Towards a Unifying Model Transformation Bus

Māris Jukšs[1], Bruno Barroca[1], Clark Verbrugge[1], Hans Vangheluwe[2,1]

[1] {mjukss,bbarroca,clump,hv}@cs.mcgill.ca
School of Computer Science, McGill University
Montréal, Québec, Canada
[2] hans.vangheluwe@uantwerp.be
Department of Mathematics and Computer Science
University of Antwerp, Belgium

Abstract. Even after the advance of model driven engineering, reusable tool integration remains one of the greatest software engineering challenges. As we venture toward generic heterogeneous modeling tool interoperability, we focus on the most commonly used application programming interface (API)-level tool integration. In this paper, we propose a unifying model transformation bus. It is a model-driven framework utilizing multi-paradigm modeling (MPM) techniques which aims towards automated interoperability realized from a specification. We demonstrate an MPM specification for integrating model transformations engines using their APIs while orchestrating method calls and data conversions. Finally, we discuss the implications of such system, its benefits, limitations and future use.

Keywords: multi-paradigm modeling, API-level interoperability, data conversion, orchestration, integration

1 Introduction

The required effort to perform complex application programming interface (API)-level integration between an arbitrary pair of programming languages often requires the knowledge of both programming native interfaces. The problem of tool integration is specially magnified when we try to apply Multi-Paradigm Modeling (MPM) solutions by reusing existing modeling tools—e.g., integrating different textual modeling languages with a graphical one[1].

A typical tool integration involves dealing with the marshaling of the data structures between several APIs including the interaction and communication issues between several tools. Adding to that, most of the reused tools are themselves made from reusing other tools, and therefore prone to evolution from operating system updates, versioning, which often raise painful incompatibilities. At the end of the day, we may face a low degree of reuse among several integration projects.

From the integration engineer point of view, one has to consider the type of interfaces that a particular set of tools provide, ranging from graphical-user interfaces, to textual command-shell interfaces, and APIs. Although APIs are the most commonly established way of performing tool integration, they do not offer service orientation as an interaction paradigm. APIs are still being defined as being merely a set of methods/functions organizing reusable algorithms over a set of predefined data structures that corresponds to the resources being manipulated by those algorithms.

In addition, while performing API-level integration, we have to face a bigger problem: in general, software tools are written in different programming languages. This is mainly due to a set of pragmatic reasons, ranging from different expressiveness and quality guarantees (e.g., type soundness), to different tool support (e.g., graphical editors, and advanced textual editors, debuggers, simulators, optimizers, and model-checkers). Therefore, typical API-level tool integration is currently performed resorting to the available programming language native interfaces (e.g., JNI, etc.) and middleware (e.g., CORBA, RMI). Having to learn and use a set of (possibly new) arbitrary native language interfaces (-NLI) in order to perform such integration is a laborious, tedious and error-prone task.

In this paper, we explore a minimal general approach to specify tool integration in a common reusable backbone. We explore how a unifying interface with advanced linguistic facilities—the unifying model transformation bus (UMTB)—can be used to decouple the recurrent burden of data conversion out from the tool's code, and push it into a reusable (model-driven-)middleware that can be systematically reused to handle data conversions among several heterogeneous tool integrations. Contributions of our work include: a solution for an MPM problem by performing data conversions in a platform independent way (i.e., using a State Chart formalism) that can be reused between APIs written in different programming languages; generation of the wrappers around APIs from an integration-specification; and using a platform independent representation (PIR) for common data representation, which is essentially typed attributed graph.

2 Related work

Concepts related to our work are model transformation (MT) chaining, tool integration and orchestration. Related work mentioned below typically addresses the integration through the use of some sort of an adapter (i.e., the wrapper) around the tool's API. However, we did not see any related work touch the issue when the tools being integrated are developed using different programming languages, and specially different from the ones used in the integration platform. When the API implementation technology does not support the integration, orchestration and communication frameworks (standards such as OSLC[3]) used. When the tool wrappers are described it is unclear how the data in the technical space of the API tool is actually accessed using the adapters as in [3, 5]. One of the main contributions of our work is the integration of APIs regardless of their implementation language with the only requirement being the common support

[3] http://open-services.net

of the network sockets on each of the languages to enable communication. Most importantly, we address how to actually access the API data and convert it into our chosen common data representation.

