Integrating Model Transformation Systems and Asynchronous Cluster Tools

Gergely Mezei, Sándor Juhász, Tihamér Levendovszky

Department of Automation and Applied Informatics
Budapest University of Technology and Economics
Goldmann György tér 3, H-1111 Budapest, Hungary
{gmezei, sanyo, tihamer}@aut.bme.hu

Abstract: Our primary research focuses on creating an efficient model transformation method that is based on graph transformations. We have found that there are cases when using single computers to compute the transformation results exhibits unacceptably slow transformations. Building clusters from computers is a well-known method to increase the computing capability in a cost-effective way. Thus, we focus on creating a distributed model transformation method based on cluster systems. In this case, the efficient communication between the computers in the clusters is essential. Therefore, a cluster management system has been chosen, which supports asynchronous, message-based communication. Cluster systems and distributable tasks in these systems are usually based on the C++ language, while our model transformation engine uses C#, which means that interoperability is required between the two fields. This paper describes the integration issues in our solution including an architectural overview and detailed information about the interoperability method.

Keywords: Cluster systems, Model Transformation, .NET, Interoperability

1 Introduction

Nowadays model-driven software development is very popular in the field of software design. Model-Driven Architecture (MDA) [1] is one of the most popular model-based approaches to facilitate the synthesis of applications from domain specific models using model transformation. Model-Integrated Computing (MIC) [2] is another model-based solution. MIC advocates the use of domain-specific languages with high level of abstraction and customizable notation, which helps to increase productivity and produces shorter development cycles. Model transformation plays an essential role in MDA, MIC and in other model-driven solution as well. The growing popularity of modeling made it necessary to create flexible transformation techniques, which are efficient with respect to performance at the same time. Several transformation approaches have been created that fulfills

these requirements. One of the most popular techniques is graph transformation. This approach handles the models as labeled, directed graphs, where the transformation means transforming the input graph into an output graph. Graph transformation is often based on graph rewriting that allows using the strong mathematical background of graph rewriting. Moreover, graph rewriting rules and the transformation itself can be modeled in a visual way using special domain specific languages. However, the complexity of a single rewriting step is $O(n^k)$, where *n* is the size of the input graph, and *k* is the size of the pattern defined in the rewriting rule. This means that the efficiency of the graph transformation is heavily affected by the size of the input graph and the size of the patterns used in the rewriting.

Visual Modeling and Transformation System (VMTS) [3] is an n-layer metamodeling environment supporting graph rewriting based graph transformation. Graph transformation has been successfully applied in several fields using VMTS, such as transforming class diagrams to database models [4], or generating source code from visual model definitions [5]. In these cases, we have found that existing graph transformation techniques are not efficient enough, because of the exponential time required by the rewriting algorithm. Although there are several algorithms that can reduce the complexity of rewriting steps in special cases, but there does not exist any universal solution. Thus, we have chosen to increase the computing capabilities instead of reducing the complexity of rewriting.

The performance of a computational system is basically determined by the speed of participating system components and by the architecture between these components. The speed of the components is limited by the applied technology, but using more than one instances of the same component, and organizing their cooperation can further expand the perspectives of the processing power. This principle is widely used at different levels of the information systems, one of the most popular among these are cluster systems. Cluster systems are composed of nodes (computers) more or less physically separated. Although using cluster systems is a popular way to increase the performance, they still lack to answer some modern requirements, because they are based on structured programming languages extended with communication primitives.

Pyramid is a cluster management system, which allows uniting PCs with heterogeneous operating systems into a single virtual computer [6]. The parallel programs written for Pyramid operate as services over a general distribution layer, and offer higher-level functions needed to produce the desired result in a distributed way. Additionally, the communication in Pyramid is based on asynchronous messages, which helps to reduce the communication overhead. The presented features of Pyramid made it an ideal solution for us to improve and distribute our model transformations.

This paper presents the integration issues of VMTS model transformation system and Pyramid cluster management tool. The paper is organized as follows. Section 2 describes the background of the paper including the introduction of VMTS and Pyramid. Section 3 describes the architecture of the distributed model transformation approach, and elaborates the interoperability issues used in the solution. Finally, Section 4 concludes and exposes future work.

