Integration of Simulation Tools and Models in a Collaborative Environment

Andriy Levytskyy
Eugene J.H. Kerckhoffs
Delft University of Technology
Faculty of Information Technology and Systems
Mediamatica Department
Mekelweg 4
2628 CD Delft, The Netherlands
a.levytskyy@cs.tudelft.nl

Keywords:
Collaborative environment, metadata, models with different formalisms, heterogeneous distributed tools, the Internet

ABSTRACT: *Scientists in collaborative teams often face technical issues, such as access to and manipulation of various models and their processing on heterogeneous distributed tools, which actually are irrelevant to the research and development process itself. In this paper we discuss our approach in achieving a uniform and generic interface to a set of off-shelf scientific tools, and the transparent sharing and processing of models in different formats and formalisms.*

1. Introduction

This paper presents ongoing research to develop a web-based collaborative environment supporting reuse and interoperability of M&S (Modelling and Simulation) components, created and/or used in a big research project in which various groups are working together. The goal of this so-called NanoComp [1] project is to investigate the feasibility of future electronics based on quantum devices. To do this it is essential to bring together various disciplines (physics, circuitry and systems); M&S components used in the various groups should be shared and reused through the planned collaborative environment. Our goal is a generic environment for collaborative research and development that shields its users from irrelevant issues in access, distribution, manipulation and processing of models in various formats and formalisms on heterogeneous distributed tools. The generic environment can be adjusted to cover the concrete situation at hand (in our case the aforementioned NanoComp project with its specific scientific tools and models).

Any scientific development process in such collaborative environments clearly needs infrastructures for data and tool integration, which involve numerous access, distribution and computation related issues. There exist a number of such environments [2 - 5] with various degrees of collaborative features that address the above issues. However, the level of abstraction they provide with respect to data and tools make them more suitable for scientists working in the same domain. Users would typically construct a meta-application using familiar models, data-files, and tools. In case of multidisciplinary collaborators it is reasonable to expect less understanding and expertise about each other's models[1] and tools. In this case, an alternative working paradigm could be based on working with scientific concepts related to modelling itself as well as to the phenomena under investigation as it was demonstrated in [6]. This research provided us with initial ideas with respect to abstraction from heterogeneity of models and tools. Supporting such abstraction requires that resources (models and tools) be enriched with additional meta-information to insure proper discovery, interpretation and processing. In the description of such networked resources (for sharing purposes) we rely on the Dublin Core Metadata Initiative [7]. The environment itself is being developed in Python [8], an object-oriented scripting language suitable for fast prototyping. At the lower system level, we employ a Distributed Object Technology (DOT) to deal with the distribution of tools. Though there exist already prominent distributed computing architectures such as CORBA [9], Java RMI [10] and HLA [11] (each with its own added value [12]), because of fast prototyping considerations we have chosen Pyro (PYthon Remote Objects) [13], which can be extremely easy integrated with the rest of our system.

In the paper we will consider our approach in achieving a uniform and generic interface to a set of off-shelf scientific tools, as well as transparent sharing and processing of models in different formats and formalisms. The paper is organised as follows. Section 2

[1] We consider everything (i.e., models in various formalisms, data files in various formats, documents, etc.) as models.

Table 1. Metadata for Resource Discovery

TITLE	:	A name given to the model
SUBJECT	:	The topic of the content of the model
DESCRIPTION	:	An account of the content of the model
CREATOR	:	An entity primarily responsible for making the content of the model
CONTRIBUTOR	:	An entity responsible for making contributions to the content of the model
PUBLISHER	:	An entity responsible for making the model available
CONTACT	:	Contact information on either Creator, or Contributor, or Publisher
IDENTIFIER	:	An unambiguous reference to the model's ARV

briefly describes how meta-level concepts are used to shield users from irrelevant low-level system details about models and tools. In section 3 we provide a description of the environment's processing platform that integrates various scientific computing tools. Section 4 briefly explains aspects of the environment configuration. Section 5 illustrates the operation of the environment with a working session example. Finally, section 6 concludes the paper with final remarks.

2. Metadata

In order to deal with heterogeneity of models and tools, the environment employs meta-level concepts for model discovery and model processing. The former (see subsection 2.1) is crucial for transparent sharing of models among collaborators and the latter (covered in subsection 2.2.) is employed to shield the users from irrelevant execution details and provides a spectrum of actions applicable to models.

