An Object-Oriented Model for the Construction of Dependable Distributed Systems

D.K. Hammer and O.S. van Roosmalen

Department of Mathematics and Computing Science
Eindhoven University of Technology
P.O. Box 513, NL-5600 MB Eindhoven
e-mail: wsindh@win.tue.nl

Abstract

In the context of the DEpendable Distributed Operating System (DEDOS) project an object-oriented model for the construction of dependable applications has been developed. This model is presented in this paper. It supports real-time, reliability and distribution requirements of applications. Three aspects of it are described in detail: the so called programming model, implementation model and development model. An essential feature of DEDOS is the dual dependability paradigm: real-time and dependability requirements are dealt with statically or dynamically depending on the real-time regime to which they pertain: hard real-time or soft real-time, respectively. The model supports both of these regimes.

1. Introduction

This paper describes part of the research that is presently carried out in the context of the Dependable Distributed Operating System (DEDOS) project [I], which is focussing on operating system architectures and techniques to support reliability and real-time requirements for application software. The intended applications of DEDOS are in the domain of embedded systems and process control. The driving forces behind this project are twofold: (1) the external demand for dependable distributed control systems, especially in the area of embedded systems and industrial control; the term dependability designates a number of non-functional system features on which the users of these type of systems depend: timeliness (real-time behavior), reliability, availability, safety, security [2] (often robustness against failures not covered by the fault hypothesis is added as an additional feature) and (2) the necessity to drastically increase the productivity and quality of application programming for distributed control.

For the implementation of dependable distributed applications DEDOS offers an application language framework supported by methods and tools. The programmer's perspective of this framework is also referred to as the DEDOS model. The DEDOS-model is based on the object-oriented programming paradigm. Four major reasons for adopting this paradigm can be given:
- It has the well known software engineering advantages (e.g. better support for modularity, reusability and extendibility).
- Fine grained concurrency control can be achieved by the (quasi) concurrent execution of objects. This enhances the possibility to meet the timeliness requirements.
- Objects can, in a natural fashion, be elementary units with respect to mutual exclusion, atomicity and recoverability.
- An object is more amenable to the analysis of timing properties since it is a small building block that unifies different but related functionalities.

A unique aspect of DEDOS is its so called dual-dependability paradigm. This paradigm distinguishes hard-real-time (HRT) tasks which have to strictly adhere to specified dead-lines, and soft-real-time (SRT) tasks which have more relaxed timing requirements. DEDOS has a different execution mechanism for each of these two types of task. Static (deterministic) methods are employed to design and verify the timing and reliability properties for HRT, while dynamic (probabilistic) methods are used for SRT. Timeliness of HRT applications is ensured by off-line scheduling, while a conventional on-line preemptive scheduling discipline is used for SRT applications. By allowing the execution of a combination of SRT and HRT tasks, a higher processor utilization is obtained without having to compromise on the real-time requirements. (For this purpose open time slots in the

0-8186-3015-9/92 $03.00 © 1992 IEEE
HRT schedule are reserved for SRT tasks). Fault tolerance for HRT executions is achieved by replication techniques while reconfiguration and graceful degradation is used for the SRT part [3]. The DEDOS model has to support both the SRT and HRT paradigms and has a number of interesting features. The static model is object-oriented and supports global objects for concurrency/exception atomicity, failure atomicity (recoverability) and hard/soft real-time synchronization (c.f. section 2.3). This allows the development of programs that are well structured in terms of resource (data) abstractions, inheritance hierarchies and uses hierarchies [4]. In addition, object invocations provide a convenient means to structure executions into scheduling blocks [5].

A DEDOS application is a collection of programs whose interfaces are explicitly stated in terms of import and export statements. Programs, in their turn, are collections of classes, also with explicitly defined interfaces. A program is identified with a single class, a so called program-class which is the root of the program. All other classes in the program are used, directly or indirectly, by this root and are contained in the program to enable compilation independent from other programs belonging to the application. A program can use other program-classes when these are explicitly mentioned in the program's import declaration. (Remote) Procedure Calls (RPCs) are the only means for communication and synchronization. This means that the distributed nature of the system is not visible at application level.

