A Comparison of Graphical Design Techniques for Parallel, Disributed Software

Mark POLMAN and Maarten VAN STEEN
Erasmus University, Department of Computer Science
P.O. Box 1738, 3000 DR, Rotterdam, {polman,steen}@cs.few.eur.nl

Abstract

We have compared three graphical design techniques, OMT, ADL, and PARSE, on their suitability for the development of parallel/distributed applications. Our method has been to use all three of them in modeling one, existing, application: a backup facility running within the Andrew File System as described in [2]. We compare and analyze the outcomes on a number of important design aspects. Based on this, we draw conclusions on each individual technique and on graphical design techniques for parallel/distributed software in general.

1 Introduction

With the increasing availability of hardware platforms, suitable to run parallel, distributed software, we can expect to observe an increase in the production of this kind of software. One obstacle to this is, that its development is found to be much more complex than that of 'conventional' software: sharing of resources, replication, communication in absence of shared memory, etc. all add to the complexity of (the dynamic behavior of) software components. Graphical design techniques, having proved their value in the development of conventional software, seem to be an answer.

A well-known one is the Object Modeling Technique (OMT) by Rumbaugh et al. [8]. Because this method is object-oriented, it could be argued that, through the natural concurrency of objects, it would be fit for parallel, distributed software as well. However, it is the question how well OMT deals with the communication and synchronization issues that are so important in parallel, distributed application: perhaps OMT needs some adaptations here.

Our approach to see if this is the case, is a very pragmatic one. We take an existing problem, we consider a number of graphical modeling techniques (among which OMT), and apply all of them to this problem. After that, we analyze the differences between the outcomes, and see if we can pinpoint some of the methods' weaknesses or strong points with regard to parallelism and distributed-ness.

Of course, we want the testcase to be a good discriminator, i.e. it must be complex enough to highlight the differences in the examined methods. On the other hand, it should be simple enough to render three models that are somehow comparable. One application that seems to have these properties is a backup facility in the Andrew File System (AFS), which has been described and implemented by Gorlen et al. [2]. In the next two sections, we will describe the essential features of the AFS and of this backup

application. We will illustrate the workings of the application by giving an outline of Gorlen's implementation. In our own models we will for the greater part adopt Gorlen's approach with a few modifications, which we explain below.

In Section 4, we will list some aspects of the application, which are of special interest to us. Distributed-ness and parallelism will play an important role here.

Then, in sections 5, 6 and 7, we model the application using our three modeling techniques, which are OMT (Object Modeling Technique) [8], ADL (Application Design Language) [11, 10, 9], and PARSE (PARallel Software Engineering) [3, 4]. About these methods we can already say that OMT is explicitly object-oriented, PARSE primarily designs in terms of communication flow diagrams, and ADL combines similar diagrams with state transition diagrams (STDs) per process. All this, however, renders a very unsubtle classification, and we will try to do all of the methods more justice in the course of this report.

In Section 8, we compare the resulting models and analyze their differences. We then draw our conclusions, being a list of positive and negative properties of all three techniques following from our experiences, and suggestions for combinations of strong points.

We would like to emphasize that we do not need very detailed models to compare our three techniques. All three techniques are, from some point onwards, based on refinement of models using decomposition and textual additions. Thus, we can keep the diagrams that we produce at a fairly high level of abstraction.

Also, it may well be possible that there are various ways of modeling this application and still getting it to work correctly. However, our primary intention was to come up with three models that were somehow comparable, while still showing where the three techniques differ. The adopted solution seems the best to us in this respect.

2 The afs

As was mentioned before, our application is based on a case study in [2]. Here, the requirements specifications are given for a backup system within the Andrew File System (AFS) [6]. The AFS is a distributed file system based on the caching of files on harddisks of workstations. Globally, the organization is as follows: client workstations are grouped in clusters, each of which is assigned a file server, which is, in turn, connected to other file servers. Whenever a process on a client workstation opens a file, the system makes sure that a local copy of the file is present on the workstation's local harddisk. Subsequent reads and writes are then performed on this local copy. On closing, the file is copied back onto the file server's harddisk. This scheme is entirely transparent to the user.

In the AFS, the notion of a *volume* exists, which is globally a (part of a) directory tree, that can be manipulated as a whole. We have, for example, one volume per user, containing the entire set of files of a user. Now, the AFS includes a number of volume-based operations. One of these concerns the possibility of copying volumes from one file server to another in one atomic operation. In order to accomplish this, the possibility exists to produce a read-only snapshot of a volume, called a *clone* volume. It is this kind of volume that is used by our backup application.

To our application, the volume-aspect and the fact, that we have a number of file

servers, each managing a group of volumes, is most important. Therefore, we will not go any deeper into the specifics of the AFS than necessary. For a more complete description we refer to [6].

3 The Backup Application

In essence, our application is expected to do all kinds of backups (weekly, monthly, full, incremental) of volumes to tape. Besides that, it should, of course, keep track of where volumes are backed up. Therefore, it also updates a databases containing this information.

Now, Gorlen et al. have already modeled and implemented the system in an object-oriented way, without using (we conclude this from the book) a formal design method. Their approach seems to be based on the application's functionality itself as well as on the existence of the NIH class library, which is a C++ library of object-classes with, among others, various types of data containers and light-weight process-classes. Here, we will give a short description of the outcome.

The application Gorlen developed utilizes a database in which *backup requests* of various kinds are placed by the *request manager* object. This request manager determines for each volume what its *backup state* is, and subsequently which backup is to be performed next. The backup requests are distributed by the request manager over several *server managers* (one server manager per file server).

A server manager receives the requests for those volumes that reside on the file server that it is attached to. Server managers prepare the backup activities to be performed. This implies, among others, the reserving of a *staging disk* for the backup requests of a file server. The staging disk is used as a buffer storage for volumes before they are copied onto tape.

When a staging disk is available, a server manager will pass it on, along with its requests, to a backup manager, who performs the actual staging and taping. In order to perform the taping, it is also necessary to reserve a *tape drive*. During the staging and taping, the backup manager will hold on to both staging disk and tape drive. Afterwards, these resources are released.

In the mean time, a database is maintained, which records where volumes are backed up, and what the backup state of each volume is. Every time a backup operation completes, the database is updated.

In *our* models, which will start in Section 5, we have only diverged from this approach in two ways:

- Firstly, we have transferred the reservation of a tape drive to an earlier stage. It will be done by the server manager (or an equivalent process).

- Secondly, we have modeled the 'managers' as non-terminating, i.e. they iterate over some sequence of actions continuously. This means that we have various parts of the system wait for events to happen, after which they become active: they are not started up explicitly.

Both modeling decisions were taken to come up with easy to understand, *comparable* models in all three techniques.

4 Our Interests

Now that we know what our application is expected to do, we can identify the aspects that are particularly interesting to us. Naturally, these all have to do with the parallel, distributed nature of this application. At this stage, we can directly identify three such aspects:

- The backup requests, in Gorlen's implementation generated by the request manager, will ultimately set the file servers in the AFS to work. Now, in order to improve performance, it appears [2] that a file server should only be handling one request at a time. This implies that requests should be handed to the file servers in a controlled manner: some form of synchronization or sequencing is required.

- Backing up a volume is done in two steps. First, volumes are copied onto a staging disk, and from there onto tape. During this entire operation, the staging disk and the tape drive should be dedicated to this backuprequest only. In other words, exclusive access to the staging disk is required throughout the operation. This implies more synchronization.

- While backing up volumes, a common database has to be maintained, part of which describes which backup volume resides where, and part of which describes what the backup state of each volume is. Both parts are updated by the backup managers, several of which can be active at any time. So, again, synchronization is necessary.

The complexity of all this synchronization, together with the passing of information (requests, staging disks, tape drives) between the application's components, calls out for abstraction, which, in our case, will be mostly graphical. In the next sections, we will model our application using the techniques of OMT, ADL, and PARSE.

5 OMT

5.1 Introduction

Our first modeling technique is OMT (Object Modeling Technique), an object-oriented design method which combines an enhanced entity-relation diagram of objects (the *object model*) in the application with a number of STDs (one for each object) and some dataflow diagrams (DFDs) of processes that can be distinguished within the application's functionality.

