SOFTWARE ENGINEERING FOR DISTRIBUTED APPLICATIONS:
THE DESIGN PROJECT

Dr. M. Muhlhauser
Universität Karlsruhe, Institut für Telematik
D-7500 Karlsruhe, W. Germany, Phone +49-721-608-3391

Abstract

The DESIGN project combines a set of new approaches to software engineering for distributed applications. Distributed applications may thereby consist of a large, varying number of interacting processes. Specific problems encountered with the development of such distributed applications are not suitably reflected by known programming languages and software engineering environments. The DESIGN system in its current version integrates consistent approaches specifically suited for distributed application development. These approaches pertain to the areas 'language support', 'performance prediction/rapid prototyping', and 'project support environment'. Most parts of the DESIGN system have been implemented and successfully applied to first sample distributed applications.

Keywords: software engineering, distributed application, distributed programming, project support environment, modeling, performance evaluation, rapid prototyping.

1 Project Overview

The development of complex distributed applications requires software engineering techniques specifically suited for this task. An increasing need for such techniques arises mainly from two trends:

1. A small number of standards for lower layers will be widely accepted. The standardization of higher layers will consolidate to a certain extent, making heterogeneous networks operational. Along with this, users will become accustomed to thinking of the 'system' they are using being a distributed system.
2. As a result, much more system programs and application programs will be designed to run on a distributed system instead of on a single processor, i.e., typical applications will be what we shall call distributed applications. At the time of program development, the topology of the target distributed system will typically be unknown. The programming of distributed applications will be a programmer's common task, just as it is the case with sequential programming today.

In project DESIGN we addressed three key areas of distributed application engineering:

- distributed programming,
- performance prediction/rapid prototyping,
- software engineering/project management.

The DESIGN system integrates these areas with the language DC (together with an appropriate runtime environment), a modeling system for distributed simulation of distributed applications under development, and the DESIGN software production environment.

In all three areas, DESIGN contains new approaches developed to meet the specific needs of distributed application development; but moreover, DESIGN fully integrates these areas: e.g., DC contains statements dedicated to modeling, the language processor generates output for the software engineering environment, etc.

In the remainder of the paper, the language DC will be introduced first. The computer assisted model construction and the overall modeling technique will be described in section 3. Basics of the software engineering environment will be presented in section 4.

2 The language DC

2.1 Overview

DC extends the well-known programming language 'C' by special constructs for formulating and developing distributed applications and their administration. 'C' was taken as a base language in order to gain experience on

- the effort necessary to complement existing widespread languages with appropriate distributed application programming capabilities,
- the additional overhead necessary to implement a distributed runtime system,
- the advantages of using a language extension as opposed to a subroutine package.

Concerning the potential complexity of distributed applications, DC had to provide a software architecture covering top down design of distributed applications, modularization, explicit naming and handling of objects common to distributed programs (communication paths, messages, processes etc.), and dynamic changes in the number and topology of these objects at runtime. In addition, a concept for communication between the modules of a distributed application was necessary. As DESIGN supports performance evaluation, software engineering, and distributed programming in an integrated manner, a modeling concept as part of DC was developed. This modeling concept had to provide hooks for the integration of DC into the development tools of DESIGN.

Two terms are of special interest, experiment and function entity. Every incarnation of a distributed application in a con-
crete distributed system shall be called an experiment.

A distributed application in DC is composed of function entities. In an experiment, function entities represent separate sequential processes from the point of view of the operating systems of the nodes. Analogously, a distributed application program is divided into a set of function entity programs which can be conceived of as function entity types. Every function entity type is stored in a different file. In addition to the function entity type files, there exists one central message type definition file where all message types exchanged between any function entity types are declared. Optionally, the user may specify heading files containing routines and declarations which are used by several function entity types. Table 1 states the different file types that form a DC distributed application.

