USING DISTRIBUTED SIMULATION FOR DISTRIBUTED APPLICATION DEVELOPMENT

M. Mühlhäuser
Digital Equipment Corporation, CEC Karlsruhe *
D-7500 Karlsruhe, W. Germany
Phone (+49)(721)661961

Abstract

The software engineering environment DESIGN integrates several approaches for the development of distributed applications. The distributed programming language DC provides for language support. A workstation based human interface integrates programming tools such as a language sensitive editor, a distributed debugger, data evaluation tools, etc. This paper concentrates on a further approach of DESIGN: performance evaluation and prototyping on the basis of ***distributed simulation***. The use of distributed simulation allowed to make effective use of the parallelism provided by a distributed system, not only after the accomplishment of a distributed program, but from the very beginning of the development. A central goal of the DESIGN approach was *computer assisted modeling*, i.e. automatic generation of the simulation model out of the program text of a network application under development; this feature substantially simplifies performance evaluation and optimization in early development phases. The distributed simulation approach, the computer assisted modeling technique, and the modeling system for distributed applications are the main topics of the paper.

Keywords: distributed applications, distributed simulation, automatic modeling, software engineering environment.

1 Introduction

The development of distributed applications requires engineering techniques including methods and tools for performance evaluation. As existing techniques showed very limited applicability, a new approach called DESIGN has been developed. DESIGN is a distributed environment for development and performance evaluation of distributed applications.

One central part of DESIGN is the language DC. This language extends the well-known programming language 'C' [Ker78] by special constructs for formulating and developing distributed applications and their administration.

The second basic feature of DESIGN is the integration of programming support tools under a workstation based human interface.

This paper concentrates on another strong point of DESIGN: performance evaluation in early phases of development as an integral part of the programming process.

Performance optimization by simulative modeling has the following advantages:

- it can be applied in early development phases,
- the potential complexity of the models is not constrained by the underlying theory,

*On leave from University of Karlsruhe, Inst. for Telematics, (+49)(721)608-3391

- the application of simulation does not require deep theoretical knowledge.

Up to now, the two major disadvantages of simulative modeling were the high efforts necessary a) to *construct models* and b) to *carry out experiments*.

In order to improve the *experiment run time* behaviour, distributed simulation methods were chosen. The net cpu time consumption of distributed simulation systems is typically higher than that of sequential simulation systems. However, the benefits of using distributed simulation as a modeling basis are manyfold:

- Distributed programming may be envisioned to occur typically in a network of processors of different power. On one hand, the average network node will not have a very heavy cpu load. On the other hand, the total cpu requirements for a simulation run is typically very high (years of experiences with sequential simulation [WoM83] showed that this is especially true for the simulation of distributed systems). Distributing a simulation experiment is an excellent means for equalizing the load among the nodes of a network in which the application development (and simulation of the applications) takes place.
- The possible parallelism can speed up execution times; yet, the existing methods for distributed simulation did not lead to a satisfying degree of parallelism and had to be changed and adopted for our purposes (cf. chapter 2).
- Major performance degradations of simulation experiments come from the massive paging and swapping imposed by large simulation models. In this field distributed simulation can also help, because the parts of a distributed simulation model running on one node are much smaller than with sequential simulation.
- Especially for simulating *distributed* applications, the clearcut interfaces required for distributed simulation modules turned out to largely improve the modularity and readability of models. In fact, the *computer assisted modeling* concept (see below) was much easier to implement given the structuring rules of distributed simulation modules.

An overview about the kernel system for distributed simulation will be given in chapter 2, concentrating on the changes and adoptions to existing methods.

In order to reduce the *modeling expense*, it was seen as one of the main objectives in developing DC and DESIGN to enable *computer assisted modeling*. Therefore, DC contains features that make it possible to use the program text of a distributed application under development as a model, without the need for separate modeling. In particular, sceletal distributed applications (prototypes), as they exist in early stages of development, can be used for simulation experiments. Provided the programmer conforms to a special programming rule ('explicit naming of undeveloped code', see chapter 3), the DC precompiler can *automatically* generate a simulation model out of the program text of the prototype. The basic features of DC as well as the technique of *computer assisted modeling* will be described in chapter 3.

Then, the modeling of the environment of a distributed application (e.g., resources, their management, users) will be presented in chapter 4.

As the human interface and software engineering environment included in the DESIGN system are not of central interest in the context of this paper, they will be described only briefly in Chapter 5.

2 Kernel system for distributed simulation

2.1 Basic features

We decided in favour of discrete event simulation since previous studies showed this simulation method to be efficiently applicable to the simulation of distributed systems [WoM83].

The runtimes of simulation experiments tend to be rather long for complex models. One of the known approaches to reduce these runtimes is that of distributed simulation where the components of the simulation program are disseminated over a distributed system. In our approach both the simulation program and the simulative runtime environment are distributed. A simulation program consists of a set of so-called *simulation modules*, the necessary simulative runtime environment is formed by a set of *simulation controllers*.

