# SOLVING TIME CONSTRAINED PROBLEMS ON THE GRID

**Nguyen Tuan Anh**
University of Technology, VNU-HCM

## 1.INTRODUCTION

High performance computing (HPC) during the pass few years has been considerably changed toward parallel distributed computing. High-cost special purpose systems have been replaced by clusters, network of workstations and distributed computing systems that span the Internet. The emerging of Grid computing [**Error! Reference source not found.**,**Error! Reference source not found.**] raises many questions on how to exploit the seamless performance of such computational environments. Old fashion programming needs to be modified to adapt to the new issues that do not exist in the conventional environment: heterogeneous, high latency, dynamic and volatile, unstable and unstructured, etc. The old resource-centric approach in which the user requests the execution on some specified resources has been step-by-step replaced by the new service-centric approach in which the user requests for some services (computing, storage,etc.) regardless of the location of resources. These changes lead to the need for new tools and new programming paradigms that are able to adapt to the environment for performance.

Many researches on application adaptivity to the heterogeneous environment have been conducted, focusing on various aspects from scheduling such as real-time CORBA[**Error! Reference source not found.**], heterogeneous task mapping [**Error! Reference source not found.**,**Error! Reference source not found.**], performance evaluation [**Error! Reference source not found.**] to programming methods in multiple variant programming [**Error! Reference source not found.**,**Error! Reference source not found.** ]. These contribute an important part in the HPC domain.

In this paper, we address the adaptivity in our framework for developing time constrained applications-applications which require that the solution be obtained within a user specified amount of time. We first briefly describe our previous works in section 2. These works are divided into two parts: the parallelization scheme providing a method to express time constrained problems and the POP-C++ tool following the service-centric approach to provide an object-oriented infrastructure for HPC. Based on these researches, in section 3, we illustrate how to use our framework for expressing time constrained problems. Section 4 presents experiment results on an emulated heterogeneous environment. Finally, section 5 gives some conclusions and our future works.

## 2.OVERVIEW OF THE PREVIOUS WORKS

We overview our previous works: the parallelization scheme and the POP-C++ programming tool for the Grid. Details of these works are described in [**Error! Reference source not found.**,**Error! Reference source not found.**,**Error! Reference source not found.**].

### 2.1.Parallelization scheme

The parallelization scheme provides users an algorithmic framework to describe and to solve their time constrained problems. In this scheme, we only address a class of problems with known complexities.

A parallelization scheme consists of a decomposition tree (DT) and a set of decomposition dependency graphs (DDGs).

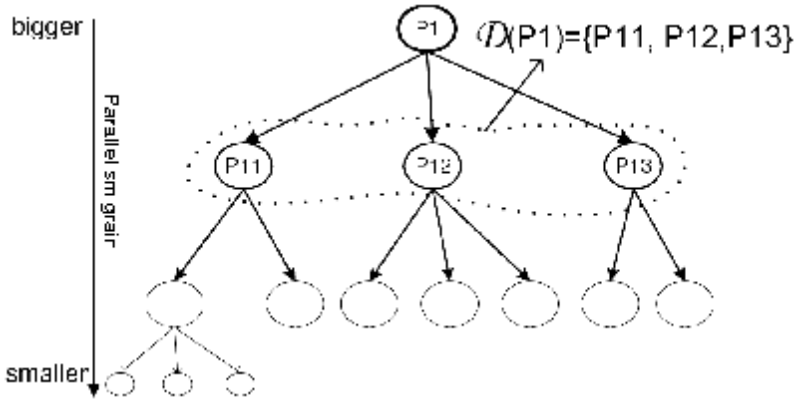


**Figure 1.** Decomposition tree

### *2.1.1.Decomposition Tree*

Decomposition tree (DT) (Fig. 1) is a tree that defines a hierarchical diagram to divide the user's problem into smaller problems. Each node of DT represents a sequential problem or sub-problem. A decomposition step is a process of dividing the problem (a node in DT) into sub-problems (its direct child nodes). The user starts at the root of DT, which is the original problem, derives new child nodes (sub-problems). The decomposition step is recursive on each node to produce new child nodes (smaller sub-problems).

