eCOMMONS

Loyola University Chicago
Loyola eCommons

Computer Science: Faculty Publications and Other Works

Faculty Publications

2007

# A Model-Driven Approach to Job/Task Composition in Cluster Computing

Yogesh Kanitkar

Konstantin Läufer
*Loyola University Chicago*, klaeufer@gmail.com

Neeraj Mehta

George K. Thiruvathukal
*Loyola University Chicago*, gkt@cs.luc.edu

## Recommended Citation

Neeraj Mehta, Yogesh Kanitkar, Konstantin Laufer, George K. Thiruvathukal, "A Model-Driven Approach to Job/Task Composition in Cluster Computing," ipdps, pp.233, 2007 IEEE International Parallel and Distributed Processing Symposium, 2007

This Conference Proceeding is brought to you for free and open access by the Faculty Publications at Loyola eCommons. It has been accepted for inclusion in Computer Science: Faculty Publications and Other Works by an authorized administrator of Loyola eCommons. For more information, please contact ecommons@luc.edu.

This work is licensed under a Creative Commons Attribution-Noncommercial-No Derivative Works 3.0 License.
Copyright © 2007 Neeraj Mehta, Yogesh Kanitkar, Konstantin Läufer, George K. Thiruvathukal

# A Model-Driven Approach to Job/Task Composition in Cluster Computing

Neeraj Mehta, Yogesh Kanitkar, Konstantin Läufer, and George K. Thiruvathukal

Emerging Technologies Laboratory
Department of Computer Science
Loyola University Chicago
820 North Michigan Avenue
Chicago, IL 60611, USA
www.etl.luc.edu
{laufer,gkt}@cs.luc.edu

## Abstract

*In the general area of high-performance computing, object-oriented methods have gone largely unnoticed. In contrast, the Computational Neighborhood (CN), a framework for parallel and distributed computing with a focus on cluster computing, was designed from ground up to be object-oriented. This paper describes how we have successfully used UML in the following model-driven, generative approach to job/task composition in CN. We model CN jobs using activity diagrams in any modeling tool with support for XMI, an XML-based external representation of UML models. We then export the activity diagrams and use our XSLT-based tool to transform the resulting XMI representation to CN job/task composition descriptors.*

## 1 Introduction

This paper is a culmination of several different but related projects. The fundamental intent of the research is to experiment with model-driven architecture in parallel and distributed computing as applied to cluster based supercomputing. We focus on the use of UML and its relevant modeling notations in our software framework—the Computational Neighborhood (CN).

Clustering is the use of multiple computers, typically workstations or rack-mounted servers, to form what appears to users as a single computing resource. Leading hardware and software companies offer clustering packages that offer scalability as well as availability. As demands on the system increase, all or some parts of the cluster can be increased in size or number to handle the increased demand. Cluster computing can also be used as a relatively low-cost form of parallel processing for scientific and other applications that lend themselves to parallel computing. An early and well-known example is the Beowulf [15] project in which a number of off-the-shelf PCs were used to form a cluster for scientific applications. Today, most so-called supercomputers are in reality a variation on the theme of cluster computing.

CN presents a similar approach toward cluster computing as does Beowulf. This approach can be described as building a supercomputer as a cluster of commodity off-the-shelf personal computers, interconnected with a local area network technology like Ethernet, and running programs written for parallel processing. The idea is to enable the average university computer science department or small research company to build its own small supercomputer that can operate in the gigaflop range. In addition to being economical, one can take advantage of the ever-evolving off-the-shelf technology by upgrading the cluster components.

