least one of the gates in a process' $blockset$, then ADL-d semantics prescribe that an effort is made to close all still open gates as soon as possible (for reasons discussed below):

$$\{(p,\mathsf{block}(),\sigma_p)\} \cup \cup_{i=1}^{n}\{(g_i,\sigma'_{g_i})\} \rightarrow \{(p,\mathsf{block}(),\sigma'_p)\} \cup \cup_{i=1}^{n}\{(g_i,\sigma'_{g_i})\}$$

if $\forall i:\ \sigma_{g_i}(proc) = p$ (1)
$\exists j\ (1 \leq j \leq n):\ \sigma_{g_j}(stat) \notin \{\bigcirc, \square\}$ (2)
$\cup_{i=1}^{n}\{g_i\} = \sigma_p(blockset)$ (3)
where $\sigma'_p = \sigma_p\{(\sigma_p(blockset) \setminus \{g_j\})/blockset\}$ (4)
$\forall i\ (i \neq j):\ \sigma'_{g_i} = \sigma_{g_i}\{0/timer\}$ (5)

Translated to English this rule says that if for process p there is at least (2) one gate g_j in p's $blockset$ (3) that is not open anymore, then all g_i are retimed to zero (5), so that they can timeout quickly. Gate g_j is removed from p's $blockset$ (4).

After this transition, at some point in the future, for every gate in p's original $blockset$, we have that it has either suc-

ceeded or timed out. At that moment, another rule can fire causing p's block() statement to end, enabling p to continue with the next statement.

Rule 3: Testing

After going through a blocking phase, a process can, one by one, test the gates involved for success or timeout, and make the appropriate transition. The transition rule for successful input is as follows:

$$\{(p, \mathsf{test}(g_{var}, v)\ \mathsf{succ}\ s_1\ \mathsf{fail}\ s_2, \sigma_p), (g, \sigma_g), (m, \sigma_m)\} \rightarrow \{(p, s_1, \sigma'_p), (g, \sigma'_g), (m, \sigma_m)\}$$

if $\sigma_p(g_{var}) = g$ (1)
$\sigma_g(proc) = p$ (2)
$\sigma_g(stat) = \otimes$ (3)
$\sigma_g(mess) = m$ (4)
$\sigma_m(val) = \alpha$ (5)
where $\sigma'_p = \sigma_p\{\alpha/v\}$ (6)
$\sigma'_g = \sigma_g\{\bullet/stat\}$ (7)

As can be seen, if process p tests on its (1,2) succeeded (3) input gate g, which points to a message m (4) with value α (5), then p's variable v will get value α (6), and g is closed (7).

4.2 Channel Semantics

Transition rules for channels' distribution semantics all bring one or more attached gates into a different state. How many depends on the channel's success criteria and whether it is post- or premedium.

Rule 4: Sending on an Asynchronous Channel

$$\{(c, \sigma_c), (g, \sigma_g)\} \rightarrow \{(c, \sigma'_c), (g, \sigma'_g)\}$$

if $\sigma_g(chan) = c$ (1)
$\sigma_g(stat) = \square$ (2)
$size(\sigma_c(buf)) < \sigma_c(cap)$ (3)
where $\sigma'_g = \sigma_g\{\boxtimes/stat\}$ (4)
$\sigma'_c = \sigma_c\{append(\sigma_c(buf), \sigma_g(mess))/buf\}$ (5)

This transition rule describes what happens if a message on an open output gate is accepted by a channel with nonzero capacity. If gate g on channel c (1) is open (2), and there is room in c's buffer (3), then g succeeds (4), and its message is appended to c's buffer (5).

Rule 5: Receiving on a Postmedium Channel

$$\{(c, \sigma_c)\} \cup \cup_{i=1}^{n}\{(g^i, \sigma_{g^i})\} \rightarrow \{(c, \sigma'_c)\} \cup \cup_{i=1}^{n}\{(g^i, \sigma'_{g^i})\}$$

if $\sigma_c(type) = post$ (1)
$front(\sigma_c(buf)) = m$ (2)
$\sigma_c(crit) =$ '=n' (3)
$\forall i: \sigma_{g^i}(chan) = c$ (4)
$\forall i: \sigma_{g^i}(stat) = \bigcirc$ (5)
where $\sigma'_c = \sigma_c\{chop(\sigma_c(buf))/buf\}$ (6)
$\forall i: \sigma'_{g^i} = \sigma_{g^i}\{\otimes/stat,\ m/mess\}$ (7)

If there are sufficiently many (3) open (5) input gates on postmedium (1) channel c, and c has a nonempty message buffer (2), then all input gates succeed simultaneously and get the first message from c's buffer (7),

4.3 Dynamic Creation and Connections

Up to now, we have omitted the notations for dynamic creation and connection-oriented communication that ADL-d includes. We did this, because the additional constructs, needed in our formalization to include these features, are complicated and space-consuming, but not very interesting from a 'formal semantics' point of view. Below, we explain why.