The dynamic programming model is based on concurrent executions rather than on concurrent heavy-weight processes or light-weight processes (threads). Since an execution is a system-wide thread of control, the problem is described in a more natural way: the programmer can concentrate on his task, the design of the control flow, rather than on implementation-issues related to this task, such as the synchronization between concurrently executing program parts, like mutual exclusion.

The first version of the programming language, named DEAL (DEDOS Application Language), that is currently being developed is an extension of C++. Therefore, the C++ terminology is adopted in this paper as much as possible. Principally, the extension allows the specification of class attributes, import/export statements and the creation of executions. The approach presently taken in this language development activity is a purely pragmatic one: to get experience with the requirement for a DEDOS language. In the long term DEAL will evolve into a high-level language that allows the expression of timeliness, reliability, distribution and concurrency requirements in an coherent and integrated fashion.

Many other research projects are concerned with timeliness, reliability, concurrency and distribution issues. A good overview of programming languages for distributed systems is given in [6] and [7].

One of these projects is the distributed operating system Clouds [8]. Some object oriented languages are being developed for this operating system (distributed Eiffel [9], CC++ [10]). There are many similarities between Clouds and DEDOS, in particular the execution mechanism of Clouds resembles that of DEDOS. Clouds is, however, not intended for real-time environments. Two related projects that are concerned with real-time applications are the CHAOS project [11], [12], and the ARTS project [13] carried out at Carnegie Mellon University. An object model for the ARTS real-time kernel has been developed as well as a particular language implementation (RTC++). Although the execution model of ARTS is different (ARTS uses the concept of passive and active objects) and dynamic scheduling techniques are employed, some of the concepts that are developed in ARTS are of direct interest to the DEDOS project.

Other research is particularly concerned with the efficient implementation of concurrency and distribution. The object-oriented language POOL [14] addresses the syntax, semantics and implementation aspects of parallel objects. The language SINA [15] concentrates on interface specification, concurrency, synchronization and communication in a distributed environment. This language also provides a flexible form of atomic transactions as a support for fault tolerance. Finally the ORCA language approaches the efficient implementation of distributed programming by a model that is based on shared variables [7], [16], [17]. Shared variables are in fact data-abstractions that also provide concurrency atomicity. For efficiency reasons shared variables are distributed to all clients, thus, fault-tolerance can easily be supported. The consistency is enforced via a 2-phase update using a reliable multicast protocol.

In the following three chapters the programming model, the implementation model, and the development model will be discussed. The first describes the semantic aspects of the DEDOS application programming language, the second describes how the DEDOS specific features are implemented, the third describes the methods and tools that are provided to transform programs into executable images.
2. The programming model

We distinguish three models each related to a certain perspective on an application program. The static model, the configuration model and the execution model. These are concerned with the static data-structure, dynamic (run-time) data-structure and the execution-behavior of programs, respectively. A summary of the DEDOS programming model is given in Table 1.

2.1 The static model

In DEDOS an application is a collection of DEAL programs. A program is the unit of compilation. It contains a set of class declarations in accordance with the object oriented paradigm. A program contains one special class, called the root of the program. This class is also called a program-class and the program can be identified with this class. There are two types of relations between classes: inheritance and use-relations. Whether a class may be inherited or used by another class depends on its visibility at the point of declaration. Two situations have to be distinguished in this context with regard to the distance between classes (see figure 1). A class is near to a program or class therein if its definition is contained in that very program, a class is remote when its definition is given in another program. A class can only inherit from a near class. A class can use a remote class only if the latter is a program-class or is used in the interface of a program-class and if it is made visible by an appropriate import statement. The interface of a program-class is a set of functions and attributes that are exported explicitly by the corresponding program (see also section 2.2). In this sense a program-class is no different from other classes. Within the limits set by the scope and visibility rules a common name-space is established for all programs belonging to an application.

