A Real-Time and Distributed System with Programming Language Abstraction

Erhan Saridogan
Department of Software Research and Technology
Turkish Navy, Software Development Center
saridogan@usa.net

Nadia Erdogan
Department of Computer Engineering
Istanbul Technical University
erdogan@cs.itu.edu.tr

1. Abstract

As processing and time requirements of computer systems increase over borders of single processor architectures, it is becoming more and more attractive to use distributed computing with additional real-time capabilities. In several cases, traditional programming languages have become insufficient to build distributed systems easily, especially when real-time issues and basic software quality factors such as reliability, correctness, robustness, ease of design, development, testing and maintenance are concerned. In this paper basic issues relevant to distributed systems are reviewed, a concurrent, object-oriented, real-time and distributed programming language, CORD-PL, with its supportive run-time system is introduced and its features are described. The new system provides an efficient solution for especially command and control systems by embedding distribution and real-time issues within the programming language structures.

Keywords: concurrent, object-oriented, real-time, distributed, programming language

2. Introduction

Real-world systems consist of many distributed components that are asynchronously interacting with their environment. When distributed computing and concurrent processing are considered, it can be observed that conventional programming languages have become insufficient due to inconvenient structures for inter-process (IPC) and network communication. These languages may be efficient for software architectures that use single processor. However, when it is necessary to distribute an application over a computer network, programming tends to become a burden on developers because of difficulties in using the underlying system facilities. Therefore it becomes essential to use a special layer, a kind of *middleware*, for decoupling applications from the underlying operating system and networking.

Generally, applications are developed using a classical language and several user defined libraries. In this case, the importance of software characteristics, methodology and development issues get the focus. One of the most suitable styles is object-oriented programming. The distributed form of this approach requires communication between distinct objects on different nodes. Tightly or loosely coupled computer systems may be used depending on the inter-object communication load. Real-time applications that require time-critical input and output operations are particularly difficult to design and implement. Such systems may also be required to have features like high reliability, efficiency, robustness and real-time. Therefore, distributed real-time programming languages have evolved to meet those requirements within language constructs, providing a high level interface.

One important application domain is the industrial and military command and control systems which require efficient infrastructure capable of handling large amount of high frequency data. Automated control systems collect data from various sensors or input devices, evaluate them and remotely control some actuators, preserving real-time constraints. Although it is possible to implement such systems using conventional methods, utilizing an efficient middleware mechanism is more attractive as it is also important to decouple application software from the underlying operating system and networking. The CORD System introduced in this paper proposes a solution for especially this domain without an explicit middleware approach, but a programming language with a special run-time support system.

3. Distributed Real-Time Computing

A distributed real-time system is usually desired to be fault-tolerant, reliable and efficient in both computing and communication. In addition to basic software engineering qualities like modularity, extensibility and reusability, programming languages to be used to develop such systems should have some extra features as reviewed below.

3.1 Object-Oriented Programming Languages

Object-oriented programming (OOP) languages have special constructs which divide programs into smaller portions called objects that can only be accessed via methods that are defined in their interfaces. OOP languages have some common properties like modularity, data abstraction, automatic memory management, class definitions, inheritance and polymorphism as specified in [1]. Languages having all properties are called *object-oriented*, while *object-based* languages have the first four properties. In the context of distributed computing, objects are classified either as activities or data. Many concurrent languages like Mentat [2], RTC++ [3], Concurrent Smalltalk [4] and Eiffel [5] have chosen to use objects as processing elements that are subject to distribution. On the other hand, Linda [6] and SR [7] use data objects while POOL-T [8] uses both types.

3.2 Concurrency

Concurrent languages use special constructs for creating processes depending on the underlying operating system. Some systems allow threads to be used within a process to support higher level of concurrency. Mutual exclusion and synchronization mechanisms must be provided to prevent concurrent access to critical resources. It is usually the programmer's responsibility to implement shared data protection and process synchronization through facilities provided by the operating system.

3.3 Communication

In a concurrent environment objects implemented as processes use IPC mechanisms of the underlying operating system. Concurrent languages like Mentat, ES-Kit C++ [9] and Pearl [10] have special constructs that provide communication in a synchronous or asynchronous manner. Ada, which is not a distributed language, but a concurrent one, uses rendezvous mechanism, like SR, to exchange information between tasks. In a distributed environment, objects communicate using messages that are passed through various network layers. Distributed operating systems such as Mach [11] hide the network level communication so that all programs seem to execute on a single machine. Common Object Request Broker Architecture (CORBA) [12] is a widely accepted and rapidly developing standard for commercial products used in implementing distributed applications over heterogeneous computer networks by using object-technology with almost any high-level language.

