On the customization of model management systems for code centric IDEs

David Méndez-Acuña
Departamento de Ingeniería de Sistemas y Computación
Universidad de los Andes
Bogotá, Colombia
df.mendez73@uniandes.edu.co

Rubby Casallas
Departamento de Ingeniería de Sistemas y Computación
Universidad de los Andes
Bogotá, Colombia
rcasalla@uniandes.edu.co

Anne Etien
LIFL CNRS UMR 8022
Université Lille 1
Lille, France
anne.etien@univ-lille1.fr

ABSTRACT

Model-based solutions are becoming more sophisticated because of the advent of new types of models, languages, and editors. To deal with this complexity, some of the current Integrated Development Environments (IDEs) offer Model Management Systems (MMSs) that provide functionalities to visualize, navigate, and search the modeling artifacts existing in a workspace. Each MMS defines the types of modeling artifacts that it supports and, commonly, furnish extensibility mechanisms for including new ones. However, the use of those mechanisms usually requires a big implementation effort. As a result, when an MMS does not support all the types of modeling artifacts that a model-driven engineer uses, he/she discards it and ends up manipulating his/her solution through file system views which is not appropriate when projects become larger. In this paper we present some of our preliminary results towards the construction of MoMS-DL, a domain-specific language to define (and automatically generate) customized Eclipse-based MMSs improving the daily work of model-driven engineers.

Keywords

Model-driven software development, integrated development environment, model management, megamodeling

1. INTRODUCTION

The use of model-driven engineering (MDE) has increased in the last years. Nowadays, there is large variety of tools, languages, and editors that model-driven engineers can use during the construction of models-based solutions. In fact, they use not only models, metamodels and transformations but also weaving models, high-order transformations, UML profiles, grammars for domain-specific languages, among others modeling artifacts [4]. As a result, the process of building a model-based solution is becoming more and more complex.

In order to improve the productivity of model-driven engineers, it is desirable to have suitable IDEs that facilitate the manipulation of modeling artifacts by offering facilities such as friendly visualization, navigation, and searching [6]. Such facilities can be easily found in IDEs designed from scratch for model-based techniques (e.g., MetaEdit+ [3]). However, the situation changes in the case of other IDEs initially conceived for code-centric paradigms that have been later adapted for supporting model-based technologies (e.g., Eclipse Modeling Framework [10]). The idea of enriching code-centric IDEs for providing model-centric capabilities has been largely studied in the literature under the concept *"model management"*. Because of this, one can find several model management systems (MMSs) built on the top of code-centric IDEs.

In general, three common elements can be identified among MMSs: (1) metadata repositories; (2) models-centric views; and (3) searching engines. A metadata repository is a software artifact where some relevant information about the modeling artifacts (such as their location) is stored. The model-centric views show the modeling artifacts organized in hierarchies defined by their role in the models-based solution instead of their location in the file system. The searching engine is composed of several queries over the metadata repository that facilitate the searching of modeling artifacts. One of the best known examples of this type of MMSs is the AMMA platform presented by Bézivin et al in [1]. In that case, the metadata repository is a model that the authors term as *megamodel*; the models-centric views show the modeling artifacts classified by their types (e.g., metamodel, transformation, UML profile); and the searching engine is a set of OCL-based queries over the megamodel.

Commonly, the scope of an MMS is limited to a predefined set of modeling artifacts and, in order to increase their functionality, these tools offer extensibility mechanisms. The idea is that model-driven engineers can extend the MMS if they use types of modeling artifacts that are not initially included in the scope of the MMS. However, the use of those extensibility mechanisms requires a deep understanding of the software architecture and, in many cases, the code. From our point of view, this is one of the main shortcomings of the current approaches because, before extending a MMS, model-driven engineers prefer to manipulate their modeling artifacts by means of the classical file system views that IDEs like Eclipse offers by default. This is not appropriate when models-based solutions become large.