Creation

As the footnote in the previous section suggested, the notations in the ADL-d diagrams represent *prototypes* of structure and behavior, of which, at runtime, multiple instances can exist, and additional instances can be created. We could, for example, enable a Contractor to create Worker instances, which is depicted in figure 6.

Here, Chan_cr is a *creation* channel, for the creation of Workers, on which Contractor has an output gate. Whenever the Contractor outputs on Chan_cr, a new instance of Worker is instantly created, including a creation of everything in its substructure (i.e. Analyzer, Chan_ac, and Calculator). This is what we want the semantics of dynamic creation to be. In our formal description, the corresponding transformation rule looks like this:

Rule 6 (concept): Sending on a Creation Channel

$$\{(c, \sigma_c), (g, \sigma_g), r\} \rightarrow \{(c, \sigma_c), (g, \sigma'_g), r'\}$$

if $\sigma_g(chan) = c$ (1)
$\sigma_g(stat) = \square$ (2)
$\sigma_c(type) = creation$ (3)
where $\sigma'_g = \sigma_g\{\boxtimes/stat\}$ (4)
$r' = r \cup$ new instances (5)

As can be seen, we introduced a new type (creation) for channels. If open gate g (2) outputs on its (1) creation

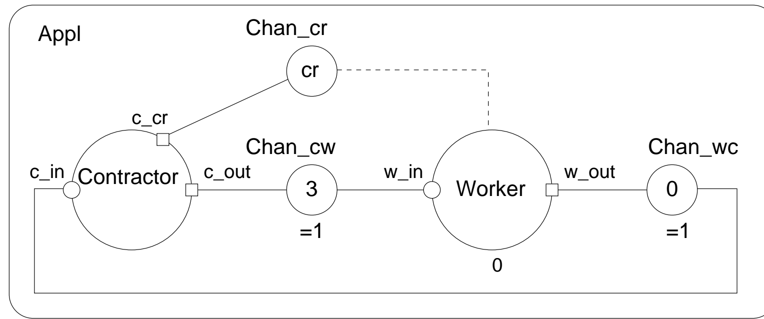


Figure 6. Example structure with creation channel

channel (3) c, then new instances are added to the existing application set (5). This set of instances is a function of the the process prototype that c refers to (which is kept in an additional, not previously shown, channel variable $proc$), and the application structure such as modeled in the ADL-d diagrams, which has to be copied among the new instances.

This copying of the application structure between new instances is a fairly complex matter, but nevertheless something that is modeled using a function executing in zero time (i.e. by the firing of one transformation rule). This latter property makes it less interesting when it comes to the interaction between notation and formalization (see below): we wanted dynamic creation of (complex) processes to be atomic, and we formalized it straightforwardly.

Connections

A similar reasoning goes for setting up connections between gates of ADL-d processes, which implies, in fact, nothing more than the dynamic creation of channels, dedicated to a select group of processes (or more exactly, gates). These creations are, in our formalization, also modeled by functions, executing atomically.

4.4 Results

Implementation

All in all, we need approximately 20 transition rules to specify the entire communication semantics of ADL-d, none of which is much more complicated than rule 5 specified above. Thus, we have a compact and simple set of guidelines to verify ADL-d implementations.

The above set of transition rules that we have chosen to illustrate matters shows a clear distinction between process-gate interaction and gate-channel interaction, with messages flowing between these objects. This, in fact, goes for the entire system, which, as was stated before, is an indication that straightforward distributed implementation is feasible. Currently, a distributed implementation is operational, serving as a testbed for new ideas, algorithms, added features etc.