In [2] automated MT chain is generated from MT repository, input model MM incompatibility is also automatically addressed. A model transformation environment [6] addresses transformation composition for Eclipse plugins. Model bus [4] is the work closest to ours as it takes a "bus" approach connecting several components. Our approach targets API level tools and is not limited to three entry points (Java Metadata Interface, CORBA, Web Service) of the model bus. *ModelBus*[4] addresses integration through common repository, it does not target heterogeneous API tool integration. *UniTI* [9] focuses on the transformation reuse and composition through unified transformation representation. Transformation composition modeling framework [7] addresses UML centric transformation composition.

3 Running example

We will now discuss an orchestration of ATL[5] and GrGen[6] transformations at their API-level and identify the challenges that UMTB shall address. Consider orchestration of two model transformations that operate on a tree, where the result of the first transformation is the input of the second. In the first, the tree initialization is performed in ATL, this constitutes a case when a transformation exists created in a popular tool from the EMF universe. In the second part, the tree manipulation transformation is implemented in GrGen. The ability of GrGen to perform fast model transformations easily motivates the integration of such tool. The actual transformations on this paper were omitted for brevity as they are considered black boxes.

Both of the tools support XMI Ecore model import. However, the interoperability of tools through a common exchange format may not always be possible. One of the ways of approaching this is to orchestrate both transformations from a single separate program. Take a programming language Python which, for the sake of an example, the integration engineer is most familiar with. The language appears to have a support for executing both Java and C-sharp binaries through third party libraries. A project *pyjnius* [7] allows for Java code execution within Python and *Python for .NET* [8] integrates C-sharp. We executed a compiled ATL transformation from within Python. In our orchestration prototype, the result of the transformation is returned and explored (i.e., using a visitor pattern) using the native method calls specific to the returned object (through pyjnius). The result of the so called parsing is stored into the Python representation of the tree. The motivation for using common representation (in this case a Python

[4] http://www.modelbus.org

[5] http://eclipse.org/atl/

[6] http://www.info.uni-karlsruhe.de/software/grgen/

[7] https://github.com/kivy/pyjnius

[8] http://pythonnet.sourceforge.net/

tree structure) is to eliminate the excessive number of conversions between the tool formats if we wish to integrate more heterogeneous tools into the solution. Next, the returned Python object is converted into the GrGen object which is then used to execute the second transformation. The result is again converted back into Python representation.

The integration of two API tools in this manner exposed a knowledge gap of integrating Java and C-sharp into Python. In addition, the integration effort required the knowledge of the native code constructs in order to realize conversion between objects (Java and C-sharp). The resulting solution, though sound, is not well suited for evolution of its respective parts. Changes in data, 3rd party APIs, and programming languages, will most certainly impose a dramatic impact on this prototype. We need a reusable integration solution that allows us to abstract from these changes. UMTB is aiming for a semi-automated integration generated from a specification.

4 Solution prototype

We now detail our vision of the prototype specification and resulting component interaction in order to solve the integration problem presented in our running example. In Figure 1 on the left, a view of a typical interaction between the UMTB components is shown using a syntax of UML Sequence Diagrams. On the right is shown the proposed UMTB architecture. The interaction results from an