3.4 Real-time Aspects

Some mission critical systems require real-time constraints to provide fast response to events occurring at non-regular rates. Soft real-time systems do not fail if a response time constraint is not met, but performance is degraded. Hard real-time systems have to meet deadlines strictly, otherwise system failure occurs. Firm real-time systems are somewhere in between, where low probability of missing a deadline can be tolerated. Generally, hard real-time systems are designed to work with special hardware due to the importance of deadlines. Timing constraints for such systems are predicted off-line, as in RT-Synchronizer [13], and sufficient resource is allocated for unpredictable events.

4. CORD Programming Language: CPL

After a thorough study of the available research work, we have decided to combine several features required by a real-time programming language into one programming language as an extension to C++. Since a higher level of abstraction to logically connect distributed processors considering real-time requirements is needed, a special layer, a kind of middleware, running on top of the operating system is built. This layer is called the *CORD-RTS* standing for *Concurrent, Object-oriented and Real-time Distribution Run-Time System*. Therefore the language is given the name *CPL* (The CORD Programming Language).

4.1 The CORD System

The CORD System is designed to allow

programmers to develop distributed soft-real time applications in CPL that are independent of operating system and network topology. Figure 1 illustrates the system architecture with a sample CPL program.

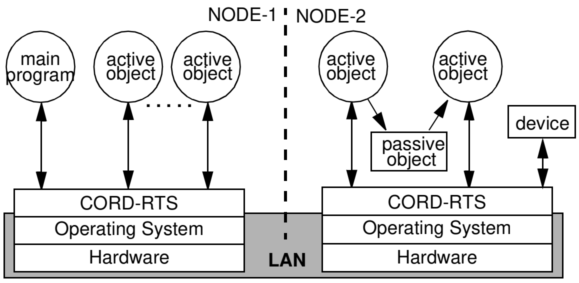

Figure 1. The CORD System Architecture

The CORD Run-Time System (RTS) **:** Each node runs a copy of the RTS whose components are illustrated in Figure 2 and described below:

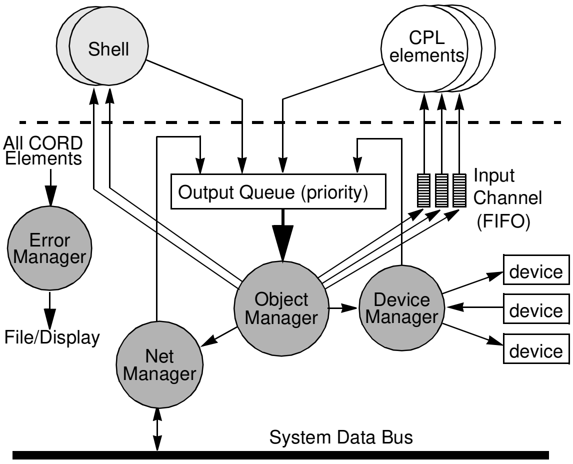

Figure 2. The CORD System Layout

Object Manager is the general controller which manages nodal communicaiton, keeps track of running programs, classes and their instances. The manager maintains a distributed database for programs and keeps it up-to-date at all times. When the manager is initiated, it creates the nodal passive data storage as a shared memory segment and the Object Output Queue which is a priority-based, one-way communication channel. The manager provides message-based communication between program components and controls the publish-subscribe mechanism providing registration and distribution of published object data. It carries out a special hand-shaking mechanism when an object is moved from one node to another.

Net Manager handles access to the network by considering the CORD network. When the manager is initiated it reads a configuration file to obtain the relevant node addresses and creates a database for them. It uses an initiation parameter to indicate the time, in milliseconds, to wait before packing messages into a network datagram for a specific destination node.

Device Manager is optionally executed on a node, if that node is to be used for device access. This manager keeps track of all the installed devices and the registered device objects, opens and sets the devices for reading or writing. It coordinates multiple accesses to a device enabling shared and remote use. It also keeps a list of active objects and their methods for a specific device object data for publish-subscribe mechanism. When new data is read from a device, the manager calls the registered method of the subscribed active objects. This subscription mechanism is separated from the Object Manager in order to speed up device data distribution.

