Extensible Global Model Management with Meta-model Subsets and Model Synchronization

Dominique Blouin, Yvan Eustache and Jean-Philippe Diguet

Lab-STICC, Université de Bretagne-Sud, Centre de recherche, BP 92116
56321 Lorient CEDEX, France
{dominique.blouin, yvan.eustache, jean-philippe.diguet}@univ-ubs.fr

Abstract. We present an infrastructure for the management of models of heterogeneous meta-models in model-based development environments. The infrastructure consists of a Global Model Management (GMM) modeling language, which allows the capture of the meta-models used in a modeling environment. Relations between meta-models and subsets of these meta-models can be declared and interpreted during model evolution for automated global model management. The infrastructure is implemented in an Eclipse EMF based EDA (Electronic Design Automation) tool. Its use is demonstrated by the generation and synchronization of AADL and VHDL code targetting an FPGA to control a self-balancing toy car.

Keywords: Global Model Management, Model Coordination, Model Transformations / Synchronization, Triple Graph Grammars, EDA Tools

1 Introduction

Model-based engineering makes use of many models of different kinds to capture all aspects of a system. As presented in [1], a significant proportion of design errors are due to inconsistencies between the heterogeneous models used to develop the evolving system. Mechanisms are required to ensure consistency of models is automatically maintained, and that proper traceability links can be established between models and maintained during model evolution. Such mechanisms should be at the heart of every model-based development tool, and it should be extensible so that modeling tools can be easily configured to target new domains making use of other modeling languages and types of relations between models.

In this paper, we present a Global Model Management (GMM) infrastructure to solve these problems. It is inspired from state of the art research and from experience gained in developing the Kaolin EDA (Electronic Design Automation) tool [2], which makes use of several rich modeling languages such as AADL (Architecture Design and Analysis Language, [3]) and VHDL (VHSIC (Very High Speed Integrated Circuits) Hardware Description Language, [4]). The tool aims at simplifying the development of electronic systems implemented on FPGAs (Field Programmable Gate

Arrays) by generating automatically the platform-specific models and VHDL implementation code from abstract functional AADL models. Our GMM language includes the concept of meta-model subset as introduced in [5], to declare sets of constraints restricting the use of complex and rich languages such as AADL and VHDL. Validation of subset constraints ensure given activities can be performed on models. In addition, automated model synchronization is provided through an extension of the core GMM language making use of an enhanced version of MoTE [6], which is based on Triple Graph Grammars (TGG) [7].

The rest of this paper is divided as follows: the next section introduces the GMM language and its interpretation semantics. Then, section 3 demonstrates the use of the GMM infrastructure through an example consisting of a self-balancing radio controlled car whose implementation code is automatically generated using interpreted GMM relations. Related work is then presented in section 4, followed by the conclusion and perspectives in section 5.

2 The GMM Infrastructure

2.1 The GMM Modeling Language

Many approaches to GMM include the concept of mega-model, for which several definitions can be found. A unified definition is provided in [8], which has the advantage of being free of implementation details, and can be extended to make use of specific transformation tools and other artifacts. Our core GMM language (Fig. 1) is inspired from this work. It includes the central concepts of *model* and *relation*. A model is defined as an element that contains models and relations between models. It is hierarchical, meaning that it can contain other models as children. This allows for better structuring of models. For instance, and as explained in [8], large languages such as UML would benefit from such structuring by having some of their model elements declared as models (e.g. class diagrams, sequence diagrams, etc.).

Models need to be related to each other, so a mega-model must be able to contain *relations* between its contained models. A relation also owns an *intention*, which describes the intended use of the relation. Inspired from [9], we further distinguish between *factual* and *obligation* relations. A factual relation must be deleted as soon as its intention is not satisfied anymore. For example, if the intent of the relation is to provide traceability between two models, then if one model is deleted, the relation must be deleted because its intention is not satisfied anymore. On the opposite, an obligation relation defines that something should always hold but does not necessarily do so between the related models. Therefore, obligation relations should not be deleted when their intention does not hold, but provide means for (re-)establishing the validity of the relation. This is represented by the *establish validity* operation, which takes as input a *modeling environment* and an *execution context*. The modeling environment provides all models *concerned* by the relation, so that they can be processed to (re-)establish validity. The *execution context* describes the context in which the validity of the relation should be established. It captures the *type* of *operation* that was performed on a *source model* of the modeling environment that was changed thus requiring validity to be re-established. Predefined types of operation follow the basic CRUD (Create, Read, Update and Delete) types used for database persistence. The establish validity operation returns a collection of models of the environment that have been updated as a consequence of (re-)establishing the validity of the relation.