Using discrete event simulation, the dynamics of an experiment is driven by so-called *events* generated and processed by the simulation modules. Every event is characterized by

- the simulation module at which it has to occur and to be processed and
- by the simulation time at which it has to occur, called *event time*.

In sequential simulation it is the task of the simulative runtime environment to synchronize the simulation modules and events, i.e. to control the order of events and to activate the simulation modules according to the event times of the events. 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, cf. [Pea80,Jef83]. For our purposes, the class of loose synchonization 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 lower bound. Methods for loose synchronization are discussed, e.g., in [Pea80,Bry79,ChM81,Jef83]. They are based on the requirement that every simulation module generates events in increasing order of event time. Common to all methods are the ideas of *minimum service times* and of *link time messages*.

The *minimum service time* of a simulation module denotes the lower bound for the difference between

- the simulation time at which an event is processed and
- the event times of new events generated in the course of this event processing.

Minimum service times may be different for different simulation modules.

Every simulation module is connected via unidirectional *links* to every other simulation module for which it may generate events; the links being unidirectional, a simulation module may distinguish between its *input links* and its *output links*. For an input link, the *link time* denotes a lower bound for the event time of the next event to be received on that input link. Whenever an event arrives on an input link, the link time is increased to the event time of that event. In addition, a link may carry so-called *link time messages* which have the only meaning to increase the link time for that link they are sent on.

The minimum of the link times of all input links is called *minimum link time*. This minimum link time gives the upper bound for the simulation time until which a simulation module may proceed without the risc to receive

further events with smaller event times.

In the above mentioned publications about loose synchronization methods it is shown how deadlocks can be avoided using link time messages, how to compute and when to send link time messages, and how to speed up simulation experiments by *time acceleration algorithms*. Time acceleration algorithms are superimposed on the normal synchronization.

2.2 Synchronization algorithms

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, it was decided

- to admit more than one simulation module per node,
- to keep different simulation times per module,
- to separate the synchronization and administration of simulation modules from the simulation modules themselves,
- to merge all synchronization and administration parts of one node into a *simulation controller*,
- to introduce bidirectional *channels* as the logical view of all links running between simulation modules of two nodes in either direction; a link from module A on node N_A to module B on node N_B is modeled as a link from module A to the channel between N_A and N_B plus a link from the other end of the channel to B.

Link times and link time messages are used as described, but now referring to a channel (i.e., a bundle of links) instead of to a single link. A channel is represented in each of the two respective simulation controllers by a data structure reflecting the corresponding end of the channel. Analogously, the link time for either direction of the channel is represented by a *minimal send time* on one end of the channel and by a *minimal receive time* on the other end of the channel. Link time messages and events will be both referred to as messages.

2.2.1 Main loop of the simulation controller

The main part of the program for a simulation controller is a loop consisting of three functional blocks:

- Receive processing: in this block, incoming messages, i.e. events or link time messages, are treated.
- Local and send processing: every simulation module is inspected here. The so-called *maximum advance* for the simulation module is computed (see below) and events with an event time lower or equal to *maximum advance* are processed. The simulation time of the simulation module is then set equal to *maximum advance*. A value called *next event time* for the simulation module kept in the simulation controller is set to the event time of the earliest event left in the event queue of the simulation module or to infinite if the queue is empty.
- Channel processing: in this block every channel is inspected. A value, also called *maximum advance*, is computed (see below). If the minimal send time of the inspected channel is lower than the *maximum advance* computed for the channel, a link time message with value *maximum advance* is sent and the minimal send time of the channel is set to that value.

2.2.2 Computation of maximum advance

In the following, both simulation modules and channels will be referred to as components for brevity. The computation of *maximum advance* for both types of components is based on the same algorithm. Let a simulation controller be given together with n local components, e.g., s simulation modules $C_1, \dots, C_s$ und c channels $C_{s+1}, \dots, C_{s+c}$, $n = s + c$. Then the computation of *maximum advance* is based on

- the minimal service time matrix $\mathcal{B}$ of dimension $n \times n$,
- the shortest path matrix $\mathcal{W}$ of dimension $n \times n$, and
- the next event vector $\mathcal{E}$ of dimension n.

The meaning of the minimal service time as introduced in, e.g., [Pea80], was changed for increased efficiency: the minimal service time is no more only one common value per simulation module but is bound to the output links of a simulation module. For each link starting at a local simulation module C_i and ending at a local or remote simulation module S, the minimal service time $ms(C_i, S)$ represents the minimal difference between the simulation time at which an event is processed at C_i and the event times of events generated in the course of this event processing for S. The value $ms(C_i, S)$ has to be initialized at the creation time of a link; for links originating and ending at the same simulation module, this value is not needed. For channels, all minimal service times are assumed as zero.

Two further notions shall be introduced:

- the relation $\bowtie$, indicating that there exists a direct link between two modules C_i and C_j (written $C_i \bowtie C_j$),
- the Set $R^*_{i|j}$ of remote simulation modules to which there exists a link from C_i which is part of channel C_j (with $i \le s$ and $j > s$).