2.1 Model discovery

In order to provide users with extended information, such as author, description, keywords, etc. with respect to the available models, every model is accompanied with metadata. This metadata is a manifestation of the model's attributes and is primarily used for model discovery. The attributes we use to describe models are based on Dublin Core Metadata Element Set (DCMES) [14] and constitute a metadata record referred to as *Metadata for Resource Discovery* (MRD). Table 1 illustrates a format of this metadata record and describes its elements.

All models (in various formats and formalisms) are distributed and maintained at the collaborative sites of their respective creators. Every model in the environment is represented and accessible via its MRD record, which is stored in a central database termed *model base.* By summarising the external attributes of a model and uniquely identifying the associated model, MRD enables users to publish and discover models by various criteria.

MRD forms the highest level of metadata, which facilitates the sharing of models in the environment. However, MRD is not sufficient to enable users to access and manipulate models. For that purpose the environment employs additional meta-level concepts (for more details see [6]) as dealt with in subsection 2.2.

2.2 Model processing

Every model has a certain amount of low-level details (such as location of a model, access methods, type of a model, etc.), which are crucial for user manipulation of a discovered model. The same holds for the heterogeneous distributed tools, which are needed to process the models. The environment encapsulates these low-level details about models and tools in a metadata record called *Abstract Resource View* (ARV).

ARVs are special structures that provide a high-level schematic view on models and tools. The ARV contains a complete totality of information pertaining to the execution of the model on its respective solver, and provides an unambiguous identifier to the underlying resource (a model or a tool). An MRD record is connected with the metadata for model processing via the value of the element IDENTIFIER, which is the ARV name of the model concerned. Each model and tool ARV instance has to be created by its users.

Table 2. ARV Record for Models

NAME:	ARV name for the model
ACCESS:	Access methods
HOST:	Address of the host
FILENAME:	Filename at the host
TYPE:	ASCII/Binary/etc
TOOL:	ARV name of the Tool
ARV_OUT:	ARV name for the output

Table 3. ARV Record for Tools

NAME:	ARV name for the tool
ACCESS:	Access methods
HOST:	Address of the host
FILENAME:	Filename at the host
TYPE:	Tool
TIMEOUT:	Tunable for efficiency
VARIABLES:	Execution time variables