Error Manager has to be initiated before the others with an option indicating the location of an error report, which may be a disk file, standard output or both. All elements of the CORD System report their errors in a specific format to this manager.

A *Shell* is provided to interpret user commands interactively to provide system control and monitoring. Users can open any number of shells to get information about the system and status of programs, classes and objects.

Main program **:** A CPL program is a collection of executables which constitute a global, distributed application. Therefore, each CPL program must have a main module, like a conventional program, which is executed on the RTS. This module, actually an executable, registers itself and all its classes to the RTS, and then creates the necessary objects. Those objects may later create other objects over the network. Terminating the main module results in the termination of all its class instances, regardless of their location.

Class Servers **:** In order to create active objects as processing units, a server process is used from which child processes are forked. There are as many active class servers as the number of active classes declared in a program. The CPL preprocessor creates an executable program for each CPL active class as a server. When a main program is loaded, all of its classes are registered to all CORD nodes. The Object Manager of each node activates the class servers, making them ready to create objects. Object creation requests causes the class server to fork a child process which registers itself as a new active object.

***Objects* :** There are three new types of objects in CPL, in addition to regular C++ objects. These are instances of active, passive and device classes that are described in the next section.

4.2 Classes

CPL introduces three new types of classes in addition to regular C++ classes. These are *active*, *passive* and *device* classes, from which active, passive and device objects are created respectively. CPL compiler also supports compile-time, static and public inheritance for these classes provided that the classes are of the same type.

***Active class* :** Active classes are implemented using Unix-like processes for class servers. Instances of this type of classes are the primary processing elements capable of calling methods of other objects. Active object methods are called remotely by using only special messages generated by the CPL preprocessor, however CPL programmers just issue a normal method call. Active classes may have static or dynamic data parts that are specified in the class definition. Dynamic data with the specified size is manipulated by the object and stored in a reliable storage on the node, controlled by the Object Manager. Static data elements are stored inside the object context. During object migration, the dynamic data part is copied to a new location in the destination node and a new active object is created initializing from the data copied to that node. These classes have priority values for scheduling that are applied to the objects when they are created. There may be any number of time triggered methods called periodically at specified intervals. Thread-based timers with millisecond accuracy are used but scheduling issues are left to the operating system. Published methods are listed in the class interface to be used by other classes. If subscription to a specific method of a class is indicated, then the object created from that class subscribes automatically to the related class methods. When a subscribed method is published by the producer object, the indicated method of the subscriber is called by the Object Manager. Active class body has a *main* part that is executed right after the elaboration. Active objects can be accessed from any node of a distributed environment provided that the necessary naming and scope resolution is achieved.

***Passive class* :** Instances of passive classes are used as a means of data storage and are not capable of calling methods of other objects. These objects can be accessed within the context of an active object located on the same node, providing very fast read or write functions. It is up to the programmer whether or not to set an explicit lock for accessing data by means of the two macros, CPL_LOCK and CPL_UNLOCK, implemented using semaphores.

***Device class* :** A device class defines a standard and coherent interface for input and output devices. The class definition specifies the type of the access, priority, data type, buffer size and speed of data transfer. All device controls are performed by the nodal Device Manager. The manager can also provide active object subscription mechanism for read access. An active object, that needs to be triggered by a device input, subscribes for that device class method with its own method. An active object can read a device data whenever it wants, by specifying the age of the data. If the age of the data maintained in the Device Manager buffer is older than the requested age, then the device is read asynchronously. Otherwise, current data is returned to the caller.

4.3 Communication

CPL object communication can be synchronous, asynchronous or broadcast. These modes are implicitly set by the CPL compiler. Actually, programmers never use and see messages; instead, they use the method calls in the class definitions. All method calls, replies and published methods are converted into system level messages with priority, source object, destination object and a time stamp. Therefore all method parameters have an indicator, *in*, *out* or *inout*, to specify the type of communication. If a method has only *in* parameters, or no parameter, it causes an asynchronous, or one-way, call, from the caller to the callee. If there is at least one *out* or one *inout* parameter, then the call is said to be synchronous, or two-way and blocked. The caller has to wait for a reply from the callee for the specified amount of time before raising an exception. Subscription is a way of asynchronous method invocation without having to know the source objects. A publisher broadcasts its method, as a message, to all nodes, and each Object Manager checks its subscriber list for this method and distributes it to each of the subscribers.