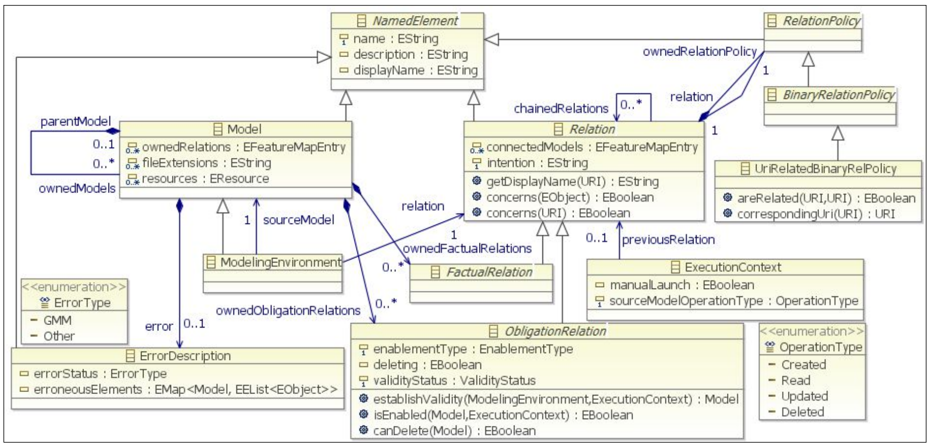

Fig. 1. The core GMM language

A model may be in an *error* state. For example, a model concerned by a relation may not be valid, and this information stored in the model can be processed by an obligation relation when re-establishing validity. In addition, the relation can itself update the error state on the models to indicate for example what prevents the validity of the relation to be established. This error information can then be displayed to users of the modeling tool, for instance through markers of the Eclipse environment.

The models that are concerned by a GMM relation must be determined in some way. For this purpose, the *relation policy* class is introduced. Subclasses can provide specific ways of relating models, for example, by determining that two models of different meta-models are related only when the file names of their resources have the same base file name, with meta-models being identified by file extensions.

A relation between models does not necessarily exist in isolation. Some relations may require other relations to hold. For example, a transformation chain can be seen as a set of chained obligation relations that must be executed one after the other to establish the validity of the chain of concerned models. For this reason, a GMM relation can declare *chained relations*.

A *meta-model* (Fig. 2) represents a specific type of model to which a model can be related through a *conformance* relation. It is also a place where useful information can be stored to be used by tools processing models of the meta-model. For instance, model comparison, which often needs to be customized per meta-model, can be specified by attaching model *comparison settings* to a meta-model.

Rich modeling languages such as UML or AADL often support a large number of *activities* (performance analysis, code generation, etc.). As pointed out in [5] for

AADL, guidance on how a language should be used for a given set of activities to be performed is essential to ensure tools interoperability. The authors of [5] proposed a DSML to capture *subsets* of modeling languages, and a revised version of this DSML is integrated in our GMM language, which introduces the concept of *meta-model subset* (Fig. 2). It consists of a set of *constraints* expressed in terms of the *cardinality* of a set of *model elements* of a given model. Various ways can be provided to construct sets of model elements, but this is still an ongoing work. The objective is to express subsets in a way that their constraints can be interpreted by tools without evaluation on a specific model, for customization according to a given subset. For example, the AADL graphical editor used in Kaolin can interpret a subset to automatically hide any element of the palette whose classifier is forbidden by the subset.