### *2.1.2.Decomposition Dependency Graph*

While the decomposition tree gives an overall view of the parallelization process at different grains, the Decomposition Dependency Graph (DDG) shows the structure of parallelization. DDG is a direct acyclic graph defining the order of solving for a given set of sub-problems (Fig. 2) in each decomposition step.

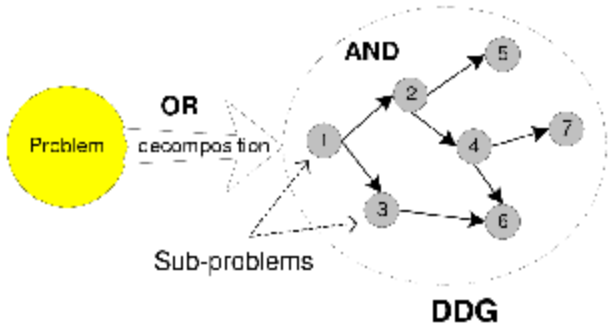


**Figure 2:** Decomposition dependency graph

### *2.1.3.How time constrained problems is solved*

We find an acceptable solution that satisfies the given time constraint T by using try-and-decompose on the DT. Try step: starting from the root, compute the needed computing power to solve the problem sequentially based on the complexity and the time constraint T; allocate the resource for the problem. If the resource is not available (try fail), Decompose step will be executed: replace the problem by a set of sub-problems (its child nodes in DT); compute the time constraints for the sub-problems and repeat the try step to solve those sub-problems in parallel. If all sub-problems are independent (the DDG is empty), the time constraint for each sub-problem is also the time constraint T of its parent in DT. A method to evaluate the time constraints of sub-problems based on the criterion "find the time constraints for sub-problems to minimize the

maximum computing power of sub-problems required on the computational resources" can be found in [**Error! Reference source not found.**].

### 2.2.POP-C++: requirement-driven parallel objects

Based on the nature of objects: each object is an relatively independent entity that can interact with other objects, we have developed a parallel object model for distributed HPC. The parallel object is a generalization of the traditional sequential object with the ability to describe its requirement during the life time through the object description (OD). A parallel object can be located on a remote machine in a separate memory address space. During the execution of the application, parallel objects can be dynamically created or destroyed. In order to allow the programmer to use high-level descriptions of parallel programs (not at the message passing level) the semantics of objects and methods invocations have been extended. Six different semantics can be attached to a method invocation (synch/asynchronous and sequential/mutex/concurrent) [**Error! Reference source not found.**, **Error! Reference source not found.**].

The parallel object model leads to the implementation of POP-C++. POP-C++ programming language extends C++ to support parallel objects. Subtracting the use of global variables, all C++ objects can be implemented as parallel objects without changing the application semantics. The object description describes the resource requirements (computing power, memory needed, network bandwidth) of the parallel object. It can be parameterized and is associated with each constructor of the object. This information will be used by the POP-C++ runtime system for the resource discovery, the resource reservation and the resource allocation. POP-C++ compiler translates the POP-C++ codes to the ANSI C++ codes which in turn will be compiled to the machine executable codes by a C++ compiler (e.g. Gnu C++). Communication between parallel objects via method invocations uses TCP/IP with Sun XDR and is transparent to the user.

## 3.THE FRAMEWORK

We built on top of POP-C++ a framework for time constrained applications using the parallelization scheme. In this framework, two types of objects: sequential objects and parallel objects co-exist and co-operate to the execution of the application. Sequential objects are used as skeletons for constructing nodes of decomposition tree and the decomposition dependency graph in the parallelization scheme. It is also responsible for creating the problem's parallel object. Parallel objects represent the real problems (or sub-problems) to be solved.

### 3.1.Expressing time constrained problems

The skeleton, illustrated in Fig. 3 consists of two main classes: `DTreeNode` and `ProbObj`.`DTreeNode` is a sequential class representing a node (or a problem) of the decomposition tree (DT). Each `DTreeNode` object associates with at most one parallel object of type `ProbObj` that implements the problem solving on a distributed resource.

- The user gets his problem solved by following the four steps bellow:
- Create the parallelization scheme.
- Set the time constraint.
- Instantiate the solution.
- Execute the parallelization scheme.

The rest of this section will explain these steps.