There are many software solutions available that enable expert computer engineers to deploy, maintain and write programs for clusters. However, the intent of our research is to simplify and extend this genre of do-it-yourself supercomputing to the masses. This means the focus is on simplicity in installation, deployment, programmability, maintainability, and upgradability and on straightforwardness and clarity of the software development model. The guiding principle for CN is simplicity for the programmer and the end user. The various design decisions and trade-offs that the development of a system like this demand, are and continue to be made in favor of usability and robustness. The belief is that such an approach would be able to steer clear of

1-4244-0910-1/07/$20.00 ©2007 IEEE.

the complexity added if the guiding principle is optimal performance and support heterogeneity. The prospective applications for CN *might* be limited by this approach; however, with the proliferation of low-cost high-performance nodes there is a significant user base that can benefit from a simple and straightforward metacomputing environment. (Users in the parallel computing community are accustomed to waiting for faster next-generation hardware to gain performance without changing their existing codes.)

In this paper, we describe our preliminary experiences in using UML to model problems arising in the area of distributed cluster computing. In Section 2, we introduce our running example. In Section 3, we summarize the CN framework. In Section 4, we illustrate how activity diagrams can be used to model job/composition in CN. In Section 5, we describe the transformation of activity diagrams to executable specifications and implementations in the target language. In Section 6, we give an overview of related projects. Finally, in Section 7, we conclude with a brief discussion future work.

## 2 Guiding Example

We have used the parallel version of Floyd's all-pairs shortest-path algorithm [8] as a guiding example in this work, since it is representative of the many typical problems faced in designing and implementing parallel algorithms. This algorithm is based on a one-dimensional, row-wise domain decomposition of the intermediate matrix $I$ and the output matrix $S$.

The algorithm can use at most $N$ processors or tasks where $N$ is the number of nodes in the graph. Each task has one or more adjacent rows of the adjacency matrix that represents the input graph and is responsible for performing computation on those rows. Fundamentally, this algorithm derives the resultant matrix $S$ in $N$ steps, constructing at each step $k$ an intermediate matrix $I(k)$ containing the best-known shortest distance between each pair of nodes. Due to this approach, in the parallel version, in the $k$th step, each task requires, in addition to the rows assigned to it, the $k$th row. Thus, we need to set up each task with the rows it computes and maintains, track $k$, in the $k$th iteration have the task with the $k$th row broadcast it and collate the results at the end.

The CN implementation of the transitive closure algorithm consists of three different tasks. The first task, `TaskSplit`, reads the input and initializes the worker tasks, `TCTask`, with the appropriate rows. Each of the `TCTask` workers keeps track of $k$, and the tasks coordinate among themselves using using the CN API for intertask communication (CN also supports communication via tuple spaces, which are not covered in this paper). The collation of the results is done by yet another task named `TCJoin`. The tasks `TaskSplit`, `TCTask`, and `TCJoin` are packaged as Java archive (jar) files conforming to the interface specified in the CN API.

This above tasks are glued together using the client program that we generate directly using UML activity diagrams. We describe the composition using UML and transformation into an executable specification (and implementation) in the subsequent sections.

## 3 CN Summary

In general, CN provides a framework to define and execute tasks in a parallel program transparently on the various nodes in the cluster and collate the final results. The ultimate goal is to develop an integrated development environment for cluster computing in which the entire process can be done in a model-driven fashion, where the user seldom is forced to deal with low-level issues unrelated to solving the actual research problem.

One of the obvious challenges faced in cluster computing is identifying the parts of the program that can run in parallel, execution and management of these parts and coordinating among them. CN does expect the user to identify these parts but provides a framework for assembling the overall solution. The CN programming model loosely imitates the multithreaded paradigm; however, the threads (tasks) run anywhere where computing power is available and can be coordinated regardless of where they actually run. This makes it easy for anybody familiar with multithreading essentials to design for and exploit CN's capabilities. Additionally, CN provides a messaging model that is similar to that of Windows (X and Win32) programming to reduce the programmer's learning curve.