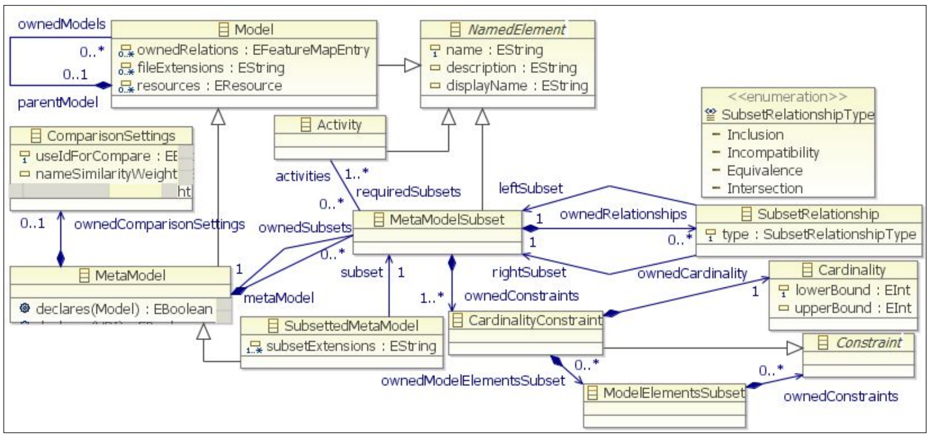

Fig. 2. The GMM concepts for meta-models and meta-model subsets

Following [5], meta-model subsets can be related to each other according to four types of relationships (*inclusion*, *incompatibility*, *equivalence* and *intersection*). A subset can be composed of other subsets through inclusion relations. It can also be associated with a given set of activities, thus allowing the analysis of tools interoperability according to their associated activities and subsets compatibility and equivalence.

A meta-model subset can be associated with a *subsetted meta-model* declared for a given meta-model. Models conformed to a subsetted meta-model are first validated against the meta-model, and then against the constraints provided by the associated meta-model subset.

2.2 Extension for Model Synchronization.

Model transformations are first class entities in model-based development. In GMM, they are represented as specific *relations* between models. Other types of operations between models could also be represented such as merge and refactoring, but it remains to be explored. Most model transformations are unidirectional and work in a batch mode, i.e. from a set of input models they can only *create* an output model instead of updating an existing model. This is not sufficient since once generated, a model may need to be modified as it provides a different view of the system that may need to be updated by users. Hence, modifications must be reflected back in the source model to maintain consistency. Often this must be performed without re-instantiating the source model, since it may contain information that is not represented in the target model. Incremental transformations, which update only parts of a model, are therefore required. This is called model synchronization.

Only a few model transformation tools can currently satisfy these requirements. Among these tools, MoTE [6] can transform models in either directions using batch or synchronization mode. In addition, an enhanced version of MoTE has recently been developed [10], providing several improvements required for synchronize models of rich languages such as AADL. Being fully EMF-based, MoTE can be easily integrated into our GMM infrastructure in the form of a language extension (Fig. 3).

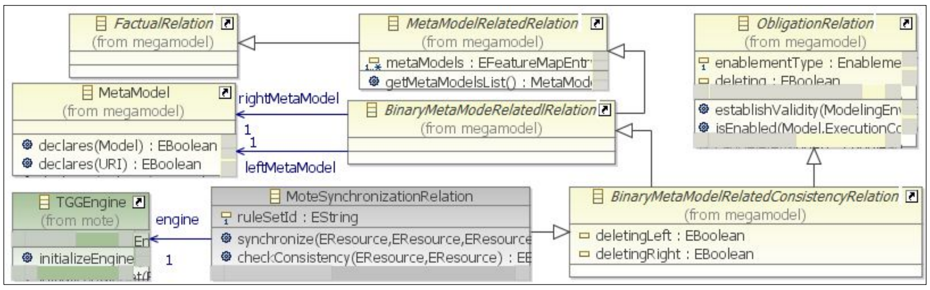
FactualRelation
(from megamodel)
MetaModelRelatedRelation
(from megamodel)
metaModels : EFeatureMapEntr
getMetaModelsList() : MetaMod
ObligationRelation
(from megamodel)
enablementType : Enableme
deleting : EBoolean
establishValidity(ModelingEn
isEnabled(Model,ExecutionCo
MetaModel
(from megamodel)
declares(Model) : EBoolean
declares(URI) : EBoolean
rightMetaModel
1
1
leftMetaModel
BinaryMetaModeRelatedlRelation
(from megamodel)
TGGEngine
(from mote)
initializeEngine
engine
1
MoteSynchronizationRelation
ruleSetId : EString
synchronize(EResource,EResource,EResource
checkConsistency(EResource,EResource) : EE
BinaryMetaModelRelatedConsistencyRelation
(from megamodel)
deletingLeft : EBoolean
deletingRight : EBoolean