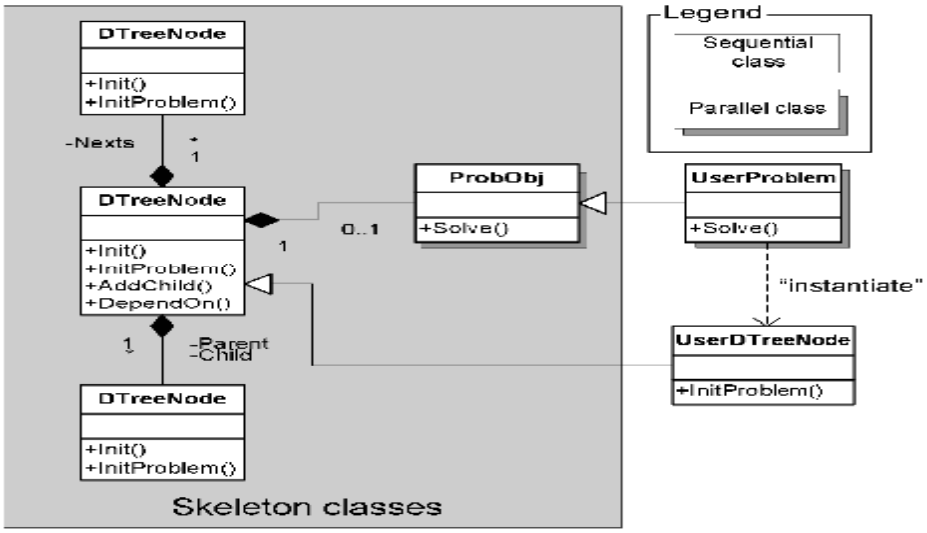


**Figure 3.**The UML class diagram of the framework

### 3.2.Creating the parallelization scheme

The user creates the DT by creating DTreenode objects and defines the parent-child relationship by calling the method `AddChild()` on each node:

```
void DTreeNode::AddChild(DTreeNode *child);
```

.The decomposition dependency graph (DDG) of child nodes within a decomposition step is constructed by defining the relationships "`DependOn`" between two nodes of the same parent:

```
void DTreeNode::DependOn(DTreeNode *prior);
```

The user can control the parallel efficiency for each decomposition by invoking the method `SetCoeff` on the parent node:

```
void DTreeNode::SetCoeff(float coeff);
```

The complexity for each `DTreeNode` should also be specified as an argument of the `DTreeNode` constructor.

### 3.3.Setting up the time constraint

After the parallelization scheme is built up, the user sets the time constraint using `DTreeNode::SetTimeConstraint`. This method only needs to be invoked on the root node of DT. For all non-root nodes, the time constraints, when necessary, will be automatically computed based on the time constraint of the root, the DDGs and the problems' complexities.

### 3.4.Instantiating the solution

This step is just a simple call to `DTreeNode::Init` on the root of DT.

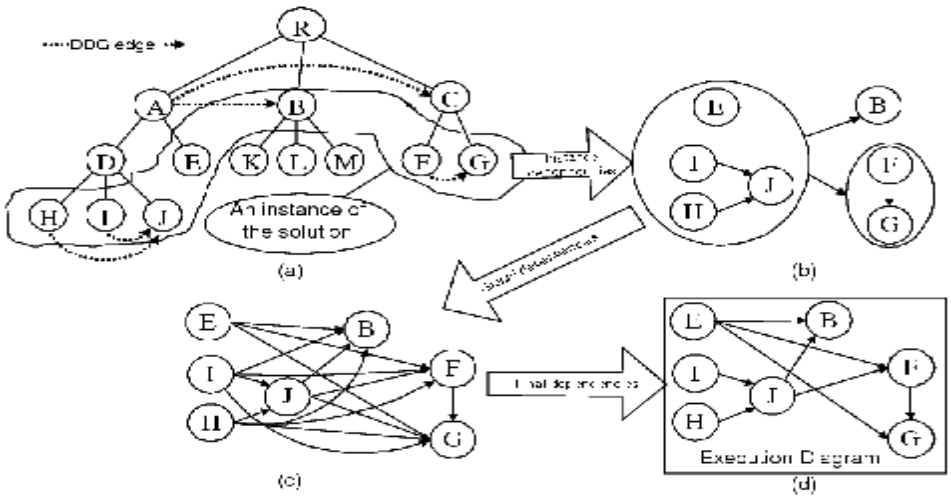


**Figure 4.**Initializing the parallelization scheme

`DTreeNode::Init` will perform as follows:

- Find an instance of the solution regarding to the availability of resources in the computational environment (Fig. 4.a). As we have described in section 2.1.c, a try-and-decompose process will be performed on DT starting from the root. "Try" will compute the resource requirements regarding to the time constraint and the complexity of the node. It then tries to allocate a parallel object based on the computed resource requirements. The POP-C++ runtime system will perform resource discovery and resource matching. If "try" succeeds, the initialization of the object is called. If not (the resource is not available),"decompose" will be executed. "Decompose" first evaluates the time constraints for all child nodes based on the DDG and the time constraint of the parent node. Then, the try-and-decompose is performed again on each child node. Decompose fails if the node is a leaf. In this case, `DTreeNode::Init` will return "out of resource". In the end, an instance of the solution is identified.
- Find the global dependencies of problems within the instance of the solution. When an instance of the solution is found, `DTreeNode::Init` will construct the global dependencies of problems within that instance by merging all hierarchical DDGs (Fig. 4.b) to generate an unique dependency graph of all problems (Fig. 4.c). All redundant dependencies will be removed to generate the final dependency graph: the execution diagram (Fig. 4.d).
- Elaborate the execution diagram to each problem's parallel object. Each ProbObj parallel object (problem) contains: a set of parallel objects that will be executed next as this problem is completely solved; and a counter that counts the number of problems that must be finished before this problem can start. `DTreeNode::Init` updates these information based on the execution diagram.

### 3.5.Executing the parallelization scheme

The last step is to call `DTreeNode::Solve` method at the root of DT to get the problem solved. `DTreeNode::`Solve looks for all "ready" nodes (nodes with no coming edge) in the execution diagram and then asynchronously invokes `ProbObj::Exec` on all ready nodes. Each time a problem (`ProbObj` parallel object) finishes, it will "fire" all next problems in its list (also by invoking `Exec` method). A problem, when being fired, will check its counter. Counter value of 0 means all previous problems have been solved. In this case, it will start solving by invoking its local virtual method `ProbObj::Solve` (the user needs to overwrite this method). Otherwise, the counter is decreased and wait for the next "being fired". The execution process is similar to that of a neural network.

## 4.EXPERIMENT RESULTS

### 4.1.Emulating heterogeneous environments

In the experiment, we built a heterogeneous environment with the following characteristics:

- High heterogeneity in computing power of processors
- Different hardware architectures
- Different operating systems
- Different network topologies

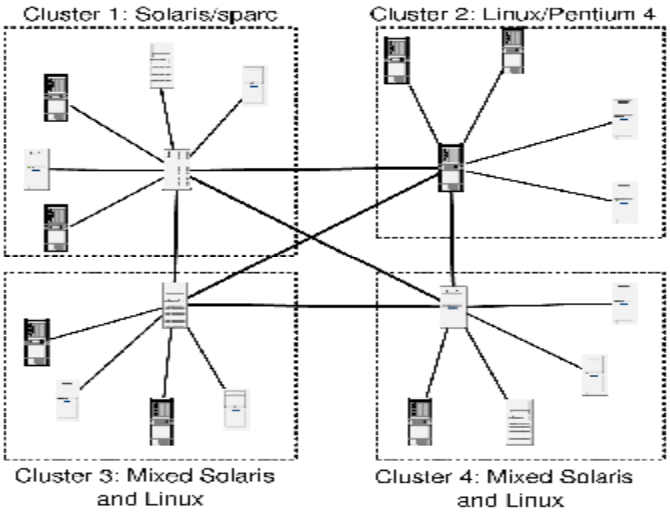


**Figure 5.**Initial topology of the computational environment

The environment consists of 130 physical workstations running Linux and Solaris, divided into 4 heterogeneous clusters. The network topology is described in Fig. 5. The effective computing powers of nodes in the environment are emulated, ranging from 60MFlops up to 1GFlops. Some machines are SMPs with 2 or 4 processors.

### 4.2.Building the parallelization scheme

We assume that we need to solve the problem that requires 50GFlop (total number of floating point operation needed to get the problem solved). We also assume that for each problem, we can decompose it into 4 sub-problems (the degree of DT). We have 4 levels of decomposition (the depth of the DT is 4). Hence, in our DT, we have total number of 341 sub-problems.