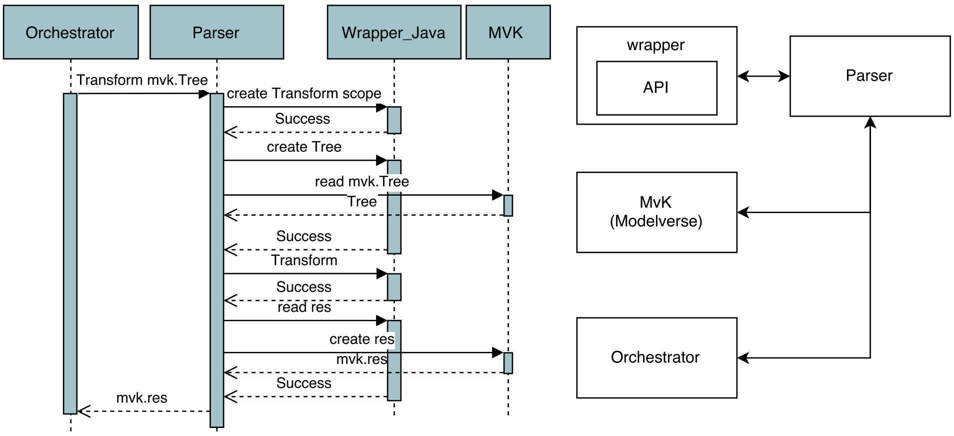


Figure 1. A view of the component interaction during an API call to the Wrapper (left) and UMTB architecture (right).

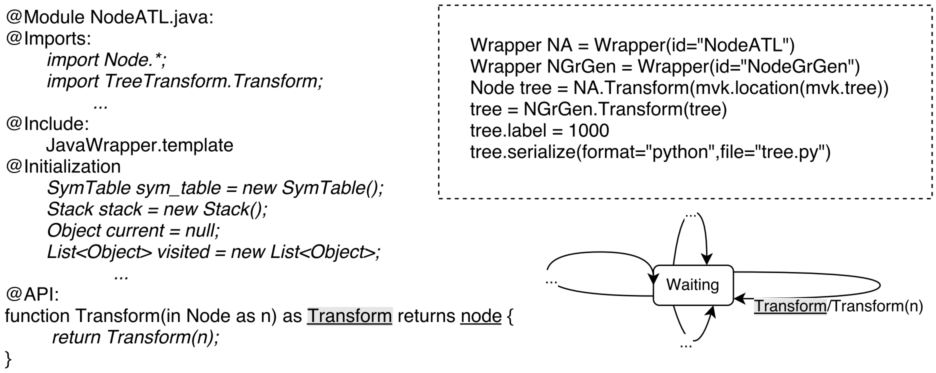
execution of a human understandable textual notation (HUTN) [8] code inside the Orchestrator upon a tree model instance located on the Modelverse Kernel component (MvK) [8]. MvK consists of a model management engine and model repository that is shared and visible among all of the components (except for the Tool's API and its Wrapper) within the UMTB. MvK internally manipulates models persisted in a PIR which is basically a typed-directed-attributed graph that is used to describe values, types and metatypes (types of types).

We envision the following process of an API method execution on the Wrapper. Every API method call initiated in Orchestrator is directed to the Parser component. The Parser component behaves as a controller that interacts with the Wrapper and MvK to create copies of data to and from these two components. The Parser, according to its configuration initiates a sequence of messages to the Wrapper. These messages are intended to: create a call-scope symbol for each API call; instantiate the parameter values in the Wrapper memory space; call the method; and return the value.

The call-scope symbol created on the Wrapper is used to store not only the values of the evaluated parameters, but also the method's return value. In the UML Sequence Diagram, the call-scope symbol is followed by the creation of the required parameter values. In this particular case, the input model resides on the MvK as do all the variables created and modified in the Orchestrator. The Parser component initiates the parameter variable creation on the Wrapper. The variable is read from the MvK, and recreated in the Wrapper during an interaction between Parser, Wrapper and MvK. The process of variable creation as well as reading the variables is guided by a multi-formalism statechart specification residing in the Parser and is described in greater detail in Section 4.2. Once all the parameter variables are created in the Wrapper memory space, the API call is executed. The resulting value is transferred to the MvK through similar statechart guided interaction between the components.

