Visual Debugging Support for Graph Rewriting-based Model Transformations

Tamás Mészáros [1], Péter Fehér[2], László Lengyel [3]

Department of Automation and Applied Informatics,

Budapest University of Technology and Economics

Magyar Tudósok krt 2, Budapest, Hungary, H-1117

[1] mesztam@aut.bme.hu, [2] feher.peter@aut.bme.hu, [3] lengyel@aut.bme.hu

Abstract— **Graph rewriting-based model transformation is an essential tool with a strong mathematical background used to process graph-based models. The most recent modeling tools tend to support the definition of rewriting rule based transformations in a visual or textual way. However, only a few of these support the continuous animation of the modifications performed on the models, which makes the traceability and the debugging of transformations more challenging. This paper presents the visual model transformation debugger realized in the Visual Modeling and Transformation System. The solution facilitates the step-by-step execution of model transformations, the visualization of the overall state of the transformation and also supports individual matches. Furthermore, it provides the possibility to influence the behavior of the transformation at runtime. The realized features are illustrated in a case study from the MATLAB Simulink domain.**

Keywords: model processing, model transformation, graph rewriting, model transformation debugging, MATLAB Simulink

I. INTRODUCTION

Domain-specific modeling is a powerful technique to describe complex systems in a precise yet understandable way. The effectiveness of domain-specific modeling lies in the application of domain-specific languages to describe a system. Domain-specific languages are specialized to a concrete application domain; therefore, they are particularly efficient in their specific target area when compared to general purpose languages [1]

Models created using such languages often need further automated processing methods to utilize the information expressed by the models in real, end-user applications. The processing is similar to source code compilers, which convert readable source code to byte-code or machine code executed by the hardware or a virtual machine, but various model-to-model transformations are also common.

When developing a model processor for a domain-specific language, the ability to efficiently trace and debug the operations performed by the model processor is also important. The debugging support is a basic requirement of a usable developing tool. Currently, almost all software development tools facilitate the debugging of source code during the execution. This is also an essential requirement related to modeling and model processing environments. The goal of our work is to build upon a visual model transformation debugger tool that provides the main features (step-by-step execution, breakpoints, variable inspection, and so on) already widely used in integrated development environments.

We have realized our approach in the Visual Modeling and Transformation System (VMTS) [2] – a general purpose metamodeling environment supporting an arbitrary number of metamodel levels. Models in VMTS are represented as directed, attributed graphs, the edges of which are also attributed. The visualization of models is supported by the VMTS Presentation Framework (VPF) [3]. VPF is a highly customizable presentation layer built on domain-specific plugins which can be defined in a declarative manner.

VMTS is also a transformation system, which transforms models using template-based text generation or graph rewriting [4]. Templates are used mainly to produce textual output from model definitions in an efficient way, whereas graph transformations can describe transformations in a visual and more formal way. In this paper, we concentrate on the graph rewriting-based engine of the VMTS.

The rest of this paper is organized a follows. Section II summarizes the related work. Section III provides a short, generalized summary of graph transformations. Section IV introduces the graph rewriting-based model transformation approach of VMTS. Section V presents how model transformations can be visually traced and debugged using our solution. Finally, conclusions are elaborated.

II. BACKGROUND RESEARCH

AToM^3 [5] is a general purpose metamodeling environment including simulation and model animation features. AToM^3 supports graph rewriting-based model transformations with a graphical editor for the definition of the rules; furthermore, interactive debugging of transformations is also possible. The processed model can be animated according to the operations of the model transformation. The transformation can be executed in a continuous or step-by-step manner. A feature which is lacking inAToM^3 is the supporting of breakpoints and the checking of the result of a successful match is also not possible.

AToMPM [6] is a web-based form of metamodeling and model transformation environment and can be considered the successor of AToM^3.. It supports the visual specification of model transformations as well as the step-by-step execution of the individual transformation steps; however, it does not support breakpoints, the runtime modification of the transformations or the host model.

978-1-4673-2232-4/13/$31.00 ©2013 IEEE

Graph Rewrite And Transformation (GReAT) [7] is a visual language and toolset to define and execute graph rewriting-based model transformations. GReAT has an approach similar to that used in VMTS in terms of graphical rule definition and the sequencing of the rules with a control-flow language. GReAT provides advanced debugging features including step-by-step execution of rules and the application of breakpoints. The results of successful matches and the rewriting rules are also logged in detail. However, inspecting the operations of the transformations is not supported in a visual way, although one can trace the transformation with the help of a textual interface.