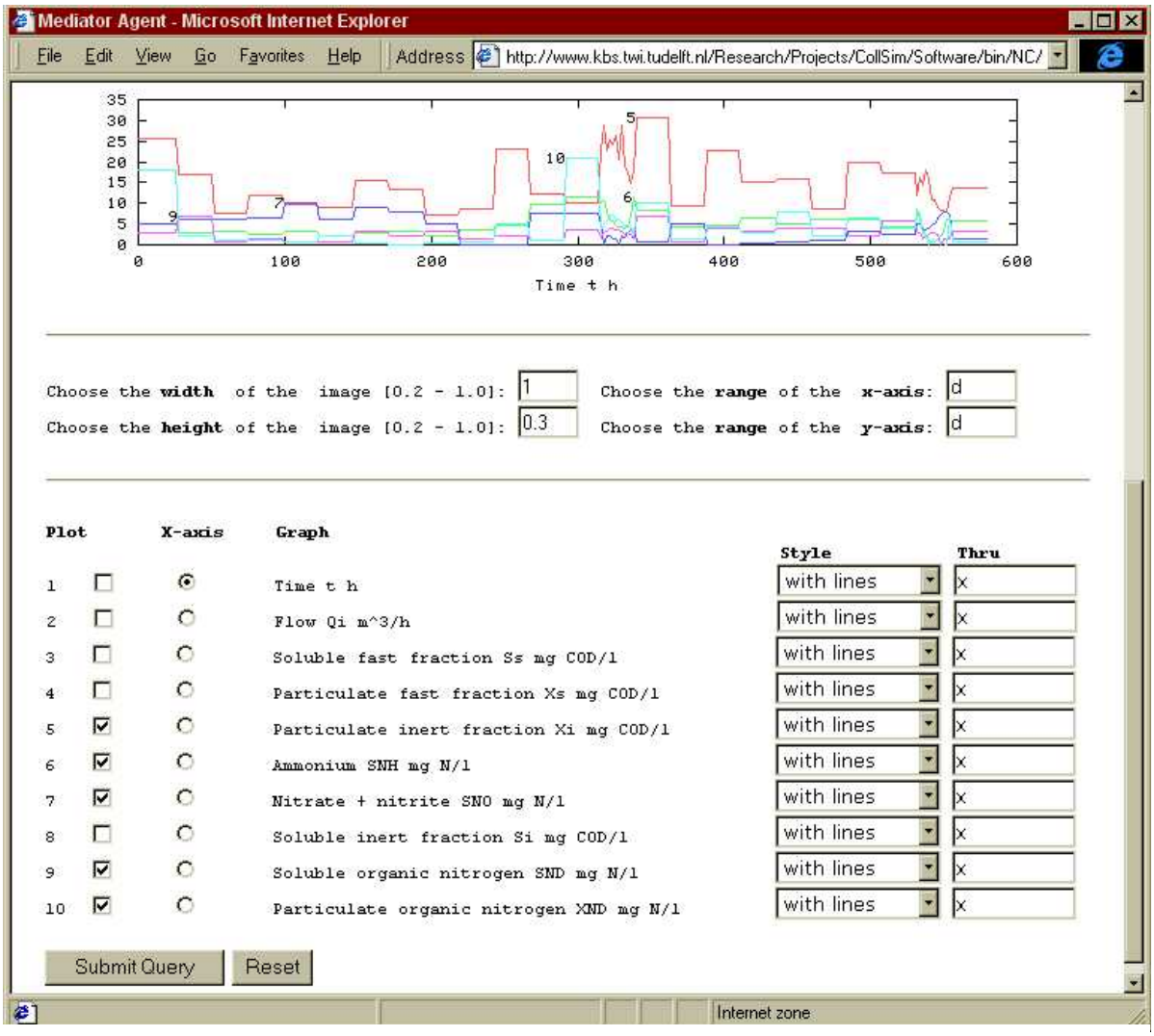

Figure 1. User Interface to the Transformation Function "Display" for Gnuplot

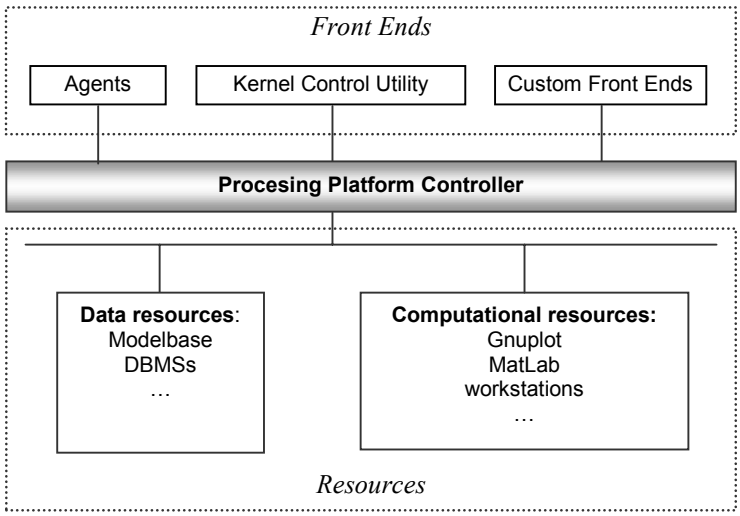

Figure 2. Architecture of the system.

Table 4. Represenation of 'Display' Transformation Instance for the Tool Gnuplot

```
Display (mdl1, mdl2)
{
  TOOL: Gnuplot
  Begin
     #gnu script
     set terminal pbm color small
     set nopolar
     set nokey
     set size 0.6, 0.5
     set xrange []
     set yrange []
     set xlabel "Time t h"
     set label 2 "2" at  148,  1221 right
     set output mdl2
     plot mdl1 thru x using 1:2 with lines
     exit
  End
}
```

There are two separate ARV types for models and tools (see Tables 2 and 3 for their respective representation). The meaning of most elements in an ARV record is self-explanatory. The value of element `FILENAME` is assumed to be another ARV name; if finding this ARV in the ARV library fails then it is assumed to be a filename. `TIMEOUT` is used to prevent deadlocks or blocking that might happen because of incorrect inputs. Element `ARV_OUT` is a name of an ARV to be associated with the produced output.

Performing meaningful actions on ARVs of models involves interaction with underlying heterogeneous (scientific) tools, which is accomplished through abstract high-level functions termed *transformations* (see Table 4 for an example). A transformation is specifically developed for a particular tool and contains a reference to that tool (the value of element `TOOL` is the name of the tool's ARV). A transformation is a containment for tool's commands (between the *`begin`* and *`end`*), which implement an action function of a transformation and are native to a specific tool. As such, transformations encapsulate the knowledge about the tool's user interface and are used to carry out an interactive session on behalf of remote users.