4.1 UMTB specification

We present the prototype of an ATL tree transformation Wrapper specification on the left in Figure 2. We omit, for brevity, similar specification for the C-sharp



```
@Module NodeATL.java:
@Imports:
      import Node.*;
      import TreeTransform.Transform;
              ...
@Include:
      JavaWrapper.template
@Initialization
      SymTable sym_table = new SymTable();
      Stack stack = new Stack();
      Object current = null;
      List<Object> visited = new List<Object>;
               ...
@API:
function Transform(in Node as n) as Transform returns node {
       return Transform(n);
}
```

```
Wrapper NA = Wrapper(id="NodeATL")
Wrapper NGrGen = Wrapper(id="NodeGrGen")
Node tree = NA.Transform(mvk.location(mvk.tree))
tree = NGrGen.Transform(tree)
tree.label = 1000
tree.serialize(format="python",file="tree.py")
```

Figure 2. Header of the Wrapper for the ATL tree transformation on the left. On the right in the dashed rectangle is the example orchestration in HUTN of two Wrappers to chain two transformations to address our running example. Wrapper server flower with one petal corresponding to an API function in bottom right.

based GrGen transformation. At the top of the specification are all necessary native Java code imports required to execute the API methods in the Wrapper and run the Wrapper specific code (italics indicate the native code snippets injected into generated Wrapper programs). In this particular instance, an ATL transformation package is imported for the *Transform* method as well as the *Node* package of the EMF code implementing the *Node* tree model. The *Node* class specification can be seen within the dashed rectangle in Figure 3.

Next follows the include section of a Java Wrapper template, that is reusable across all Java APIs. The Wrapper template contains the minimal server implementation including the communication channel. Wrapper template is constructed using a template language (such as EGL from Eclipse, for example) to allow for native code snippet injection indicated in specification. The resulting Wrapper after generation is a program in the native code, that links (imports) the API and needs to be compiled.

The communication channel for the purpose of this paper is assumed to be socket based TCP/IP connection between bus components, ensuring the exchange of messages and the data encoded into strings. The choice of communication channel however is open for flexibility. The use of sockets was motivated by their typically universal interface across platforms and languages as well as the freedom from using third party communication libraries.

The server, or simply the server loop of the Wrapper, if expressed using a statechart, resembles a flower with transition loops initiating from and terminating at a "Waiting" state, see the bottom right of Figure 2. The pre-conditions of the transitions are taken from the Wrapper Getter, Setter, and API specifications highlighted and underlined in all listings. The actions of the transitions are triggering the actual native code snippets in the bodies of Getter, Setter, and API specifications.

The initialization section of the specification contains the native code executed at the instantiation of the Wrapper. The variables created in this section are global and will be used in the Getter, Setter and API specifications. Notably the symbol table *sym_table* is created with an intent to keep the variables instantiated in the Wrapper memory-space. The container *visited* is used to keep track of visited elements during navigation of the data structures. A *current* object is used as a pointer during visiting the data structure in the Wrapper. This section in the specification also includes other objects the integration engineer may need, a stack for example. There is also an initialization section in the Parser not shown here. It is intended to create variables used in the MvK specific HUTN code, as shown in Section 4.2, and it also contains the exchangeable PIR data metamodels in this case the metamodel of the tree structure we are transforming. This ensures that the data coming to and from PIR is properly modeled.

The API section of the specification contains the API methods we want to expose. In this example, the result of Transform API function call is returned from a function. The body of this function (as do the bodies of Getters and Setters) is injected into Wrapper code during generation. When the Wrapper receives the message *Transform* the body of the function will be executed provided that the parameters are instantiated within the call-scope of the Wrapper memory space, and the return of the function call is then stored in the same call-scope.