The object model describes static information on an application. In an object model we see what an application consists of. It is the most important model in OMT. The STDs describe the dynamic behaviour of objects, and provide operations to the objects. The DFDs provide us with a functional overview of the application, and provide operations as well.

Using OMT, a designer starts with identifying objects from the requirements specifications. By identifying the relations between these objects, a global object model can be produced. Then, in a process of constant iteration, this model is adapted and refined right to the point where we have an actual program which can be compiled and

run. This refining is supported by the other two OMT diagramming techniques, which provide objects with operations and which also provide the designer with a new look on inter-object relations.

In this section, we will first look at the global object model of the AFS, the application domain. Then, we introduce the objects that are specific to our application in a second object model. After that we will consider the most important STDs in the dynamic model and the DFDs in the functional model.

5.2 The Object Model

Notation

In the entity-relation diagram, objects are given by rectangles in which the object class' characteristics are listed. Relations are denoted by labeled lines between object classes. The multiplicity of one side of a relation is indicated by open dots (zero or one), or closed dots (zero or more). The default is one. Other notations we will use in our diagrams will be discussed later on. However, not *all* symbols available in OMT will be used. An overview of the entire set of notational possibilities can of course be found in [8].

Global object diagram

Our first object model is given in Figure 1, and provides an overview of the essential objects in the AFS and the relations between them. In the upper part of the diagram, the meaning of the used symbols is given.

Globally, the diagram can be divided into four parts: the user part, the file part, the hardware part, and the volume part. We will describe each of them separately:

user objects On the lower right of the figure the user objects are given. Here we see that a user can run multiple processes, served by the same server stub. This stub handles the `open` and `close` calls of user processes; it makes sure that the corresponding files are copied from the file server to the local harddisk and vice versa. This is accomplished by communication with the file server, serving the corresponding cluster. The file servers in the AFS run a system called *vice*, which enables them to communicate with each other and with their clients, in order to provide for the right volumes at the right places.

From the relation between users and client stations, we can see that users can be logged in on several machines at the same time, and that several users can use the same machine simultaneously.

file objects The upper right part of the diagram describes the objects concerning files in the AFS. Here, we see that of every file there is a primary copy (residing at some file server) which can be cached at several workstations' local harddisks. On some of these copies a callback exists which implies that the copy is 'recognized' by the file server. The latter will then notify the client in the case of certain events. We see, that not all local copies have callbacks. This is indicated by the *subset constraint.*

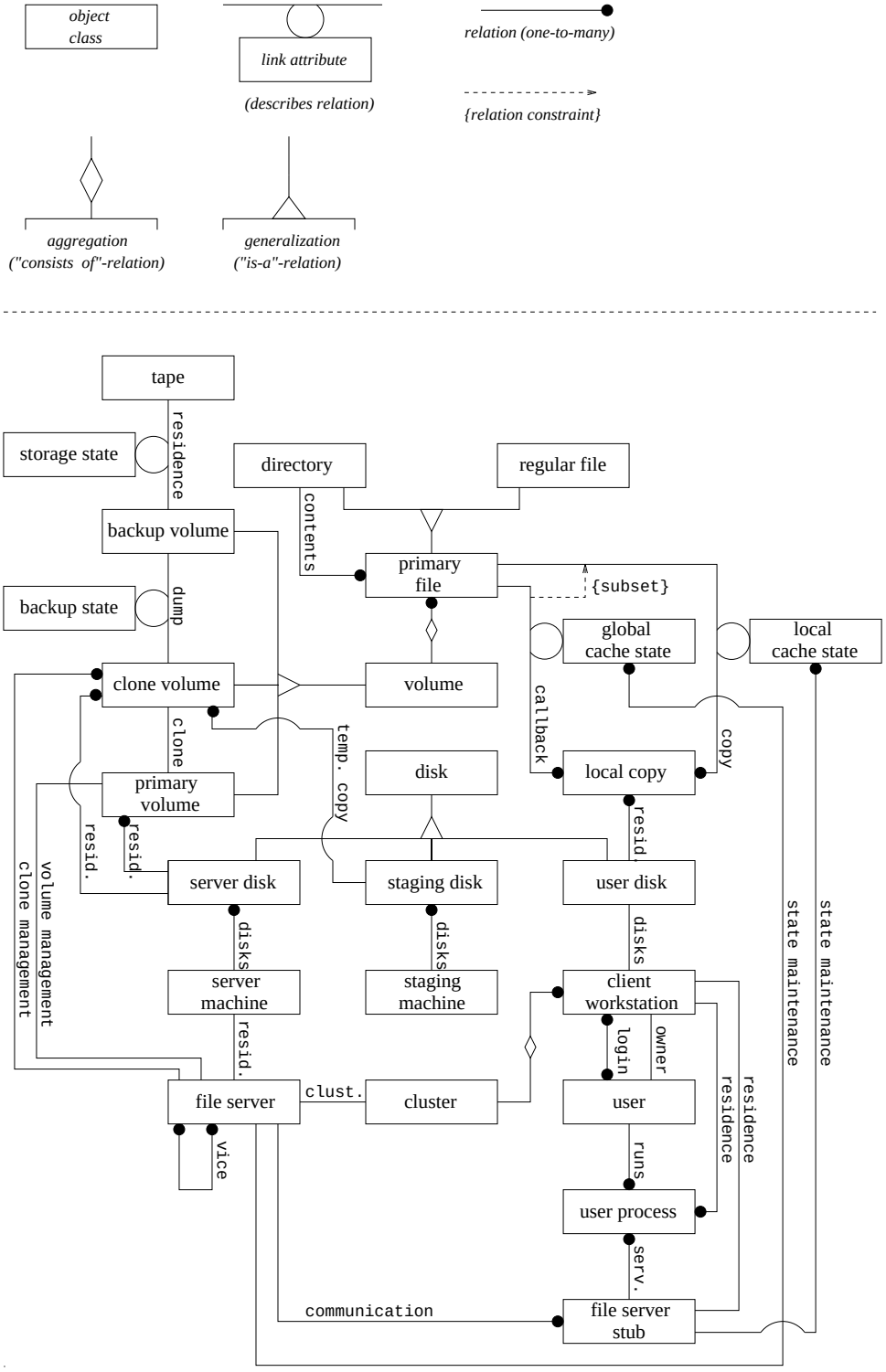

Figure 1: Global object diagram of the application domain

The global cache state of a primary copy describes where it is cached, and is maintained by the file server. The local cache state is maintained by the client stubs, and contains information on, for example, callbacks. Global and local cache state are objects describing *relations.* Therefore, they are called *link attributes.*

hardware In our diagram, we distinguish three different machines: the file servers, the staging machines, and the client workstations. Each of these contains one or several harddisks. Every file server machine handles the file operations of a *cluster* of client workstations.

volumes AFS volumes come in three kinds: primary volumes, which reside on a file server's harddisk; clone volumes, which reside there as well, but which are also copied on staging machines' harddisks in the backup process; and backup volumes, which reside on tapes. The relation between clone volumes and backup volumes gives us the backup state, describing how a volume is backed up (full, incremental, weekly, etc.). The relation between backup volumes and their tapes gives us the storage state, specifying where a volume is backed up.

Two symbols appearing in the model have not been described yet: the specialization and the aggregation. Specializations are denoted by triangles. They indicate an 'is a' relation, e.g. a disk *is a* server disk, a staging disk, or a user disk. Aggregations, represented by diamonds, describe 'part of' relations: a primary file is a *part of* a volume.

Application specific objects

So far, we have just modeled the *application domain* in terms of objects. Of course, this is not enough to model the entire application. What we need is a set of objects implementing the activities described before. To be more specific, we need objects to carry out the process of building a backup request table, of scheduling the requests to be performed, of synchronization among processes to make correct and efficient use of resources such as tapes, staging disks and file servers, and of performing the actual staging and backing up. Furthermore, we need objects to contain the data used by the above operations.

Let's first look at the objects we need and how they fit in our model so far. Their placement can be observed from Figure 2. From this figure, we have, for clarity, stripped some objects and relations that we consider not entirely relevant for our application: most importantly, the 'user part' has been left out.