If a class uses a near class it can invoke member-functions declared public for this class. Such an invocation is named a near call. If functions of a remote class are called we speak of a remote call. The syntax and semantics (within the fault-hypothesis) of both calls are identical (location transparency), although there are some restrictions on the types of parameters that can be passed with a remote call. Two situations can be distinguished with regard to remotely accessible functions. (1) The function is exported by a program-class. (2) The function is an operation on an exported attribute of a program-class; the class of the attribute must declare the operation as public. In either case the class with the remotely available member-function is called a service class and must be declared as such (see section 2.4). Member-functions of a service class must satisfy the above mentioned restrictions on function-parameters.

Certain restrictions on the usage of language constructs are imposed to simplify analysis of DEDOS programs that have to satisfy strict timing (HRT) properties. For instance, repetitions must be bounded by constants or constant-expressions such that an upper-bound for the number of times the body is executed can be inferred at compile-time. Recursion is not allowed at all in such programs.

2.2 The configuration model

The collection of compiled programs belonging to an application, together with a system description file is called a system. When a system is started, a program-instance, or process, is created for the root-program of the application which is identified by the system description file. The root-process may create instances of other program-classes. Information on the location of such programs, the processor(s) on which they are to be loaded (global schedule) and, in case of HRT programs, the local execution schedules are contained in the system description file. A process keeps an identifier of the processes it creates and may pass this identifier to other processes.

Thus, a process is a (run-time) instance of a program-class just like a regular object is a (run-time) instance of a class. The interface description of a process is given in the corresponding program, that of an object in the corresponding class. The only substantial difference between a process and an object is that a process carries with it a set of objects. Some objects in this set may have global scope within the process, i.e. all objects in the process may be able to use them. The only application-wide global objects are processes and exported process-attributes. Upon the creation of a process an initialization-function called the constructor, is executed. This function comprises the creation and initialization of all objects with global scope. In the first version of the DEAL language, the role of constructor is played by the function main() of an extended C++ program.

In the same spirit as for classes and programs, the distance of objects observed from a process is defined as near or remote depending on whether an object is created by the same or by another process.

When a uses relation exists between two classes, the corresponding objects are said to have a client-server relation. A client can access public members of
Processes are marked HRT or SRT in the system description file, and DEDOS will start executions accordingly. An execution may perform a DEDOS system call to start additional executions. Such a system call returns an execution-identifier. HRT executions can only create additional HRT, and SRT only additional SRT executions.

The execution of a process starts with the constructor which initializes all objects with global scope. To enable static analysis, a HRT process may only start additional executions and may only create new processes in the constructor. Real-time execution according to an off-line calculated schedule commences after initialization. The total number of HRT executions remains constant from then on. SRT executions can be created liberally at run time. Since they are not as time-critical as HRT executions they are scheduled dynamically (on-line) using a preemptive priority scheduling policy.

Apart from the executions that originate from a process itself, executions may enter via remote service-requests (c.f. figure 2). HRT executions may only enter HRT processes and SRT executions only SRT processes. An exception to this rule is made for HRT-SRT communication for which special arrangements exist [18]. Thus, although the total number of HRT execution remains constant during the life-time of the application, this number may vary per process.

Within a process, each execution is managed by a separate system entity, called a thread. A thread keeps track of that part of the context of an execution which is unique to that execution. Data shared by all threads of a process are managed by that process. Always, when an execution performs a remote call and thereby enters a server process, a new thread in this server is allocated to the execution to handle the service request (figure 3). Since entire processes are allocated to a processor, all threads within a process reside on the same processor. The assignment of processes to processors is called global scheduling.

A thread is the unit of dispatching. The assignment of processor-time to all threads running on a processor is called local scheduling. Naturally, threads are executed (quasi) concurrently. As mentioned before, SRT threads (carrying SRT executions) are

![Figure 1. DEDOS object types.](image)

The central concept in this model is that of an execution. An execution is a causally related, system-wide sequence of instructions. For example, when an execution encounters a remote call, it moves from the client to the server-process until control is passed back (figure 2). This server process may be located on the same or on a different processor. Hence, an execution is not confined to a process or processor.