CN provides a modular framework comprising four main components: Job, Task, Job Manager and TaskManager. A Task is defined to be a unit of work that the user wants to perform. A Job is defined as a collection of Task objects. The Job and Task creation, control and coordination is all done using CN API (a factory). The user is responsible, usually toward the beginning of the parallel program, to acquire a reference to the CN API. A JobManager is a conduit between the client CN application and the Job in CN to talk to each other. TaskManager executes the various Tasks of various Jobs and is transparent to the user. (For those familiar with X Windows, the CN design uses the term client similarly; a user-developed application is considered a client of the CN system.) The components of the CN framework and implementation are shown in Figure 1.

More specifically, JobManager and the TaskManager are part of the same process, CNServer, which is a *servant* (since it acts as a client and a server). The JobManager can support multiple Jobs. The client links to the CN API, which exposes the following capabilities:

**CN Server** CN Servers run on the various nodes of the cluster.

**CN API** Client programs use the CN API to execute and exploit the various resources of the cluster.

**CN Intelligent Object Editor** The user could specify the details required to generate the Client program using this graphical use interface.

**CNX** CNX (XML) is a compositional language that captures the details of the client program.

**CNX2Java** CNX2Java is an XSLT that translates CNX to compilable JAVA code.

**XMI2CNX** XMI2CNX is an XSLT that translates UML model in XMI format to CNX.

**Prototype** Web interface to the CN cluster that accepts UML model in XMI format, translates the model to an executable, executes model and displays or makes the results available for download.

**Figure 1. CN framework components**

- Initialize CN API (using the factory)
- Create Job in JobManager
- Create Tasks for the Job
- Start the Tasks
- Get Messages from Tasks
- Send Messages to Tasks

Requests to JobManager are communicated using multicast. JobManagers respond to multicast requests for JobManagers if they have free resources and are willing to be JobManagers. A JobManager is selected based on User specified Job requirements from the list of willing JobManagers. The Job is subsequently created in the selected JobManager.

A Task is typically packaged as a self-sufficient JAR file that has a class that conforms to the Task interface defined by CN API. The JobManager solicits TaskManager for the Tasks that requested to be created by the User program. If a willing TaskManager is found the JobManager will upload the JAR file to that TaskManager. TaskManager in turn sets up a message queue for each Task and then executes each Task in a separate thread when the User program requests to start the Task.

CN uses messages as the fundamental information between the CN and the client. CN has well-defined messages that define the Message Request, expected Message Action and expected Message Response. Besides the well-defined messages, CN also allows user-defined messages that only the application (client and its tasks) understands. Thus, in the case of user-defined messages CN merely provides a message delivery mechanism.

CN can be deployed in multiple configurations. One could install CN servers on all the machines of a subnet and a user could run their client programs from any machine on the subnet. The other deployment configuration is through a web portal so that the user does not need to log on to the subnet.

## 4 Activity Diagrams

In this section, we provide an overview of activity diagrams in UML [12] and discuss their applicability to modeling the composition of tasks in parallel programs.

An *activity diagram* is a visual representation of an activity graph. An activity graph is a state machine whose states represent actions or subactivities and where transitions out of states are triggered by the completion of the corresponding actions or subactivities. Activity diagrams are thus intended for modeling computations whose control flow is driven by internal processing. In a UML model, activity diagrams are usually attached to a *classifier*, such as a use case or a package.

In parallel computing, a computational job typically consists of one or more concurrent tasks whose dependencies form a directed acyclic graph. In CN, a *client* is composed from one or more such jobs. While the details of jobs and tasks are implemented modularly in the target programming language, the jobs and tasks need to be "glued" together outside of the implementation itself. An example of a CNX client descriptor for computing the transitive closure of a graph is provided in Figure 2.

It is helpful to have a visual notation to express the composition of jobs from tasks and the dependencies among tasks. Since transitions between CN tasks are triggered by internal task termination, activity diagrams provide a natural notation for modeling the composition of tasks in CN. Furthermore, it is natural to attach an activity diagram for a client computation to the package containing the other elements of the model (class diagrams and interaction diagrams) for that client.