What is not shown in the examples (because of space limits) is that some forms of communication require quite an amount of administration. For example, with premedium message distribution to multiple receivers, in order to prevent receivers from getting the same message more than once, records have to be kept showing which processes have received which messages. This is what, in our formalization, we use the $recvrs$ field in our messages for. In our semantics, we have been careful not to overspecify the transition rules in the direction of one implementation or another, where it comes to keeping this administration. Specifying semantics in a declarative manner, in terms of state-changes in gates, allows for many implementations. It also keeps the number of transition rules down, making the system simpler, and allowing us to focus mainly on the interaction aspects.

With these results, our main objective, to generate implementation guidelines that are nonrestrictive towards details on the algorithmic level, is met.

Feedback

As far as the interaction between formalizing the ADL-d semantics and the notation itself is concerned there are several results to be reported, the most important of which are listed below.

Single Select State Originally, the block and the test actions in ADL-d were integrated into one blocking state with two outgoing transitions: one for success and one for timeout. Also, ADL-d included a construct called the single select state, which was devised to enable processes to block for communication over several gates simultaneously, with the guarantee that after leaving the blocking state, at most one gate had succeeded. Clearly, we did not succeed in incorporating such a construct in our transition system, since success for different open gates is determined independently by different objects in the system. Suppose process P has

opened gates p_1 and p_2. If these gates are attached to different channels, then guaranteeing that only one of them succeeds would require some intricate coordination between the channels involved, when the situation on the channels is such, that p_1 and p_2 can, in principle, both succeed. In fact, any attempt to incorporate such semantics without enormous communication overhead or highly centralized solutions was feeble. A compromise was found using the language building blocks that were already present in our formalism: block and test, and make them explicit in the ADL-d notation. The only guarantee that is given after leaving a blocking state, is that none of the gates involved is open anymore. Consequently, they have to be tested one by one for success at a later stage. A problem that remained was the possibility of an input gate succeeding and not being tested before being opened again, allowing the old value to be overwritten. We have blocked this by a rule specifying that a block on a gate in state $\otimes$ leaves this state intact (see the remark below Rule 1).

Unicast and Broadcast From the start, ADL-d distinguished only between unicast and broadcast in its distribution semantics, and included special channel symbols for these. However, in searching for an elegant model to capture both, we came up with success criteria in terms of receivers reached. But this construct allowed for far more possibilities for channel semantics than just unicast and broadcast (i.e. various types of multicast), which all made sense, too. Hence, we decided to incorporate explicit notations for success criteria in ADL-d, thus covering a more complete range of communication patterns.

Nonblocking Opening A similar phenomenon can be observed with the gate states open, closed succeeded, and failed. After identifying these states from the intuitive ADL-d semantics, it is only natural to incorporate explicit commands in our process language that use/manipulate these states (in the form of block and test). However, the semantics of block can still be split into a separate open, just for opening the gate, followed by a block, for just blocking. This separate opening could then be incorporated as a gate action in ADL-d's STD notation, allowing processes to open gates without immediately blocking for them. Currently, we are investigating the consequences of incorporating a separate gate action to open gates.

Group Communication A weak point of ADL-d used to be its lack of group communication facilities. However, although only briefly discussed above, ADL-d does include constructs to model connection-oriented communication, in which a connection is established between a number of senders and receivers over one or more channels, before actual data transfer starts.

This can be considered a case of *static* group communication: a group is established, data transfer occurs, and the group disintegrates. However, formalizing the semantics of connections has revealed possibilities to let gates dynamically join and leave connections. This would be another example of successful feedback.

5 Related Work

Any design notation striving to be a candidate for automated code generation is likely to have some sort of formalization of its semantics. The idea of having your semantics defined in natural language or by an actual implementation in some programming language is not very appealing.

There are numerous formalisms to choose from, each with its own strong points and weaknesses. An important factor in the selection of a formalism is what the formalization is used for. For example, our operational semantics has the advantage of its intuitive appeal, which makes it easy to see what communication patterns are generated by ADL-d channels. Also, deriving channel implementations from these semantics is straightforward. These were exactly the properties we were looking for. However, transformation systems are hard to execute directly for verification and simulation purposes, which *is* possible when using Petri nets.

PARSE

A technique similar to ADL-d is PARSE [3]. PARSE's notations are formalized using Petri nets. Its BSL programs, which are used to specify individual process behavior, are easily converted to Petri nets. Coupling these Petri nets according to what is specified in PARSE's *process graphs*, which define an application's structure, renders a net describing an entire application. This can subsequently be used for verification and simulation on the level of abstraction that Petri nets provide.