The Attributed Graph Grammar System (AGG) [8] is an environment for developing graph rewriting-based transformations. One can follow the execution of a transformation in AGG visually, including the applied rewriting rule and the host graph. The manual definition of matches is also supported in this environment. A transformation can run continuously or in a step-by-step manner, however, the process cannot be paused by predefined breakpoints in the rule-application sequence.

ATL [9] is a model transformation language and execution environment realized in the Eclipse framework [10]. ATL provides a textual language to define model transformations and it is also made possible to debug the transformations using the services of Eclipse. The debugging features include step-by-step execution, breakpoints, visualization of variables and matches. However, it does not support the runtime modification of the matches, the runtime editing of the transformation, or the underlying host model.

III. Graph Rewriting-based Model Transformation

Recall that in VMTS, models are represented as directed, attributed graphs. Model elements are represented by nodes, and the connections between the elements are defined by the edges of the graph. This representation demonstrates the applications of various graph transformation algorithms. Graph rewriting [4][11] is a powerful technique for applying graph transformations. A graph transformation consists of rewriting rules and each rewriting rule has two parts: a Left Hand Side (LHS) and a Right Hand Side (RHS). The LHS defines a model pattern (graph pattern) which must exist in the input model, while the RHS describes a substitute pattern to replace the match of the LHS. A rewriting rule most often contains additional constraints as well as imperative instructions. Using the constraints, one can further refine the LHS by ensuring that the attributes of the model nodes, matched by the LHS, satisfy several conditions. The imperative instructions may modify the values of the attributes of the matched nodes in case of a successful match.

Several rule description languages have been explained by the different tool developers. Some of them use a textual approach (like ATL [9], VIATRA2[12], GrGen.Net[13]) that describes both the graph pattern itself and, the constraints and the imperative code with textual languages. While other tool vendors prefer the visual representation of the graph patterns (like VMTS [2], AToM^3 [5] or AGG [8]) and let the user define the pattern to match and replace graphically.

A typical graph transformation consists of several such rewriting rules, and the transformation defines the order in which these rules should be applied. Depending upon the tool and the formalism, there are many ways to define the execution order of the individual rules. They may be ordered visually, using a control flow graph, as implemented in VMTS or AToM^3, or through imperative code such as in ATL or GrGen.NET. Both approaches apply the same concept: the transformation developer defines precisely which rules to apply and when to apply them. Another popular approach (originating in graph grammars [11]) is to organize the rules into layers, executed in a predefined, sequential order,, but the order of the execution of the rules within one layer is not defined (as in AGG).

IV. Graph Transformation in VMTS

In the VMTS Transformation Framework (VTF), the LHS and the RHS of the rewriting rules are not separated. They are defined in the same transformation model and the role of the elements (i.e., matched or deleted for elements in the LHS; and created for the RHS) is specified with the Action attribute of the rule items: (i) an element, x, of the LHS graph which is not deleted in the RHS graph is marked as *x.Action = 'match'*, (ii) elements created by the RHS graph are marked as *x.Action = 'create'* and (iii) elements deleted by the RHS graph are marked as *x.Action = 'delete'*. The creation and deletion of an element is also indicated in the concrete syntax: created elements have a blue background while deleted elements have a red base color.

Rewriting rules in VMTS [14] do not use the abstract syntax of the host model but rather a custom Domain-Specific Modeling Language (DSML) developed specifically for this purpose. The type of the model element created/matched by a rule element is defined by the *TargetTypes* attribute of the rule element. As it is permitted to specify multiple potential types for a single rule element to be matched, the *TargetTypes* attribute can hold more than one type of identifier as well.

The execution sequence of rewriting rules in VMTS can be defined using a control flow language [14]. With a control flow model, one can precisely define the execution order of the individual rewriting rules. In such a control flow model, we must have exactly one start node, at least one stop node, and one or more rule containers which indicate the execution of one specific rewriting rule (defined in another model). An example of such a rewriting rule and control flow model can be seen in Fig. 1. Here, the rewriting rule (*Get_OutEdges*) matches two node types and an edge (*outgoingEdge*) connecting them, and later deletes this edge in the rewriting phase. The control flow model executes this rule exhaustively (see the continuous arrow in the upper right corner of the rule container) meaning the same rule is executed as long as a match exists. Then, the transformation proceeds to the following rule.