In this paper we present our preliminary results towards the construction of an approach that facilitates the customization of MMSs. To do so, we first motivate the problem by illustrating some of the shortcomings that model-driven engineers have to deal with when they do not use any MMS. Then, we introduce a tool that provides: ❶ a domain-specific language, called "MoMS-DL", that enables the definition of

MMSs; and ❷ a generation process that, from a MoMS-DL script, automatically generates a customized MMS that overcomes the studied shortcomings. Each of our generated MMSs is composed of a metadata repository and a set of features for friendly visualization and searching of modeling artifacts within a models-based solution.

This remainder of this paper is structured as follows: Section 2 introduces a set of definitions we use along the rest of the paper. Specifically, it clarifies the vocabulary concerning to metadata registries and megamodeling. Section 3 presents a motivating example that we use in Section 4 for describing the shortcomings of code centric IDEs for MDE. Also, we use this example in Section 5 for illustrating the MoMS-DL language and explaining the generation process for MMSs. Section 6 discuses the related work and Section 7 concludes the paper and presents the future work.

2. BACKGROUND

The use of the term *"metadata"* has become very popular to the point that it is difficult to find a definition that encompasses all the meanings that this term has received [5]. Indeed, the abstract idea of metadata as *"data about data"* seems to be the only point where all these definitions agree. In this sense, it is important to identify: (1) what are the data that need to be considered and (2) what are the data (about the data to be considered) that are relevant and that should be stored. From the implementation point of view, the *"metadata repository"* is the software artifact where the metadata is physically stored whereas the *"metadata schema"* is the definition of the structure that the metadata repository must conform to.

In the case of the MMSs for MDE, the data to be considered correspond to the set of modeling artifacts involved in a model-driven solution whereas the data to be stored correspond to the location of the modeling artifacts and the relationships existing among each of them. Furthermore, metadata repositories can be implemented as a special type of models (termed *"megamodels"*) whose structure is defined in metamodels [11, 4].

Figure 1 illustrates the previous definitions by using a simple situation in an Eclipse workspace that contains a model-based solution composed of three modeling artifacts: a metamodel `MMA`, a metamodel `MMB`, and a model-to-model transformation `A2B` that produces models conforming to `MMB` from models conforming to `MMA`. These three modeling artifact correspond to the data to consider. The metadata repository (implemented as a megamodel) contains the information about their location and the relationships among them. In this case, the relationships are the dependencies between the transformation and the metamodels i.e., source and target. The megamodel conforms to a metamodel that defines the supported types of modeling artifacts i.e., metamodels and transformations, and the interesting relationships.

3. MOTIVATING EXAMPLE: AN MTC FOR PROTOTYPING MAZE-GAMES

We propose, as motivating example, a software development scenario where a model-driven engineer builds a model-based solution for maze-games automatic prototyping [7]. Specif-

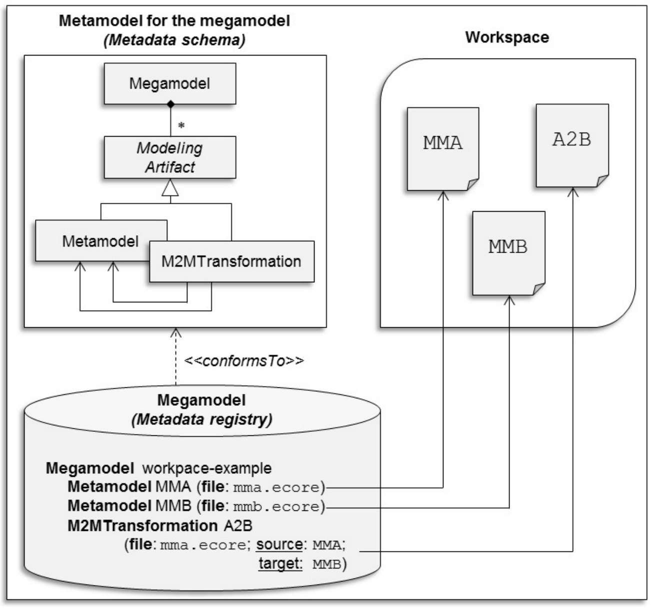

Figure 1: A simple megamodel (metadata repository)

ically, this solution is a model transformation chain (MTC) that, from a high-level game model, automatically produces the code of the game that can be deployed in the games engine Torque2D [2]. We chose this example because, although simple, its implementation requires more than models, metamodels, and transformations. Weaving models and domain-specific languages are also involved.