Transformations consist of two parts: the transformation itself and a part that is responsible for user interface to that transformation (if applicable). Such combination introduces a degree of interactivity into the execution of transformations. For example, a "Display" action for a data model (columns of data) is a transformation that involves the tool 'Gnuplot' to draw a graph. There is an interface counterpart, which provides a user with a web interface to a subset of 'Gnuplot' functionality related to the action (see Figure 1). This allows the user to customize default transformation parameters before initiating a request for the transformation. Table 4 shows the resulting instance of the "Display" transformation with the concrete parameters provided by the user. This instance contains a script for the tool 'Gnuplot', uses `'mdl1'` as input data source, and plots a graph into `'mdl2'`. The resulting file of this transformation will be associated with the ARV, which name is indicated in element `ARV_OUT` of the ARV record for `'mdl1'`.

Transformations available for a tool form a spectrum of actions that can be applied by a user to the class of models (more exactly to their ARVs) associated with that particular tool. Transformations are relatively small and simple to code and develop, thereby it is quite easy to extend the library of actions available to users.

Finally, all the information about a class of models and the transformations applicable to it is encapsulated in a metadata structure termed *manipulation structures* (M-structure). By putting the ARV and transformation concepts together, M-structures represent the spectrum of available transformations applicable to a given class of models.

3. Processing Platform

In our design of the Processing Platform (PP) we would like to address the following needs: transparent operation of heterogeneous off-shelf tools, sharing/reusing tools, dealing with license restrictions, workload control, integrity of multiple users' experiments being executed on shared tools and automation of routine processes. Crucial in our approach is the application of a discrete-event simulation kernel in order to meet our goals as indicated above. Together with a DOT system, the kernel (further on referred to as Processing Platform Controller (PPC) or simply Controller) maintains an abstract view on the state of the tools in the back-end and synchronises requests to them.

3.1 Architecture overview

The environment is based on a three-layer architecture (see Figure 2): (i) front-end layer (high-level front-ends), (ii) middle layer (*Processing Platform Controller*), and finally, (iii) back-end layer (distributed heterogeneous tools). In this subsection we give a brief overview of the layers with emphasis on the second and third layer, which form the Processing Platform.

The front layer consists of a number of client-side *front-ends* (e.g., an agent environment). They provide users with a high level GUI to the system and enable them to operate in the environment.