<table>
<thead>
<tr>
<th>File types contained in a distributed application</th>
<th>count</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>function entity type</td>
<td>1-n</td>
<td>sequential process type</td>
</tr>
<tr>
<td>message type definition</td>
<td>1</td>
<td>common messages</td>
</tr>
<tr>
<td>heading</td>
<td>0-m</td>
<td>routines &amp; declarations</td>
</tr>
</tbody>
</table>

Table 1: file types of a distributed application program

2.2 Software Architecture

2.2.1 Hierarchical modularity

Known approaches to hierarchical modular structuring decompose each system into a set of 'logical' modules or subsystems, having clear-cut interfaces hiding the internal structure of the subsystems on the top level of abstraction. In lower levels of abstraction, one can find each subsystem further divided into 'logical' subsystems, and so forth, until the bottom level is reached where subsystems are designed as the 'working' subsystems of the system.

For DC, this structuring approach has been modified. A 'logical' subsystem is not only just a conceptual collection of several further subsystems, but, moreover, is represented at its interface by a function entity. This function entity controls and manages on one hand the interface to the outside world and on the other hand the subsystems contained in it. The 'working' subsystems are of course also represented by function entities.

The interface of a function entity consists of a set of interface ports each of which has a number of valid incoming and/or outgoing message types associated with it, plus a number of arbitrary creation parameters instantiated at creation time of the subsystem.

A function entity may decide upon creation, deletion, and interconnection of its subsystems. Creation and deletion of subsystems are controlled using the operations create and terminate. If a subsystem represents only one function entity, (i.e. what we called a 'working' subsystem above), just this function entity is started by create. If a subsystem consists of a function entity together with further subsystems (which are then invisible at the current level of abstraction), create only starts the visible function entity. It is then the task of this function entity to create its own subsystems.

terminate applied to a function entity recursively stops the issuing function entity together with all its subsystems, whereas exit_fe only stops the issuing function entity, leaving possible subsystems alive, but prohibiting further administrative operations on them.

For interconnecting its subsystems, a function entity decides on the connection of the interface ports of the subsystems:

- among each other using the operation connect,
- to its own interface ports and thereby to the outside world using the operation map,
- to its so-called inner ports also by means of the operation connect. Inner ports enable message exchange between a function entity and its administered subsystems.

Besides mapping an interface port to one of its subsystems, a function entity may reserve an interface port for message exchange between itself and the outside world. This is indicated by the operation serve.

There is one common reverse operation to map, connect, and serve, called free.

For function entities with no subsystems, the only possible operation for interface ports is serve. For such function entities, serve is automatically inserted into the program text by the DC language processor.

Table 2 summarizes the administrative operations.

<table>
<thead>
<tr>
<th>Administrative Operations in DC</th>
</tr>
</thead>
<tbody>
<tr>
<td>create: start function entity representing a subsystem</td>
</tr>
<tr>
<td>create &lt;subsystem&gt; on &lt;node&gt; [parameters pl...];</td>
</tr>
<tr>
<td>terminate: stop subsystem</td>
</tr>
<tr>
<td>terminate &lt;subsystem&gt;;</td>
</tr>
<tr>
<td>exit_fe: stop this function entity, leave subsystems alone</td>
</tr>
<tr>
<td>exit_fe;</td>
</tr>
<tr>
<td>exit_as: stop this function entity and all subsystems</td>
</tr>
<tr>
<td>exit_as;</td>
</tr>
<tr>
<td>connect: link subsystem-port or inner port to subsystem-port</td>
</tr>
<tr>
<td>connect &lt;subsys.&gt; @ &lt;port&gt; with &lt;subsys.&gt; @ &lt;port&gt;;</td>
</tr>
<tr>
<td>connect &lt;inner port&gt; with &lt;subsys.&gt; @ &lt;port&gt;;</td>
</tr>
<tr>
<td>map: link interface port to subsystem-port</td>
</tr>
<tr>
<td>map &lt;interface port&gt; onto &lt;subsystem&gt; @ &lt;port&gt;;</td>
</tr>
<tr>
<td>serve: use interface port inside this function entity</td>
</tr>
<tr>
<td>serve &lt;interface port&gt;;</td>
</tr>
<tr>
<td>free: reverse connect, map, or serve</td>
</tr>
<tr>
<td>free &lt;subsystem&gt; @ &lt;port&gt;;</td>
</tr>
</tbody>
</table>