2 Backgrounds

2.1 Visual Modeling and Transformation System

The Visual Modeling and Transformation System (VMTS) is a highly customizable modeling environment for creating, or editing visual languages. VMTS also provides a model transformation framework to transform the models to other models, or to source code. VMTS consists of several subsystems (Fig. 1).

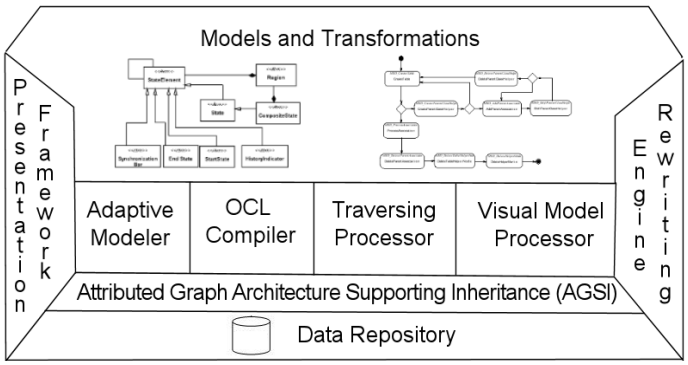

Figure 1
VMTS architecture

Attributed Graph Architecture Supporting Interface (AGSI) offers a high-level interface for the other components to reach the underlying data repository. VMTS Presentation Framework (VPF) is solution to support creating, displaying and editing the models in a graphical environment. Adaptive Modeler is an application based on the VPF; it provides an easy-to-use user interface for the core functions of VMTS. Model transformation is applied using Rewriting Engine and Visual Model Processor. VMTS offers another built-in way to process models: Traversing Processors are available for the traditional model interpretation in C#. Constraints in modeling and model transformations are compiled to a validation binary by the OCL Compiler module. This binary is used every time, when validation is required.

2.2 Model Transformations in VMTS

VMTS uses a visual approach, the VMTS Control Flow language [7] to describe the control flow of the transformation. This language defines a strict execution order between the transformation steps. The approach uses stereotyped UML activity diagrams as graphical notation. VMTS Control Flow can be modeled in the same visual way as other models. The steps of the transformations specify the operational behavior of model processing, where the main elements are graph rewriting rules. The control flow also supports conditional branching, parameter passing (called *external causalities* [8]), and hierarchical decomposition of the transformation.

The steps of the transformations, i.e. the graph rewriting rules consist of a Left-Hand Side (LHS) and a Right-Hand Side (RHS). Applying the rewriting rule means to find the pattern defined in LHS in the host model (in the input model) and replacing it with the pattern defined in RHS [8]. VMTS uses metamodel-based rewriting, thus the LHS and the RHS are built from the metamodel elements. It is possible that LHS and RHS use different metamodels (the input and output model of the transformation can differ). Transformation rules can contain OCL constraints to define additional constraints for the pattern [8].

2.3 Pyramid

The main idea of Pyramid is to build a high performance virtual computer from desktop computers. The computers of the clusters may enter, or leave the system any time, thus, besides the heterogeneous operating systems also the dynamic changes in the participating computers must be supported. Pyramid was designed to exploit the event-based programming method, which enables the service-based use of shared remote resources.

The components in a Pyramid cluster are represented by tasks that are running instances of different services. A service is a task prototype, containing a set of predefined functions that can be reached through messages. A computer in the cluster can host any number of Pyramid nodes, it can run any number of tasks. A service describes the accepted and generated message types, their parameters and the protocol that must be followed during its usage. The Pyramid services are organized and stored in dynamically (run-time) loadable modules. For a service to become available, its module must be registered.

The architecture of Pyramid system has a layered structure (Fig. 2), built up of three main components: the host operating system, the distribution layer, and the Pyramid tasks. The distribution layer is a thin layer responsible for location transparency. This layer provides primitives for communication, process handling, and graphical functions. The layer also includes a node handler task called *Pyramid Manager* is running on each node, i.e. on each computer in the cluster.