On these premises, the minimal service time matrix $\mathcal{B}$ is computed as follows:

$$\mathcal{B}_{i,j} = \begin{cases} ms(C_i, C_j) & if\ i \neq j\ \wedge\ i,j \le s\ \wedge\ C_i \bowtie C_j \\ \min_{r \in R^*_{i|j}} (ms(C_i, r)) & if\ i \le s\ \wedge\ j > s\ \wedge\ R^*_{i|j} \neq \emptyset \\ 0 & if\ i > s \\ \infty & otherwise \end{cases}$$

The shortest path matrix $\mathcal{W}$ expresses the minimal differences $\mathcal{W}_{i,j}$ between the simulation time at which an event is processed at C_i and the event times of events processed at C_j originating either directly *or indirectly* from the event processing at C_i. $\mathcal{W}$ is computed directly from $\mathcal{B}$ via a shortest path algorithm. $\mathcal{W}_{i,j}$ is the minimal sum of the minimal service times over a path from C_i to C_j with z intermediate components $C_{i_1}, C_{i_2}, \dots C_{i_z}, z \ge 0$, where intermediate components have to be simulation modules. $\mathcal{W}_{i,j}$ is the minimal value of

$$\mathcal{B}_{i,i_1} + \mathcal{B}_{i_1,i_2} + \dots + \mathcal{B}_{i_z,j}$$

for all paths considered.

The elements of the next event vector $\mathcal{E}$ are determined as follows:

$$\mathcal{E}_i = \begin{cases} \text{event time of the first event in the event queue of } C_i & if\ i \le s \\ \text{minimal receive time of } C_i & if\ i > s. \end{cases}$$

Given $\mathcal{E}$ and $\mathcal{W}$, the *maximum advance* for a component C_i is obtained as

$$\min_k(\mathcal{E}_k + \mathcal{W}_{k,i})$$

Fig. 1 gives an example for the computation of *maximum advance*. The simulation modules 1, 2, and 3 are shown together with their minimal service times (value in upper half, assumed as identical for all links to simplify

matters) and their next event time (value in lower half). Channels I and II are labeled with their minimal receive times.

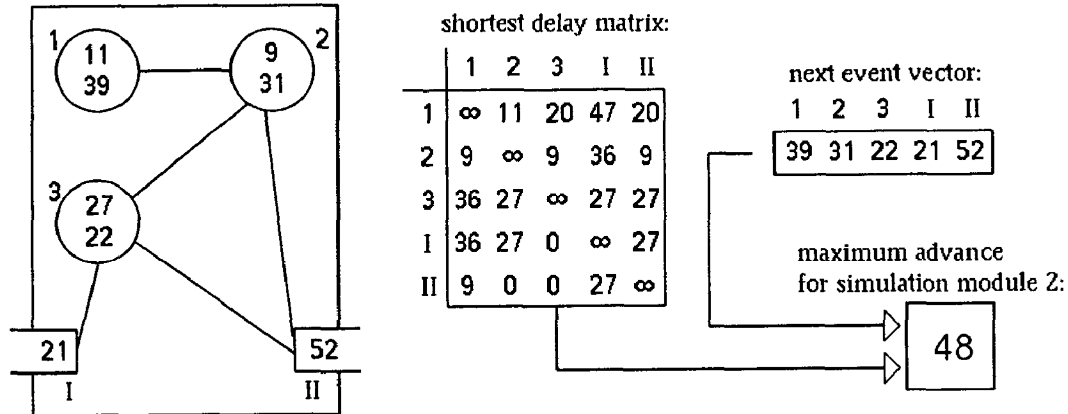

Fig. 1: computation of *maximum advance*

2.2.3 Time acceleration algorithm

With link time based synchronization, the advancement of simulation times (in our case via *maximum advance*) essentially depends on the minimal service times. This may cause significant performance degradations and imply large numbers of link time messages in phases where the simulation time difference between two consecutive events is typically large compared to the minimum service times.

A time acceleration algorithm is superimposed on the link time based synchronization and executed periodically. It computes the global, i.e. network wide next event time in a test phase via test messages. It then advances the next event times of all simulation modules to at least this value in a set phase via set messages.

The time acceleration algorithm developed for DESIGN gives better performance than other proposals [Bry79], because the number of messages per cycle is in the order of m instead of n^2 (where m is the number of simulation controllers and n the - possibly much larger- number of simulation modules). A detailed description of the time acceleration algorithm can be found in [MüD86].

3 DC and computer assisted modeling

3.1 Requirements

Concerning the potential complexity of distributed applications, DC has 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 is of great importance. As DESIGN is thought to support performance evaluation, software engineering, and distributed programming in an integrated manner, a *modeling concept* as part of DC is necessary. This modeling concept has to provide hooks for the integration of DC into the development tools of DESIGN.

In the following, every incarnation of a distributed application in a concrete 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 as *types* of function entities.

3.2 Software Architecture

3.2.1 Hierarchical modularity

DC decomposes each distributed application into a set of *logical subsystems*, represented at their interface by a function entity. Such an "administrative" function entity controls and manages on one hand the interface to the outside world and on the other hand further subsystems contained in it.