Table 2: Administrative operations in DC

Fig. 1 shows these operations in an example to illustrate the structuring concept for distributed applications. Function entities are drawn in trapezoidal shape, dashed lines surround subsystems, and solid line rectangles stand for ports.
2.2.2 Dynamics at runtime

Complex distributed applications tend to change their configuration, e.g., the number of processes, network nodes, and communication paths, not only from experiment to experiment, but also within the runtime of every single experiment. To provide for an indefinite, unrestricted number of objects and object references, a list concept was included in DC. List operations refer to the addition and removal of objects at any index position, to indexed access, allowing scanning of a list, or return the length of a list. Indexed access is indicated by surrounding the index number with the symbols "<" and ">". Lists of, e.g., function entities, network nodes, and communication ports allow these objects to change in type and number at runtime, while each object remains identifiable through its listname and index.

2.2.3 Naming rules

DC requires the explicit naming and declaration of all communication ports together with the message types exchanged via these ports, and of the subsystems administered. This makes the overall system structure and message flows visible to the programmer and to the software engineering environment, the latter supporting compile-time and runtime checks and the generation of, e.g., graphical displays of the distributed application structure and the flow of events. In addition to ports, port pointers may be declared as references to ports:

```plaintext
port <portspec> <identifier>;  
```

where <portspec> specifies one specific, a number of, or all ports to be valid objects the port pointer may point to:

```plaintext
type <port> | union "<port>..." | all
```

The message types used in the distributed application are declared in a unique message definition file. This enforces a common view of the message structures over all function entities. The code for all function entities is under control of the software engineering environment, enabling, e.g., a check of administrative operations against the interface definition of the subsystems involved.

Message variables have to be specified within the scope of a function entity. The same applies to message pointers, whose declaration syntax is fully analogous to that of port pointers (the keyword being msgptr).

The flexible declaration syntax for message and port pointers renders full selection capabilities for the message receive operation get (see next paragraph).

All possible asynchronous events, called exceptions, have to be named in a specific program section, together with the actions to carry out when these exceptions occur. The actions may only be executed at specific points in the program, indicated by the programmer with the handle-exception statement. Examples of exceptions are the disconnection of communication paths, the termination of a subsystem, etc.

Structuring rules also apply to the program text for a function entity: as shown in table 3, function entity programs are divided into five main sections.

### Table 3: Sections of a function entity type

<table>
<thead>
<tr>
<th>Sections of a function entity type</th>
<th>Representation to the 'outside world' declaration of administrated subsystems</th>
<th>Handling of asynchronous events</th>
<th>Main data and code</th>
</tr>
</thead>
<tbody>
<tr>
<td>interface configuration</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>initialization</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>exceptions</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>body</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

2.2.4 Communication

The communication concept introduced in DC was developed with regard to three requirements:
1. the broad variety of possible communication needs, and
2. the need for automated transformation into a simulation model.
3. the need for efficient communication methods, including synchronized send, monitors, but also transaction-oriented communication, remote procedure call, etc.

Therefore, requirement 2) is met too. In fact, for future DC versions it is planned to provide so-called 'packages' implementing different higher level communication methods, giving the programmer the possibility to choose at will. Section 3 will show how the requirement 3) is met.

Communication is based essentially on the two operations transmit and get, for message transmission and receipt. The basic operands are an output or input port specification and a message specification.

get enables input message selection according to reception port and/or message type. Arbitrary combinations of ports, port lists, and message types may be selected in a get statement through the use of port pointers and/or message pointers declared in the beforementioned way. The syntax is as follows:

```plaintext
get <msgspec> at <portspec> [error <rout>];
```

get selects the message to be received according to the valid message types and valid ports. The error routine is called if
an unexpected event occurs while get is carried out. Message selection by get plus conditional branches according to the type of message received provide programming features like those of guarded commands \[^{4}\], get is implemented synchronously, i.e. if no corresponding message is stored in the input ports of the function entity, the function entity waits until a corresponding message arrives. To avoid undesired waiting phases, a test function is available:

```plaintext
    test ( <msgspec>, <portspec> );
```

with `<msgspec>` and `<portspec>` analogous in syntax to the get operation. The boolean function test returns true if a corresponding message is stored in the input ports of the function entity.