Specifically, each job is represented as an activity, and each task is represented as an action state; the dependencies among tasks are represented as transitions between the action states. The data flow between tasks is an implicit part of the control flow. (Tasks can also communicate asynchronously, but such communication is deliberately not represented in this model. Instead, asynchronous communication would be part of the dynamic model for each task.) Finally, a client consisting of more than one job is represented as an activity that performs the jobs in some partial order (allowing for a mix between sequential and concur-

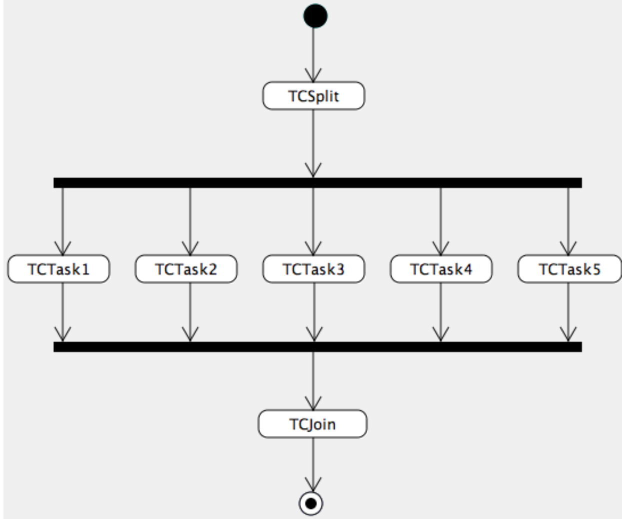


**Figure 3. Activity diagram for transitive closure using explicit concurrency.**

| jar | `tctask.jar` |
|---|---|
| class | `org.jhpc.cn2.trnsclsrtask.TCTask` |
| memory | `1000` |
| runmodel | `RUN_AS_THREAD_IN_TM` |
| ptype0 | `java.lang.Integer` |
| pvalue0 | `2` |

**Figure 4. Tagged values for `TCTask2`.**

rent execution); the tasks themselves then become nested activities.

An activity diagram that visualizes the task composition of the transitive closure CN client is shown in Figure 3. This example shows a splitter task, a fixed number of worker tasks that execute concurrently, and a joiner task.

UML's *tagged values* allow us to model all of the information present in a CN client descriptor, including the implementation class of each task, the archive containing the implementation class, as well as various other task configuration parameters. Figure 4 shows the tagged values for the worker task `TCTask2`, whose parameter `pvalue0` has value `2`.

When modeling a parallel computation, it is sometimes desirable to leave the number of concurrent invocations of a task open until run time, dependent on system load or other external factors. Activity diagrams support this approach through their *dynamic invocation* notation. The number of concurrent invocations is determined by a run-time expression that evaluates to a set of actual argument lists, one for each invocation. An activity diagram for the transitive closure job involving dynamic invocation is shown in Figure 5. Here, the multiplicity is zero or more; a specific run-time argument expression would be specified separately.

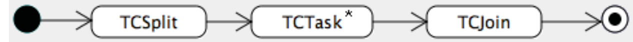


**Figure 5. Activity diagram for transitive closure using dynamic invocation.**

## 5 From Model to Executable Specification: XMI to CNX

In this section, we describe the transformation of the UML model of a CN computation to an executable version of the computation, including a CNX client descriptor. At a high level, this transformation involves the following steps, which are also illustrated in Figure 6.

1. The UML model for the CN computation is created in the form of an activity diagram.
2. The UML model is exported as an XMI document [11].
3. The XMI document is transformed, using XSL Transformations (XSLT) [16], to a CNX client descriptor.
4. The CNX client descriptor is transformed, using XSL Transformations, to a client program in the target language (currently Java).
5. The resulting client program is deployed to a CN server along with the archives containing the compiled classes.
6. The client computation is executed by the CN server.