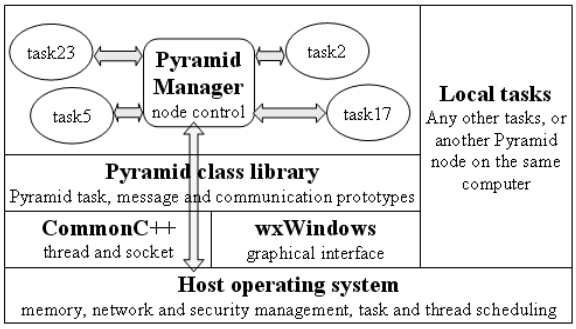

Figure 2
Architecture of a node in a Pyramid cluster

Pyramid Managers form the run-time part of the distribution layer; it allows handling inter-node message passing and node administration efficiently. To make the task and the system itself portable at source code level, the distribution layer is based on two portable class libraries: the thread, socket and synchronization management is provided by *Common C++* [9], while the common graphical user interface is built on *wxWindows* [10].

Pyramid tasks are built on the top of the distribution layer. Pyramid is based on object-oriented technology, more precisely on *Pyramid Class Library*, which contains the objects for messaging, addressing and task management. To create a task in Pyramid, a new class must be derived from the generic base class *PTask*, so the (i) message processing loop, (ii) the node Manager connectivity, (iii) the cluster level message queue handling mechanism, (iv) the remote console, (v) monitoring facilities and (vi) other features are simply inherited from the base class. The new class is compiled to create a dynamically loadable assembly module that can be loaded and used as a service at run-time by the any node Manager. Each task should by identified by a unique service identifier.

The C++ language combined with object-oriented technology makes efficient and easy-to-use the assembly and the extraction of inter-node messages. A series of extraction and assembly operators are defined for the Pyramid messages, being capable to serialize not only the predefined types, but also any custom type derived from an empty template data type called *PObject*. The only rule to respect is that the message content must be read in the same order as it was packed with the operators.

The communication channels between the tasks are provided by the Pyramid Managers. Since Pyramid uses an event-based message passing system, communication means dealing with the message flow among the tasks. Every task possesses a message queue, where the incoming messages must be placed, while the way and the order of their processing are left to the decision of the task. Each message of a node arrives to the node Manager, because all the tasks of a node

appear to the outer world through the same TCP port present in their identifier. The Manager maintains an inventory of all the tasks on the nodes, thus, it is capable of distributing the received messages to the appropriate message queues. In Pyramid, the Manager plays an active role in the message sending operations as well. This solution avoids blocking the sender task during the slow message sending through the network. To avoid serialization of message handling inside the Pyramid Manager, multiple communication threads are used to support more incoming and outgoing messages at the same time. Despite the parallelism, the mechanism is guaranteed to preserve the message order between any two specific tasks.

To find the services registered on different nodes and to keep record of the continuously changing system configuration, it is crucial to have an information service. This role is played by the *directory service* in Pyramid. The directory service keeps record of all the nodes available in the cluster and the services registered on them. Pyramid Managers know the location of the directory service from their configuration files; Managers forward this information to each new task of the node at their creation. As all the cluster information is present at the very same location known by each task, it is possible the find any service in the cluster simply by questioning this main database. To avoid performance bottlenecks and to increase scalability actually the directory service is a hierarchy of directory service providers. Every Manager is configured to connect to one preferred server, which is responsible for maintaining information of only a part of the whole cluster. The directory servers are arranged in a tree, and are able to organize their cooperation for information retrieval in this structure. The *single point of failure* situation is eliminated by using backup directory services. The backup directory services are present at every level of the tree, and their content is synchronized by the corresponding primary service at predefined time intervals. If the primary service fails, the secondary service takes its place, and acts like a primary service until the original directory service recovers.

2.4 Interoperability

Software systems often require interoperability solutions to exchange their data. Interoperability is especially popular if the communicating systems are based on different software platforms. In the distributed transformation approach, presented in this paper interoperability is needed between Pyramid and VMTS, because Pyramid is mainly based on standard, unmanaged C++ code, while VMTS components use managed assemblies. Managed code means here that the Common Language Runtime (CLR) [11] controls the code execution. According to [11] there exist three main ways to create a bridge between unmanaged C++ assemblies and managed libraries: (i) Platform Invoke, (ii) COM interop and (iii) C++ interop.

Platform Invoke is possible the most popular technique among the interoperability solutions. It allows managed code to call unmanaged functions exported from unmanaged dynamic link libraries (DLLs). In this case, CLR handles loading of the DLL file and hides parameter marshalling from the programmer. Platform Invoke was originally created to reach unmanaged code from managed code, but the inverse direction can also be applied by using callback functions.