To enhance the synchronization capabilities, transmit offers the option to delay the transmission of messages. Thus, e.g., a time-out mechanism can be implemented by a delayed transmit to the transmitting function entity itself. The full syntax of transmit reads:

```plaintext
    transmit <msgspec> at <portspec>
    [ delay <time> | id <ident> ] [ error <rout> ];
```

The delay, id, and error parts are optional. `<ident>` stands for an identification variable with which the delayed transmission of a message can be withdrawn using the operation:

```plaintext
    cancel <ident> ;
```

### 2.3 Compilation

The program text of a distributed application in DC is translated via the DC language processor and a C compiler into machine readable code. For an experiment in a real-system environment, DESIGN:

- controls the definition of experiment parameters, monitoring constraints, etc.
- distributes and installs the function entities on the respective target nodes, and
- supports experiment control and debugging.

In the course of the experiment, the real-system runtime environment controls the function entities, performs the administrative operations, and monitors performance data.

### 2.4 Distributed Runtime System

The distributed runtime system is the part of the runtime environment which is necessary to make a distributed application executable on a distributed system. It consists on one hand of **additions** to the function entities and on the other hand of **administration entities**. Every node included in the experiment contains one administration entity. The distributed runtime system is responsible for the execution of most of the DC specific operations (administration, communication, list operations). Complex communication protocols are involved in rendering the port administration concept safely executable; in fact, the distributed runtime system is one of the largest and most complex parts of the DESIGN system.

### 2.5 Experiences

The experiences gained with the design, implementation and usage of DC can be shortly summarized as follows:

- It needs much more than a communication concept to extend a sequential programming language to a distributed application programming language; in the case of DC, the additional enhancements included additional support for modularizationstructuring, extensive administration aids, a concept for handling dynamically alterable objects, additional typing, exception handling, and modeling support (see next paragraph).
- The runtime support for a sophisticated distributed application programming language requires complex communication protocols and substantial amounts of code, especially for administration support.
- Using a language extension as opposed to a subroutine package allows for structured programming and an amount of compile time checks and code generation that substantially facilitates the programmer's work.

### 3 Modeling

#### 3.1 Overview

With the computer assisted modeling technique of the system DESIGN, the DC program text of the distributed application under development is used as it is, translated by the DC language processor into special simulation-oriented C code. A DC distributed application model consists of three major parts to be described in the next three subsections:

- the **simulation-oriented function entities**, automatically generated out of the DC code (see subsection 'computer assisted model construction'),
- a **modeling system**, modeling the 'outside world' of a distributed application at experiment runtime, and
- a **kernel system for distributed simulation**. This kernel system for distributed simulation allows for the model of a distributed application to be distributed over a computer network.

Using the DESIGN modeling system it is possible, for instance, to run the model of a distributed application on a ten node network as if it ran on a one hundred node network; thereby, the final distributed application may be modeled although it only exists as a prototype, and there is no need to write a simulation model, neither for the distributed application nor for its 'outside world'.

#### 3.2 Computer assisted model construction

##### 3.2.1 Prerequisite features

The DC program text for a distributed application under development and the program text for a simulative model of the final version of the distributed application do not differ at all. The differences between real experiments and simulative experiments lie in the different runtime environments and in the translation by the language processor into either the real system or a simulative model. This transparency of DC source code was achieved by introducing four DC features, namely **virtual function entities**, the common message exchange basis, the distinction of computing phases and interaction points, and the explicit naming of undeveloped code.