Fig. 3. The GMM extension for model synchronization with MoTE

The extension consists of a *MoTE synchronization relation*, which factually relates two models through their meta-models (*binary meta-model related relation*). At the same time, it is also an *obligation relation establishing* that the two related models must be maintained *valid* by ensuring their consistency. This is achieved by a MoTE *TGG engine* used by the MoTE synchronization relation.

2.3 GMM Model Interpretation

Our GMM language and its interpreter are deployed in the Eclipse Integrated Development Environment (IDE) as depicted in Fig. 4. A mega-model declaring the meta-models, their subsets and their relations is stored in the workbench configuration directory. A GMM controller listens for model change or read events (e.g. editor opening) sent by the Eclipse platform. For a given model source of a received event, the controller instantiates a modeling environment containing the models in the workspace and an execution context whose operation type reflects the type of the event. It then calls the GMM engine that interprets all relations declared in the mega-model that concern the models of the modeling environment. Obligation relation is currently the only type of relations interpreted in our GMM language. The GMM engine calls the *establish validity* operation passing the created modeling environment and execution context objects. For a MoTE synchronization relation, the operation consists of calling the appropriate operation on the associated MoTE TGG engine according to the specified execution context and for each target model of the modeling environment. For an execution context with a read operation, this means that the model loaded in memory may be updated in a next operation. If the model corresponding to the source model does not exist, the relation calls the engine to perform a batch transformation to create the model. Otherwise, a check consistency operation on the TGG engine is performed. Both of these operations cause a TGG correspondence model to be created between the models and stored in the TGG engine's memory. Later on, when the model is updated, a resource change event is sent to the GMM controller, which is translated into an execution context of type *Update* sent to the GMM engine. The MoTE relation then calls the TGG engine to synchronize the target models of the modeling environment. When a model is deleted, the corresponding model may be deleted or not, as specified by the relation's deletion properties.

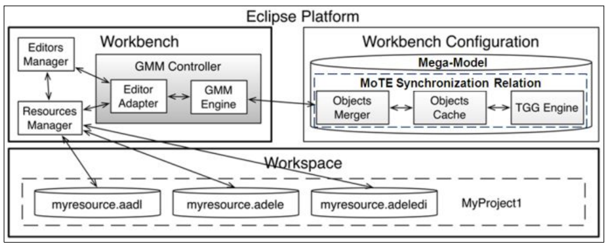

Fig. 4. The GMM infrastructure integrated in the Eclipse platform

The MoTE synchronization relation also takes care of creating appropriate errors carried by the models in case inconsistencies are detected during the creation of the TGG correspondence model. Model objects that are not mapped by the correspondence model are inspected according to a model elements coverage policy associated with the relation, which determines if unmapped elements should have been mapped or not. In the former case, this indicates that the models are inconsistent, and appropriate errors are set for the model elements. The MoTE relation will not process models until the inconsistencies are resolved through manual update of the models. The relation makes use of a cache of the model objects, which are linked by the correspondence models stored in the TGG engine memory. Changes made by any tool to a model are first merged into the cache, which preserves the object instances, thus ensuring the links of the correspondence model used to synchronize the models remain valid. The merge layer is implemented with EMF Compare [11], using comparison settings defined per meta-model declared in the mega-model.