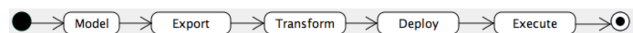


**Figure 6. Transformation of UML model to executable CN client specification.**

We will now illustrate this process in the context of our transitive closure example. The activity diagram for this example was shown in Figure 3. A fragment of the exported XMI document can be seen in Figure 7; this fragment corresponds to the `TCTask2` state whose tagged values were shown in Figure 4. Finally, the resulting CNX client descriptor was shown in Figure 2.

## 6 Related Work

The Askalon project [13] utilizes the UML extension mechanisms to customize UML for the domain of performance oriented distributed and parallel computing. This project focuses on modeling and performance analysis, and visualization. In contrast, our work attempts to go a step further and generate executable code from the model.

The remaining literature to one degree or another is related thematically but does not employ UML and model-driven architecture specifically. Nevertheless, some of these works do provide support for higher-level modeling and are included for completeness.

The Frugal system [5] transforms Jini-enabled networks into metacomputers. The Frugal System allows users on any machine in the network to run their Java jobs on any other machine in the network. It uses advanced decision-making algorithms to automatically place these jobs on the best machine, guaranteeing near-optimal end-to-end performance of all submitted jobs.

The Globus project [9] is developing fundamental technologies needed to build computational grids. Grids are persistent environments that enable software applications to integrate instruments, displays, computational and information resources that are managed by diverse organizations in widespread locations.

Legion [10], an object-based metasystems software project at the University of Virginia, is designed for a system of millions of hosts and trillions of objects tied together with high-speed links. Users working on their home machines see the illusion of a single computer, with access to all kinds of data and physical resources, such as digital libraries, physical simulations, cameras, linear accelerators, and video streams.

Beowulf [15] is a high-performance massively parallel computer built primarily out of commodity hardware components, running a free-software operating system like Linux or FreeBSD, interconnected by a private high-speed network. It consists of a cluster of PCs or workstations dedicated to running high-performance computing tasks.

SNIPE [7] is a metacomputing system that aims to provide a reliable, secure, fault-tolerant environment for long-term distributed computing applications and data stores across the global Internet. This system combines global naming and replication of both processing and data to support large scale information processing applications leading to better availability and reliability than currently available with typical cluster computing and/or distributed computer environments.

Titanium [17] is an explicitly parallel dialect of Java developed at UC Berkeley to support high-performance scientific computing on large-scale multiprocessors, including massively parallel supercomputers and distributed-memory clusters with one or more processors per node. Other language goals include safety, portability, and support for building complex data structures.

The Distributed Object Migration Environment (DOME) project [1] aims to build sets of distributed objects which can be used to program heterogeneous networks of computers as a single computing resource. Dome addresses the problems of load balancing in a heterogeneous multiuser environment, ease of programming, and fault tolerance.

The GLOBE project [3] research focuses on a powerful unifying paradigm for the construction of large-scale wide area distributed systems: distributed shared objects. In this model, the universe consists of a vast number of shared objects, each of which has some associated methods invokable by authorized users.

The Condor project [6] aims to develop, implement, deploy, and evaluate mechanisms and policies that support High Throughput Computing (HTC) on large collections of distributively owned computing resources.

The Pangaea [14] project provides an automatic distribution environment for Java. Given a centralized Java program, and certain requirements for distribution, the environment automatically creates a distributed version of the program.

Orca [4] is a language for parallel programming on distributed systems, based on the shared data-object model. This model is a simple and portable form of object-based distributed shared memory.

Cluster JVM [2] for Java virtualizes the cluster, supporting any pure Java application without requiring that applications be tailored specifically for it. The aim of Cluster JVM for Java is to obtain improved scalability for a class of Java Server Applications (JSA) by distributing the application's work among the cluster's computing resources.

## 7 Conclusions and Future Work

In this paper, we have focused on our early work in applying UML to an application domain in which more modeling is truly needed and relevant to improve the way software is presently being developed. We do not claim to have the complete solution in hand; however, our early work suggests that the Activity Diagram framework of UML can play a significant role in conceiving and developing software for cluster computing.