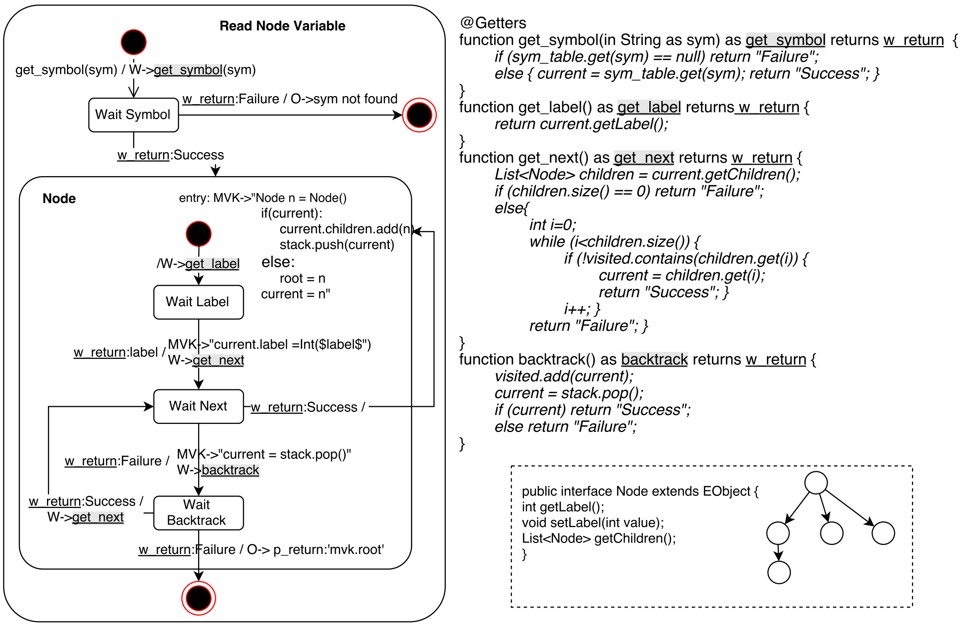
The notable syntax elements of the API specification, as well as Getter and Setter function specifications, are the following. The pre-condition or the triggers of the Wrapper server statechart (the message identifiers of the petal transitions) are specified after the keyword *as*. The return identifier is specified after keyword *returns*. The return identifier is used to tag the messages containing primitive return values (strings). Note that communication channel is only allowed to sends primitive strings. The variable values, the primitive ones such as integers or floats in this prototype solution need to be converted to and from strings at the level of communication channel. It is possible however to specify the API, Getters and Setter functions to only return strings thus taking care of primitive type conversions explicitly. The name and the type of the function parameter(s) is indicated inside the brackets. The use of Getters and Setters is described in the following section.

4.2 Reading and creating Wrapper variables

The data conversion between bus components is central in our approach. We propose that every data structure (model) participating in the integration is converted to and from the PIR according to the specification. We identify two cases of data exchange, first is reading the variable from the Wrapper using Getters. This is necessary to get the return value of an API call (reading can also be used on demand). The second case is of creating the variables in the Wrapper using Setters to execute methods requiring instantiated objects.

To get the result of an ATL transformation into PIR we need to make a copy of the tree object in the MvK/Modelverse by exploring the Java tree object. We start with the Wrapper Getters on the right in Figure 3. They are a part of the Wrapper specification from Figure 2 and are used to read the data structures in the Wrapper. Note that the native code in this paper does not handle exception scenarios, for brevity. On the left in the Figure is the statechart describing the interaction between the Parser component, the Orchestrator, the Wrapper and MvK. The Getter (and Setter) definition shares the API definition syntax. In addition, we highlight and underline the message identifier in both Getter specification and the statechart to aid the comprehension. The return identifiers are underlined.

The statechart is designed by integration engineer with an intention to visit in depth first manner the tree object in Java Wrapper space and copy it to MvK tree model. The visitor paradigm is achieved by advancing the *current* pointer over the data structure by calling the Getters' code in the Wrapper. The copying is ensured by executing MvK HUTN code specified in the statechart. The *current* pointer is also maintained on the MvK side by the Parser. Essentially, in this example, both pointers are associated with the same data structure element during the copying process.



```
@Getters
function get_symbol(in String as sym) as get_symbol returns w_return {
        if (sym_table.get(sym) == null) return "Failure";
        else { current = sym_table.get(sym); return "Success"; }
}
function get_label() as get_label returns w_return {
        return current.getLabel();
}
function get_next() as get_next returns w_return {
        List<Node> children = current.getChildren();
        if (children.size() == 0) return "Failure";
        else{
                int i=0;
                while (i<children.size()) {
                        if (!visited.contains(children.get(i)) {
                                current = children.get(i);
                                return "Success"; }
                        i++; }
                return "Failure"; }
}
function backtrack() as backtrack returns w_return {
        visited.add(current);
        current = stack.pop();
        if (current) return "Success";
        else return "Failure";
}
```