3 Example

This section presents an example illustrating the use of the GMM infrastructure, where many details are omitted due to lack of space. It consists of an electronic system implemented on an FPGA to control a self-balancing toy car (hereafter RC Car) from a smart phone (lower left part of Fig. 5). AADL is used to specify PIM and PSM models for the system. From the AADL PSM, a VHDL model is generated, which can be taken as input by FPGA vendor tools for synthesizing the circuit in the FGPA.

AADL is a component-based language for modeling both the software and hardware parts of embedded systems. It supports the specification of systems as an assembly of software and hardware components divided into categories. Software categories are thread, thread group, data, process and subprogram. Hardware categories are processor, virtual processor, memory, device, bus and virtual bus. Hardware and software component classifiers can be declared in libraries or hierarchically organized in systems for reuse. AADL components interact through features (interaction points) and connections, which together model data or control flows between components.

The first step to design a system in Kaolin is to create an AADL functional model independent of any execution platform (diagram of Fig. 5). It is then transformed into an AADL PSM, which describes the FPGA chosen by the user and the synthesized functions taking into account execution platform-specific details. The GMM language is used to specify the AADL and VHDL meta-models, including three subsetted meta-models for the AADL PIM and PSM, and for a subset of VHDL that can be handled by FPGA synthesis tools (synthesizable VHDL).

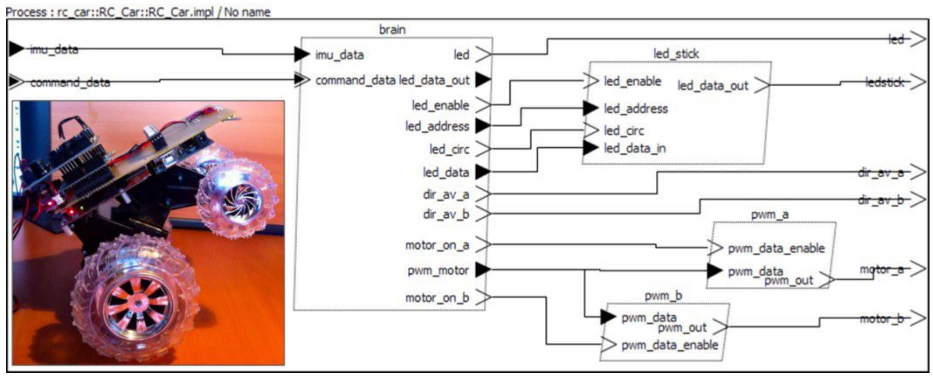

Fig. 5. The self balancing toy car and an AADL functional diagram for its control system

The objective of the AADL functional subset is to ensure AADL is used correctly for PIMs to be transformed into AADL PSMs, The functional subset includes a first subset that restricts the AADL language to its software part, which consists of forbidding the use of hardware constructs (processor, virtual processor, memory, bus, device and bus access). Additional constraints are then added to the functional subset to ensure only AADL *threads* and *data* subcomponents are used and contained in a single AADL *process* (Fig. 5).

From a functional AADL model, an AADL PSM is generated, conformed to an execution platform subsetted meta-model ensuring execution platform models are properly defined to be transformed into synthesizable VHDL code. Similar to the PIM subset, the PSM subset includes a subset restricting the constructs to hardware AADL elements. Then, another subset describing how FPGAs must be modeled with AADL is provided, following an AADL extension developed to model FPGAs [12]. It includes the AADL hardware subset. Finally, the last required subset is *Synthesizable VHDL*, which cannot be described here due to the lack of space. Other VHDL subsets ensuring simulatability, testability and reusability, and as implemented by tools such as the Leda RTL checker [13] could also be modeled with our GMM language.

Once the required subsetted meta-models have been created, relations between models conformed to these subsetted meta-models can be declared to transform / synchronize the models. These relations are implemented as MoTE synchronization relations, allowing to maintain the consistency of models as they are updated, but also to check their consistency. The most complex relation is the *functional to FPGA execution* platform relation, which generates from an AADL PIM (Fig. 5) an AADL PSM (Fig. 6) describing the content of the synthesizable component of the specific FPGA platform selected by the user.