As one can see from this figure, we have introduced a number of object classes specific for our application. Firstly, we need *server managers.* They shield a fileserver from concurrent read requests for different staging operations. This ([2]) appears to be necessary from an efficiency point of view: concurrent read requests are bad for performance.

One instance of the server manager object class is associated to each file server. Each server manager gets its own local request table, consisting of requests for backups of volumes that reside at its corresponding file server.

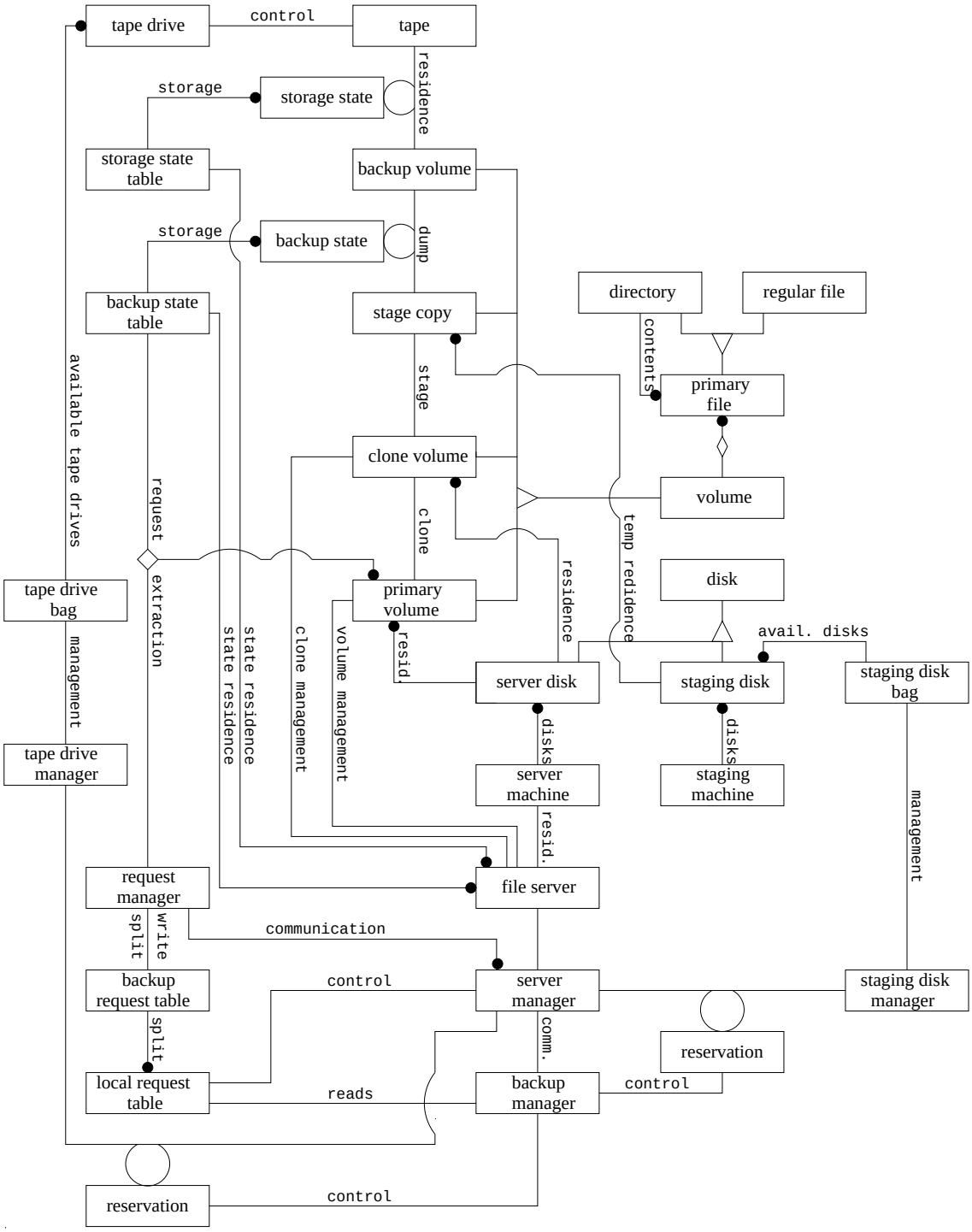

Figure 2: Global object diagram of application objects

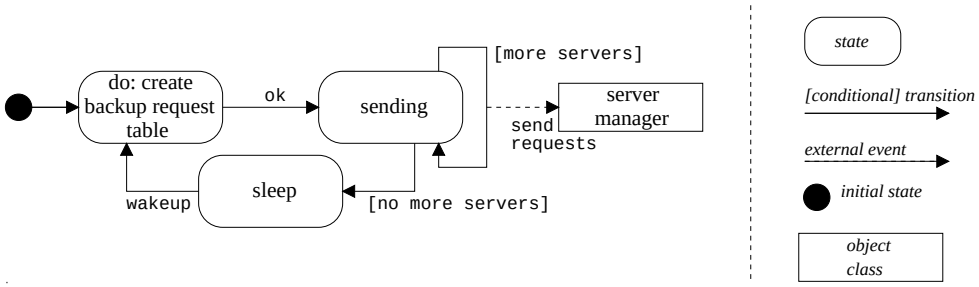

Figure 3: Request manager: STD

There are two more object classes besides file servers with respect to which other objects will have to synchronize. These are the staging disks and the tape drives. They can be involved in only one backup 'operation' at a time. To this end, they will be reserved by a server manager for the time needed to process the backup requests of the server manager. In our diagram the available disks and drives are put into bags. Associated with a bag is a manager object responsible for the distribution of resources over waiting clients, in this case, the server managers.

The actual staging and taping of volumes is performed by a backup manager, which also updates the backup and storage states of the backed up volumes. These states are saved in the backup and storage state tables, residing on some file server machine's harddisk. After a backup manager has processed its backup request table, it releases the staging disk and tape drive that were reserved for it.

5.3 The Dynamic Model

Notation

In the OMT the dynamic behaviour of an application is specified on an object class basis by STDs in the form of state charts ([5]). The diagramming technique includes symbols for states (rounded boxes) and transitions (arrows between boxes). Usually, transitions from one state to another are triggered by internal or external events, which appear in text near the arrows. Other notations will be discussed after the application's diagrams.

States of objects, in which the object is performing some activity, can often be decomposed into a collection of substates with transitions in between them. This can be very easily done in OMT, globally, by making sure that the incoming and outgoing transitions of the subdiagram of a state match those of the original diagram.

In this paper, we will only consider three 'manager objects': the request manager, the backup manager and the server manager, for they display the most interesting behaviour. Decompositions of states will not be shown. The subject of decomposition will come up again in PARSE, where it is most important.

The request manager

The dynamic behaviour of the request manager can be observed from Figure 3. The righthand-side of the diagram gives an inventory of used symbols. Globally, the request manager acts as follows: first, the backup request table is constructed from the existing volumes in the system and the backup state table. After that, the server managers are

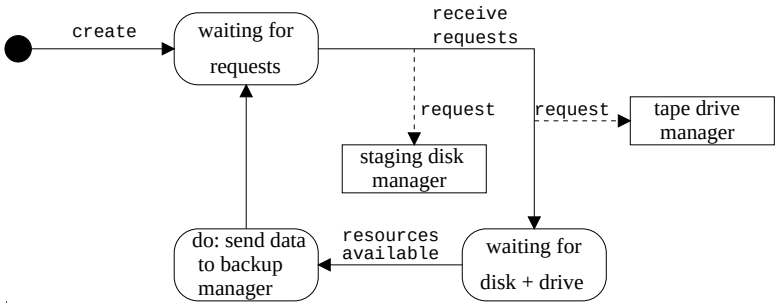

Figure 4: Server manager: STD

provided with the backup requests for their corresponding file servers. This is modeled by a so-called *external event*, *send requests*, to the server manager class. An external event is an event triggered by one object class, and affecting another object class. It is denoted by a dashed, labeled arrow to the receiving object.