A problem with Petri nets is their static structure, which renders difficulties when modeling applications with dynamically changing/expanding structures. This was one of the reasons we stuck to an operational semantics description of our notations.

Regis and Darwin

The configuration language Darwin [4] (with both a graphical and a textual representation), is used to describe the binding between self-contained components with well-defined communication interfaces, as, for example, used in Regis [5]. The language can be used to design dynamically evolving communication structures. Its formal semantics are de-

scribed in Milner's π-calculus [7]. In the π-calculus, it is possible to name communication channels and transmit these names over other channels among processes so that new bindings can be formed.

SDL

As was pointed out, our notation for modeling behavior has been derived from SDL [2]. In fact, the role of SDL state diagrams is very similar to that of our state transition diagrams. In SDL, state diagrams are used to model a process that appears in a block diagram. A block diagram corresponds to our notion of a (complex) process. A major distinction between SDL and ADL-d is that ADL-d emphasizes communication modeling between processes. It is for this reason that we have different communication channels, and strictly separate processing from communication by means of gate-based interfaces.

SDL has been formalized using the Meta-IV language, a very complete language, with a 'programming like' notation. As such, Meta-IV is very much oriented towards implementation, which was the objective of the formalization in the first place.

Besides PARSE, Darwin and SDL, many other modeling and design techniques exist, but not as many are explicitly targeted towards development of parallel and distributed programs. Instead, what we observe is that distributed computing is often supported at the implementation level by means of middleware solutions such as CORBA [8] and DCOM [6], or advanced communication libraries like MPI [13].

A research area related to ours is that of the object-oriented modeling techniques. Here also, work has been done on the formalization of graphical notations. For an overview, see [14].

6 Conclusions and Future Work

Using only about 20 transition rules, we have been able to formally define ADL-d's communication semantics. Together, they form an implementation model, providing a minimal set of criteria to which implementations must conform.

The maintained abstraction level has turned out to be sufficiently close to a distributed implementation to reveal several flaws and unrealistic assumptions in the 'intuitive' ADL-d semantics. The formalization has even triggered changes in the original ADL-d notation itself.

Future work will include the testing of different algorithms for internal channel behavior (of course conforming to the semantics described here). Also, as was stated before, we are looking into possibilities for providing more complete group communication facilities. The first results on this subject are expected shortly.

References

[1] J. d. Bakker and E. d. Vink. *Control Flow Semantics*. The MIT Press, Cambridge, Massachusetts, 1996.

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

[3] I. Gorton, J. Gray, and I. Jelly. Object-Based Modelling of Parallel Programs. *IEEE Parallel and Distributed Technology*, 3(2):52–63, July 1995.

[4] J. Magee, N. Dulay, and J. Kramer. Structuring Parallel and Distributed Programs. *IEE Software Engineering Journal*, 8(2):73–82, Mar. 1993.

[5] 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, Sept. 1994.

[6] Microsoft Corporation. *DCOM Technical Overview*, 1996.

[7] R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes. Technical Report ECS-LFCS-89-85 and -86, Lab. for Foundations of Computer Science, Edinburgh University, Edinburgh, 1989.

[8] OMG. The Common Object Request Broker: Architecture and Specification, revision 2.0. OMG Document 96.03.04, Object Management Group, Mar. 1996.

[9] M. Polman and M. v. Steen. Design Level Support for Parallel and Distributed Applications. In *High-Performace Computing and Networking*, pages 812–819, Brussels, Belgium, Apr. 1996. Springer-Verlag, Berlin.

[10] M. Polman and M. v. Steen. Designing Distributed Programs with Dynamic Communication Structures. In *Proceedings of ICA3PP'96*, pages 271–278, Singapore, June 1996.

[11] M. Polman, M. v. Steen, and A. d. Bruin. A Structured Design Technique For Distributed Programs. Technical Report EUR-CS-96-04, Erasmus University of Rotterdam, Rotterdam, Nov. 1996.

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

[13] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI: The Complete Reference*. MIT Press, Cambridge, MA., 1996.

[14] R. Wieringa and G. Saake. A formal analysis of the Shlaer-Mellor method: towards a toolkit for formal and informal requirements specification techniques. *Requirements Engineering*, 1:106–131, 1996.