3.1 Maze-games

In general, a maze game consists of a playground where different objects can be placed. The game turns around a main character, usually controlled by the player, that has to develop a particular purpose during the game. Depending on the game, the main character must interact with other elements in order to achieve the goal. Depending on their behavior, these elements can be either dynamic or static depending. Dynamic elements develop some actions whereas static elements are either obstacles or boundary delimiters for the playground. Pacman is one of the most famous example of this type of games. In this case, the main character is the yellow smiley-face that dies if touched by the ghosts (dynamic elements) and that feeds the intermittent strawberries (static elements) for increasing the score.

3.2 MTC for maze-games prototyping

The inputs of the generation process (shown at the top of Figure 2), are threefold: (i) a maze game model; (ii) an instances model; and (iii) a weaving model between (i) and (ii). The maze game model contains the definition of the game and includes a characterization of the elements existing in the game (dynamics and statics). These elements are defined in terms of their properties and the way in which they interact each other. The instances model specifies the amount of each type of element that should be instantiated for a particular level and their initial positions. Because the definition of the element types and the instances are defined in separate models, it is necessary to provide a weaving model for relating each element type with each instance type. These models are input of a model-to-model transformation that creates a platform specific model with the definition of the game in terms of the Torque2D concepts. Then, this model is transformed to TorqueScript code that can be finally deployed in the games engine.

4. SOME SHORTCOMINGS OF CODE CENTRIC IDES ADAPTED FOR MDE

In this section we use the motivating example presented above to illustrate some of the shortcomings of classical IDEs that model-driven engineers have to deal with when they do not use any MMS.

4.1 Models visualization

The first shortcoming refers to the visualization and manipulation of a model-based solution where models have to be manipulated in code-centric views such as the package explorer or project navigator. In such views, models are treated as data files without any differentiation from code or documentation artifacts. As a result, a model-based solution looks like a set of projects and the semantics of each of them is only stored in the mind of the engineer.

Figure 2 illustrates this fact. At the top there is an abstract representation of the MTC for generating maze-games code. This representation is very close to the way in which the model-driven engineer imagines the MTC. At the button, we show the way in which the model-driven engineer actually visualizes the MTC in a code-centric view. Notice that those images are not similar at all. Indeed, the model-driven engineer is responsible for understanding how the projects represent each of the artifacts that compose the model-based solution.

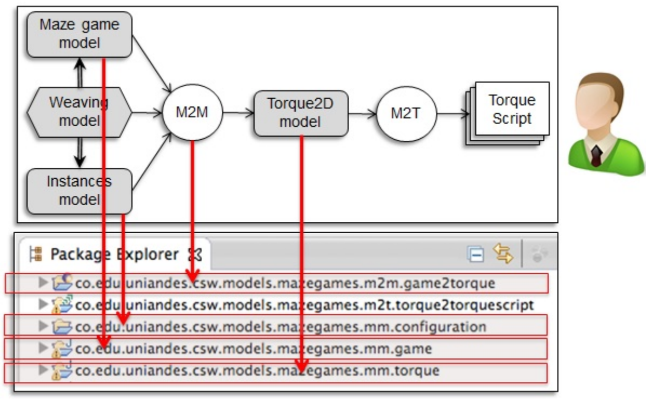
Maze game model
Weaving model
Instances model
M2M
Torque2D model
M2T
Torque Script
Package Explorer
co.edu.uniandes.csw.models.mazegames.m2m.game2torque
co.edu.uniandes.csw.models.mazegames.m2t.torque2torquescript
co.edu.uniandes.csw.models.mazegames.mm.configuration
co.edu.uniandes.csw.models.mazegames.mm.game
co.edu.uniandes.csw.models.mazegames.mm.torque

Figure 2: Code-centric views for model-based projects

4.2 Models searching

The second shortcoming is related to the way in which model-driven engineers search models. While java developers have enhanced searching mechanisms that allow to perform specialized searches in terms of classes, methods, attributes, and interfaces; model-driven engineers have to comply with the classical file search. Since this type of search is exclusively based on syntax matches, there is a large amount of non-relevant search results when the files are written in modeling languages such as UML, XMI, or ECORE. Suppose for example that the model-driven engineer needs to know the set of models conforming to a given metamodel. He/she would have to perform a file search with the URI of the metamodel. As a result, the model-driven engineer will obtain not only the models but also all the artifacts that contains that URI including java code and configuration files.