The interface of a function entity consists mainly of a set of *interface ports* each of which has a number of valid incoming and/or outgoing message types associated with it.

If a function entity controls and manages subsystems, it may decide upon their creation, deletion (using operations CREATE, TERMINATE and EXIT) and interconnection. 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 using the operation CONNECT.

There exist reverse operations and others not mentioned here. Inner ports are invisible at the interface to the outside world, meant only for message exchange between the function entity and the administered subsystems.

Fig. 2 shows the above 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.

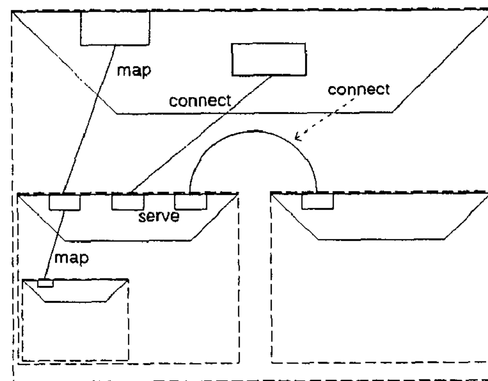

Fig. 2: Structuring of a distributed application

3.2.2 Dynamics at runtime

The DC specific *list concept* provides for an indefinite, unrestricted number of objects and object references, e.g., function entities, network nodes, and communication ports.

3.2.3 Naming rules

DC requires the explicit naming and declaration of all communication ports, message types, and subsystems administered. This allows, e.g., compile-time and runtime checks and the generation of graphical displays of

the distributed application structure. Moreover, all possible asynchronous events, called *exceptions*, have to be named in a specific program section, together with their handling. Structuring rules also apply to the program text for a function entity.

Communication: For a high degree of flexibility, a relativly simple basic communication concept was introduced. 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 and a message or message pointer.

GET enables input message selection according to reception port or message type. Arbitrary combinations of ports, port lists, and message types may be enumerated in a GET statement: either one specific port, or a set of ports, or every port may be specified valid, respectively; the same applies to message types.

Selective GET statements may be followed by a SWITCH statements, rendering a functionality comparable to the so-called 'guarded commands' [Dij75]. 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, 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. Delayed transmission of a message can be withdrawn using the operation CANCEL.

3.3 Computer assisted modeling

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 precompiler into special simulation-oriented C code. The DESIGN system combines the resulting simulation-oriented function entities with a special *simulative runtime environment* described in chapter 4, both running on top of the *kernel system for simulative modeling* described in chapter 2. The modeling system comprises models for the central resources (cpu and main memory), for disk i/o subsystems, for terminal i/o subsystems, and for the communication subsystem. In addition, the simulative runtime environment contains parts which are almost identical to those of the real-system runtime environment, like certain monitoring facilities, the function entity administration system, etc.

The above means that the differences between real experiments and simulative experiments lie in the different runtime environments and in the translation by the precompiler into either the real system or a simulative model. The DC program text for both experiment types does not differ at all. This transparence of DC source code was achieved by introducing four DC features, namely *virtual function entities*, the *common message exchange basis*, *computing phases and interaction points*, and the *explicit naming of undeveloped code*.

3.3.1 Virtual function entities

In DC, terminal i/o and disk i/o are programmed in the same manner as the communication between function entities: e.g., a function entity intending to do disk i/o does this by communication with a so-called *virtual function entity for disk i/o*. The opening of a file corresponds to the connection between a port of the function entity to a port of the virtual function entity for disk i/o where the port name reflects the file name. Putting a record onto a file corresponds to TRANSMITting a message to the virtual function entity at the corresponding port (there are predefined message types corresponding to file records). A record is read from a file by issuing a

GET on the corresponding port; a file is closed by disconnecting the ports.

Terminal i/o is analogously performed by communicating to a *virtual function entity for terminal i/o.*

The *virtual function entity for command input* enables a user to transmit simple commands to a distributed application using the standard system command level of the operating system he is using.

In addition, further system-specific virtual function entities may be developed on request, providing interfaces to, e.g., special communication services, databases, application, process control systems, etc., which are to be accessed by or incorporated into a distributed application.

Relation to computer assisted modeling:

Translating a distributed application program for real-system experiments, the precompiler translates communication with virtual function entities directly into disk i/o, terminal i/o, etc., i.e. in real-system experiments the virtual function entities do not really exist but provide a consistent view of peripherals or services to be accessed by a distributed application.

In the simulative runtime environment, the virtual function entities are represented by parts of the modeling system, i.e. the virtual function entities stand for *models* of the peripherals or services to be accessed. The common treatment for every possible communication partner, may it be another part of the distributed application, a peripheral, or a service, makes it possible for the precompiler to easily switch between real usage of peripherals/services and modeling of such (in fact, a user may request that, e.g., both the modeling of disk i/o *and* real disk i/o take place in a simulation experiment).