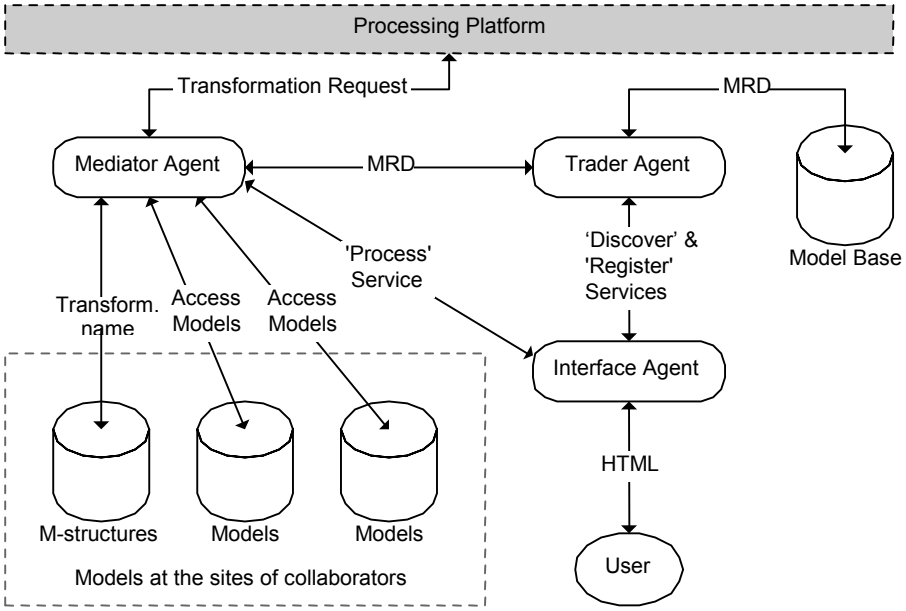

Figure 3. Agent Environment

The middle layer of the environment consists of one central persistent *Controller* (PPC). This Controller synchronises transformation requests to the tools and manages queues of requests per tool. There are a few aspects characteristic to the Controller. First of all, the core of the Controller is a run-time system for a small process-interaction simulation language. The former is referred to as Process Oriented Kernel (POKer), and the latter as POKer language [15]. The language is used to model a resource-centric view on the tools in the environment. Each resource in the Controller's model represents a logical view on an actual (scientific) tool in the back-end. In general, the relation of a resource to a tool's instances is *one* to *many*. In order to configure a new tool into the PPC (for more details see section 4), one has to create a new resource in the Controller's model, and configure the resource's attributes. Most important of the attributes are the current *capacity* of the resource (it shows whether the resource can still serve more simultaneous requests) and a FIFO *queue* of delayed transformation requests on behalf of clients. The former is influenced by a number of factors, such as licence restrictions, server configuration (shared, dedicated) of the tool and workload considerations. Based on the capacity state of the resource in the Controller's model, the Controller synchronises the clients' requests to access and perform transformations on the respective tool in the back-end. If necessary, blocking occurs. For more details on the Controller, see [16].

The back-end layer consists of a number of tools distributed over the network and accessible via a DOT system to the Controller. There is always a controlled number of a tool's instances running for each respective resource registered to the PPC. The maximum number of instances allowed for a tool equals the resource's capacity for that tool.

3.2 Implementation

The agent environment consists of trader, mediator and interface agents. Applied to our situation, the *trader agent* maintains a model base with MRD records on the available models and offers users a mechanism to search them. A *mediator agent* is positioned between distributed model sources (at remote hosts) and users. Tasks that are associated with mediation are, for instance, accessing, merging models from different locations, supporting abstraction, generalisation, and processing the underlying model(s). Finally, the *interface agent* provides user interface to the functionality (*services*) of the other agents. An overview of the agent environment is shown in Figure 3.

The PPC was developed in *Python* [8], a scripting object-oriented language. The implementation of POKer was driven by the ideas and operational semantics behind *πDemos* [17] – a small process-oriented discrete-event system simulator.

The DOT system, we have chosen for the time being is *Pyro* [13], which closely resembles Java's Remote Method Invocation [10]. Though Pyro is rather basic when compared to such general systems as, for example, CORBA, it is small, simple to set up (especially if the rest of the system is being developed with Python), free of charge and provides sufficient

services. All of the above makes Pyro an attractive alternative for fast prototyping of distributed systems within the Python community.

The Processing Platform (PP) is built on distributed scientific tools (such as, for example the numerical package MatLab) registered to Pyro *Naming Service* (NS) as server objects. These objects are created with custom wrappers (see *wrapper* in subsection 4.1) that have to be developed and tailored for every tool in the back-end.

Tools in the back-end operate in one of the two server modes: shared or dedicated. A shared server is shared by multiple clients and there is never more than one instance of a shared server running. A dedicated server is dedicated to a single client. Each client that requests a connection to a dedicated server will cause a separate instance of it to be launched, and that server will not be shared with any other clients. Therefore, there can be several instances of a dedicated server running simultaneously.

Communication between the agents relies on the Common Gateway Interface (CGI). Agents communicate with the PPC using a custom protocol through sockets. The Controller communicates with the Pyro Naming Service in the back-end via a PYRO protocol (based on top of TCP/IP). All the interactions between the Pyro server and client objects also use the PYRO protocol.

4. Configuring the Environment

4.1 Micro-level

The *M-structures* are in fact models on their own and they are registered, discovered, and processed just like any other model by the trader and mediator agents. Their registration also requires an MRD record, and their location is not specific to any part of the environment. However, because of its importance to the functional operation of the environment, all the M-structures are typically registered by the administrator and are stored in a certain controlled location.

Abstract Resource Views are managed in the same way as M-structures. By default, the mediator agent has access to at least two simple transformations that enable users to *view* ARVs and to *create* their instances.

Transformations are implementations of useful high level functions that involve and are developed for a particular tool. All transformations are stored in a library local to the PPC, and their interface counterparts are usually stored local to the agents (Sun and Linux in Figure 4). Transformations do not necessarily involve interaction with scientific applications, but may also execute simple 'helper' functions.

Wrappers encapsulate the I/O of tools in a generic way and comply with the DOT system used for a particular environment instance. In our case, a wrapper consists of two parts (Python scripts): the server part (implementation of the encapsulation itself) and the (very thin) client part. The former creates a server object and registers it to the DOT system; the latter creates a dynamic proxy object of the server object on the client side. The server part is stored local to the respective tool, and the client part is stored in a library local to the PPC (WinNT and Sun in Figure 4).