We plan to explore UML to model other aspects of CN development, including the lower-level details of the computation or problem being solved and complex interactions between tasks. We are currently working on model and tool support to facilitate development of the computational tasks themselves. At present the user has to hand-code these tasks using the target language API. (Java is presently the only supported language.) Nevertheless, the preliminary steps

we have taken allow higher-level problem-solving strategies to be employed in high-performance computing—a needed evolutionary step for the field to go beyond niche status.

Finally, we see potential to apply the results of this work to other high-performance computing environments with a different but related goals. In particular, grid computing, which exposes a number of low-level interfaces for developing wide-area high-performance computing applications, could benefit from higher-level model-driven architecture. We will be investigating this and other possibilities through our future work and collaborations.

# 8 Obtaining the Computational Neighborhood Code

The code for Computational Neighborhood is available via the Google Code Project Hosting site `http://code.google.com/p/neighborhood/` and can be checked out anonymously using Subversion as an Eclipse project as follows:

```
svn checkout http://neighborhood.\
  googlecode.com/svn/trunk/cn-java
```

# References


[1] J. Arabe, A. Beguelin, B. Lowekamp, E. Seligman, M. Starkey, and P. Stephan. Dome: Parallel programming in a heterogeneous multi-user environment. Technical Report CS-95-137, School of Computer Science, Carnegie Mellon University, 1995.

[2] Y. Aridor, M. Factor, and A. Teperman. cJVM: A single system image of a JVM on a cluster. In *International Conference on Parallel Processing*, pages 4–11, 1999.

[3] A. Bakker, E. Amade, G. Ballintijn, I. Kuz, P. Verkaik, I. van der Wijk, M. van Steen, and A. S. Tanenbaum. The globe distribution network. In *Proceedings of the USENIX Annual Conference*, pages 141–152, 1999.

[4] H. E. Bal, M. F. Kaashoek, and A. S. Tanenbaum. Orca: a language for parallel programming of distributed systems. *IEEE Transactions on Software Engineering*, 18(3):190–205, 1992.

[5] R. S. Borgstrom. *A Cost-Benefit Approach to Resource Allocation in Scalable Metacomputers*. PhD thesis, Johns Hopkins University, Sept. 2000.

[6] D. H. J. Epema, M. Livny, R. van Dantzig, X. Evers, and J. Pruyne. A worldwide flock of condors: load sharing among workstation clusters. *J. on Future Generations of Computer Systems*, 12, 1996.

[7] G. E. Fagg, K. Moore, and J. J. Dongarra. Scalable Networked Information Processing Environment (SNIPE). *Future Generation Computer Systems*, 15(5–6):595–605, 1999.

[8] Floyd, R. W. Algorithm 97: Shortest Path. *Comm. ACM*, 5:345, 1962.

[9] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, Summer 1997.

[10] A. Natrajan, A. Nguyen-Tuong, M. A. Humphrey, and A. S. Grimshaw. The Legion Grid Portal, 2001. Submitted for publication.

[11] Object Management Group. XML Metadata Interchange (XMI). Technical Report formal/02-01-01, Object Management Group, Jan. 2002.

[12] Object Management Group. Unified modeling language. Specification v1.5 formal/2003-03-01, Object Management Group, Mar. 2003.

[13] S. Pllana and T. Fahringer. UML Based Modeling of Performance Oriented Parallel and Distributed Applications. In *Proceedings of the 2002 Winter Simulation Conference*, San Diego, California, USA, December 2002. IEEE.

[14] A. Spiegel. PANGAEA: An automatic distribution front-end for JAVA. In *IPPS/SPDP Workshops*, pages 93–99, 1999.

[15] T. Sterling, D. Savarese, D. J. Becker, J. E. Dorband, U. A. Ranawake, and C. V. Packer. BEOWULF: A parallel workstation for scientific computation. In *Proceedings of the 24th International Conference on Parallel Processing*, pages I:11–14, Oconomowoc, WI, 1995.