Finally, the request manager sleeps for some time and returns to the state where it builds up a new request table.

The server manager

The dynamic behaviour of a server manager is given in Figure 4. It starts out by waiting for its local request table to arrive from the request manager. Notice, that the incoming event *receive requests* corresponds with the external event *send requests* by the request manager. Hence, we have modeled communication between these two classes.

When the local request table arrives, a server manager enqueues itself to wait for a staging disk and a tape drive. To this end, it will generate *external* events to the staging disk manager and the tape drive manager, who manage the resource bags.

When staging disk and tape drive are both available, the server manager will pass them on to the backup manager, together with its local request table.

The backup manager

The most interesting dynamic behaviour comes from the backup managers in the system. It can be seen in Figure 5. As soon as data (i.e. the local request table) comes in, the backup manager will get the first backup request and start staging. Then it proceeds with the next and so on, until either the staging disk is full or there are no more requests. In either case, the taping begins. After performing all the required backups, the backup manager will update the database containing the backup data and release the staging disk and the tape drive. External events will be sent to the resource managers (the disk and drive are returned).

Obviously, after taping some volumes, a backup manager will only resume staging if there are still some requests left. So, if taping has succeeded, we have a *conditional* transition either to staging or to updating. This is denoted by a statement between square brackets over the event name (e.g. the condition ‘no more requests’).

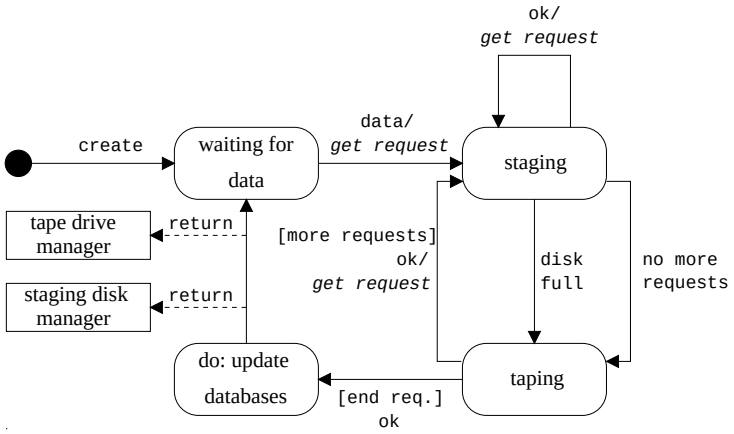

Figure 5: Backup manager: STD

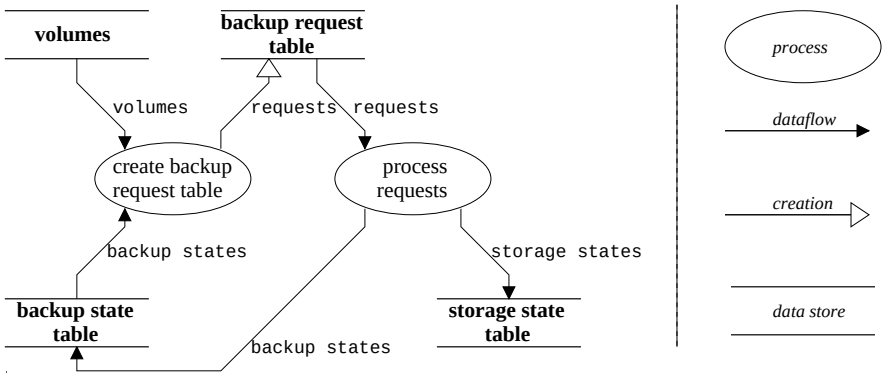

Figure 6: High-level functional model

5.4 The Functional Model

Notation

The functional model in OMT consists of diagrams specifying the flow of data between processes within the application. In these models, a process is denoted by an ellipsis, while a dataflow is represented by an arrow with a label indicating the sort of data that 'flows'. Often, data is obtained from, or sent to, an object where no further processing is done. For this we use the normal object notation (i.e. a rectangular box). Also, there are entities which merely serve to store data and intermediate results, the so-called data stores. These are denoted by two horizontal lines with the data store's name in between them. Examples of these are typically files and other containers such as lists/tables.

Global functional model

For now, we will only consider a very high-level DFD. Later on we will discuss the problems with its refinement. The diagram is given in Figure 6. In the diagram, we have modeled our application as consisting of two processes, namely one in which the backup request table is generated, and one in which the backup requests are processed.

The processes communicate by means of the backup request table and (backwards) by the backup state table. The generation of a data store (or and object) is indicated by an open arrow to it (instead of a filled one). Other data stores relevant for our application are the backup and storage state table. The diagram does not show any objects as data sources or sinks.

6 ADL

6.1 Introduction

The next modeling technique we'll discuss is ADL (Application Design Language). ADL is a process-based graphical design language, which combines communication flow diagrams on various levels of decomposition with STDs for each process. As we will see, the notations used are quite different from what we have seen above, which results in some desirable features of the language, especially with regard to replication and types and semantics of communication.

We will first present the global communication flow diagram of the application, followed by STDs for three processes with interesting dynamic behaviour.

6.2 The Flow of Communication

Notation

The flow of interprocess communication in ADL models is given in so-called *structure diagrams*. Here, processes are represented by circles. These processes communicate to each other through input and output gates (small circles and squares on the edge of a process symbol) via communication channels. The channels are represented by medium-sized circles containing a symbol for the type of communication that goes on between the processes: synchronous, asynchronous (queued) or by semaphores.

An important feature in ADL communication flow notation is replication. Replication of entire substructures of a model can be represented by connecting a number of process and channel symbols to the replication symbol (a diamond).

In ADL it is also possible to decompose a process into several sub-processes with communication channels between them. In the subdiagram, communication to and from the high-level (composite) process will be denoted by rectangles, indicating *external* gates. In this paper, we will not go into decomposition of processes any further, until, as stated before, we arrive at the PARSE process notation.

Other notational features of ADL will be discussed along with the application's diagrams, which are discussed next.

Global Communication Diagram

In Figure 7, we have modeled the flow of communication between our application's processes.

The core of the application consists of three processes. The first of these, the request manager, is responsible for generating the backup request table, using information on volumes and the backup state (notice that these are also modeled as processes). The backup request table is split up and each part is communicated to the (replicated, see

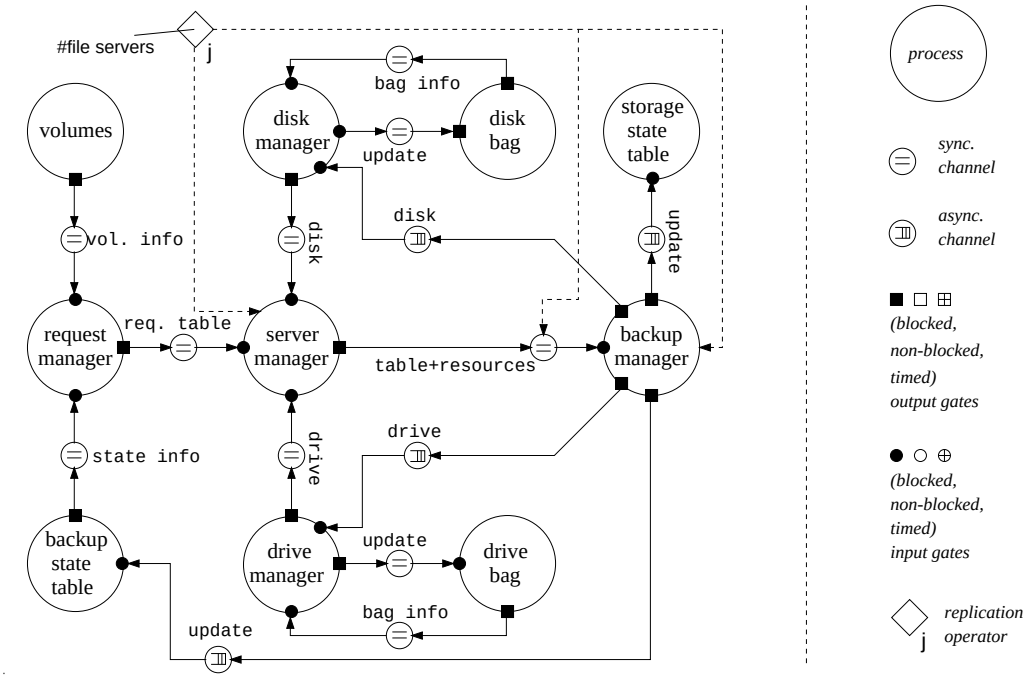

Figure 7: Global ADL structure diagram

below) server manager, preparing the backup activities over a synchronous channel (indicated by the '=' symbol). This process reserves a staging disk and a tape drive from the disk manager and the drive manager, respectively. The disk and drive as well as the request table are then passed on synchronously to the third process, which actually performs the backups. Afterwards it releases its resources and updates the relevant data (backup and storage state) asynchronously (the queue symbol indicates this). Notice, that the three processes behave just like the request-, server-, and backup manager in OMT.