COM interop allows managed components to interact with COM objects trough COM interfaces and clients. COM components must be registered for interop before using them. Registration procedure is easy to use if the component does not change, but it is inflexible in other cases. Additionally, COM interop means a relatively large interoperability overhead.

C++ interop, also called implicit Platform Invoke, allows managed and unmanaged code to co-exist in the same application. This means a higher level of comfort when in creating the interoperability bridge between VMTS and Pyramid. C++ interop requires the managed components to use Visual C++.

3 Integration

3.1 Distributed Model Transformations

In the case of graph transformations, parallel execution can be applied on the level of transformation steps, and on the level of rewriting rules. Existing graph transformations usually focus on the acceleration of the second case, more precisely on parallel pattern matching in graphs. In VMTS, both kinds of parallelization are supported. Although there are several differences between the cases on the implementation level, the architecture is the similar (Fig. 3).

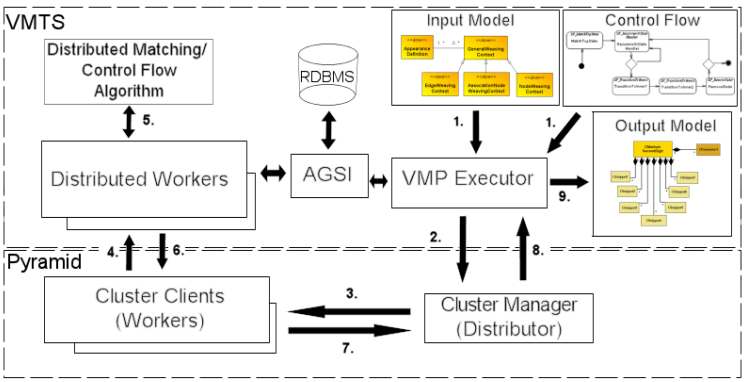

Figure 3
Architectural overview

Although database relations are represented in the architectural overview with two headed arrows, they are not detailed here.

The steps of the distributed transformation are the following: (i) The input (host) model and the transformation control flow are selected, and they are sent to Visual Model Processor Executor (VMP Executor), which is the graph transformation engine. (ii) VMP Executor initializes the transformation using AGSI and decides the order of execution between the transformation steps by analyzing the transformation control flow. The initialized data is sent to the Cluster Manager component. (iii) The Cluster Manager component obtains information about the available worker units (Cluster Clients) using the library service of Pyramid. Then, Cluster Manager decides the distribution strategy according to the number of task and the number of available workers. The description of the transformation information (referred to as *Worker Package*) is sent to Cluster Clients. (iv) Cluster Clients forward the received data to the Distributed Workers module. (v) Distributed Worker completes the task using a distributed algorithm and AGSI to obtain the underlying model items. (vi) The Distributed Worker sends the results to Cluster Clients and (vii) to Cluster Manager. In this phase, according to the initial parameters, Cluster Coordinator can send a message to other Cluster Clients. For example, if only one possible match were requested, the other Cluster Clients should be shut down. (viii) The result is sent to VMP Executor, which (ix) creates the output model.

The distribution of rewriting rules works similarly, but in that case, the *Worker Package* describes not a whole transformation step, but a sub-match of the rewriting rule under execution. In this case, the Workers (Clients) try to find the rest of the LHS pattern and apply the rewriting. This distribution is especially useful, if all possible matches are needed. To distinguish between the two cases the expressions, the terms *Transformation Worker Package* and *Rewriting Worker Package* are used.

3.2 The Distribution Method

On the level of rewriting rules, the key of efficient parallelism is to find independent matches in the host model. Starting nodes (the nodes matched first) are chosen using statistical information of the host models. Parallelism is based on these start nodes, since matching does not modify the host graph, and at least one model item is different in the partial matches. A worker package is created for each possible start node, thus, matching is applied in *n* parallel thread, where means the number of possible start nodes. For example if the first node of the pattern can be matched to three nodes in the host model, then three *Rewriting Worker Package*s are created, and they can be executed in parallel.

Parallelism is a bit more complex on the level of transformations, where distribution points can be set manually, or they can be detected automatically by analyzing the transformation control flows.