Tools are any software packages or executable programs that support at least a standard-I/O interface. Most of the legacy systems fall into this category. The spectrum of tools can vary from complex applications such as the numerical package MatLab to home-made executables that implement simple 'helper' functions, like transforming data from one format to another. Tools can run on various hardware and software platforms, as long as the DOT of choice can run there too.

4.2 Macro-level

Adjusting the PP to a concrete project or adding a new scientific tool can be seen as a two step process. First of all, for each new tool a wrapper, that would encapsulate it as DOT object and provide an interface to it, has to be developed. Secondly, the administrator of the environment has to install the tool and the wrapper on one of the environment's servers, create and register a new server object to the DOT system, and update the Processing Platform Controller's model.

In our case, the above-mentioned wrapper contains a remote Python class, which implements a Pyro server object on behalf of the new one. An administrator creates a Pyro Daemon object (if it doesn't exist yet) on the same machine (see WinNT in Figure 4), and connects an instance of the remote class to the daemon as server object under a distinct name (which later will be used by clients to find the server object). Consequently, the server object is registered to the NS.

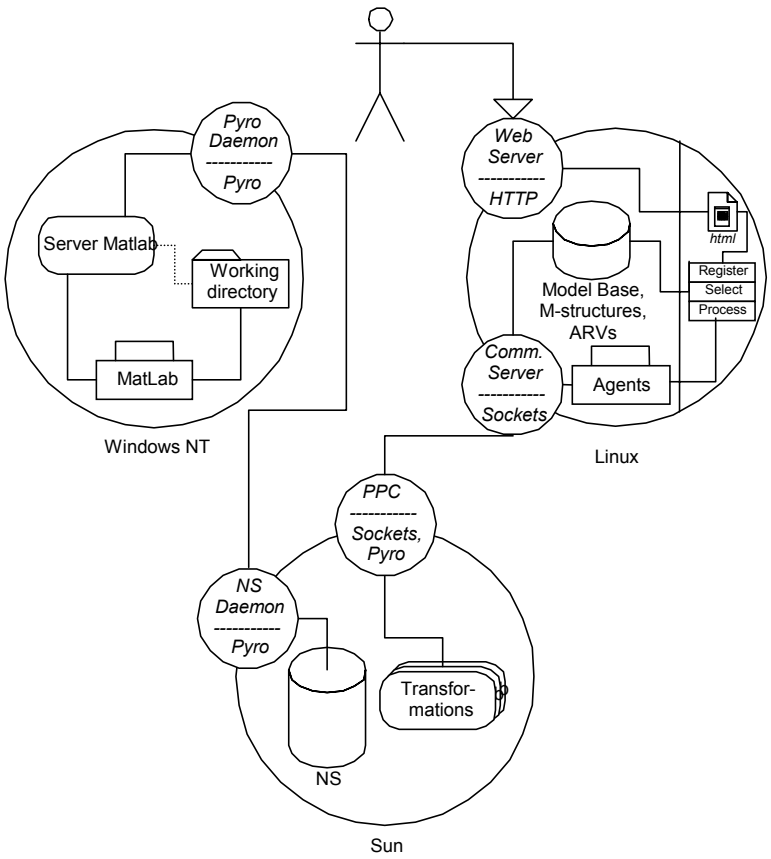

Figure 4. An Instance of the Proposed Environment

There can be many instances of the same tool running in the back-end, but only one server object is created and registered to the Pyro NS. Once this is done, the client part of the new wrapper can be added to the library of wrappers local to the PPC and the creator of the wrapper has to create an instance of ARV for the new tool, as well as a new ARV class for models supported by that tool.

As new transformations are being created for the ARVs of the configured tools, these transformations are stored in a library local to the PPC. Consequently, the creator of the transformation has to create and register to the trader agent a new M-structure, or update an existing one.

The final step is to update the model of the Controller: Using the POKer language, the administrator creates a new instance of a resource class with a desired capacity and adds it to the model. In case of a shared tool, the capacity of the respective resource in the PPC should be set to 1. Then multiple clients working with the shared tool will be synchronised in time by the PPC to prevent them from affecting each other's sessions. In case of a dedicated tool, it is reasonable to control the workload caused by the instances a dedicated tool can launch. Because of the heterogeneity of tools (often legacy ones), the granularity of control occurs at the level of tools' instances. This control is executed by the PPC via the capacity attribute of the resource. The value of the capacity defines how many instances of the particular tool are allowed to run concurrently. The PPC will immediately grant any incoming requests as long as the number of the instances is below the capacity limit. Otherwise, it will synchronise requests like in the case of a shared tool.