Active objects convert method calls into messages indicating the destination and send them to the RTS. The sender object does not have to know where the receiver object is located, or in what state it is. The

RTS finds the location of the destination object and forwards the message either to the object itself (if it is located on that node) or to the RTS on the node where the object resides. The RTS on the destination node transfers the message to the object.

4.4 Real-time Issues

CPL and its accompanying run-time system is designed to develop soft real-time systems easily. Therefore, some linguistic timing constructs are defined in CPL in order to specify real-time constraints at statement level in millisecond domain. Special language structures such as timed loops, time-triggered method calls and action blocks are provided. Process scheduling is left to the existing operating system.

As expected from a real-time programming language, CPL is a strongly typed language, enabling exception handling, dynamic memory management, abstract data type support and modularity.

Real-time filters are used inside the CORD System. Each manager checks the time-out value of a message to decide if there is sufficient time for communication delay and processing. Otherwise, it does not forward the message and sends a system reply to the sender.

All active objects send their messages to the RTS with an assigned priority ranging from 1 to 10 and a time-out value defining the validity period of the message. The programmer should assign suitable priorities and time-out values to method calls explicitly for efficient resource utilization, otherwise system defaults are used.

Priority inversion feature is provided for the objects that are running with lower priority. When a high priority message is received, the receiving object increases its scheduling priority up to the priority of the incoming message. After the message is processed, the scheduling priority is lowered again.

4.5 Distribution

CPL object creation is performed in a similar way used in C++ constructors with additional location information as a string. Objects use four-level naming convention. Of these, the first level uses program identification which enables multiple copies of the same distributed program to run simultaneously. The second level is the class type, the third one is class identification, and the fourth level is the object identification. In order to speed up processing, numeric representation of logical names are transparently used within the system.

Migration of active objects is performed by stopping the process, copying the contents of the dynamic data and starting a new process at the new location. For this purpose, active classes declare if they want to use dynamic data storage which is in a safe memory area.

4.6 System Characteristics

CPL applications are isolated from the computer infrastructure. However, standard C++ libraries as well as operating system calls can directly be used in addition to the Application Program Interface (API).

The CORD-RTS heavily relies on a globally accurate system time as it processes the time stamps of each message for time-out evaluation. Each node has to be synchronized up to a certain level for assigning time stamps. The current development environment uses NTP 4.0 over ATM providing synchronization up to 100 microseconds.

The CORD-RTS must be initialized before a program is executed. The target network system configuration can be specified in a special instantiation file which is read during initialization. This procedure includes defining the logical network and starting up the necessary managers.

CPL objects are created global to all nodes with a unique identification. An object can be accessed by any active object within the same network. This seems in contradiction with some of the software engineering rules like abstraction and information hiding. However, CPL classes use the same visibility and access rules as C++ so that an object is visible only in the block it is declared and can only be accessed via its methods.

CPL enables programmers to develop a distributed program without concentrating on the low level communication primitives and the physical layout of the system. Programmers write CPL modules as if they were writing a regular C++ module, either using full capabilities of C++ or CPL specific keywords and structures for distribution. CPL modules are then fed to the CPL compiler to generate separate client and server stub code to access the RTS. The resultant modules are compiled with an existing C++ compiler to generate the necessary executables for each active class and for the main module. These files are then loaded onto the target nodes and executed.

4.7 Keywords

It is becoming a common practice to use keywords rather than calls to library functions, in order to make programming process less problematic. Besides, learning the rules of a language is easier than learning a given library. Hence, we added extra keywords, as described in Table 1, to the existing C++ grammar.