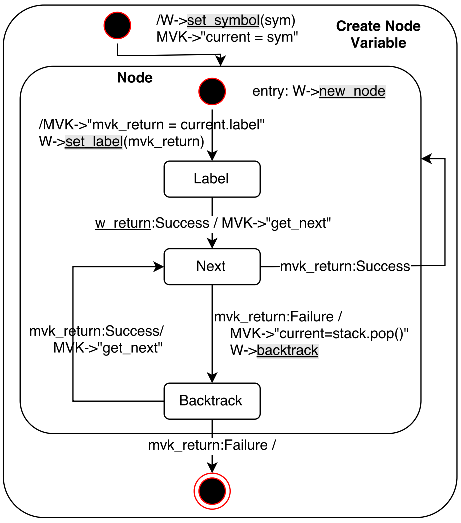
```
public interface Node extends EObject {
int getLabel();
void setLabel(int value);
List<Node> getChildren();
}
```

Figure 3. On the left a statechart implementing the copying of the tree from ATL Java Wrapper into PIR in MVK by visiting the tree in depth first manner. Java specific Getters on the right. EMF tree interface in dashed rectangle.

Inside the dashed rectangle in Figure 3, we show the possible tree structure for visualization, and its definition from an EMF project. Also note that the integration engineer could have designed the statechart differently, such as visiting the tree in breadth-first manner. The statechart also relies on Getter functions, such as *get_label* in order to receive the data associated with the structure (such as the node label, in our example).

The statechart post-condition issues narrow-cast messages with one of W, O or MVK identifiers followed by a $\rightarrow$. These messages are sent to Wrapper, Orchestrator and MvK respectively. At the entry point into the statechart in Figure 3, the symbol table is queried for an existence of a variable. In case of success, the statechart enters the $Node$ state where the entry action creates a node in the MvK. Subsequent entries into the $Node$ state will create the tree in MvK, and advance the *current* pointer in MvK. The *current* pointer within the Wrapper is also advanced using the *get_next* Getter. The statechart is also performing backtracking when the tree leafs are reached by using the stack in both Wrapper and MvK spaces. Statechart terminates when the Wrapper tree visiting is over. The location of copied MvK object can then be returned to the Orchestrator for further processing.

In Figure 4, we demonstrate the creation of the variable inside the Wrapper space. Description of Getters applies here. For brevity, certain MvK code snippets in the statechart were encapsulated into functions such as *get_next*. The



```
@Setters:
function set_symbol(in string as sym) as set_symbol returns  w_return {
        sym_table.Add(sym,null);
        current_symbol = sym;
        return "Success";
}
function new_node() as new_node returns w_return {
        INode n = graph.AddNode(nodeType);
        if (current) {
            graph.AddEdge(nextType, current, n);
            stack.push(current);
        } else {
            sym_table.Add(current_symbol,n);
        }
        current = n
        return "Success";
}
function set_label(in string as label) as set_label returns w_return {
        current.setLabel(Int32.Parse(label));
        return "Success";
}
function backtrack() as backtrack returns w_return {
        current = stack.pop();
        return "Success";
}
```

Figure 4. On the left a statechart implementing the copying of the tree from PIR in MVK to GrGen C-sharp Wrapper by visiting the tree in depth first manner. C-sharp specific Setters on the right (error handling omitted).

behavior of this function is to advance the *current* pointer similar to what the Java *get_next* Getter is doing just using the HUTN code. In this example, we show the Setters from GrGen C-sharp Wrapper. Notice that both statecharts for reading and creating Wrapper variables do not contain the native code and only platform independent HUTN code. This allows for a good deal of reusability of these statecharts. Statechart for creating tree structure in ATL Java Wrapper can now be repurposed for C-Sharp by reimplementing the Setters. Also note that both statecharts in this section are not exactly the same. Since the intention of both is to navigate tree structure in depth first manner, they appear similar in structure, and actually could be made almost exactly the same in structure (i.e., completely reused) except for their transition specification and entry actions. Difference in statecharts was intentionally made to demonstrate several engineers working on an integration. This statechart visits MvK tree and makes a copy in the Wrapper.