5. OUR APPROACH

As already said earlier, these limitations can be overcome by means of the construction of well-engineered MMSs. The problem arises when a model-driven engineer requires to use modeling artifacts that are not supported by the selected MMS and, hence, customization is required. Most of the MMSs that offer some type of customization provide strategies based on extensibility. In that sense, the software developer that wants to extend the MMS should: understand the extensibility strategy; build the extensions (probably writing code and/or models); re-compile the code of the platform; and re-install it. In this process, software developers need to understand some implementation issues of the MMS and, in most of cases, they are not necessary willing to deal with all this additional work. As a result, the use of MMSs is usually discarded when those does not fulfill by default all the initial requirements.

In order to avoid this limitation, we propose raising the level of abstraction in which customization is performed. Hence, we provide MoMS-DL (Model Management Systems Definition Language), a domain-specific language where the requirements of a specific model-driven engineer can be expressed and we offer a generation process that automatically produces the code of the MMS. In other words, we enable customization by automatic generation instead of customization by extensibility.

5.1 MoMS-DL: language concepts

We believe that the requirements that a model-driven engineer demands to a MMS can be expressed in terms of the types of modeling artifacts involved in a model-driven solution and the existing relationships among them. Our hypothesis is that, if we have that information, we can automatically produce a MMS that offer model-centric capabilities specialized in those artifacts. In that sense, MoMS-DL is a language intended to provide the expressiveness enough for defining a megamodel (i.e., a model that represents a set of modeling artifacts and the relationships among them).

Figure 3 shows the MoMS-DL metamodel. In that metamodel the central concept is the `MetaMegaModel` that is composed by a set of modeling artifacts represented by the `ArtifactType` concept. Each artifact type has references and attributes. A reference represents a relationship between two types of modeling artifacts whereas an attribute refers to a typed characteristic i.e., string, integer, double, float. Both, references and attributes have a multiplicity definition.

5.2 Using MoMS-DL

Let us illustrate the use of MoMS-DL in the motivating example. Listing 1 is a segment of the MoMS-DL script that would produce the MMS supporting the MTC for maze-games. Because of the space limitation, we only include the

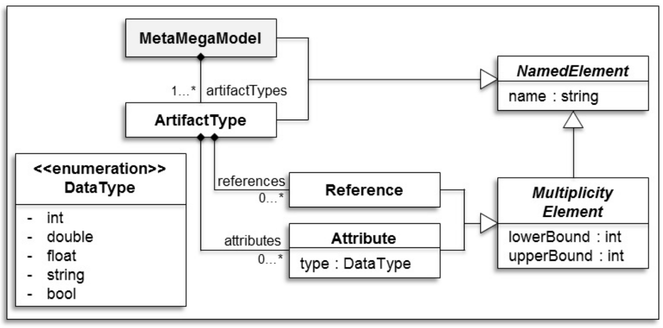

Figure 3: MoMS-DL metamodel

definition of models, metamodels, model-to-model transformations, and weaving models.

- A **metamodel** is an artifact type that has two attributes: the path and the URI. The path is a string representing the location of the file in the workspace whereas the URI is a string with an address that can be used for locating the metamodel in the web.
- A **model** is an artifact type that, besides its only attribute `"path"` referring to its location in the workspace, has a relationship called `"conformsTo"` to the type Metamodel that represents the conformance relationship.
- A **model-to-model transformation** is an artifact type that, besides its only attribute `"path"` referring to its location in the workspace, has two references to metamodel referring to the `"source"` and `"target"` metamodels of the transformation.
- A **weaving model** is an artifact type that has the `"path"` attribute and three references. The first reference is called `"weavingMM"` and refers to the metamodel that contains the well-formedness constraints of the weaving. The other two references called `"leftModel"` and `"righModel"` refer to the left and right parts of the weaving.