In addition to the complied transformation engine, VMTS also has an interpreted solution that is ideal for the realization

978-1-4673-2232-4/13/$31.00 ©2013 IEEE

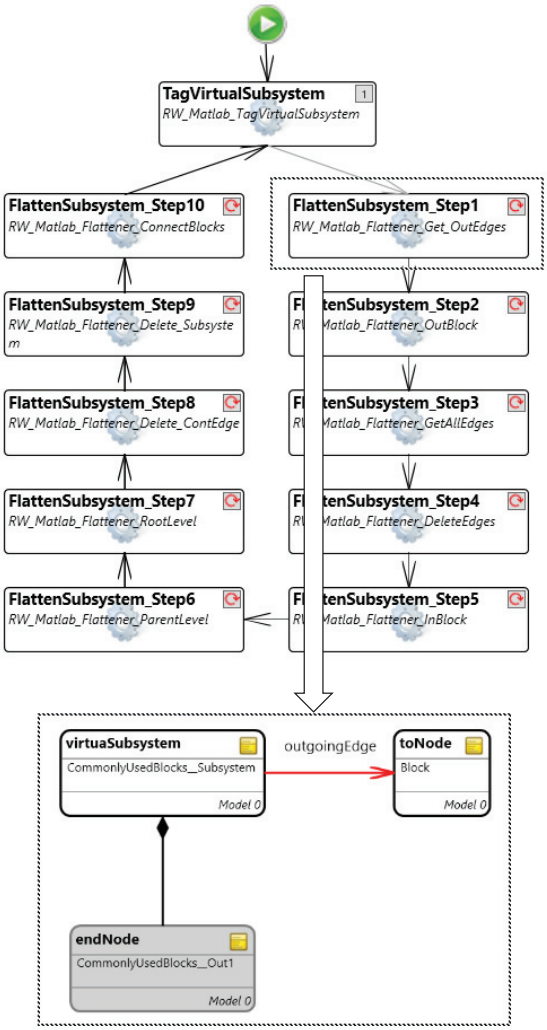

Fig. 1: Example control flow model and a rewriting rule.

of the model transformation debugger. While in the complied version both the execution logic of the individual rewriting rules and the execution order of the rules are precompiled, thus hardwired into the transformation binary. The interpreted solution provides more control over the execution steps and the execution order of the rules. In the next section, we introduce the visual debugging support of the approach.

V. Debugging Support

The general aim of a debugger is to enable the tracing of the runtime behavior of an application and intervene if necessary. We have focused on these two goals when designing and developing our solution to support model transformation debugging as well.

In the following, we summarize the main features of our approach and provide a case study from the MATLAB Simulink [15] domain to illustrate them further. The aim of the transformation is to flatten virtual subsystems (hierarchical sub models) in the Simulink model. This operation is usually performed prior to the translation of a Simulink model into executable source code. Fig. 2 illustrates a fragment of the complete transformation consisting of 27 rewriting rules. Fig. 3 presents the original host model both with the Simulink syntax and its representation in VMTS. In this model, we can see a subsystem that performs a data type conversion and also contains a feedback loop.

A. Stepwise execution

It is necessary that, during the debugging of an application, the developer can execute the code, line-by-line, and inspect the behavior of the application at each step. We have adapted this requirement into our model transformation debugging approach as well: we facilitate the execution of the transformation, either in continuous or step-by-step mode. In the latter case we can perform the individual steps of a transformation (matching, rewriting, and proceeding to the next rule), separately. However, at any time we may also switch from continuous to step-by-step mode and vice-versa. The overall state of the transformation is also reflected through the highlighting of the item currently being executed in the control flow. An example is shown in Fig. 2, the rule *Get_OutEdges* is highlighted. Fig. 4 shows the debugging control options in the VMTS toolbar.

B. Breakpoints

Often the activity to be inspected occurs somewhere deep in the transformation and we do not want to visually trace all the steps before that or simply jump over them. If the transformation has been started in continuous mode, it is possible to stop at predefined edges or rules by placing a *breakpoint* on them. This can be achieved by switching the *HasBreakpoint* attribute of the related control flow element to true, or by hitting the F9 button while an element in the control flow is selected.