![Figure 2. The DEDOS programming model: executions](image)
Threads are the units of light-weight concurrency.

Threads are execution contexts: Thread = Process ∩ Execution-entry.

Figure 3. The DEDOS programming model: threads

preemptable and are scheduled on-line. HRT threads (carrying HRT executions) are dispatched as specified by an off-line calculated schedule. When an execution performs a remote call or system call the corresponding thread is blocked until the execution returns with the reply. For HRT executions only a limited number of additional suspension or synchronization points exist: boundaries (i.e. entry and exit points) of special near objects. Two types of such special objects can be distinguished, namely: (1) delimited objects and (2) system objects. An object can be declared delimited by the programmer for the sole purpose of introducing synchronization points on its boundaries. System objects are global resource-management objects provided by DEDOS, such as timers, device-managers and semaphores. The HRT-scheduler attempts to calculate a feasible schedule given the synchronization points.

An object with a global scope (within one process or with respect to the entire application), called a global object, is shared between all executions. An object with a local scope, a local object, is always created for the exclusive use by a single execution. Local objects are only visible in the block in which they are declared. They will be automatically removed when they get out of scope.

These rules simply and directly link the scope of objects within a process or set of processes (global/local) with their accessibility from executions (shared/non-shared). (See figure 1).

Two types of global objects exist: static and dynamic ones. Objects declared outside all scopes in a program are static and global. An object created by an execution is dynamic and may be global (e.g. if it is created through the C++ construct new). Dynamic global objects are used to support variable-size data-structures. They can only be made visible to other executions through a global reference or pointer. HRT executions may not dynamically create global objects for reasons of efficiency and predictability (see table 1). Since, for HRT programs, the temporal extent (number of iterations) and, related to this, the spacial extent (size of data-structures) of each program part must be know at compile time, there is no need for dynamic global objects. They can all be replaced by static ones. Thus, in HRT executions global objects may only be static.

Static global objects are created at system initialization and exist for the complete life-span of the system. Dynamic global objects can only be removed by statements to that effect inserted into the program by the software developer. It is his responsibility to produce systems that deal with such objects correctly: there are no runtime facilities in DEDOS for guaranteeing absence of orphan-objects or dangling pointers. Hence, these issues of safety provide additional reasons to prohibit the use of dynamic global objects in HRT.

The communication paradigms used in DEDOS can be summarized as follows. Processes and objects communicate, synchronously, via (remote) calls. Executions communicate, asynchronously, through global objects. For the synchronization of executions system-objects, semaphores, are available.

Since global objects are shared between different executions, measures have to be taken to guarantee object-consistency. For this purpose object-classes can be given the attribute atomic, implying concurrency and exception atomicity of its instances. Concurrency atomicity implies that each operations on a shared object is carried out completely or not at all, with a result identical to a situation where operations on the object are carried out sequentially (serializability). Exception atomicity, in addition, requires this to be true in the presence of failures that do not affect consistency of memory. In addition to concurrency atomicity DEDOS supports the declaration of objects which are recoverable or failure atomic (see Figure 1). Such declarations guarantee persistence of the object-state reached after an operation even in the presence of failures that do affect memory-consistency (crash failures). DEDOS takes care that access to an atomic or recoverable object by different executions appears mutually exclusive, in the described manner, with the understanding that access to a shared server is also considered access to the client.
2.4 The DEAL language

In first version of DEAL a number of new constructs that implement aspects of the DENS model have been added to the C++ base-language. These constructs can be categorized according to the following concepts:

1. Process interface definition.
2. Concurrency atomicity.
3. Creation of multiple executions per process.
4. Synchronization points.