Now, every process involved in some communication is prepared to *block* until communication completes. This is indicated by filling the gate symbols. In the case of non-blocking communication the gate symbols would be left open. Another option in ADL is *timed* communication, indicating that a process is willing to block only for a certain amount of time for communication to take place. This is denoted by a '+'-symbol within the gate, and an annotation concerning the maximum amount of time spent blocking.

Another observation about this model concerns the fact, that the entire structure of preparing and backing up is *replicated*: we have one such structure for each file server in the system. The *intended* meaning of all this, is that the request manager sends parts of its request table to a number of server managers processes, which all communicate to their own backup managers. However, there is a bit more to this. ADL's communication flow diagrams support replication of *channels* only if the connected processes are also replicated. In this way, replication of parts of the design does not change the *interface* of any process. The consequence of this is that a request manager can only send its table parts over *one* channel. Now, the ADL semantics of these sends are, that *any* server manager ready to receive them, can pick them up. However, *we* wanted to model that

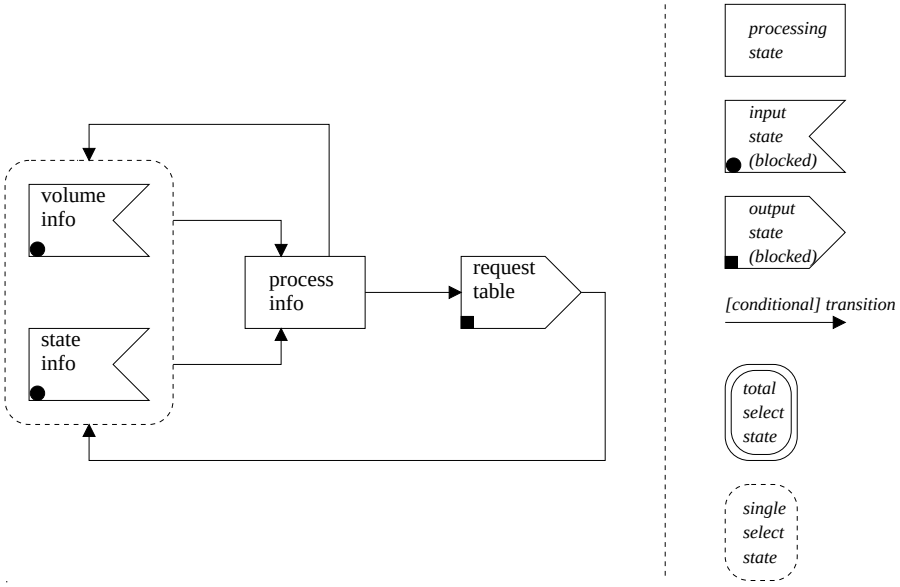

Figure 8: Dynamic model for request manager

specific parts of the table were intended for specific server managers, implying that server managers should be individually addressable by the request manager. Thus, we reveal a shortcoming in ADL's modeling possibilities. Possible solutions are currently being worked on.

6.3 Dynamic Behaviour

Notation

The dynamics of the processes in an ADL model are specified using STDs. ADL distinguishes two different states: processing states in which a process is processing data, and communication states in which a process is communicating or trying to communicate. The latter can be further divided into input and output states. Transitions from communication states are triggered by the success or failure of communication (in blocking communication only success, of course). Transitions from processing states are dependent on the outcome of the 'processing' itself; they can be annotated with boolean expressions. Processing states are denoted by rectangles, communication states look like processing states with one side bended to the outside or inside for output or input states, respectively. They are named after the communication channel they correspond with. In the lower left corner of a communication state symbol, the kind (i.e. blocking, non-blocking or timed) of communication is copied from the corresponding gate in the structure diagram.

The request manager process

From Figure 8 we can observe the dynamic behaviour of the request manager process.

Initially it will be in one of two input states: it is either trying to receive volume information or backup state information. This is denoted by a so-called single select state, which takes the form of a dashed box surrounding the states that can be selected.

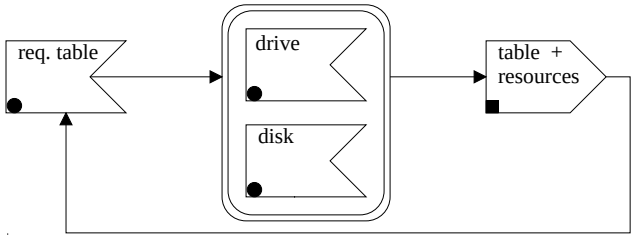

Figure 9: Dynamic model for server manager

In the following processing state the process generates the backup requests. From there it either returns to the initial state (if more information is to be received), or proceeds to sending the request table to the next process. By the last transition back to the original state, we can see that the process is modeled as *ongoing*.

The server manager process

Figure 9 shows the dynamics of the server manager process. This process starts out by waiting for its request table to come in. When this happens, it tries to reserve a staging disk and a tape drive. By the *total select state*, denoted by a double box, we indicate that it does not matter which of the two resources becomes available first. However, both states must have been entered before proceeding to the last. This is the state where the request table and the resources are sent to the backup manager process.

The backup manager process

Consider Figure 10. Here, we have the behaviour of the process actually performing the backups, the backup manager. Initially, it waits for its data to come in. Then, it will alternately be staging or taping, until all requests are processed. Afterwards, the backup and storage state tables are updated, and the resources released. The latter two states are again total select states, both containing two substates.

7 PARSE

7.1 Introduction

Just like ADL, the PARSE (PARallel Software Engineering) method concentrates on the communication between the various processes that can be distinguished within an application. However, PARSE does not include STDs to specify their dynamic behaviour. On the other hand, the notation to specify the flow of communication is somewhat richer than ADL's.

Again we will start out by considering a very global process communication model, which will be refined in subsequent 'sub'-models.

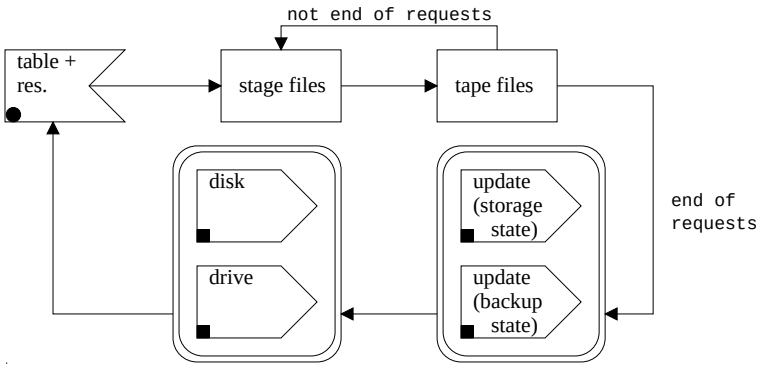

Figure 10: Dynamic model for backup manager

7.2 The PARSE model

Notation

Processes in PARSE come in four kinds, namely data servers, control processes, functions and external interfaces. Every kind has its own symbol: a rectangle, rounded box, ellipsis and filled rectangle, respectively. Communication between these processes can be synchronous, asynchronous, bi-directional synchronous and by broadcast, where every one of these has a timed variant as well. This communication is indicated by lines connecting the processes, where an additional symbol indicates the type of communication.