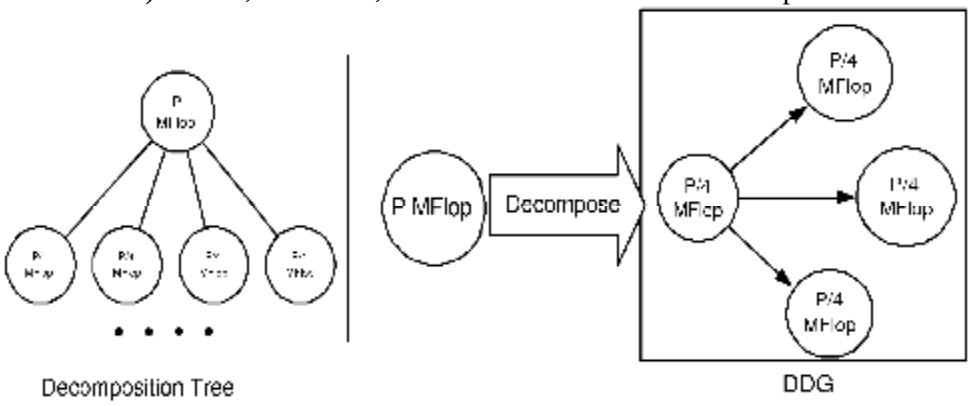


**Figure 6.**Decomposition Dependency Graph for each decomposition step

In order to classify the decompositions based on the dependencies of sub-problems, we introduce a new metric called dependency factor. Dependency factor is defined as the ratio between the maximum number of problems that can be solved in parallel and the total number of problems.

Dependency factor is a number in between 0 and 1. Dependency factor is 1 if all problems are independent and can be solved in parallel and 0 if all problems are dependent and they must be solved sequentially one after the other.

In the first test, all sub-problems within a decomposition step are independent and each sub-problem requires 1/4 computing power of its parent. The dependency factor is 1. In many practical problems, dependencies are inevitable due to the nature of the decomposition. These will degrade the degree of parallelism. So, in the second test, we create a DDG for each decomposition step as in Fig. 6: in each decomposition step, 25% of the computing power is spent to solve one sub problem sequentially and after that three other sub-problems can be solved in parallel. Here, in this case, the dependency factor is 75%.

We constructed 3 classes: `MyDTreeNode` (sequential, from DTreeNode), `MyProbObj` (parallel, from ProbObj) and `LogDataObj` (parallel class used to log execution progress information). The two first classes are used to construct the parallelization scheme. `LogDataObj` is a shared parallel object among all `MyProbObj` object. Each `MyProbObj` object will invoke methods on the shared `LogDataObj` object to store information about its execution states. In many real applications, `LogDataObj` can be replaced by the data source, the output or the monitoring parallel objects. The `MyProbObj` object will simulate the real computation by a counting loop. The time for running the loop depends on the computing power of the resource and the complexity (total computing power) of the object (this information is obtained from the parallelization scheme).

### 4.3.The results

We run the tests on the built heterogeneous environment with different time constraints. The parallel efficiency for each node is 0.95. The real computation time is measured and compared with the time constraint. For each run, the number of parallel objects that reflects the degree of parallelism is also counted