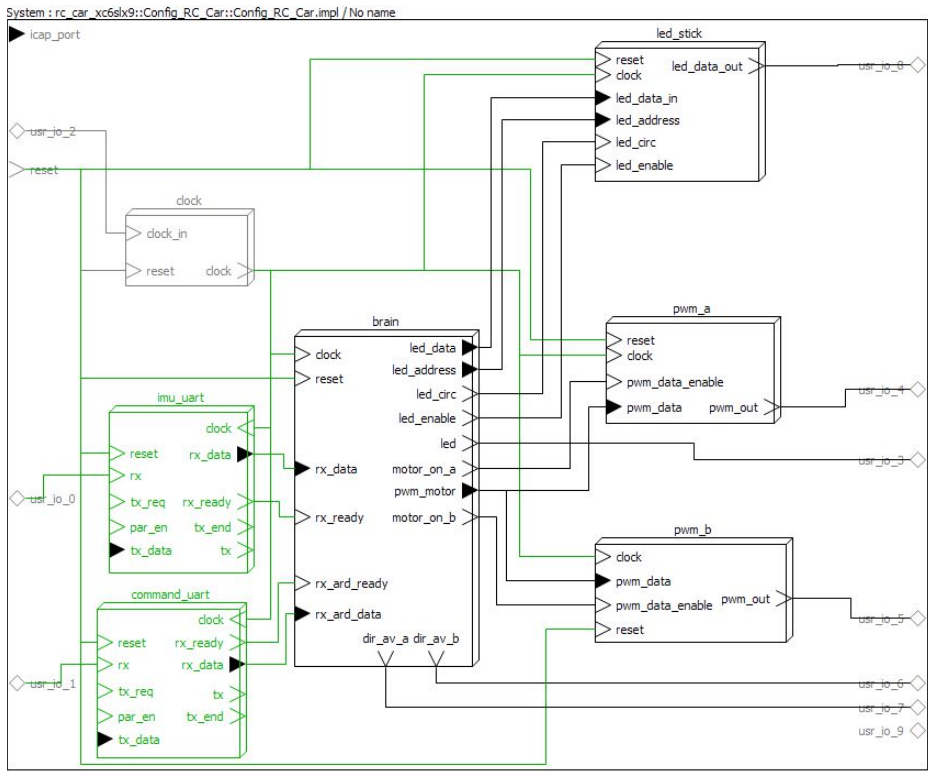

Fig. 6. An AADL execution platform model generated from the RC Car functional model

Each thread of the functional model is transformed into a processor subcomponent, which exhibits execution platform details such as clock and reset signals. The generated AADL FPGA component extends a template for the selected FPGA target. Grey elements on the diagram are inherited from the template, which in this case includes a clock. Green elements on the diagram have been added after the transformation to take into account requirements for the specific FPGA. In this case, 2 UART (Universal Asynchronous Receiver/Transmitter) controllers (in green) have been added to fix communication incompatibility between the brain controller and FPGA ports predefined in the template. Adding these controllers is currently performed by a Java procedure called at the end of the transformation, but the intent is to implement this as a GMM relation. In this way, new refinement relations can be integrated in the mega-model to target other execution platform specific needs.

4 Related Work

Several approaches have been proposed for GMM, most of them making use of mega-models. A summary is presented in [8], with our GMM language derived from the proposed unified definition. In [9], dynamical traceability management is proposed through the categorization of relations into factual and obligation types, which was also introduced in our language. In [14], another infrastructure for GMM is proposed and implemented in Eclipse, which combines mega-models with model weaving. However, it only supports basic functionality such as model navigation through traceability links. Automated production of the links and model synchronization are not supported. Our work enhances these approaches with meta-model subsets and model synchronization based on automated traceability link production. Our approach is also extensible so that new relations and tools can be integrated as needed.

A difficulty in GMM is to identify the relationships that are needed between models. The GEMOC initiative [15] proposes an initial categorization of relations in three different forms: interoperability, collaboration, and composition. Interoperability supports the exchange of information across models with minimum coupling between the models. It seems similar to model weaving proposed in [14]. Collaboration relationships support *coupled development* of models, where the development of a model directly influences the form of other models. This is similar to our synchronization relation, which influences the form of the associated models by maintaining their consistency during model evolution. Finally, composition relationships combine information from several models to create new forms of models. This is similar to EMF views [16], where several meta-models can be combined to provide new views on models, similar to database views.