For example, in the control flow presented in Fig. 2, a

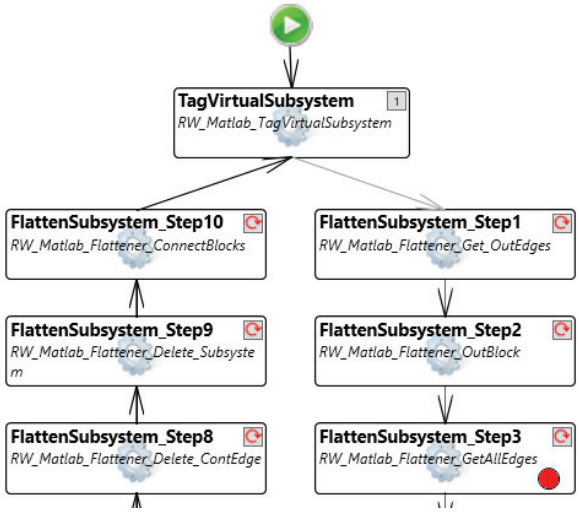

Fig. 2: Highlighting the current rule that also has a breakpoint.

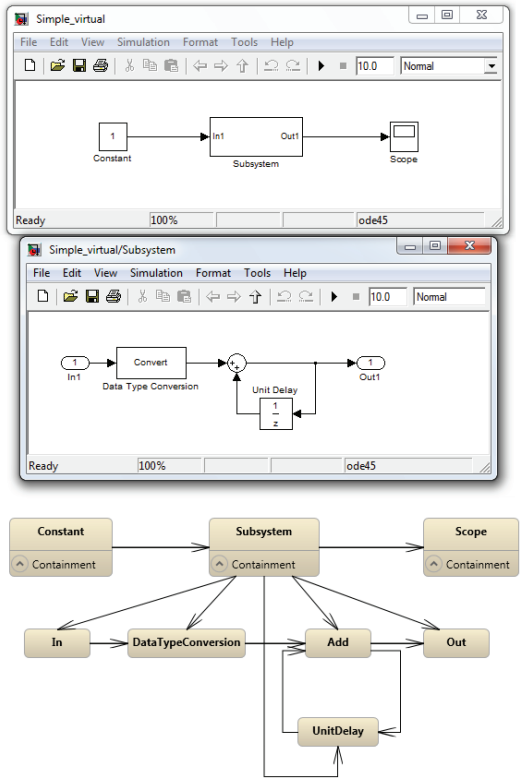

Fig. 3: The host model in Simulink and in VMTS.

Fig. 4: The debugging controls.

breakpoint is placed on the second rule container (indicated by the red filled circle in the corner). This means that the transformation engine automatically stops running before the execution of this rule and the rule can be examined in detail.

C. Visualizing matches

During the stepwise execution, after a match has been successfully identified, the pairing between the elements of the LHS and the host graph are clearly indicated. A unique order index is assigned and shown in a small tag next to each element in the LHS. And the same indices are also used in the host graph to visualize which element in the host graph corresponds to which element in the LHS. This function is depicted in Fig. 5. E.g., the node *virtualSubsystem* in the rule is mapped to the node *Subsystem* in the host model (tag "3"). By hovering the mouse over an element in the LHS, the corresponding element in the host graph is automatically highlighted. These are the nodes included in the actual match.

D. Modifying matches

After a match has been found, but the rewriting operation has not been performed yet, it is possible to modify the match and completely override the initial pairings. This can be achieved by clicking with the mouse on an element in the LHS while holding the *Alt* button and then clicking on the new pair in the host model – the element in the LHS will then become the focus (indicated by a red outline). The candidates (based on the expected metatype) in the host model are highlighted via a green outline. The modification of the match is also possible prior to the matching phase: the matcher will then attempt to extend the initial pairings to a valid match. Using this feature, we can coordinate the transformation engine and select the appropriate match if there are multiple potential candidates.

An example is displayed in Fig. 5. The top part of the figure depicts the state when the block to change in the LHS model has been selected. The candidate blocks of the host model are highlighted with a green outline. The bottom part of the figure shows the models after the match has been changed. Here, we have changed the corresponding host model element of the rule node *simpleBlock* from *DataTypeConversion* to *Add* (tag "0").

E. Modification of the host model

It is common that the underlying host model has invalid settings, while the transformation definition does not contain errors. Thus, we assist in editing the host model – both structurally, by creating and deleting elements, and also changing the attributes of existing elements – on the fly, while a transformation is being executed (and paused). This feature allows for the editing of the host model as it is normally done.