3.3.2 Common message exchange basis

The common message exchange basis means that all interactions of a function entity with its outside world are effected by the four communication operations TRANSMIT, GET, TEST and CANCEL. For message exchange between function entities this is trivially true; for disk and terminal i/o, command input, and specific services, the common message exchange basis is introduced via the virtual function entities. As for the administrative operations of a function entity, like MAP, SERVE, CREATE, and so on, these operations are translated into messages which are TRANSMITed to the runtime environment of a distributed application, more precisely called *local administration entity* (this term describes that part of the runtime environment which resides on the same network node as the function entity).

Relation to computer assisted modeling:

Regarding the four basic operations one finds that, for modeling purposes, TEST and CANCEL can be emulated by specific TERMINATE and GET operations exchanged with the model of the local operating system (see below). Therefore the computer-assisted modeling technique has to distinguish in principle only *two basic types of interaction*, whereas all other types of interaction (i/o through virtual function entities, administrative operations, TEST, CANCEL) base on these two operations.

3.3.3 Computing phases and interaction points

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 two 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.

Relation to computer assisted modeling:

All points in the program text of a function entity where interactions are performed may easily be identified, as they are well-defined; administrative operations and message exchange operations - dealing with either real or virtual function entities - are identified by keywords of the language DC and can be processed by the precompiler. Moreover, after translation of administrative operations, TEST, and CANCEL, the computer assisted modeling technique has to deal with no more than *two* types of interaction points.

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, and an interaction point may be either a TRANSMIT or a GET.

3.3.4 Explicit naming of undeveloped code

This term denotes the programming rule a programmer has to conform to, so that computer assisted modeling 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 sceletal structure of what the function entity is thought for, the parts of computing phases left out have to be explicitly denoted by statements of the form

to_develop <ident> '| *' <description> '* |'

<unit> <magnitude> *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 <magnitude> 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 pragmatical 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.

In order to assure a consistent view of the abstraction level for all function entities at a certain development phase, a *central message definition* is used where the types of all messages to be transmitted between function entities have to be defined.

The relation of the explicit naming of undeveloped code to computer assisted modeling will become evident in the next two sections.

3.3.5 Modeling technique - simplified:

In the next two sections, we will show how the precompiler 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 consupmtion of central resources, i.e., main memory and (mainly) cpu. As for the modeling of

disk i/o, terminal i/o, etc., the precompiler 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:

COMPUTING PHASE:

Iinside a computing phase, the DC preprocessor adds statement to the program code which sum up the amounts of central resource (cpu and main memory) required by the computing phase. These amounts are determined by the precompiler according to the type of statement 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 precompiler sum up these amounts of central resource in local variables at runtime, so that, e.g., an arbitrary number of loops 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* (described in the following chapter). 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.

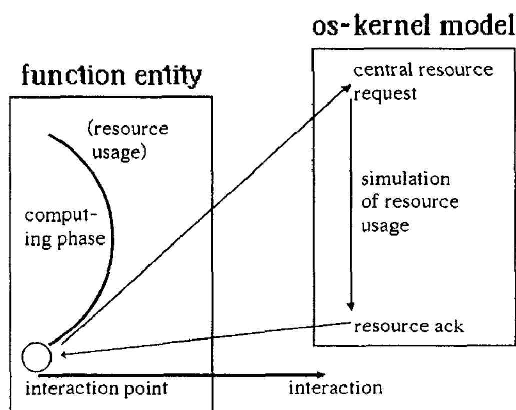

Fig. 3: Message flow for the simplified modeling technique.

3.3.6 Modeling technique - optimized:

The technique described can be optimized further. It is obvious 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 might synchronize with the os-kernel model not before *every* interaction point.

The progress of the function entities according to simulation time, however, is by far mutually dependant

through the competition for central resources. Therefore, a function entity willing to transmit a message does not know the simulation time associated with the TRANSMIT operation unless the os-kernel-model has acknowledged the resource pertaining to the preceding computing phase. The modeling technique used in the DESIGN system is therefore (roughly) as follows:

- the precompiler adds statements to the function entity code such that a function entity passes computing phases as well as TRANSMIT-type interaction points without interferring with the os-kernel model. While doing so, the function entity builds up a so-called *sequence* message consisting of 'resource requests' (for every computing phase, see above) and of 'TRANSMIT descriptions' describing TRANSMIT-type interactions. The real messages pertaining to a TRANSMIT-type interaction are immediately transmitted to their destination, but are treated 'nonvalid' by the destinated function entity, see below. Reaching an interaction point of type GET, a 'GET description' is entered into the sequence. The furhter flow of actions depends on whether or not a valid message corresponding to the GET has already arrived. If so, the GET-type interaction is satisfied and the next computing phase is handled as described above. If not, the function entity transmits the *sequence* to the os-kernel model and waits for a *resource acknowledgement* as described for the simplicied modeling technique.
- The os-kernel model simulates the consumption of central resources according to the different sequences received by the different function entities (see next chapter). Reaching a 'TRANSMIT description', the os-kernel model knows the simulation time pertaining to the corresponding TRANSMIT. At that time, the os-kernel model sends a *TRANSMIT acknowledgement* to the destinated function entity of the TRANSMIT which turns the message (that was already received from the originating function entity) from 'nonvalid' to 'valid'.