PARSE also includes a notion called the *path constructor*, which concerns the way a collection of incoming paths to a process is handled. The constructor symbol used determines whether the communication through the paths should be handled deterministically (sometimes prioritized), nondeterministically or concurrently. The corresponding symbols will be discussed along with the PARSE application model.

Now, PARSE does not include explicit graphical means of showing behaviour in time: for a description of a process that cannot be decomposed (a *primitive* process), a textual description is assumed. To get to these primitive processes a designer has to perform process decompositions. This decomposition in PARSE has two aspects. Firstly, we have decomposition of the processes themselves, which will render a collection of subprocesses. Secondly, we have a decomposition of the incoming communication to the composed process involving primarily the path constructors attached to it. For this latter decomposition, a set of well-defined rules apply, e.g. a concurrent path constructor of two incoming paths should be split up at the lowest level over two *different* primitive processes.

Finally, PARSE, just like ADL, includes replication notations. This is done by annotating the replicated process by the number of instances. All incoming and outgoing paths of a replicated process are assumed to be replicated as well, but this may be overruled by the designer using textual additions.

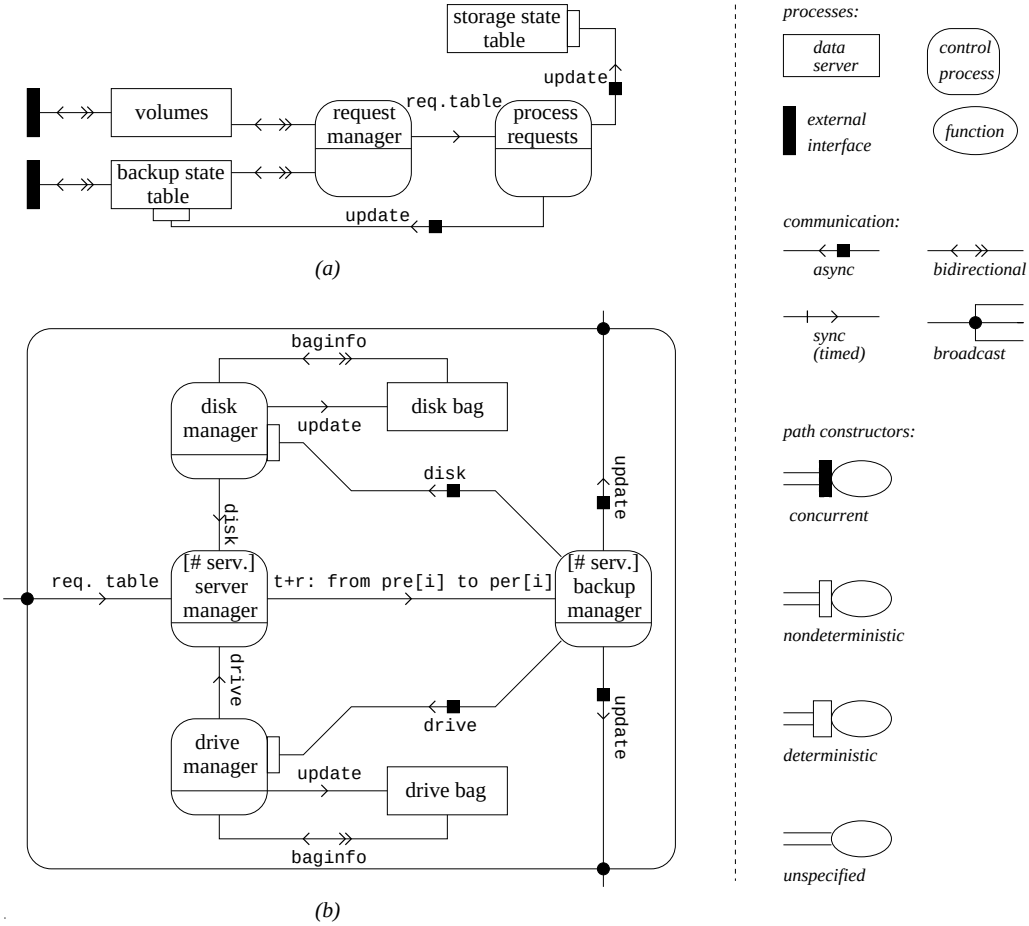

Figure 11: a: Global PARSE model; b: Decomposition of second process

Global PARSE model

In Figure 11a, we have given a very global PARSE model of our application. The process generating the request table, the request manager, will get information about volumes and their backup states by *bi-directional synchronous* communication with the data servers 'volumes' and 'backup state table'. To show how external interface processes are used, we have modeled the data servers as communicating via an external interface to, say, a database management system, managing information on volumes and states.

The request table is forwarded synchronously to the process responsible for handling the requests after which the database updates take place. Attached to the state tables are path constructors (the open rectangles). These indicate that updates on these tables are performed nondeterministically and sequential. This is because the 'process requests' process contains multiple subprocesses (see below), which all issue updates. The order in which these come in is not specified. The nondeterministic path constructor is one of four path constructors. The PARSE designer can also use a concurrent and a deterministic constructor, as well as leave the order of processing unspecified. The corresponding notations can be found in the symbols overview at the right of the figure.

PARSE Decomposition

Now, the process handling the requests can be decomposed, as we see in Figure 11b.

Notice, that the combination of this model with the higher level model from Figure 11a reveals a lot of similarity with the global ADL model.

Let's see what our decomposition is about. It contains two replicated processes: the server manager and the backup manager process. Now, since, by default, there would be a 'table+resources' path from every server manager to every backup manager, we will have to overrule that. After all, we want a server manager to communicate to one only backup manager. This results in the *path restriction* from the diagram, where it says that table plus resources from the i^{th} preparer are sent to the i^{th} backup manager.

Furthermore, it is crucial that the server managers are individually addressable by the request manager. Fortunately (in this case), the default semantics in PARSE of communication from one to many, is that the port on the 'one'-side is *vectored*. However, in other situations, this might not be desirable at all, as we will discuss later.

Another important observation to make is that the returns of disks and drives to their managers will come in from multiple backup managers. These, again, have to be processed sequentially and non-deterministically. Hence, we get the corresponding path constructor to the managers. The returns of resources are modeled here as asynchronous.

Dynamic behaviour

As was mentioned before, PARSE includes no graphical means of specifying behaviour in time for processes. Initially, it was intended to use Petri-nets for this. With these, correctness proofs could be given, if needed. Later in the development of PARSE, the decision for only Petri-nets was dropped to give the designer the opportunity to use a technique of his/hew own choice. So, it should be noted, that the PARSE *method* does include specification of dynamic behaviour of processes as part of the design process.

8 Analysis of Models

In the previous sections, we have modeled the application graphically, using our three modeling techniques. Because of the fact that we took one model (as proposed by Gorlen) as a basis and tried to express this in all three techniques, we ended up with three models that can be very well compared to each other, which we will do in this section.

Here, we will start with a discussion on the key concepts in a design technique. After that, we look at our three techniques and see how well they deal with these concepts. Along with this comes a list of several strong points and weaknesses of the techniques, and suggestions for the combination of strong features. Of course, we will frequently return to our application to illustrate the discussion and to motivate our beliefs.

8.1 Important Concepts

What are important issues to keep in mind, when looking for useful stuff to incorporate into a graphical design technique for parallel, distributed applications? We list a number of them below:

Data

In software engineering, we observe a growing interest in object-oriented design. This technique is essentially data driven: it focuses on the application domain first (i.e. the data to manipulate) and builds functionality on top of it. This functionality can then be easily changed in the future, which lowers software maintenance costs.

To capture data aspects in an application adequately, we concentrate on:

- Data modeling, i.e. describing the *structure* of data.
- When and how data is used in the application.

In our application, for example, we would want to model exactly, *what* a request table consists of, as well as *when* and *how* it is used by the request manager.

Communication

One of the reasons, that the development of parallel, distributed software differs from that of other software, is that communication gets much more complicated as synchronization with regard to shared software components comes in. Hence, we want to be able to model the following communication/synchronization aspects in a clear and well-defined manner.