Table 1. CPL Keywords

Keyword	Meaning
active, passive, device	Each keyword defines a CPL class type.
declare	A block of global declarations to be used by each class.
inherits	A list of CPL classes used for public inheritance.
uses	Specifies a list of CPL class to be included by a CPL class.
includes<F>	F is the name of a file to be included during preprocessing.
priority <P>	P specifies the processing priority of the objects created from an active class, ranging from 1 to 10.
data	A list of data member declarations, for both active and passive kept inside the object context.
dynamic <B>	A list of active class data member declarations mapping to the safe memory whose amount is specified as B in bytes.
methods	A list of CPL methods to be called by active objects.
in, out, inout	CPL method parameter specifications.
periodic <T>	A list of parameterless active class methods to be activated at every T milliseconds.
publish	A list of active class methods, without any body, as an interface to publish mechanism. These methods are not activated unless they are called explicitly from one of own class methods.
subscribe-to	A list of declarations indicating which method is to be called when the specified publish-method of another active class is actually published.
body	Each CPL class must have a body part that contains the implementations of its constructor and call methods.
class_main	Active classes may have main parts in the class body containing any number of statements to be executed right after object elaboration.
program	Specifies the main module of a distributed CPL program.
read-write	Specifies the device access type either for reading or writing.
buffer <B>	B indicates the number of elements to be held in object buffer.
speed <S>	S indicates the device data transfer rate in bauds.
do-every <T>	Denotes a timed loop executed every T milliseconds.
do-until <T>	Denotes a timed loop executed until current time equals T.
on "S"	S specifies on which node an object is to be created.
timeout { ... }	If a time-out occurs the statement inside block are executed.
retry <N> { ... }	If an error occurs during a call of maximum N times, the statements inside the block are executed.

Sample CPL class declarations are shown below:

```
declare {
   typedef int Natural;
}

active class Class_B inherits Class_A {
priority 5;
data:                 //within an object context
   int          mem_size
   Natural      num_elem;
dynamic 1024:         //amount of safe memory needed
   Link_List          element_list;
methods:
   Start();                       //asynchronous call
   GetNumOfElem(out: Natural N);  //synchronous call
   Compute(in: int X; out: int Y);//synchronous call
   Calculate(inout: int Z);       //synchronous call
   Insert(in: int Num);           //asynchronous call
periodic 1000:              //millisecond
   Do_Processing();
publish:
   NewData(int Val);        //broadcast when called
subscribe:
   Compute to Class_C.NewData;//own method to another
};
```

```
body Class_B {
GetNumOfElem(out: Natural N) {
   N = num_elem;
}
...//other method bodies
class_main:
   cout << "This is class B| << endl;
}

passive class Class_P {
data:
   float        values[100];
methods:
   Insert(in: int Key, out: float Value);
   Get(in: int Key; out: float Value);
};
body Class_P {
Insert(in: int Key, out: float Value) {
   CPL_LOCK;
   values[Key] = Value;    //regular assignment
   CPL_UNLOCK;
}
Get(in: int Key; out: float Value) { ... }
}
device class Serial_Comm {
read int;            //data type to read from device
priority 4;          //message priority
buffer 10;           //number of elements in buffer
speed 19600;         //data transfer rate
};                   //Implicit method: Read_Data
```

Object creation and method calls:

```
Class_A   obj1; obj1.Bind(); //bind to an existing one
Class_A   obj2(initial_val) on "NODE_4"; //new object
Class_A*  obj3 = new Class_A on"NODE_4"; //new object
obj1.GetA(val, PRIO_3, 200) timeout { ReportTimeout(); }
obj1.GetA(val, PRIO_4, 50) retry 4 { ReportError(); }
```

Timed loops:

```
do {
   Sensor.Check_Status(status);
} every 1000;     //activated every 1 second exactly

do {
   a.Read();
   b.Write();
   CORD_RTS.Sleep(1000);   //API call
} until 20:45:00.0;        //continue until this time
```

5. Performance Analysis

During the system development, various tests were performed in order to discover the system bottlenecks and time consuming parts. Table 2 shows some performance figures in microseconds obtained by averaging the time consumed by one object calling another object's method.

The first test environment consists of two Sun Sparc4 (100 MHz) workstations with SunOS 4.1.3 on 10 Mbs ethernet. The second one is a stand-alone Sparc notebook (70 MHz) with Solaris 2.5 and the third one consists of four Sun Ultra60 (300 MHz) workstations with Solaris 2.5 on 155 Mbs-ATM network with LAN emulation. The test results have shown that inter-object communication can be reduced dramatically if passive objects are used. However, active object communication latency increases as the number of active objects requiring parallel execution increases.