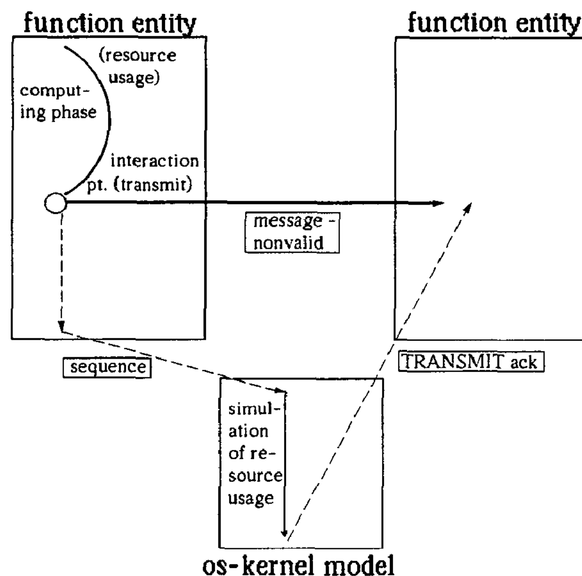

Fig. 4: Message flow for the optimized modeling technique.

4 Modeling system

4.1 Components of the simulative runtime environment

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 fore each type of workload).

The *modeled communication subsystems* build up the model for the network interconnecting the *modeled nodes*.

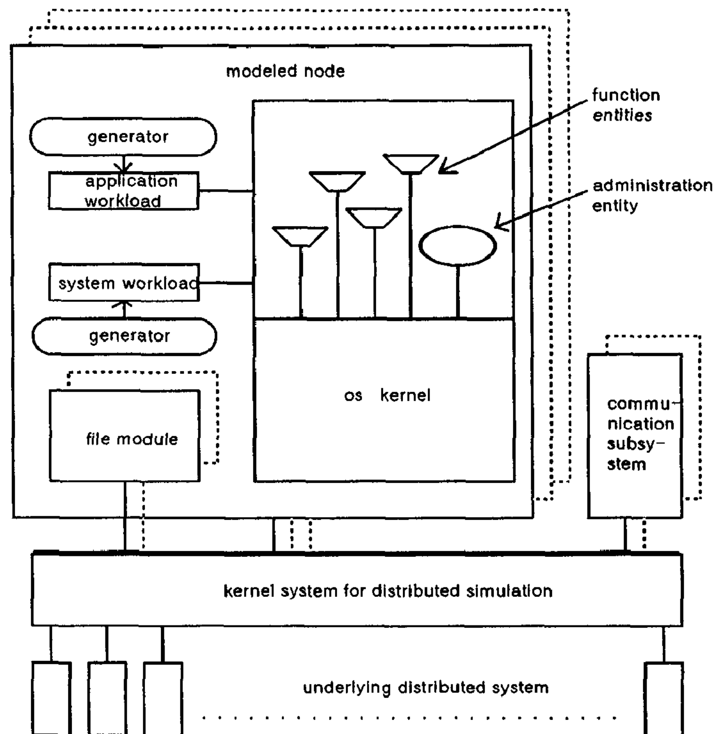

Fig. 5: Distributed application model

Fig. 5 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 5 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 [MüD86].

4.2 Os-kernel

The structure of the central part of the os-kernel model is described using a pseudo code similar to DC. The os-kernel manages all *entities* contained in a network node, where the function entities, the administration entity, the virtual function entities, and the models of the system workload processes are denoted as entities. The allocation of the central resources to all sorts of entities and the interactions between entities are controlled by the os-kernel. The resource management is controlled using the above described sequence messages, the elements of which consist of a cpu time request for a computing phase and of a description of the succeeding interaction.

Every entity is described by a struct which contains the actual sequence message, a boolean variable MATCH indicating whether the GET which terminates the sequence message has been satisfied already by a matching message, and a pointer to management data about the entity. The sequence message is stored as a DC list named seq≪≫, where '≪≫' is the DC symbol for lists.