'Virtual function entity' stands for a system service external to a distributed application, like the terminal i/o system.
the disk I/O system, a database service, a special higher layer communication service, etc. The term is used because all such services are accessed by DC function entities in the same manner as other function entities (i.e., through the operations connect, transmit, get, etc.). Different service calls are thereby reflected by different predefined message types to be sent to the service (indications by the service are handled in analogy). The transformation of a new system service into a new virtual function entity may be easily performed by the DESIGN system manager.

The concept of virtual function entities is an essential prerequisite for the common message exchange basis. This feature, which was a design principle for DC operations, makes it possible that all types of interaction between a function entity and its outside world (I/O through virtual function entities, administrative operations, test, cancel) can be internally transformed in a well-defined way into the two basic operations terminate and get.

The points in the program text where interactions (administrative operations, message exchange with other function entities, or communication with virtual function entities) are performed are called interaction points as opposed to computing phases. Computing phases are program sections which do not contain any of these operations. The effects of any actions undertaken in a computing phase remain internal to the function entity until the next interaction point is reached; the only external effects of computing phases lie in their consumption of the central resources cpu and main memory.

To recapitulate, a function entity can be viewed as an alternating sequence of computing phases and interaction points; a computing phase has no side effects but consumption of central resources. Interaction points may easily be identified because they are well-defined as language statements, even in the case of access to external services (through the 'virtual function entities'); internally, interaction points can be transformed to either a transmit or a get through the 'common message exchange base'.

The fourth feature to be explained in this subsection, the explicit naming of undeveloped code, denotes the programming rule a programmer has to conform to when he writes a rapid prototype, so that computer assisted model construction can be applied. When, in early phases of the distributed application development, the programmer outlines function entities by programming only essential parts of the function entities, i.e., those parts which form the skeletal structure of the function entity, the parts of computing phases left out have to be explicitly denoted by statements of the form

```
todevelop <ident> ' | * <description> ' * | ' 
  <unit> <mag> substitution <statement>; 
```

where <ident> uniquely identifies the section to be developed, <description> gives a verbal description of what the section to be developed is supposed to do, <unit> and <mag> represent the programmer's estimate of the amount of cpu necessary, stated as either the number of statements, or lines of code, or milliseconds of cpu. The amount of main memory occupied by the function entity is stated as one of the simulation parameters. The substitution part indicates interim code necessary to ensure the semantical and pragmatically correctness of the function entity code in the absence of the code to be developed.

Analogous to computing phases to be developed there may be message parts to be developed - i.e. in early development phases dummy messages or message parts may be used. If such dummy messages are shorter in length than expected for the final messages, this has to be stated in the program text, too.

The use of the central message definition file (cf. subsection 2.1) assures a consistent view of the abstraction level for all function entities at a certain development phase.

The relation of the explicit naming of undeveloped code to computer assisted model construction will become evident in the next subsection.

### 3.2.2 Modeling technique, simplified

In this subsection, it will be shown how the language processor modifies the source code of prototype function entities in order to construct a model of the future distributed application.

We will concentrate on the modeling technique as pertaining to the consumption of central resources, i.e., main memory and cpu. As for the modeling of disk i/o, terminal i/o, etc., the language processor mainly has to modify the messages exchanged with the corresponding virtual function entities; for the sake of shortness, these parts of the modeling technique shall be left out.

It was shown how the concept of virtual function entities and the common message exchange basis enable the view of a function entity as consisting of a sequence of computing phases and interaction points. On a simplifying level of abstraction, the modeling technique may be viewed as follows:

```
function entity

<resource usage>

computing phase

interaction point

interaction

<resource usage>

os-kernel model
```