5 Conclusion and Perspectives

Our experiment in using GMM for our EDA tool shows that several benefits can be obtained through explicit representation of the used meta-models and subsets, along with relations between models. Interpretation of relations ensures models are properly managed during their evolution to prevent errors introduced early in the development process. We think every model-based IDE should include a GMM infrastructure. One advantage of our GMM is that it was developed using rich and realistic modeling languages, which revealed important needs such as meta-model subsets and improved automated model synchronization.

However, many aspects of our infrastructure require improvements. Despite our enhancements, the TGG language would benefit from a complete review to improve aspects such as reuse of TGG elements across several TGGs. The study of other types of relations as proposed in [15] is also of interest, and in particular the integration of EMF views in GMM implementing meta-model composition relations. Finally, model to meta-model conformance could be enforced during meta-model evolution, represented as a conformance relation of obligation type, making use of frameworks such as Edapt [17].

6 References

1. P. H. Feiler, *Model-based Validation of Safety-critical Embedded Systems*, Aerospace Conference, pp. 1-10, 2010.
2. Y. Eustache, D. Blouin, M. Lanoë, J-P. Diguet, P. Coussy, Kaolin, a Model-based EDA Tool to Program, Reuse or Retarget Embedded Systems on FPGAs, Data Automation and Tests in Europe (DATE) conference, University Booth, 2014.
3. SAE International, Architecture Analysis and Design Language (AADL), http://standards.sae.org/as5506b/.
4. IEEE Standard VHDL Language Reference Manual, ANSI/IEEE Std 1076-1993, 199.
5. V. Gaudel, A. Plantec, F. Singhoff, J. Hugues, P. Dissaux, J. Legrand, *Enforcing Software Engineering Tools Interoperability: An Example with AADL Subsets*, IEEE International Symposium on Rapid System Prototyping (RSP), pp. 59-65, 2013.
6. The Model Transformation Engine (MoTE), http://www.mdelab.de/mote/.
7. A. Schürr, *Specification of graph translators with triple graph grammars*, in Graph-Theoretic Concepts in Computer Science, LNCS Volume 903, pp. 151-163, 1995.
8. R. Hebig, A. Seibel, H. Giese, *On the Unification of Megamodels*, Proc. of the 4th International Workshop on Multi-Paradigm Modeling (MPM), volume 42 of ECEASST, 2011.
9. A. Seibel, S. Neumann, H. Giese, *Dynamic hierarchical mega models: comprehensive traceability and its efficient maintenance*, Softw. Syst. Model, Volume 9, Issue 4, pp. 493-528, 2010.
10. D. Blouin, A. Plantec, P. Dissaux, F. Singhoff, J-P. Diguet, *Synchronization of Models of Rich Languages with Triple Graph Grammars: An Experience Report*, International Conference in Model Transformation (ICMT), pp. 106-121, 2014.
11. The EMF Compare Framework, http://www.eclipse.org/emf/compare/.
12. D. Blouin, D. Chillet, E. Senn, S. Bilavarn, R. Bonamy, C. Samoyeau, *AADL Extension to Model Classical FPGA and FPGA Embedded within a SoC*, International Journal of Reconfigurable Computing (IJRC), 2011.
13. The Leda RTL Checker, http://www.synopsys.com/tools/verification/.
14. F. Jouault, B. Vanhooff, H. Bruneliere, G. Doux, Y. Berbers, J. Bezivin, *Inter-DSL coordination support by combining megamodeling and model weaving*, Proceedings of the 2010 Symposium on Applied Computing (SAC), 2010.
15. B. Combemale, J. DeAntoni, B. Baudry, R. B. France, J-M. Jezequel, J. Gray, *Globalizing Modeling Languages*, Computer, vol.47, no.6, pp.68-71, June 2014.
16. The EMF Views Project, http://emfviews.jdvillacalle.com/.
17. The Edapt Project, http://www.eclipse.org/edapt/.