The entity descriptions of running entities are stored in two lists, computable≪≫ and waiting≪≫. The list of computable entities is ordered according to the cpu time request of the first element in their sequence message in increasing order. Therefore the first entity, described by computable≪ 0 ≫, is the one with the minimal cpu time request to be satisfied before the next interaction point. The main loop of the os-kernel roughly executes as shown in figure 6 (cf. explanations below).

```
loop begin
     service_time = compute_svt(computable≪ 0 ≫.seq≪ 0 ≫.cpu_time,
                    computable.length);
     start_time = time;
     delay (service_time);
     if pre_empted
        then cpu_time_passed = get_cpu_time_passed (time, start_time);
        else cpu_time_passed = computable≪ 0 ≫.seq≪ 0 ≫.cpu_time;

     for ( i = 0; i++; i < computable.length ) begin
         computable≪ i ≫.seq≪ 0 ≫.cpu_time =- cpu_time_passed;
         if computable≪ i ≫.seq≪ 0 ≫.cpu_time == 0 then
            switch ( computable≪ i ≫.seq≪ 0 ≫.type ) begin
            case transmit:
                 transmit (computable≪ i ≫.seq≪ 0 ≫.interaction);
                 remove_first_element (computable≪ i ≫.seq);
                 reschedule (computable≪ i ≫);
            case get:
                 if computable≪ i ≫.match then begin
                    computable≪ i ≫.match = false;
                    remove_sequence (computable≪ i ≫);
                    receive_new_sequence (computable≪ i ≫);
                    reschedule (computable≪ i ≫);
                 end else
                    put_into_waiting_list (computable≪ i ≫);
            end;
     end;
     if pre_empted then transmit (preempting_message);
end;
```

Fig. 6: Pseudo code for main loop of os-kernel

The os-kernel first computes the expected time necessary to satisfy the cpu time requested until the next interaction point. This time, called *service_time*, is computed out of the number of computable entities, computable.length, the cpu time requested, computable≪ 0 ≫.seq≪ 0 ≫.cpu_time, and of some system parameters

(size of main memory etc.). Note that service_time is a simulation time.

The os-kernel is then suspended for a time period of length service_time. The os-kernel is reactivated, either after service_time has passed or when the os-kernel is pre-empted by a 'TRANSMIT description' arriving from a virtual function entity (for the notion 'TRANSMIT description' see 3.3.5). In either case, the cpu time requests of all entities are decreased by the cpu time corresponding to the time passed.

If no pre-emption occurred, at least the first entity in the computable list has (in simulation) reached its first interaction point. In its sequence, this interaction may correspond to either a 'TRANSMIT description' or a 'GET description'.

If the interaction point is a TRANSMIT, the corresponding 'TRANSMIT acknowledge' is transmitted. If the TRANSMIT matches a GET terminating a sequence message of a computable entity, the variable match of the receiving functing entity is set true. If the TRANSMIT matches a GET of an entity in the waiting list, the corresponding entity will be activated by the 'TRANSMIT acknowledge' sent; in this case, a new sequence message is awaited from this entity and the entity description is scheduled into the computable list according to its first cpu time request, as above.

If the interaction point is a GET and the GET was matched already, then the matching message has already activated the corresponding entity and a new sequence message can be received. After the arrival of the sequence message, the entity is rescheduled as above. If the GET has not yet been matched, the entity description is transferred into the waiting list.

If a pre-emption occurred, a 'TRANSMIT acknowledge' is transmitted according to the 'TRANSMIT description' received from a virtual function entity.

4.3 Workload

Workloads are modeled by *workload generators*, one per node for the system workload and one per node for the distributed application workload. Every workload generator is configured by the user at experiment definition time. The user may define an arbitrary set of arrival processes, each of which is characterized by a statistical distribution, by the parameters of the distribution, and by the user-defined name of the so-called *load type*.

In addition to the simple load types offered by DESIGN, the user may program its own load types in DC.

4.4 Further modules

For file modules and communication subsystems DESIGN offers standard models. To a certain extent, these models can be adapted to the distributed system to be modeled by parameters like mean access time for disks, protocol parameters and performance data for communication subsystems, etc. In addition the user may build its own models and implement them as separate modules of type 'file module' or 'communication subsystem'. Sceletal DC programs are offered for such extra modeling so that, e.g., the interface definition is compatible to that of standard models.

5 The System DESIGN

In this chapter a short overview is given about the embedding of DC and the simulative runtime environment into the system DESIGN. The main parts of DESIGN are the user interface, tools, project data and runtime environments.

Project data: As distributed applications may be large and complex, several programmers may be involved in their development; a distributed application will normally be kept in a series of versions where, e.g., simple versions may be used as simulation models for more detailed versions. Regarding these facts, DESIGN defines a so-called *working context* which is characterized by

- the actual distributed application,
- the current version,
- the actual function entity,
- the runtime environment (real-system, simulative, or other),
- the experiment definition,
- the name of the programmer, and
- the actual node in the underlying distributed system.

These classification points form the basis for the structuring hierarchy for project data in DESIGN. The DESIGN user interface centrally controls access to and usage of all project data; specific 'rights lists' control access rights of different users.

Software lifecycle: DESIGN conceives a distributed application development as an iterative walk through the four basic phases CONSTRUCT, COMPOSE, CONTROL, and CONCLUDE; in addition, there is a phase called MANAGE which can be entered from any other phase. In each phase, the programmer may chose from a number of *actions*.

Tools: The large set of tools offered to the user includes those pertaining to the runtime environments, like experiment definition and experiment monitoring tools. Most tools are specific to a certain phase and implemented as actions in these phases, e.g., a language specific editor in phase CONSTRUCT, graphical aids in phase MANAGE, a playback tool and a distributed debugger in phase CONTROL, etc.