- Firstly, we have communication structure; we want to model which software component communicates to which others.
- Secondly, the dynamic aspects of communication deserve attention. We want to be able to indicate when communication takes place.
- Communication semantics are equally important. We want to define exactly what, for example, a communication line between two processes stands for.
- Finally, we are interested in the contents of communication.

So, we're interested in *who* communicates, *when* this happens, *how* it is done and *what* data is communicated. The latter of the four has, of course, a significant link to data modeling.

In modeling the backup facility, we would like to express, for example, that a request manager sends a specific part of its request table to each individual server manager waiting for it. This alone already involves all aspects mentioned above.

Parallelism and replication

Parallelism occurs in application models for two main reasons: firstly, because often it is just natural to model a part of an application as a number of parallel processes, and secondly, because we can gain speed of execution by having activities performed in parallel. For example, in our application it is very natural to set all file servers to work in parallel. Besides that, we expect that it will give us a significant speedup.

In the discussion on the support of parallelism by our three techniques, we concentrate on *replication* of software components. Basically, we want to be able to handle:

- Existency constraints, indicating whether a process is replicated, and if so, how many times, dynamically or statically, etc.
- Replication structure: if parts of a design are replicated, how does that influence the communication channels between them? What kind of complex communication/replication structures can we handle?

In the backup application, we want to model that there is a server and a backup manager for each file server. Besides that, we want to know exactly what replication of server managers means for the communication with the request manager. The situation that replicated objects/processes have communication between *themselves* (in, for example, a pipeline or grid structure) does not occur in our application. However, in other applications, this might well be the case. Hence, we want to be able to model that as well.

Orthogonality and Decomposition

Of course, the purpose of any graphical modeling technique is to get a grip on the complexity of an application. This is usually done through a divide and conquer approach: we split up the application into separate parts, model the individual pieces and glue them back together. Now, three issues are important here:

- Firstly, we have the issue of *orthogonality of notations*. Globally, this means that used symbols represent clearly separated modeling concepts. Thus, we get a set of basic building blocks able to constitute higher level concepts. An orthogonal notation has advantages with regard to, for example, flexibility, code generation, and reasoning about functionality.
- Secondly, there is *orthogonality of techniques*, meaning that different diagramming techniques within the same method can be used independently with remaining links as well-defined as possible.
- As a third issue, we mention decomposition of diagrams to reach higher levels of detail, keeping the links between a diagram and its decompositions well-defined.

To illustrate notational orthogonality, we return to communication modeling. In communication modeling, we can use the message queue (with capacity ranging from zero to infinity) to model an (a)synchronous communication channel. After that we

can connect senders/receivers to this channel by (timed) ports/gates, without altering the kind of communication: here, ports and queues are orthogonal concepts.

An example of orthogonal techniques can be found in the separation of communication flow modeling and dynamic behaviour modeling: one aspect can be modeled separately from the other.

Decomposition can be applied to, for example, processes. In the decomposition we then have to make sure, that every connection to/from the high-level process returns on a lower-level process.

Automated code generation

It is claimed that OMT, if properly used, will take a designer from requirements specifications to implementation in a seamless refinement process. In particular, implementing a design in some object-oriented language should be no more than a rather mechanical step. This indicates that this final step could be (partly) automated. Now, all three groups working on OMT, ADL and PARSE are interested in automated code generation, and, in some cases, have partially succeeded.

We will concentrate on three requirements for significant code generation from a graphical design:

- We need clear, unambiguous semantics of all our notations.
- We need a description of dynamic behaviour of objects/processes.
- Orthogonality of notations and techniques eases automated code generation.

Clearly, if we don't know for sure what a notation exactly means, we can generate no code for it whatsoever. Also, if we want to generate more code from a design than just some skeleton code (header files, configuration files, object definitions, etc.), we need a specification of dynamic behaviour of software components. Finally, the more orthogonal different techniques and notations are, the easier it becomes to generate code *per symbol/model*, which gives us more flexibility.

Design Method

As a final important aspect to graphical design techniques we would like to mention the design *method*, a series of guidelines to let a designer use a technique optimally, taking him/her from analysis all the way to implementation. Often, a graphical technique and a method for using it are closely linked together. For example, it is hard not to design in terms of functional decomposition, if task graphs are the models which are used to visualize matters.

8.2 Comparing the Three Techniques

Now that we have identified some of the important design issues for parallel, distributed software, we will see how well OMT, ADL and PARSE perform on each of them. In doing so, we will maintain the order in which they were discussed above.

Data

Clearly, data modeling is best supported by OMT. In our application's design, all data(structures) needed for the state maintenance and the reservation of resources can be modeled adequately in OMT. Some of the classes introduced can be refined even further. We could, for example, derive tables and bags from a general 'container' class, or provide manager classes with a queue to help sequentialize requests.

The use of data in OMT is modeled in STDs and DFDs. Here, the operations on data are described and the way data travels through the application.

Primarily focusing on the way data travels through an application, ADL and PARSE do very little on data structuring, apart from a possibility (we did not show this in our models) to annotate communication channels with types (ADL) or protocols (PARSE). Here, some of the structure of communicated data is revealed. About the data(structures) internal to processes nothing is specified. In PARSE, there is the notion of a data server, which is used for processes showing the kind of behaviour we would expect from a 'container' class in OMT. This is, of course, positive from an intuitive point of view, but it has a disadvantage as well, as we will see later.

We conclude that OMT is the only one of our design techniques supporting data structuring in an adequate manner. Both other techniques concentrate on the active processes in an application, while ignoring to a large extent the (structure of the) data that these processes use.

Communication

Communication structure in OMT is modeled in the object model. However, this is primarily on a *class* level. To model structure between objects, we have to use textual constraints in the form of relation keys. For example, to model the fact that every server manager object communicates to one backup manager, we can annotate the communication relation between these classes by something like $\{server - manager.index, backup - manager.index\}$. However, this may not be powerful enough to grasp more complex communication structures.

The moment of communication as well as its contents are modeled in OMT's STDs. Objects only send/receive messages when they reach a state that allows this. For example, in Figure 3, the request manager only sends the local request tables if the request table is completed; a server manager will then only receive its local request table if it is waiting for it (Figure 4).

The contents of these messages are usually operation names and parameters [1]. On the subject of DFDs (where communication contents are described as well) we come to speak later.

Serious problems arise with the semantics of communication, since OMT specifies nothing in that area. Hence, we have no means other than (textual) circumscription in STDs (such as the 'waiting for requests' state of a server manager in Figure 4) to describe if communication is synchronous/asynchronous or blocked/non-blocked/timed. Furthermore, the semantics of, for example, the one-to-many communication relation

[1] We did not show operation names in our models. Of course, in later design stages, these would have to be added to the objects.

between the request manager and the server managers in Figure 2 could be anything between a send to one of the objects on the 'many'-side and a broadcast.

As an aside, we mention here, that OOD ([1]), which is another object-oriented design technique, *does* incorporate some explicit communication notations in its diagrams. For example, synchronous vs. asynchronous, timed etc. However, the semantics of these are still largely undetermined and the notation is not orthogonal. The improvement over OMT's notations is therefore not so evident.

ADL and PARSE, on the other hand, are very rigid in their approach to communication. The structure of communication, its semantics and its contents (see previous paragraph) are all described in one diagram. Also, in ADL's STDs, the moment of communication by a process can be found. PARSE supports this dynamic behavior aspect only to some extent.

Both techniques have the means to describe complex communication structures with multiple senders and/or receivers. They support this through their explicit replication notations, by which parts of a system can be replicated. In PARSE, we can only replicate processes, which has default implications for the communication paths between them. These can then be overruled textually. In our application's design, for example (Figure 11b), we had to annotate the channels between server managers and backup managers. In ADL, we can explicitly replicate the channels, but not without replicating the processes connected by them as well. Again, this replication of processes has well-defined consequences for the semantics of communication.

However, as was discovered in the modeling of communication between the request manager and the server managers, not all complex structures can be models in ADL and PARSE. Also, we revealed a difference in their modeling possibilities (see, for example, the discussion on vectored ports/gates in Sections 6 and 7). This latter observation comes primarily from a difference in ADL's and PARSE's viewpoints as to what determines communication: *ports* (as in PARSE) or *channels* (as in ADL). We will further discuss this below.