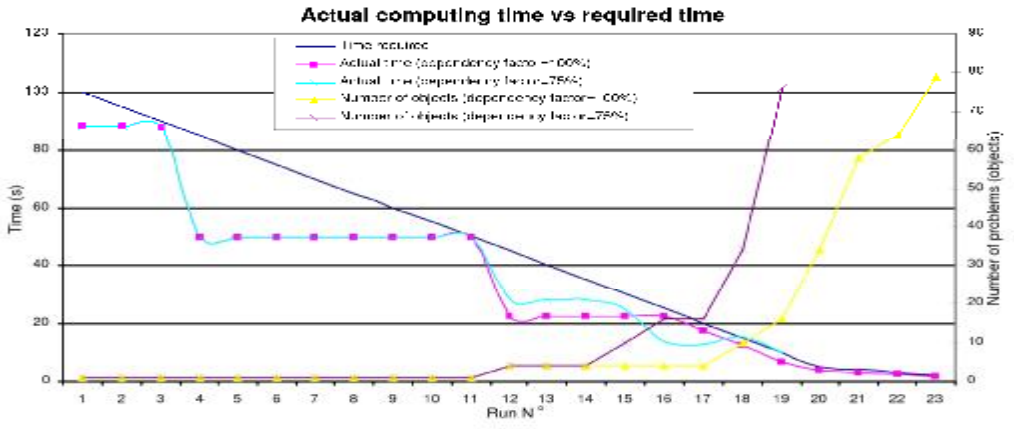


**Figure 7:** Emulation results with different time constraints in a heterogeneous environment

Figure 7 shows the experiment results. When the time constraint is greater than or equal 50 seconds, the problem is solved sequentially because there exists some 1GFlops machines. The actual solving time in this case depends on the resource discovery process but it is always smaller than the required time. As the required time decreases, the problem starts to be decomposed

(number of parallel objects increases). For tests with dependency factor of 100% (there is no dependency of sub-problems), the problem can be solved as fast as 2 seconds (67 sub-problems). Below that, the problem can not be solved sequentially due to the lack of resources. In the case where dependency factor is 75% (section 4.2), the dependencies reduce the capacity of parallelism and increase the demand for resources. Therefore, we got "out of resource" message when the time constraint is below 10 seconds. The time constraint of 10 sec leads to a decomposition of 64 sub-problems and the actual running time is 9.67 sec. Nevertheless, in both cases, the time constraints have been guaranteed.

## 5.CONCLUSIONS

Solving problems with time constraints in widely distributed heterogeneous environments such as the Grid is a challenge. We address this challenge by providing: a parallelization scheme to express the algorithmic parallelism; POP-C++-a requirement-driven parallel object infrastructure for heterogeneous HPC; and a framework that implements the parallelization scheme on POP-C++.

Primary experiments have been performed on the emulated heterogeneous environment with different processor architectures, processor capacities and different OS. The results show that we can achieve the time constraints by automatically instantiating a suitable grain of parallelism based on the availability of resources. In other word, we have tailored the HPC application to the computational environment.

Many issues were not fully taken into account yet in this paper: evaluation of the effective performance of resources, the fault tolerance, security and usability. Above all, real applications have not yet been tested. All of these issues will contribute to our future works.

## REFERENCES


[1]. T.D. Braun, H. J. Siegel, and A. A. Maciejewski. *Mapping heuristics for tasks with dependencies, priorities, deadlines and multiple versions in heterogeneous environments*. In Proc. of the 16 International Parallel and distributed Processing Symposium, (2002).

[2]. Foster and C. Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, (1998).

[3]. Foster, C. Kesselman, and S. Tuecke. *The anatomy of the grid: Enabling scalable virtual organizations*. International J. Supercomputer Applications, 15(3), (2001).

[4]. R. Gruber, P. Volgers, A. De Vita, M. Stengel, and T. M. Tran. *Parameterisation to tailor commodity clusters to applications.* Future Generation Computer Systems, (19):111–120, (2003).

[5]. J. Gunnels, C. Lin, G. Morrow, and R. van de Geijn. *Analysis of a class of parallel matrix multiplication algorithms.* In Proc. of the First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing, pages 110–116, (1998).

[6]. P. Jdrzejowicz and I. Wierzbowska. *Scheduling multiple variant programs under hard real-time constraints.* European Journal of Operational Research, 127:458–465, (2000).

[7]. M. Maheswaran, S. Ali, H. J. Siegel, D. Hensgen, and R. F. Freund. *Dynamic mapping of a class of independent tasks onto heterogeneous computing systems.* Journal of Parallel and distributed Computing, 59(2):107–131, November (1999).

[8]. T. A. Nguyen and P. Kuonen. P*arallelization scheme for an approximate solution to time constraint problems.* In Proc. of the International Conference on Computational Science 2003, June (2003).

[9]. T. A. Nguyen and P. Kuonen. *Paroc++: A requirement-driven parallel object-oriented programming language.* In Proc. of the 8th International Workshop on High-Level Programming Models and Supportive Environments, April (2003).

[10]. T. A. Nguyen and P. Kuonen. *Programming the grid with POP-C++*. Journal of Future Generation Computer Systems, (23):23–30, (2007).

[11]. *Object Management Group*. Real-Time CORBA specification. http://www.omg.org.