**Fig. 2:** Message flow for the simplified modeling technique.

#### Computing Phases:

Inside a computing phase, the DC language processor adds statement to the program code which sum up the amounts of central resources (cpu and main memory) required by the computing phase. These amounts are determined by the language processor according to the types of statements and operands using predefined values specific for the type of computer to be modeled; these predefined values are generated by a dedicated calibration tool. The additional statements inserted by the language processor sum up these amounts of central resource in local variables at runtime, so that, e.g., an arbitrary number of boops can be considered. For undeveloped code the programmer's estimates are used as given in the to-develop-statement, multiplied by a machine-specific factor calculated by the calibration tool mentioned.
Interaction Point:
At the end of a computing phase, before an interaction point, statements are added for communication with a model of the local computer and its operation system, called os-kernel-model. First, a message called 'resource request' is transmitted containing the amount of central resources requested in the passed computing phase. The os-kernel-model then simulates the consumption of these central resources. Then, a 'resource acknowledgement' is sent from the os-kernel-model to the function entity. Finally, the interaction takes place as specified in the interaction point.

The technique described can be optimized further. The optimization makes use of the fact that only for interactions of type get the flow of actions of a function entity may depend on the progress of other function entities (i.e., of the time and sequence of the arrival of messages), whereas interactions of type transmit do not further influence the flow of actions of the transmitting function entity. Therefore, a function entity does not have to synchronize with the os-kernel-model before every interaction point. The optimized modeling technique, which is the one used in DESIGN, will not be described in more detail.

3.3 The modeling system
The modules of a simulation experiment are distributed among the network nodes on which the experiment shall actually run; these (physical) network nodes are called the underlying distributed system. The underlying distributed system is to be distinguished from the modeled distributed system which consists of modeled nodes and of modeled communication subsystems.

The kernel system for distributed simulation is distributed over the underlying distributed system and makes the latter transparent to the simulation model.

A modeled node consists of the following parts:
- the function entity models destined for that node,
- the local administration entity,
- the os-kernel model,
- file modules (modeling 'virtual function entities' for disk i/o, see above),
- the system workload (modeling the behaviour of interactive users and their processes),
- the application workload (modeling users of the distributed application), and
- workload generators (one for each type of workload).

The modeled communication subsystems build up the model for the network interconnecting the modeled nodes. Fig. 3 shows an example of the model of a distributed application. The figure does not show 'logical' communication links between the model parts, but illustrates the modularization from the point of view of the kernel system for distributed simulation. All the model components shown in figure 3 exist as predefined building blocks, parametrized and configured by an experiment definition tool and an experiment control tool which are part of the DESIGN software production environment (see next section). A more detailed description of the modeling concept can be found in [8].

![Distributed application model](image)

Fig. 3: Distributed application model

3.4 Kernel system for distributed simulation

3.4.1 Known approaches to distributed simulation

We decided in favour of discrete event simulation since previous studies showed this simulation method to be efficiently applicable to the modeling of distributed systems [15].

The runtimes of simulation experiments tend to be rather long for complex models. One of the known approaches to reduction of these runtimes is that of distributed simulation where the components of the simulation program are disseminated over a distributed system. The DESIGN approach to distributed simulation belongs to the class of 'distributed simulation with model distribution', where both the the kernel system for distributed simulation and the simulation model (i.e., in our case, the modeling system together with the model of the distributed application) are distributed. A simulation model consists of a set of so-called simulation modules, the kernel system for distributed simulation is formed by a set of simulation controllers. The experience with this type of distributed simulation has shown that one of its major advantages lies in the enforced modularity of the simulation program, providing clear-cut interfaces between the simulation modules.

Using discrete event simulation, the dynamics of an experiment is driven by so-called events generated and processed by the simulation modules. An event is associated with the simulation time at which it has to occur, called event time. In sequential simulation it is the task of the kernel ('runtime') system to synchronize the simulation modules, i.e. to control the order of events and to activate the simulation modules according to the event times. Prior to the activation of a simulation module, a system wide simulation time is set to the event time.

In distributed simulation, events are treated as messages being sent from the generating simulation module to the simulation module where the event has to occur. The distribution of events and the intention to have several simulation modules...
work in parallel render the task of synchronization much more difficult. There are several classes of synchronization methods [12,5]. For our purposes, the class of loose synchronization proved to be adequate. Loose synchronization methods allow every node to keep its own simulation time and give a lower bound until which the simulation time of a node can be increased. For this lower bound it must be guaranteed that no more events may arrive with an event time lower than the given lower bound. Methods for loose synchronization are discussed in [12,2,3,14], showing how deadlocks can be avoided using so-called link time messages, and how to speed up simulation experiments by time acceleration algorithms. Time acceleration algorithms are superimposed on the normal synchronization.