User interface: The DESIGN user interface is based on a workstation with a high resolution monitor, multi-windowing and keyboard plus graphics input (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 alphanumerical 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. A typical screen layout of the user interface is shown in Figure 7.

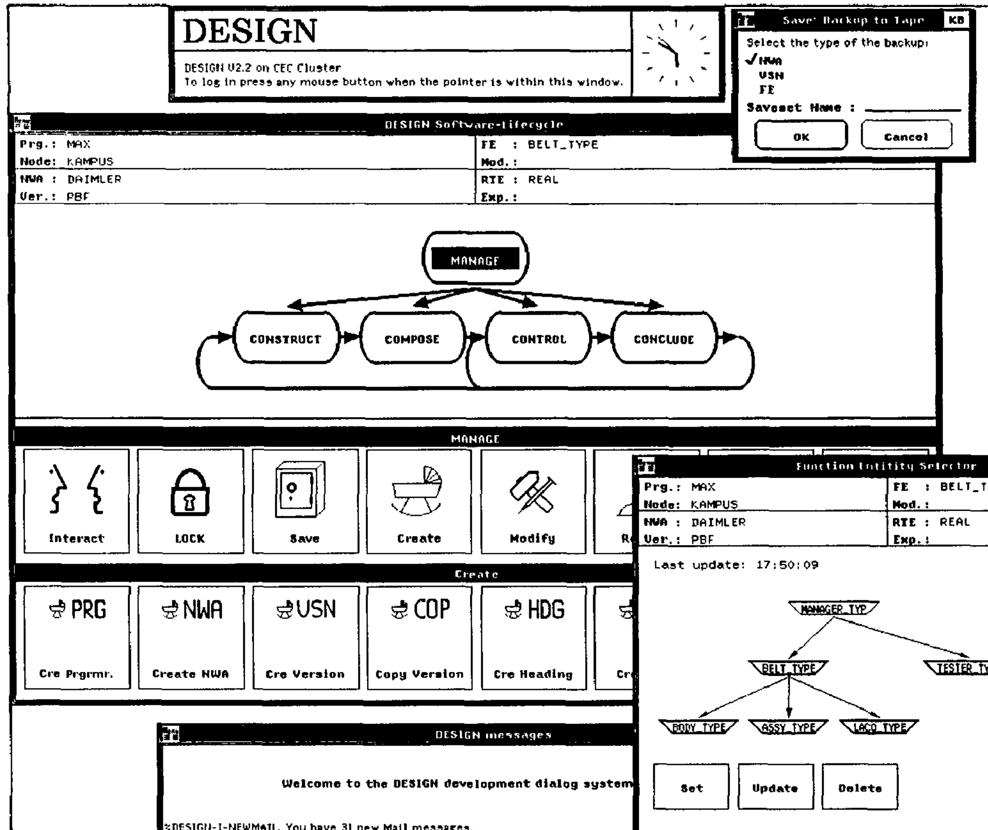

Fig. 7: Screen layout of the DESIGN user interface

6 Conclusion

The paper shows how distributed simulation can be used for the development and performance evaluation of distributed applications. A main result is that the integration of the real programming and of the simultive modeling in one common software engineering environment can essentially facilitate the use of simulative performance evaluation.

The simulation oriented features form only a part of DESIGN; the power of DESIGN lies in its integration of distributed programing, software engineering, and performance evaluation. Almost all parts of the DESIGN system according to the current concept are fully developed and operational. Experimental applications have been developed using the system. Some work is still undertaken in order to fully integrate the modeling part of design with the other parts.

References

[Bry79] Bryant, R.E.:
Simulation on a Distributed System.
Proc. IEEE, CH1445-6/79, 1979

[ChM81] Chandy, K.M., Misra, J.:
Asynchronous Distributed Simulation via a Sequence of Parallel Computations.
CACM, Vol. 24, No. 11, 1981, pp. 198-206

[Dij75] Dijkstra, E.W.:
Guarded commands, nondeterminacy, and formal derivation of programs.
CACM, Vol. 18, No. 8, 1975, pp. 453-457

[Jef83] Jefferson, D.:
Virtual Time.
University Of Southern California, Los Angeles, CS Dept.; TR-83-213, May 1983

[MüD86] Mühlhäuser, M., Drobnik, O.:
Integrated Development And Performance Evaluation Of Network Applications Using Distributed Simulation.
in: Shoemaker, S. (Ed.): Computer Networks And Simulation III,
North Holland 1986, pp. 361-383

[Ker78] Kernighan, B.W., Ritchie, D.M.:
The C Programming Language.
Prentice-Hall 1878

[Pea80] Peacock, J.K.:
Distributed Simulation Using a Network of Processors.
Dept. of CS and CCNG, University of Waterloo, Ontario, Canada; CCNG T-Report T-87, January 1980

[WoM83] Wolfinger, B., Mühlhäuser, M.:
Construction Of A Validated Simulator For Performance Prediction Of DECnet-based Computer Networks.
Performance Evaluation Review, 8/83, pp. 138-150