5 Conclusion and Future Work

In this paper we explore the UMTB, a model-driven alternative to the use of the Native Language Interfaces on performing API-level tool integration. We show how statechart models located in the Parser can be reused for data conversion between API tools in different programming languages. We aim towards automated generation of Wrapper component from a specification using native code injection specific to the integrated API. The common data representation in MvK component was used as a middle-ground for data conversion. In our approach, we do not impose a particular communication channel to be used to exchange messages between UMTB components. Possible communication channels could include process pipes for example.

In future work, we need to investigate the construction effort and the reusability gains of UMTB, especially for data conversion. We will also explore other formalisms for specifying data converted on the bus, such as for instance class diagrams. Ideally, these formalisms should be analyzable so that we can reason about these data conversion specifications, and prove properties such as their correctness: such as for instance termination. The process of specifying UMTB can be augmented to aid the engineer with code completion when using native code of choice, syntax directed editing for the diagram construction, and wizards. Also, we could provide design-time navigation through the data structures in an API of choice, in order to assist the UMTB specification. Finally, automated deployment decisions of the UMTB is another topic that needs a further development.

References

1. Barroca, B., Mustafiz, S., Mierlo, S.V., Vangheluwe, H.: Integrating a neutral action language in a devs modelling environment. SIMUTOOLS '15: Proceedings of the 8th EAI International Conference on Simulation Tools and Techniques (2015)
2. Basciani, F., Di Ruscio, D., Iovino, L., Pierantonio, A.: Automated chaining of model transformations with incompatible metamodels. In: Model-Driven Engineering Languages and Systems, Lecture Notes in Computer Science, vol. 8767, pp. 602–618. Springer International Publishing (2014)
3. Biehl, M., El-Khoury, J., Loiret, F., Törngren, M.: On the modeling and generation of service-oriented tool chains. Softw. Syst. Model. 13(2), 461–480 (May 2014)
4. Blanc, X., Gervais, M.P., Sriplakich, P.: Model bus: Towards the interoperability of modelling tools. In: Aßmann, U., Aksit, M., Rensink, A. (eds.) Model Driven Architecture, Lecture Notes in Computer Science, vol. 3599, pp. 17–32. Springer Berlin Heidelberg (2005)
5. Clavreul, M., Barais, O., Jézéquel, J.M.: Integrating legacy systems with MDE. In: Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 2. pp. 69–78. ICSE '10, ACM, New York, NY, USA (2010)
6. Kleppe, A.: MCC: A model transformation environment. In: Rensink, A., Warmer, J. (eds.) Model Driven Architecture Foundations and Applications, Lecture Notes in Computer Science, vol. 4066, pp. 173–187. Springer Berlin Heidelberg (2006)
7. Oldevik, J.: Transformation composition modelling framework. In: Distributed Applications and Interoperable Systems, 5th IFIP WG 6.1 International Conference, DAIS 2005, Athens, Greece, June 15-17, 2005, Proceedings. pp. 108–114 (2005)
8. Van Mierlo, S., Barroca, B., Vangheluwe, H., Syriani, E., Kühne, T.: Multi-level modelling in the modelverse. MULTI 2014 – Multi-Level Modelling Workshop Proceedings 1286, 83–92 (2014)
9. Vanhooff, B., Ayed, D., Van Baelen, S., Joosen, W., Berbers, Y.: UniTI: A unified transformation infrastructure. In: Model Driven Engineering Languages and Systems, Lecture Notes in Computer Science, vol. 4735, pp. 31–45. Springer Berlin Heidelberg (2007)