5.3 Tool implementation and generator of model management systems

We implemented MoMS-DL with EMFText [12]. Consequently, we can offer a friendly editor with content assistance and syntax coloring. Meantime, the generation of model management systems from MoMS-DL scripts is a process composed by three different code generators. They produce the metadata repository; the models-centric views; and the models searching engine. Let us explain each of those generators.

5.3.1 Generation of the metadata repository

The result of this generation process is a metadata repository implemented as a megamodel. To achieve this, we need: (1) a metamodel for the megamodel; and (2) a mechanism for manipulating megamodels by creating, deleting and updating the references to the concrete files so the metadata repository can be synchronized with the file system.

Listing 1: Definition of the MetaMegaModel

```
MetaMegaModel MetaMegaModelForMazeGames{

   artifactTypes{
      ArtifactType Metamodel{
         attributes{
             attribute path : string [1]
             attribute uri  : string [1]
      }

      ArtifactType Model{
         attributes{
             attribute path : string [1]
         }
         references{
             reference conformsTo : Metamodel [1]
         }
      }

      ArtifactType M2MTransformation{
          attributes{
             attribute path : string [1]
          }
          references{
             reference source : Metamodel [1...*]
             reference target : Metamodel [1...*]
          }
      }

      ArtifactType WeavingModel{
         attributes{
             attribute path : string [1]
         }
         references{
             reference weavingMM : Metamodel [1]
             reference leftModel : Model [1]
             reference rightModel : Model [1]
         }
     }
}
```

- **Construction of the metamodel for the megamodel:** The metamodel of the megamodel can be built directly from the MoMS-DL script since it contains the types of modeling artifacts and the relationships among them. Hence, we just need to transform the MoMS-DL script to a metamodel. To do so, we first parse the MoMS-DL script and obtain a model that conforms to the MoMS-DL metamodel.

  Figure 4 shows the metamodel for the megamodel that would be generated for our motivating example expressed in Listing 1. Notice that it is just two different representations of the same thing. However, having this information expressed in a metamodel allows to have a model where the metadata is directly stored. Figure 5 illustrates this idea by presenting a segment of the megamodel (represented as an objects model) corresponding to the metadata of the model-to-model transformation from maze-games to Torque2D. In that model, each modeling artifact is represented by an object that is an instance of a particular artifact type of the metamodel. As a result, we can have a structured metadata that can be manipulated and consulted.

- **Manipulation of the megamodel:** In order to maintain the megamodel synchronized with the file system, we automatically generate a set of CRUD operations that allow to manipulate the megamodel programmatically by creating, removing and updating megamodel elements (i.e., references to concrete files). Then, we offer a set of menu options that enable the invocation of such CRUD operations on the megamodel by including new artifact types. In that sense, the model-driven engineer is responsible for using those menu options in