We conclude for now, that ADL and PARSE are superior to OMT in describing the semantics of communication by using very explicit notations and assuming very rigid semantics for these. One option for improvement in OMT in this is to introduce a separate inter-object communication diagram to describe communication. Here, ADL and PARSE ideas can be used.

Parallelism and replication

OMT has no ways of explicitly indicating how many instances of an object class there are in an application, although a lot can be specified by the use of one-to-one and one-to-many relations. In Figure 2, for example, there is a one-to-one relation between file servers and server managers, which is probably sufficient. Both PARSE and ADL have explicit replication features, making it possible to exactly specify the number of instances.

Now, some of the problems with complex communication structures among multiple senders and receivers were already discussed in the previous paragraph. However, one issue has not been addressed yet. This is the situation in which a process is replicated and the instances communicate with each other, for example in a pipeline or grid structure. In ADL these are indeed the two possible structures: a replicated process can

be supplied with gates in four directions to communicate to other instances, rendering pipes or grids. In PARSE we can accomplish the same things by annotating paths with expressions. However, for more complex replication structures, for example, trees, neither of the two techniques is adequate. OMT does not explicitly support replication structures. However, a notation using generalizations and relation keys might be a promising possibility to denote these structures. Of course, the semantics of communication would then have to be determined, which, as we saw earlier, is not the case in OMT.

Orthogonality and decomposition

The most prominent example of notational orthogonality can be found in ADL. Here, communication channels are based on the notion of a queue. Whichever process wants to listen to the channel, should be given a gate to it. At the gate level, we can then describe how communication is timed. This timing and the semantics of the channel are completey decoupled: they are orthogonal concepts. Clearly, orthogonality in notations has been an important point to the developers of ADL, rendering a very clean, concise notation.

In PARSE, we have less orthogonality of notations. For example, timed communication issues are located on the path level, making it impossible to indicate *how* a process wants to communicate without specifying *where* communication goes. Besides this, separate notations are introduced for various kinds of processes (i.e. data stores, interfaces, etc.), also not based on orthogonal concepts.

As to orthogonality of techniques, we can say that, of course, PARSE has none since only one kind of diagram is supported. In OMT and ADL the relation resp. communication structure is orthogonal to the description of dynamic behaviour per object/process. OMT's third technique, the DFD plays a somewhat different role. There is significant overlap with both the object diagram and the STD. Its primary purpose seems to be supplementary, i.e. providing a different look upon an application to help construct the other diagrams.

In conclusion, we can say that ADL has accomplished quite a lot in orthogonality. This makes the technique clean and simple, semantically very clear and thus (see also below) easier to generate code from.

Automated code generation

If we look at the requirements for code generation from graphical designs, we see, that ADL fulfils all of them to a large extent: its notations are simple, semantically well-defined, unambiguous and orthogonal. Also, ADL contains a description of dynamic behaviour. Code generation from OMT is mostly limited to class frames, incorporating to some extent specializations and aggregations. With the communications part of OMT in the STDs we can do fairly little. Code generation from PARSE seems a more difficult task than from ADL, because dynamic behaviour is only specified to some extent, and communication notations are less orthogonal.

Design Method

ADL has, in contrast to OMT and PARSE, no real design *method.* The technique is there and has even been implemented in a graphical environment, but guidelines as to how it should be used to design successfully are few to none. OMT has a very explicit method of using the available techniques, especially in the analysis and initial modeling phases. PARSE has this as well, which reflects precisely the difference in the approaches that the makers of ADL and PARSE have taken. ADL was intended to be a clean technique with a small number of basic concepts, making it semantically well-defined and easier to generate code from. PARSE was developed from a software engineering point of view. Its notations are based on an inventory of intuitively more appealing notions that designers work with in practice.

9 Conclusions

Now we come to draw some conclusions regarding graphical design techniques for parallel, distributed software. One of the main points we want to make, is that OMT's suitability for developing this kind of software is limited: while excelling in data modeling, its representation of communication (especially the semantics thereof) is incomplete. Now, this is probably not a big issue in the development of conventional software. However, in parallel distributed software, we consider it important for various reasons given above.

ADL is probably the most concise, orthogonal technique of the three. Therefore, it has been able to remain well-defined and thus an excellent candidate for automated code generation. However, this does put some pressure on the extent to which ADL is intuitively appealing, a shortcoming that is strengthened by the fact that there is no real method provided for using ADL.

The presence of a method and intuitive modeling are precisely the strong points of PARSE. Besides that, PARSE has found a way (via path constructors) to determine on higher levels of design the sequencing of incoming communication. However, being based on practical, intuitive software engineering concepts, PARSE is semantically far less well-defined than ADL. Also, it has no specification of dynamic behaviour. All this makes code generation and (formal) analysis of designs difficult.

Neither of the three techniques has very sophisticated means to express geometrical communication structures of a replicated process. ADL and PARSE have some explicit notations, which are, however, of limited power (primarily pipes and grids). The OMT notations perhaps have the potential to describe these structures (by the use of generalization structures and relation keys) but lack the formal semantics for this.

If we are to develop a general technique for designing parallel, distributed software, we need to combine the strong points of all three (and may be some other ones as well) techniques used here. The question is, whether the modeling features within the object-oriented and the process-based paradigms are easy to merge. A study on combining OMT with PARSE has already been conducted in [7]. A particular interesting question is whether a new technique (or an extension of an old one) should be essentially object-oriented or process-based. Whereas for flexibility of software, object-orientation claims to be the solution, PARSE and ADL have chosen the approach of message passing between

processes, because this maps naturally onto most target hardware models. Our research right now aims at taking ADL as the technique for communication modeling, while using OMT notations for data modeling. To accomplish this, we have to investigate to what extent data modeling and communication modeling are orthogonal concepts.

Also, we ask ourselves the question whether the communication media in ADL are high-level enough. We have, for example, no explicit means for modeling remote procedure call (rpc) behavior, other than circumscription by multiple channels in combination with STDs. Our approach here would be to introduce only higher level concepts that can be captured by combining lowel level ones as well. In this sense, we can speak of a *kernel* ADL with only the low-level concepts and an *extended* ADL with more elaborate constructions.

After that, we will come to a set of rules to translate graphical designs into programming code for distributed systems, for example, a network of workstations. One thing that has to be taken into account here is the limited power of ADL's STDs. Here, nothing is specified about what goes on in a processing state. In order to generate code for these, they need to be decomposed.

References

[1] G. Booch. *Object Oriented Design.* Benjamin/Cummings, 1991.

[2] K.E. Gorlen, S.M. Orlow, and P.S. Plexico. *Data Abstraction and Object-Oriented Programming in C++.* John Wiley, 1990.

[3] J. Gray. Definition of the PARSE Process Graph Notation. Technical Report PARSE-TR-2a, University of Wollongong, Australia, december 1993.

[4] J. Gray. Definition of the PARSE Process Graph Notation. Technical Report PARSE-TR-2b, University of Wollongong, Australia, March 1994.

[5] D. Harel. STATECHARTS: A Visual Formalism for Complex Systems. *Science of Computer Programmming*, 8(3):231–274, 1987.

[6] J.H. Howard, M.L. Kazar, S.G. Menees, D.A. Nichols, M. Satyanarayanan, and R.N. Sidebotham. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems*, 6(1):55–81, February 1988.

[7] C. Knowles and P. Collingwood. Parallel Software Development Using an Object-Oriented Modelling Technique. December 1993.

[8] J. Rumbaugh, M. Blaha, W. Premerlani, E. Frederick, and W. Lorensen. *Object-Oriented Modeling and Design.* Prentice Hall International, 1991.

[9] M.R. van Steen. ADL: A Graphical Design Language for Real-Time Parallel Applications. Hamlet EUR-CS-93-06, Erasmus University Rotterdam, Department of Computer Science, April 1993.

[10] M.R. van Steen. Developing Parallel Real-Time Applications in the Hamlet Application Design Language. Hamlet EUR-CS-93-14, Erasmus University Rotterdam, Department of Computer Science, September 1993.

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