5. Working Example

We now describe an example user session to show how the environment operates from the top to the bottom (see Figure 5). The demonstration will involve a transformation on the tool MatLab and be carried out on the instance of the environment as shown in Figure 4.

Let us take the situation that a scientist has discovered a MatLab model (located somewhere on the Internet) via the trader agent. The model itself is located at a site maintained in (let us assume) London. The processing platform with the tool MatLab is located in Amsterdam. The selected model belongs to the class of 'MatLab' models, for which there exists an associated M-structure, which enables the mediator agent to automatically provide the user with a list of available

actions applicable to that class of models (Linux in Figure 4). Upon selection of an action by the user (e.g., 'Open'), the agent de-references the ARV name for the action implementation, a transformation that knows the appropriate tool and knows *what* to interact to that tool. If applicable, an interaction with the user occurs via the interface part of the transformation, before the agent initiates a request to the PPC to execute that transformation on MatLab.

The Controller located on Sun (see Figure 4) reacts to the request according to the state of the tool that is addressed by the request: the request is either granted or delayed until all the previous requests (if any) from the waiting queue for the tool MatLab have been processed. Once the state of the resource MatLab in the Controller's model becomes free (as the result of finishing processing all previous requests), our request is granted: the PPC locates the transformation by its name in the library of transformations, and processes it with a *Session Manager* (SM) in a separate thread.

The SM manages a transformation session with the target tool MatLab on WinNT machine (see Figure 4). In this session a MatLab server object is located via the Pyro NS; the model is downloaded from the remote location in London and transferred into the temporary directory local to tool MatLab, and the server object interactively executes MatLab commands as is indicated in the transformation body. For transformation 'Open', it would be the following sequence: *load*, *execute*, and *save*. Respectively, an instance of MatLab loads and executes the model (the M-file as well as all the other linked M and MAT-files), saves the resulting workspace into a temporarily file and clears its workspace for the next session. This concludes the current session with the tool MatLab. As the last step, the SM creates an ARV instance for the output model (a MAT-file), and passes this instance as an argument to the mediator agent. This allows the agent to pre-associate available meaningful actions (derived from the respective M-structure) with the output model.

The agent provides the user with a web page, which enlists the actions applicable to the produced output. For example, the scientist can choose between 'Display' and 'Register'. These actions would initiate execution of the respective transformations. Steps in the execution of these transformations are similar to those just outlined above. The process as is depicted in Figure 5, can be applied again to the result of the previous iteration. This can be continued until there are no more ARV's can be associated with the output or/and the produced output becomes very basic (e.g., a jpeg file).

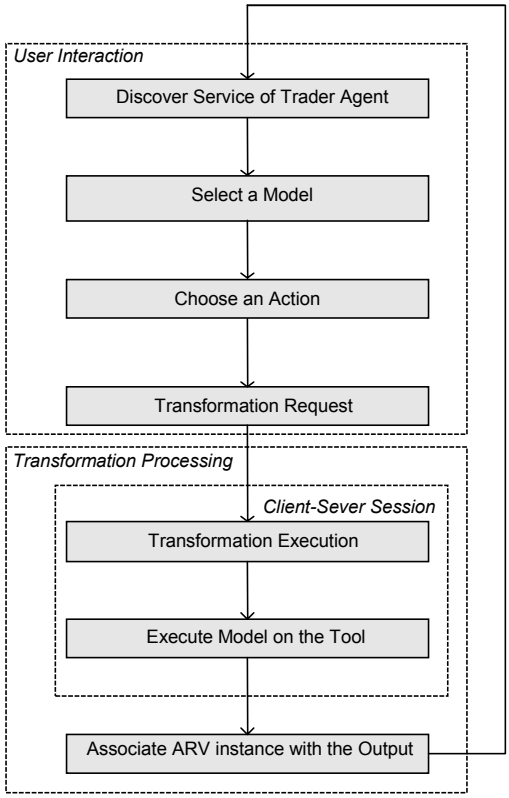

Figure 5. Top to Bottom View on the Execution Process

6. Final Remarks

The work considered in this paper aims at developing and building a generic collaborative environment to support on-line research and development processes, in which simulations on distributed solvers, sharing and reuse of various models in different formalisms and formats are involved. We have presented an approach in which meta-level concepts are used to deal with heterogeneity of models and tools, and a process-oriented discrete-event system simulator is used to complement a distributed objects system with access control to the objects in the back-layer.