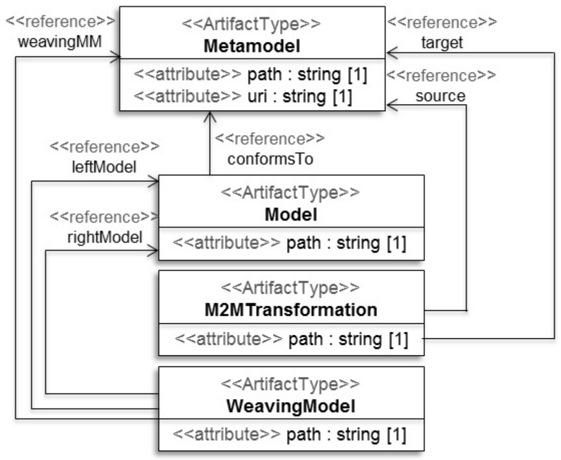

Figure 4: Metamodel of the metamodel for the maze-games example

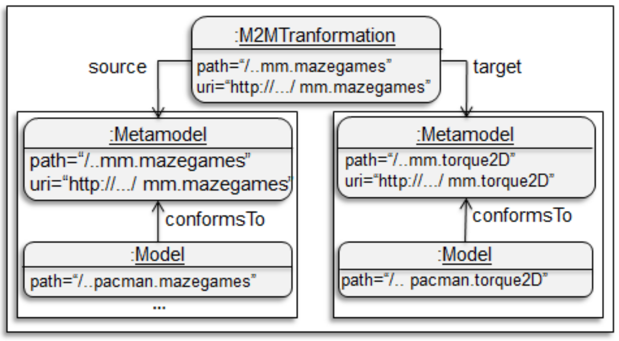

Figure 5: Megamodel for the maze-games example

order to maintain the megamodel up to date.

5.3.2 *Generation of the model-centric project views*

As we said before, the classical file system views provided by code centric IDESs show the modeling artifacts by using a file system structure. From our point or view, a model-centric project view should give a global perspective of the modeling artifacts existing in the workspace classified them by their types and not by their location in a folder hierarchy. Hence, from each MoMS-DL script we generate a model-centric project view where, for each `ArtifactType` we include a modeling artifact category. This generation process is based on a model transformation chain that produces a model-centric view from the MoMS-DL script. Figure 6 illustrates the generated view for the case of our motivating example.

5.3.3 *Generation of the searching engine*

In order to improve the way in which the model-driven engineer searches modeling artifacts, we provide a *model searching engine* based on the relationships among modeling artifacts defined in the MoMS-DL script. For each relationship, we generate two OCL queries for bi-directional searching. Each query is implemented as an OCL statement that is executed over the instance of the megamodel for the current workspace. The results are manipulated and displayed in

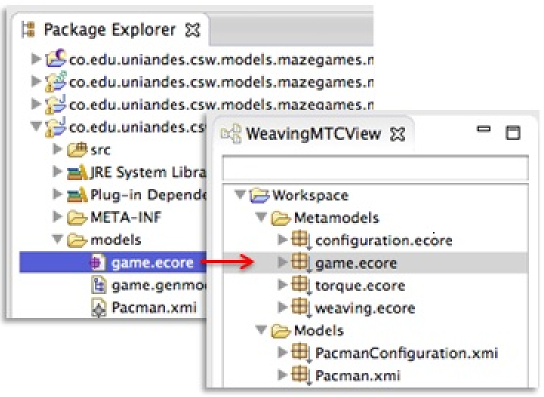

Figure 6: Code-centric view vs. models centric view

the corresponding searching results view.

6. RELATED WORK

Despite research in model management is not new, so far there is not a consensus about what that term exactly means. Indeed, there are (at least) two conceptions about how model management should be addressed. 1) operations-based model management systems: they intent to facilitate the definition and execution of diverse operations on models such as transformation, merging, or comparison; 2) indexation-based model management systems: they intent to facilitate manipulation of interdependent modeling artifacts by means of indexation (i.e., capability of maintaining metadata registries that store the references among models).

In the first case, the prime example would be the EPSILON suite [8] that aims at providing a family of interoperable languages. Those languages are ”task-specific” what means that each of them is intended to support a special type of operation. Thus, the model-driven engineer has a specific language to implement transformations, another for merging, matching, and so on. There are other approaches, such as [9], that offer customization mechanisms for operations-based MMSs. Thus, model-driven engineers can define generic-operations that can be applied to different types of modeling artifacts. For example, a merging operation that may be used on both models and metamodels.

In this paper we are more interested on indexation-based model management systems where one of the most famous example is the AMMA platform [1]. AMMA provides a complete infrastructure for supporting indexation of modeling artifacts following the concepts of modeling in-the-large introduced by Jean Bézivin [1]. This platform provides a megamodel, i.e., the metadata registry, and, based on it, a pool of functionalities for facilitating the tasks of model-driven engineers. Some of those functionalities are: Eclipse views for navigating modeling artifacts; and searching engines. The Eclipse views are intended to display modeling artifacts involved in a models-based solution classified by their types instead of the file system hierarchy where they are located. The models searching engine [4] is based on a domain-specific language that provides not only specialized searching for modeling artifacts but also programmatic execution of operations for models manipulation. Similarly to AMMA, it focuses on supporting the idea of "manipulating collections of related models" by offering Eclipse-based tool support for models metadata and operations execution. This approach offers also a graphical editor where the model-driven engineer manually indexes the modeling artifacts in a registry that, latter, serves to automate execution of the operators.