3.4.2 DESIGN approach to distributed simulation

The kernel system for distributed simulation used in DESIGN is based on the loose synchronization methods presented above, but with two basic extensions. One of these extensions is called central synchronization. For simulation experiments in DESIGN, the number of simulation modules is expected to be essentially larger than the number of nodes in the underlying distributed system. To support a high degree of parallelism and to keep the synchronization overhead as low as possible, DESIGN allows to run more than one simulation module on a node, thereby keeping different simulation times per module. The link time based synchronization algorithms used in known approaches had to be adapted to the requirements of the DESIGN approach. In addition, a new algorithm had to be developed for synchronization of the different simulation modules running on a single node.

The second basic extension to known approaches relates to the above mentioned ‘time acceleration’. The time acceleration algorithm developed for DESIGN gives better performance than the original proposal [2], because the number of messages per cycle is in the order of m instead of n square where m is the number of simulation controllers and n is possibly much larger number of simulation modules. In addition, it is optimized for the modelling of distributed systems in contrast to, e.g., [14].

An in-depth description of the DESIGN approach to distributed simulation and of the algorithms mentioned can be found in [11, details about the modelling approach in [8,10].

4 The DESIGN Software Production Environment

4.1 Project data

As distributed applications may be large and complex, several programmers may be involved in their development, working possibly at different development nodes. A distributed application will evolve in a series of versions. Since simple versions may be used as models for more detailed versions, the software production environment has to support the coexistence of several versions. Considering the structure of distributed applications and of experiments, every running part of the system DESIGN has an actual working context which is characterized by the tuple \((N,V,F,M,L,E,K)\) with

- \(N\) the actual distributed application (classified as 'NWA' for historical reasons),
- \(V\) the current version,
- \(F\) the actual function entity,
- \(M\) the name of the module (function entity substructure),
- \(L\) the runtime environment (real-system, simulative, ...),
- \(E\) the experiment definition,
- \(P\) the name of the programmer, and
- \(K\) the actual node in the underlying distributed system.

The first five classification points form the main structuring hierarchy for the project data, too; a DESIGN system is considered to be composed of distributed applications, which may exist in several versions, etc. The project data may belong to different levels in this hierarchy, e.g., an experiment definition belongs to the experiment-level and comprises all function entities pertaining to a certain distributed application, version, and runtime environment. Normally, a programmer's responsibility is restricted to one or few subsystems. He obtains programming instructions by the programmer of the function entity which contains these subsystems. These instructions are given on one hand verbally and on the other hand by specification of the interface description of the function entity representing the subsystem.

4.2 Software lifecycle

DESIGN conceives a distributed application development as an iterative walk through the four basic phases CONSTRUCT, COMPOSE, CONTROL, and CONCLUDE. In each phase, the programmer may choose from a number of actions. During the phase CONSTRUCT, the distributed application is specified and implemented. Examples for possible actions in phase CONSTRUCT are specify (specify and document programming instructions), edit (edit source code), propose change (propose a change in the programming instructions to the advisory programmer), etc. The phase COMPOSE serves, e.g., to perform consistency checks and to integrate, compile and install the version of a distributed application. The phase CONTROL comprises the definition, execution and monitoring of experiments, while CONCLUDE serves to analyze an experiment in order to draw conclusions for the improvement of the distributed application. Several iterative CONTROL and CONCLUDE phases may be needed in order to draw such conclusions.

Besides these phases, there is a special phase called MANAGE which can be entered from any other phase in the software lifecycle. In the MANAGE phase, a sufficiently authorized programmer may create, modify or remove function entities, versions, distributed applications, programmers' responsibilities, etc.; there are also services such as mailing, backup of files, and context management.