Explicit distribution points are supported by *Fork* and *Join* elements in the VMTS Control Flow Language. These elements specify where the parallel executing should begin and where it should terminate, i.e. where the parallel branches should merge. *Fork* elements produce several *Transformation Worker Package*s at the same time, one for each parallel branch. Each worker computes one of these branches. Computing results are summarized in *Join* elements, where the transformation engine waits until the result of the last branch is received. Additionally, VMTS can simulate parallel execution on a single computer, thus, it is recommended, but not required to use a cluster system, when running control flows with these special kinds of elements.

Finding the parallelizable steps automatically is a more complex task. Here, the solution is based on Parallelism theorem for metamodeling environments [12]. The detailed introduction to these theoretical results is beyond the scope of this paper, but we point out that the examination of independency between the transformation steps produces a dependency matrix that is used to decide which steps can be executed in parallel.

3.3 Integration Issues

According to the architectural overview (Fig. 3) and its explanation, integration is required in steps 2, 4, 6 and 8. This integration is based on message passing between the modules of VMTS and Pyramid. The main difficulty of this approach is that VMTS and Pyramid use different programming platforms as mentioned earlier. More precisely, AGSI is based on managed code and C# language. This means that all VMTS components have to communicate with this managed assembly. In contrast, Pyramid can handle tasks only inherited from *PTask* class and written in unmanaged C++. Thus, interoperability between VMTS and Pyramid cannot be avoided. Using COM interop is in this case would mean a significant communication overhead, because the wrapper upon Pyramid modules and accessing COM components would have limitations on usable data types. Additionally, managing COM components would also require manual administration of the participating modules. Platform Invoke would be a better, more flexible solution, but in this case, the unmanaged assemblies have to be extended by explicit DllImport attributes to define entry points for the managed code. C++ interop offers a better type safety, it is easier to follow the changes of the unmanaged libraries (changes of Pyramid). C++ interop also makes performance enhancements available, which are not possible with Platform Invoke. The main advantage of C++ interop is this performance tuning possibility that allows the programmers to reduce the interoperability overhead by writing

optimized code. Besides the advantages of C++ interop, it has some drawbacks as well. C++ interop works only with C++ managed code, thus, model transformation modules need to be written in managed C++, not C#. Additionally, in case of C++ interop marshaling the data in communication is handled implicitly. Therefore, the structure of the interface between VMTS and Pyramid can heavily affect the performance of interop communication. According to the presented requirements, VMP Executor and Managed Worker (Distributed Worker) component is written in managed C++, while Cluster Manager and Cluster Client uses unmanaged C++ code. Here component means binary assembly (a .dll file).

The main communication steps between the components are established as follows (the parameters of the methods are not shown, for sake of simplicity):

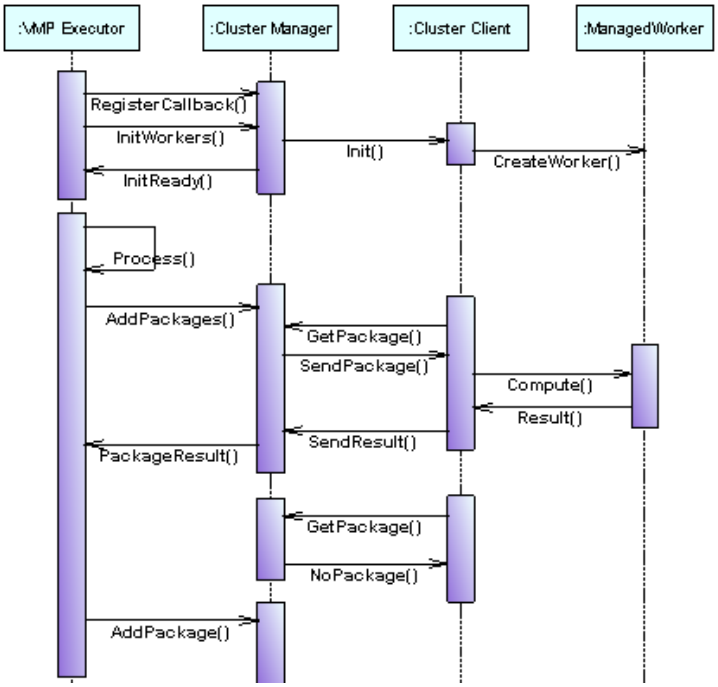

Figure 4
Communication between components

After selecting the host model and transformation, VMP Executor initializes the Cluster Manager component and registers a callback function, which is used by Cluster Manager, when it sends a message to VMP Executor. After the callback is registered, VMP Executor starts initializing the workers. Initialization in distributed workers mainly consists of (i) creating the managed component (Managed Worker), and (ii) setting the transformation and host model passed by the *Init* method.