The process interface-definition is given in the program heading. This heading contains export and import blocks. Classes and objects mentioned in these blocks must be of service-class type. Thus the interface definition, but also atomicity and synchronization, are effectuated by introducing special classes through the insertion of class-specifiers. Finally, an execution-statement is introduced to enable the initiation of additional executions. At this stage no constructs for reliability (i.e. exception atomicity and recoverability) have been incorporated. It should also be noted that the initial implementation of the DEAL preprocessor focuses on HRT-programs.

The main purpose of the present language implementation is to gain experience, such that a more complete description of the language requirements can be obtained at a later stage. Therefore, the design strategy that has been adopted is to minimize the number of deviations from the C++ syntax and semantics, and to keep the DEAL-specific elements as syntactically isolated as possible. This considerably reduces the magnitude of this first implementation effort. The new language constructs will be briefly discussed here. Syntax rules are given in the usual C++ notation. A more extensive description is given in [19].

Program heading: within an application, each program has its own name: the program-class name. A program defines a main-function which is the constructor of the program-class, a number of functions which may be exported and a number of global objects which may be exported as attributes. The program heading has the following syntax:

```
program:
  program-head export-block , , import-block , , declaration-list
```

```
program-head:
  program typedef-name ;
```

The program heading forms the beginning of each DEAL program and may not be omitted.

Export block; on the program level members may be exported using an export block (zero or one per program):

```
export-block:
  export { member-list , , }
```

```
member-list:
  member-list , , member-declaration
```

An exported service may either be a function or an object. Functions and objects are exported by including their specification in an export block. Exported objects are global as if they were declared in the outermost scope of the program. The class of an exported object must be a service class (see below).

Import block; a program-class may be imported by including its name in an import block. A DEAL program may contain zero or one import blocks.

```
import-block:
  import { typedef-name-list }
```

```
typedef-name-list:
  typedef-name-list , , typedef-name
```

Special classes: classes can be declared as system, service, atomic and/or delimited. Only instances of service classes may be exported. Atomic classes are executed concurrency atomic. The entry and exit points of instances of a delimited class and a system-class are synchronization points. Non-delimited classes may not use delimited ones.

It should be noted that these concepts are not orthogonal. For instance, service classes are by their very nature delimited and the delimited attribute may be omitted in their declaration. Syntax rules for these special classes are:

```
special-class-spec-list:
  special-class-spec-list , , special-class-specifier
```

```
special-class-specifier:
  system | service | atomic | delimited
```
3. The implementation model

Implementation issues pertaining to scheduling, atomicity and recoverability will not be discussed in this paper. They are discussed in [5], [18] and [20]. The present description focusses on the memory-management and communication aspects of DEAL.

3.1 The memory model

In the previous section two entities were introduced that keep memory management information: processes and threads. The type and purpose of the memory areas that are assigned to each of these two entities is called the memory model. Per process a code area and a global-object area (heap) is assigned. The heap contains static and, in case of SRT processes, dynamic global objects. All threads in a process share these code- and global data-areas. In addition every thread has its own stack for local objects, and naturally its own register values, including the program counter and stack pointer (figure 4). Access to secondary memory (e.g. disks) and devices is managed by the process via system objects. This implies that file- and device-objects are global, and that the state of each of these objects, as observed from different threads in a process, is identical.

3.2 The communication model

As explained in section 2.1, the distance of an object has no effect on the syntax or semantics of a call-statement. The DEAL preprocessor, automatically generates the correct implementation. Remote object-invocations are implemented using the remote procedure call (RPC) inter-process communication paradigm. Each process has at least one mailbox for sending and receiving service-requests. If both HRT and SRT requests can arrive (in case the process enables HRT-SRT interaction [18]) two separate mailboxes, one for each type of request, are used. A process has communication ports that are linked during system initialization to mailboxes according to its export and import declarations. A port is nothing more than a reference to a mailbox and is introduced to allow for dynamic client-group membership and efficient client or server relocation. The concepts of port and mailbox are analogous to the UNIX sockets and message queues, respectively [21]. For the communication with remote objects,
There are chiefly two aspects to DEDOS that make the system-development process a non-standard one. First, DEDOS introduces concurrency and requires proper object management for concurrently executing threads. Second, DEDOS attempts to guarantee timeliness for HRT applications through their deterministic execution according to a schedule which is determined off-line. The impact of the first aspect on the development process is that programs have to be written in a language fit to describe aspects of concurrency. Since the DEDOS application programming language is based on an existing sequential language (C++), programs have to be pre-compiled. The second aspect, timeliness, has a much more profound influence on the development of applications. The programmer has to provide information on the timing-requirements imposed by the environment. In addition, pre-compilation and post-compilation analyses have to provide supplementary information on execution times and resource requirements to enable the off-line computation of a schedule.