7. CONCLUSIONS AND FUTURE WORK

The main contribution of our work is to provide mechanisms for helping to the model-driven engineers to create customizable model management systems (MMSs). With the MMSs the model-driven engineers can have a better user experience with the IDE to develop their model-driven solutions. To have these suitable MMSs is important because model-driven solutions have become more and more sophisticated in terms of the amount and divers types of modeling artifacts they use. The contribution is in concrete the MoMS-DL language that allows model-driven engineers to define arbitrary megamodels. These megamodels are used as metadata repositories that contain the relationships existing among modeling artifacts. Besides, from the scripts in MoMS-DL and using the metadata registry we generate automatically a pool of plugins to improve the productivity of the developer.

There is still a long path to follow. Our on-going work includes extensions to the functionalities our plugins can offer.We are working also on including the ideas of operations-based model management systems by providing a work-flow language to define new operations (simple and composite) to manipulate modeling artifacts and the generation of the corresponding tooling.

8. REFERENCES

[1] J. Bézivin, F. Jouault, P. Rosenthal, and P. Valduriez. Modeling in the large and modeling in the small. In *Model Driven Architecture*, volume 3599 of *Lecture Notes in Computer Science*, pages 33–46. Springer Berlin Heidelberg, 2005.

[2] GarageGames. Torque2d: An engine to 2d games development, 2013.

[3] S. Kelly, K. Lyytinen, and M. Rossi. Metaedit+ a fully configurable multi-user and multi-tool case and came environment. In J. Bubenko, J. Krogstie, O. Pastor, B. Pernici, C. Rolland, and A. S¯lvberg, editors, *Seminal Contributions to Information Systems Engineering*, pages 109–129. Springer Berlin Heidelberg, 2013.

[4] W. Kling, F. Jouault, D. Wagelaar, M. Brambilla, and J. Cabot. Moscript: A dsl for querying and manipulating model repositories. In A. Sloane and U. A√ümann, editors, *Software Language Engineering*, volume 6940 of *Lecture Notes in Computer Science*, pages 180–200. Springer Berlin Heidelberg, 2012.

[5] K. Laskey. Metadata concepts to support a net-centric data environment. In R. Ladner and F. Petry, editors, *Net-Centric Approaches to Intelligence and National Security*, pages 29–54. Springer US, 2005.

[6] D. Lucrédio, R. M. Fortes, and J. Whittle. Moogle: a metamodel-based model search engine. *Software and Systems Modeling*, 11:183–208, 2012.

[7] L. Morales, D. Méndez-Acuña, and W. Montes. Model-driven game development - case study. a mtc for maze-game s prototyping. *Revista electrónica en construcción de software PARADIGMA*, 5(3):1–15, 2011.

[8] R. Paige, D. Kolovos, L. Rose, N. Drivalos, and F. Polack. The design of a conceptual framework and technical infrastructure for model management language engineering. In *Engineering of Complex Computer Systems, 2009 14th IEEE International Conference on*, pages 162 –171, june 2009.

[9] L. Rose, E. Guerra, J. Lara, A. Etien, D. Kolovos, and R. Paige. Genericity for model management operations. *Software and Systems Modeling*, 12:201–219, 2013.

[10] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF: Eclipse Modeling Framework 2.0.* Addison-Wesley Professional, 2nd edition, 2009.

[11] A. Vignaga, F. Jouault, M. Bastarrica, and H. Brunelière. Typing artifacts in megamodeling. *Software and Systems Modeling*, pages 1–15, 2011. 10.1007/s10270-011-0191-2.

[12] C. Wende, M. Seifert, F. Heidenreich, S. Karol, and J. Johannes. Emftext. concrete syntax mapper, 2013.