After initializing the workers, VMP executor starts the transformation. It traverses the transformation control flow and creates worker packages, if required. These packages are sent to Cluster Manager that has a package list about the currently waiting packages.

When cluster clients have finished the initialization of their Managed Worker, starts to get packages from Cluster Manager in an infinite loop. They do not have information about the length of the package list in Cluster Manager, thus, sometimes they receive empty packages only (*NoPackage*). If the received package is not empty, then Cluster Client forwards it to Managed Worker that computes the result.

To support parallel branches in the control flow, VMP Executor does not always wait for the result of the packages, but continues the execution on different parallel paths. The results are summarized at the end of the parallel branches.

Interoperability mainly consists of parameter passing of the transformation, or partial match identifiers. These identifiers – originally stored in AGSI classes – are serialized to strings when sending the data from managed to unmanaged code, and restored by applying the reverse direction.

Conclusions

Model transformation plays an essential role in model-driven software engineering approaches. Graph transformation is one of the most popular model transformation methods. Existing transformation solutions cannot always compute the results of the transformation enough fast. Since the complexity of the transformation method cannot be reduced in general, it is a natural idea to accelerate the transformation by increasing the computing capabilities. Cluster systems offer a cost-efficient way to reach the required performance. To be able to use the advantages of cluster systems, distributed model transformations are required. The theoretical basis for this distribution has been created, but none of the existing transformation engines has succeeded to create an implementation for distributed transformations in cluster systems.

This paper has presented the integration experiences gained from integrating our metamodeling and model transformation tool, VMTS and an asynchronous cluster system, Pyramid. Although the presented solution relies on the concrete implementation of VMTS and Pyramid, it contains general ideas, applicable in any other similar approach. The paper has presented not only an architectural overview of our solution, but it has also elaborated the interoperability issues and the main communication steps between the components.

Future work focuses mainly on how to improve the detection of parallelizable steps in the transformation control flow and in partial matches of the rewriting rules.

Acknowledgement

The found of 'Mobile Innovation Centre' has supported, in part, the activities described in this paper.

References

[1] OMG Model Driven Architecture homepage, www.omg.org/mda/

[2] MIC Official Homepage, http://www.isis.vanderbilt.edu/research/research.html

[3] VMTS Official Homapage, http://vmts.aut.bme.hu/

[4] L. Lengyel, T. Levendovszky, G. Mezei, H. Charaf: *Model-based Development with Strictly Controlled Model Transformation*, In Proc. The 2nd International Workshop on Model-Driven Enterprise Information Systems, MDEIS 2006, Paphos, Cyprus, pp. 39-48

[5] Forstner B., Lengyel L., Levendovszky T., Mezei G., Kelényi I., Charaf H.: *Model-based System Development for Embedded Mobile Platforms, In Proc* 13th Annual IEEE International Conference and Workshop on the Engineering of Computer-based Systems (ECBS)

[6] Juhász S, Charaf H.: *Building Clusters on Modern Desktop Operating Systems*, International Conference on Applied Informatics (section: Parallel and Distributed Computer Networks), February 10-13, Innsbruck, Austria, 2003, Proceedings pp. 547-552

[7] L. Lengyel, T. Levendovszky, G. Mezei, H. Charaf: *Control Flow Support in Metamodel-based Model Transformation Frameworks*, EUROCON 2005 International Conference on 'Computer as a tool', Proceedings of the IEEE, Belgrade, Serbia and Montenegro, November 21-24, 2005, pp. 595-598

[8] László Lengyel PhD Thesis, Budapest University of Technology and Economics, 2006

[9] Common C++ Official Homepage, http://cplusplus.sourceforge.net

[10] wxWindows Official Homepage, http://www.wxwindows.org

[11] MSDN - Interopability, http://msdn2.microsoft.com/en-us/library/ms235292.aspx

[12] Tihamér Levendovszky, PhD Thesis, Budapest University of Technology and Economics, 2005