Future works will concern further elaboration of the meta-level concepts in order to accommodate semantic aspects of models (such as formalism, experimental frame, etc.), complex experiment scenarios and extend the functionality of the environment with a modelling service. On the implementation side, we plan to provide a user-friendly modelling front-end and consider moving from Pyro to a more general, system and language independent, Distributed Object Technology system.

Acknowledgement

The research reported in this paper is done in the framework of the NanoComp-project [1], sponsored by the TU-Delft.

References

[1] NanoComp homepage: http://nanocom.et.tudelft.nl

[2] A. D. Malony, J. E. Cuny, J. L. Skidmore and M. J. Sottile: "Computational experiments using distributed tools in a web-based electronic notebook environment" Elsevier Science, Future Generation Computer Systems, Vol. 16 (5) (2000), pp. 453-464.

[3] T. Haupt, E. Akarsu and G. Fox: "WebFlow: a framework for web based metacomputing" Elsevier Science, Future Generation Computer Systems, Vol. 16 (5) (2000), pp. 445-451.

[4] P. S. Coe, F. W. Howell, R. N. Ibbett and L. M. Williams: "Technical note: A Hierarchical Computer Architecture Design and Simulation Environment" ACM Transactions on Modeling and Computer Simulation, Vol. 8, No. 4, pp. 431-446, October 1998.

[5] S.Scherber, K.Wöllhaf, C.Müller-Schloer: "An Industrial Approach for the Simulation of Complex Heterogeneous Systems" SCS: paper of the 14th European Simulation Multi-conference (ESM), pp. 318-322, May 2000.

[6] A. Saran, D. Agrawal, A. El Abbadi, T. R. Smith and J. Su: "Scientific Modeling using Distributed Resources" Proceedings of the fourth ACM workshop on Advances on Advances in geographic information systems, pp. 68 – 75, 1997.

[7] Dublin Core Metadata Initiative Homepage: http://dublincore.org/index.shtml

[8] Python Homepage: http://www.python.org

[9] CORBA Homepage: http://www.corba.org

[10] Java RMI: http://java.sun.com/j2se/1.3/docs/guide/rmi

[11] High LA Homepage: http://www.dmso.mil/index.php?page=64

[12] A. Buss and L. Jackson: "Distributed Simulation Modelling: a Comparison of HLA, CORBA and RMI" Proceedings of the 1998 Winter Simulation Conference, pp. 819-825, 1998

[13] Python Remote Objects Homepage: http://sourceforge.net/projects/pyro

[14] Dublin Core Metadata Element Set: http://purl.oclc.org/dc/documents/rec-dces-19990702.htm

[15] A. Levytskyy and E.J.H. Kerckhoffs: "A Plain Python Simulator to Control a Collaborative Environment" SCS: paper of the 14th European Simulation Multi-conference (ESM), pp. 719-727, May 2000.

[16] A. Levytskyy and E.J.H. Kerckhoffs: "POKer, a Process-Interaction Simulator and Controller for use in Collaborative Simulation" SCS: paper of the 2nd Middle East Symposium on Simulation and Modelling (MESM), pp. 12-20, August 2000.

[17] G. Birtwistle and C. Tofts: "An operational semantics of process-oriented simulation languages: Part 1 πDemos" Trans. Soc. Comput. Simul., 10 (4), pp. 299 – 333, December 1994.

Author Biographies

EUGENE J.H. KERCKHOFFS holds a MSc-degree from Delft University of Technology (1970, Physical Engineering, thesis on analog and hybrid computer simulation) and a PhD-degree from the University of Ghent (1986, Computer Science, thesis on parallel continuous simulation). Currently, he is an associate professor at Delft University of Technology (Faculty "Information Technology and Systems", Department "Mediamatica", Group "Knowledge-based Systems"). He is also chairholder of the SCS Chair in Simulation Sciences at the University of Ghent, Belgium.

ANDRIY LEVYTSKYY graduated from Chernivtsi State University, Ukraine and holds a M.Sc. degree in Computer Science. Currently, he is a Ph.D student at Delft University of Technology, Faculty "Information Technology and Systems", Department "Mediamatica", Group "Knowledge-based Systems".