Table 2. Performance Figures (in microseconds)

Test	Sparc-4	Sp.Book	Ultra-60
Asynchronous (one-way) method call, same node	850	1300	550
Asynchronous (one-way) method call, different node	2400	-	1100
Published method, same node	900	1300	600
Published method, different node	2200	-	1200
Published device data, same node	1100	1500	700
Synchronous (two-way) method-call, same node	2600	3500	1550
Synchronous (two-way) method-call, different nodes	6700	-	2250
Reading passive object with Lock/Unlock	180	220	50
Reading passive object without Lock/Unlock	0.18	0.25	0.15
Asynchronous read from a local device	2200	2400	1800
Active object creation, same node	< 8000	< 10000	< 5000
Passive object creation, same node	< 4000	< 5000	< 2000
Active object migration	-	-	< 15000

6. Fault Tolerance

The CORD-RTS itself can be considered as fault tolerant as all RTSs are synchronized with each other. Restarting an RTS on a node causes it to request program, class and object database update from other RTSs. Active class servers acting as hot stand-by servers and shadow objects increases fault-tolerance. In case an active object fails, its server can detect this immediately and recreates a new object. In case of a node failure, a mechanism at the application level which detects the fault can easily re-initiate the active objects from the class servers, constructing the object from its last saved state. An implicit mechanism is not provided in the current version of the CORD-RTS.

If an active object crashes, it has to be recreated immediately and its previous internal state have to be retrieved. This can be achieved by keeping the state variables outside the object context. Passive objects or dynamic part of class declaration can be used for this purpose. Replicating this data over the network and synchronizing them is another part of this study and shall be implemented in future versions.

7. Conclusion

Using linguistic mechanisms for embedding real-time constraints and distribution simplifies program design and implementation. Such a capability even reduces the need for system level programming skill enabling the programmer to concentrate on functional behavior rather than complex, low level communication. For that reason, the ability to express distribution issues and real-time constraints through language constructs is focused. Military or industrial command and control systems with soft real-time requirements, distributed knowledge-based systems or parallel computing are potential areas for CPL implementation. Since CPL provides an abstraction over the CORD-RTS and the operating system, it is always possible to easily port application code written in CPL to new platforms. Simple modifications to the CORD infrastructure will be sufficient for adaptation.

8. Reference

[1] *Object-Oriented Software Construction*, B. Meyer, Prentice Hall, Englewood Cliffs, N.J., 1988.

[2] A.S. Grimshaw, *"Easy-to-Use Object Oriented Parallel Processing with Mentat"*, IEEE Computer, May 1993.

[3] Y. Ishikawa, H.Tokuda, C.W.Mercer, *"An Object-Oriented Real-Time Programming Language"*, Computer, Oct. 1992.

[4] Y. Yokote, *"The Design and Implementation of Concurrent Smalltalk"*, World Scientific, Vol.21, 1990.

[5] B. Wyatt, K. Kavi, S. Hufnagel, *"Paralellism in Object-Oriented Languages: A Survey"*, IEEE Software, Nov. 1992.

[6] P.G. Robinson, J.D. Arthur, *"Distributed Process Creation Within a Shared Data Space Framework"*, Software-Practice&Experience, Vol.25(2),Feb. 1995.

[7] G.R. Andrews, *"The Distributed Programming Language SR - Mechanism, design and implementation"*, Software-Practice & Experience, Vol.12(8), Aug. 1982.

[8] P. America, *"POOL-T: A Parallel Object-Oriented Programming"*, Research Directions in Object-Oriented Programming, B.D.Shriver, P.Wegner, MIT Press, Cambridge, Mass. 1987.

[9] K. Smith, A. Chatterjee, *"A C++ Environment for Distributed Application Execution"*, Tech.Report ACT-ESP-275-90, Micro-electronics Computer Technology Corp., Austin, Tex. 1990.

[10] A.D.Stoyenko, W.A.Halang,*"Extending Pearl for Industrial Real-Time Applications"*, IEEE Software, July 1993.

[11] D.Kirschen, *"An Overview of the Mach Operating System"*, Operating Systems Technical Committee Newslettter, 3(2), 1989.

[12] Object Management Group, *The Common Object Request Broker: Architecture and Specification Rev.2.1*, Aug. 1997.

[13] S.Ren, G.A.Agha, "*RTsynchronizer: Language Support for Real-Time Specifications in Distributed Systems*", ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems, June 1995.