The approach presently taken in DEWS is to separate the functional issues from the temporal ones. This means that the programmer, at first, will develop programs and concern himself only with the functional demands on the application. Subsequently, tools are at his disposal to analyze the timing-properties of the environment and the developed programs and to calculate a schedule. If a feasible schedule is found, this means that the timeliness properties of the implementation have been successfully verified against the timing properties of the environment. In case a schedule cannot be found, the programmer has to, partially, redesign his application to eliminate possible bottlenecks. Naturally, through experience, a programmer may learn to successfully anticipate and avoid problems in the second stage during the first. A description of the scheduling model can be found in [5].

Figure 6 shows the development process of DEDOS systems. Extended C++ programs are translated by a preprocessor into regular C++ programs. DEDOS specific features are translated into appropriate sequences of system invocations, with the use of the DEDOS library. For HRT programs, the preprocessor also delivers an execution graph.
The pre-processor extracts the precedence-relations from the HRT programs and constructs the execution graph. The system-developer is then required to supply timing information from the environment, such as external event-timing and action time-windows. Using the C++ compiler output and the execution graph produced by the pre-processor the timing analyzer will compute for all objects execution times of their member-functions. Finally these execution-times are combined into times for non-interruptable program blocks, so called scheduling blocks. The off-line scheduler combines all timing information, and attempts to find a feasible schedule with the help of a heuristic search algorithm. When the scheduler succeeds it produces a system description file. This file contains the global schedule and a local-schedule description-table per processor.

5. Conclusion

In this paper a description is given of the DEDOS-model intended for the development of object-oriented dependable applications. A first version of the DEDOS application language DEAL, which implements this model has been described in detail. Presently, work is in progress to built tools that enable the calculation of off-line schedules, such as a execution-graph generator and a timing analyzer. Extensions of the language for the support of stringent reliability requirements are also being studied at present. At the current stage of the research, the functional and timeliness properties of an application are developed separately. This is not considered the ideal situation. One of the research goals is to have DEAL evolve into a high-level language that supports the integrated development of functional, timeliness, reliability, concurrency and distribution properties of an application.

Table 1. The DEDOS programming model: summary

<table>
<thead>
<tr>
<th>Unit of development</th>
<th>Process</th>
<th>Execution</th>
<th>Thread (Context)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Program</td>
<td>Program</td>
<td>Implied</td>
</tr>
<tr>
<td>Timeliness type</td>
<td>HRT/SRT</td>
<td>HRT/SRT</td>
<td></td>
</tr>
<tr>
<td>Scheduling domain</td>
<td>Global</td>
<td>Local</td>
<td></td>
</tr>
<tr>
<td>Scheduling type</td>
<td>HRT:Static</td>
<td>HRT:Fixed</td>
<td>Implied</td>
</tr>
<tr>
<td>Number of instances</td>
<td>HRT:Dynamic</td>
<td>SRT:Dynamic</td>
<td></td>
</tr>
<tr>
<td>Memory type</td>
<td>Primary</td>
<td>Primary</td>
<td></td>
</tr>
<tr>
<td>Memory allocation</td>
<td>Code &amp; Heap</td>
<td>Stack &amp; Reg</td>
<td></td>
</tr>
<tr>
<td>Lifetime</td>
<td>HRT:Application</td>
<td>Pooling</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SRT:Dynamic</td>
<td>Process</td>
<td></td>
</tr>
</tbody>
</table>

References