4.3 User interface

The DESIGN user interface is based on workstations with a high resolution monitor, multiwindowing and graphics input capabilities (e.g., mouse). The user interface guides through the phases of the software lifecycle and provides a common access method for all tools, where the parallel usage of different tools is possible by the multiwindowing feature. Commands are selected using icons, and where alphanumeric input is requested, command
boxes with preset values are available. Structuring rules apply
to the monitor display, including positioning rules for status and
message texts, help displays, and icons.

The interface was designed respecting the principle of 'no
preemption', i.e. the user is not forced to do any actions in a
certain order. This applies, e.g., to the phases and actions of
the software lifecycle and to the inputs to command boxes.

An *extendable tool interface (ETI)* provides a standardized,
fast and easy way of hooking tools into the interface (as actions
in one of the phases, see above). In fact, such an integration
process is a matter of minutes.

Figure 4 shows a typical screen layout of the DESIGN user
interface.

4.4 Tools

Most tools are specific to a certain phase and implemented as
actions in these phases, e.g.,
- a language specific editor edit in phase CONSTRUCT,
- the DC language processor compile in COMPOSE,
- graphical aids in phase MANAGE which, e.g., display the
  function entity type hierarchy,
- the tool playback in CONCLUDE, a graphical aid for event
  playback, showing a 'film' of the events which occurred during
  an experiment, or
- *debug in CONTROL*, which provides sophisticated distributed
  debugging aids.

The distributed debugging tool, too, makes use of the windowing
capabilities of the graphics workstations. It provides access
to every function entity on the network, allows inspection and
manipulation of function entities by a high level command inter-
face, and addresses the major problems of distributed debugging
(interruption, time delay, consistent view, etc.) with specific ap-
proaches.

Some tools are also specific to a certain runtime environ-
ment (cf. next subsection), like experiment definition and ex-
periment monitoring tools, or the *calibration tool* for simulative
experiments. The latter is used to calibrate a specific exist-
ing computer network node for, e.g., its execution speed, and
to create an execution speed reference table so that the corre-
sponding computer type can be named and used in simulative
experiments.

4.5 Runtime environments

Two different runtime environments are currently supported:
the *real-system runtime environment*, and the *simulative run-
time environment*. The *real-system runtime environment* is the
one which controls the normal execution of a distributed appli-
cation. It consists mainly of the following parts:
- Experiment definition: Definition of parameters, nodes to
  be included in the real-system experiment, logical names,
etc.
- Experiment control: Start, stop, status reports, etc.
- Experiment observation: distributed debugging/playback fa-
cilities.
- The distributed runtime system as described in subsection
  2.4.

The *simulative runtime environment* is used for modeling
distributed application under development. In addition to
the real-system runtime environment, this one includes the ker-
nel system for distributed simulation, the modeling system, an
extended experiment definition facility, and extended measure-
ment/calibration facilities.

5 Project Status and Outlook

The DC language processor and the distributed runtime sys-
tem have been developed and tested. Sample applications of
substantial size have been developed and demonstrated. The
implementation was based on systems of type VAX [9] with the
operating system VMS and the communication software DEC-
net.

For the modeling part of DESIGN, the kernel system for
distributed simulation is stably running for more than a year,
the modeling system implementation is also completed, except
for the workload generation. The extensions to the DC lan-
guage processor for computer assisted model construction are
still under work.

Simple versions of the experiment definition and experiment
control are available.

The software production environment is fully operational.
The project data base, the user interface including workstation
graphics and extendable tool interface are completed. The im-
plementation is based on high resolution graphics workstations
of type VAXStation. Available tools include, e.g., the DC lan-
guage sensitive editor and the distributed debugger.

For a detailed description of the whole DESIGN system, the
reader familiar with german may consult [9].

The main shortterm objective of the ongoing DESIGN project
is the final integration of the modeling system. For a follow-up
version of the DESIGN system we intend to incorporate object
oriented programming and object/relational data storage.
Fig. 4: Screen layout of the DESIGN user interface

References