[16] World Wide Web Consortium. XSL Transformations (XSLT), 1999. http://www.w3.org/TR/xslt.

[17] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: A high-performance Java dialect. In ACM, editor, *ACM 1998 Workshop on Java for High-Performance Network Computing*, New York, NY 10036, USA, 1998. ACM Press.

```xml
<?xml version="1.0"?>
<cn2>
 <client class="TransClosure" log="CN_Client1047909210005.log" port="5666">
  <job>

   <task name="tctask0" jar="tasksplit.jar"
         class="org.jhpc.cn2.transcloser.TaskSplit" depends="">
    <task-req>
     <memory>1000</memory>
     <runmodel>RUN_AS_THREAD_IN_TM</runmodel>
    </task-req>
    <param type="String">matrix.txt</param>
   </task>

   <task name="tctask1" jar="tctask.jar"
         class="org.jhpc.cn2.trnsclsrtask.TCTask" depends="tctask1">
    <param type="Integer">1</param>
    <task-req>
     <memory>1000</memory>
     <runmodel>RUN_AS_THREAD_IN_TM</runmodel>
    </task-req>
   </task>
   .
   .
   .
   <task name="tctask5" jar="tctask.jar"
         class="org.jhpc.cn2.trnsclsrtask.TCTask" depends="tctask0">
    <param type="Integer">5</param>
    <task-req>
     <memory>1000</memory>
     <runmodel>RUN_AS_THREAD_IN_TM</runmodel>
    </task-req>
   </task>

   <task name="tctask999" jar="taskjoin.jar"
         class="org.jhpc.cn2.transcloser.TaskJoin"
         depends="tctask1,tctask2,tctask3,tctask4,tctask5">
    <task-req>
     <memory>1000</memory>
     <runmodel>RUN_AS_THREAD_IN_TM</runmodel>
    </task-req>
    <param type="String">matrix.txt</param>
   </task>

  </job>
 </client>
</cn2>
```

**Figure 2. Client descriptor for transitive closure**

```
<UML:ActionState xmi.id = ’a89’ name = ’TCTask2’
  isSpecification = ’false’ isDynamic = ’false’>
    <UML:TaggedValue xmi.id = ’a91’ isSpecification = ’false’
      dataValue = ’1000’>
      <UML:TaggedValue.type>
        <UML:TagDefinition xmi.idref = ’a13’/>
      </UML:TaggedValue.type>
    </UML:TaggedValue>
    <UML:TaggedValue xmi.id = ’a92’ isSpecification = ’false’
      dataValue = ’RUN_AS_THREAD_IN_TM’>
      <UML:TaggedValue.type>
        <UML:TagDefinition xmi.idref = ’a16’/>
      </UML:TaggedValue.type>
    </UML:TaggedValue>
    <UML:TaggedValue xmi.id = ’a93’ isSpecification = ’false’
      dataValue = ’tctask.jar’>
      <UML:TaggedValue.type>
        <UML:TagDefinition xmi.idref = ’a7’/>
      </UML:TaggedValue.type>
    </UML:TaggedValue>
    <UML:TaggedValue xmi.id = ’a94’ isSpecification = ’false’
      dataValue = ’org.jhpc.cn2.trnsclsrtask.TCTask’>
      <UML:TaggedValue.type>
        <UML:TagDefinition xmi.idref = ’a10’/>
      </UML:TaggedValue.type>
    </UML:TaggedValue>
  </UML:ModelElement.taggedValue>
  <UML:StateVertex.outgoing>
    <UML:Transition xmi.idref = ’a95’/>
    <UML:Transition xmi.idref = ’a96’/>
  </UML:StateVertex.outgoing>
  <UML:StateVertex.incoming>
    <UML:Transition xmi.idref = ’a78’/>
  </UML:StateVertex.incoming>
</UML:ActionState>
```

**Figure 7. Sample XMI for transitive closure job**