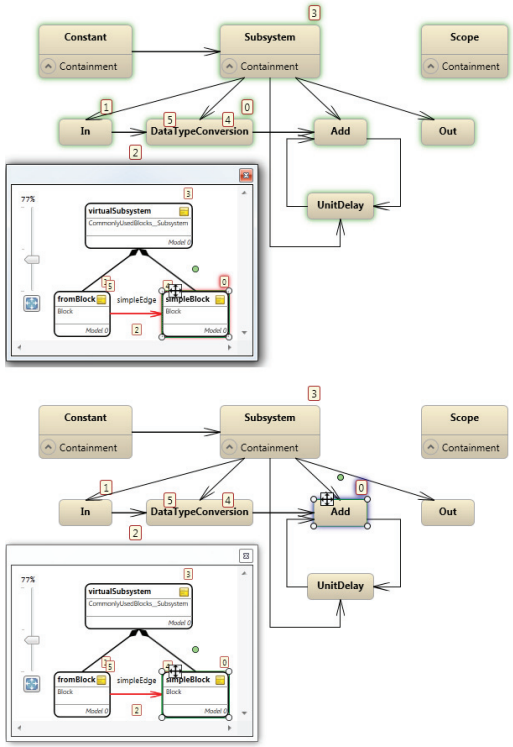

Fig. 5: Changing a match already found

978-1-4673-2232-4/13/$31.00 ©2013 IEEE

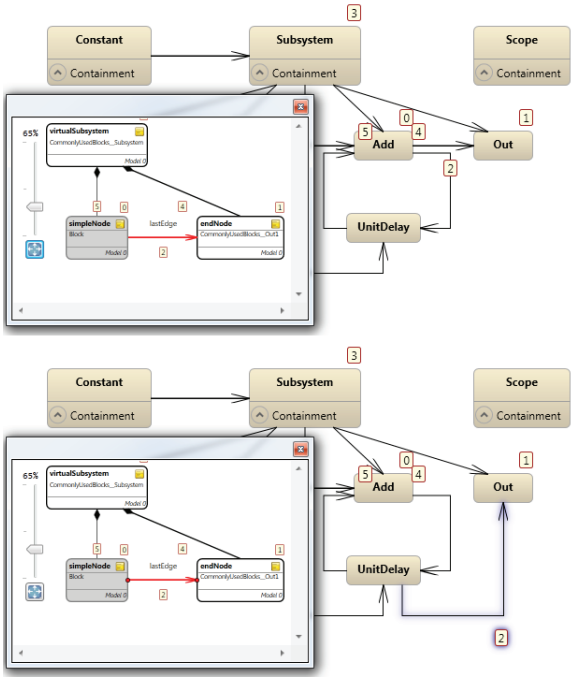

Fig. 6: The process of changing the host model during the processing

After modifying the model, we can resume the transformation. The same feature is also useful if the host model was valid, but a rule creates some unexpected modification. After fixing this modification manually, we can continue the transformation to test the remaining rules without restarting the whole process.

Fig. 6 shows an example for modifying the host model during the processing. In this case, the source of the incoming edge to the *Out* block is changed from the *Add* block to the *UnitDelay* block. After the modification, the match can and also should be changed.

F. Modifying the transformation

Similarly to editing the host model, we can also modify the transformation specification (both the control flow and the rules) at runtime. Of course it is advised to pause the transformation before doing so, but it is not necessary. We may freely add new elements or edit existing ones in the transformation definition, and continue with processing. However, the current implementation does not allow changing the textual constraints or the imperative code attached to the rewriting rules.

An example for modifying the transformation rule is depicted in Fig. 7. In this case, the rule is modified to additionally delete the *Out* block and the containment edge related to it.

VI. Conclusion

In this paper, we have introduced the visual debugging solution of the Visual Modeling and Transformation System. The main features of the visual debugging solution are as follows:

- It allows the manually controlled, step-by-step execution of graph rewriting-based model transformations.
- The transformation may be executed in continuous mode and can also be paused by placing breakpoints into the transformation definition.
- It is made possible to visually follow the host model, the control flow graph and the actual, executed transformation rules during the transformation process.
- The approach allows the modification of identified matches and the attribute values during the execution of the model transformation.
- The approach also facilitates the runtime modification of the transformation definition.

We believe that the debugging feature of the VMTS is unique in the field of these and similar types of frameworks. Furthermore, the solution provides the feeling that the environment is a user-friendly development tool.

Future work contains the extension of breakpoints to support conditional breaks. This means we are able to pause the transformation only if a predefined condition holds (e.g., an attribute has a specific value). Also, the main advantage of the solution is also a drawback: the continuous visualization of the transformation process slows down the entire process. It would be beneficial to be able to run the transformation in the background until a breakpoint is activated, and visually debug it only afterward. We also aim to extend the capability to modify the transformations at runtime. It is not currently possible to edit the textual constraints or imperative attribute setting codes while the transformation is running since they are precompiled before beginning the transformation. An

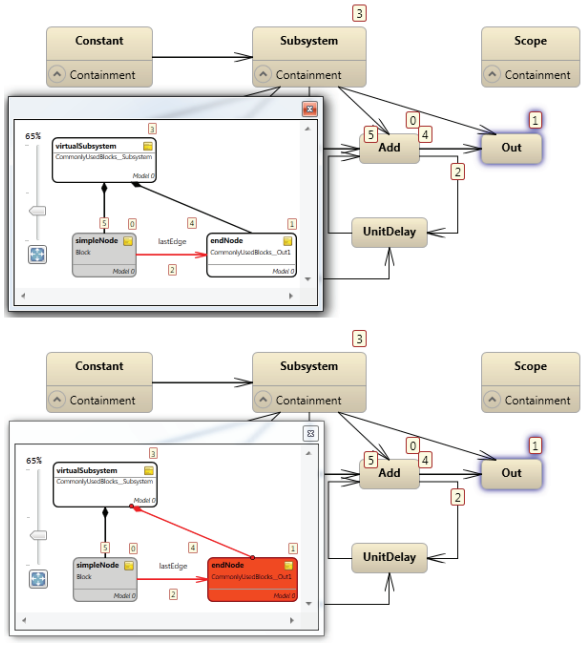

Fig. 7: The process of changing the applied transformation rule.

978-1-4673-2232-4/13/$31.00 ©2013 IEEE

interesting feature would be to recompile and reload these libraries at runtime, if anything changes are realized during the transformation. Another useful feature would be facilitating direct jumps in the transformation control flow to skip or repeat some parts of the transformation which are in demand.

ACKNOWLEDGMENT

This work was partially supported by the European Union and the European Social Fund through project FuturICT.hu (grant no.: TAMOP-4.2.2.C-11/1/KONV-2012-0013) organized by VIKING Zrt. Balatonfüred.

REFERENCES

[1] Kelly, S., Tolvanen, J.P. 2008. Domain Specific Modeling, Wiley.Fowler, M. 2010. Domain-Specific Languages, Addison-Wesley Professional.
[2] Angyal, L., et al., 2009., Towards a fast, efficient and customizable domain-specific modeling framework, Proc. of the IASTED International Conference on Software Engineering, Innsbruck
[3] Mészáros, T. et al., 2008., A Flexible, Declarative Presentation Framework For Domain-Specific Modeling, Proc. of the 9th International Working Conference on Advanced Visual Interfaces, Naples
[4] Ehrig, H. et al., *Fundamentals of Algebraic Graph Transformation*, Springer, 2006
[5] de Lara, J., Vangheluwe, H., 2002., AToM^3 A Tool for Multi-Formalism Modeling and Meta-modeling. In ETAPS/FASE'02, LNCS 2306, pp. 174-188. Springer
[6] Mannadiar, R., 2012., AToMPM User's Manual, *http://msdl.cs.mcgill.ca/people/raphael/files/usersmanual.pdf*
[7] Balasubramanian, D. et al., 2006, The Graph Rewriting And Transformation Language: GReAT, *Proc. of the Third International Workshop on Graph Based Tools*, Natal
[8] Taentzer G., 2000., AGG: A Tool Environment for Algebraic Graph Transformation, LNCS 1779, pp. 333-341, Springer
[9] Jouault, F. at al., 2008., ATL: A model transformation tool, Science of computer programming vol. 72, pp 31-39, Elsevier
[10] Eclipse Website, http://www.eclipse.org
[11] Rozenberg, G., *Handbook of Graph Grammars and Computing by Graph Transformations*, World Scientific, 1997
[12] VIATRA2 homepage: http://www.eclipse.org/viatra2/
[13] Geiß, R. at al., A fast SPO-based graph rewriting tool, in *ICGT*, pp. 383-397, 2006.
[14] Asztalos, M., Madari, I., 2009., An Improved Model Transformation Language. Proc. of Automation and Applied Computer Science Workshop, Budapest
[15] Simulink, 2012, http://www.mathworks.com/simulink/

ICT-14: Wireless Technologies 1

978-1-4673-2232-4/13/$31.00 ©2013 IEEE