A Language-oriented Approach for the Maintenance of Megamodel-based Complex Systems

El Hadji Bassirou Toure1,2,3, Ibrahima Fall1,2,3, Alassane Bah1,2,3, Mamadou S. Camara1,2,3, Mandicou Ba1,2,3 and Ahmad Fall1,2

1École Supérieure Polytechnique, ESP, Dakar, Senegal
2Université Cheikh Anta Diop de Dakar, UCAD University, Dakar, Senegal
3Institut de Recherche pour le Développement, IRD Institute, Dakar, Senegal

Abstract: Model Driven Engineering (MDE) provides the concept of a runtime megamodel to represent the dynamic structure of a given system, to which it is causally connected. A system changes at runtime therefore implies frequent and dynamic changes of its related megamodel. In a previous work we have proposed to automate change management through a runtime megamodel evolution management approach. In such an approach, a megamodel manipulation, a kind of programming in-the-large activity, is considered as a mega-program which is modified throughout Global Operation Models (GOMs). Then we proposed a safe execution of GOMs as the solution for megamodel consistency preserving during evolution. In this work, we propose LAMEME, a domain-specific language for the management and the evolution of megamodels, and its axiomatic semantics. LAMEME gives the possibility to express an evolving megamodel as a mega-program and therefore defines a framework that supports our previously proposed approach.

1 INTRODUCTION

Configuration management is the discipline of managing change in large, complex systems. Its goal is to manage and control the numerous maintenance activities (corrections, extensions, and adaptations) that are applied to a system over its lifetime. Software configuration management (SCM) is configuration management applied to software systems. In the early 80s SCM focused on programming-in-the-large (versioning, rebuilding, composition) (Estublier, 2000). In a seminal paper entitled Programming-in-the-large vs. Programming-in-the-small (DeRemer and Kron., 1976), De Remer and Kron claimed that programming languages are well suited to describing algorithms and data structures, but not the structure of complex versioned software products. According to Favre, programming-in-the-large focuses on concepts related to the management of large software systems, taking into account four aspects: architecture, manufacture, evolution and variation (Favre, 1997).

A large software system is a system consisting of a large number of components and many dependencies, being dynamically developed (Murier and Bonatti, 2010). Such systems are inherently hard to manage largely because of their complexity. Indeed the complexity of such systems has exploded from isolated computers to global systems. It is for this reason that such systems are also called "complex systems".

To address the problems related to complex systems, De Rosnay advocates the use of a macroscope which can be considered as the symbol of a new way of observing, understanding, controlling and acting on complex systems (Rosnay, 1975). According to Barbero, MDE (Schmidt, 2006), may now provide the right level of abstraction to move the macroscope from its status of a symbolic instrument to a set of concrete and practical tools, ready to be used by engineers when they collectively work on complex computer-based systems (Barbero et al., 2008). The idea promoted by MDE is to use models at different levels of abstraction for developing systems, thereby raising the level of abstraction in program specification. An increase of automation in program development is reached by using executable model transformations (operations). Higher-level models are transformed into lower level models until the model can be made executable using either code generation or model interpretation.
Such concrete and practical tools, to which Barbero refers to, resulted in a kind of a special MDE model called a megamodel which is intended to provide a macroscopic representation of a complex system (Bezivin and al., 2004), (Bezivin and al., 2005b), (Vignaga A., 2009).

Subsequently a megamodel has to hide fine-grained details that obscure understanding and focus on the "big picture", i.e. system structure, interactions between models, assignment of models as parameters or results for model transformations, etc. Indeed, the notions of megamodelling (Bezivin and al., 2004) and modeling-in-the-large (Bezivin and al., 2005b) are introduced by analogy to respectively megaprogramming (Boehm and Scherlis., 1992) and programming-in-the-large (DeRemer and Kron., 1976) to address the development of large-scale systems from a modeling perspective called Global Model Management (Vignaga A., 2009).

Thus the task of manipulating the megamodel is considered as a programming-in-the-large one (DeRemer and Kron., 1976), where the megamodel acts as the mega-program which is manipulated through mega-modules represented by GOMs (Toure and al., 2018). Such megamodel manipulations coincide roughly with the needs of maintenance activities. Indeed nearly all systems particularly the complex ones will need maintenance over their lifetime because something in the system needs fixing (corrective maintenance), or external changes force a change to the system (adaptive maintenance), or something can be improved (perfective maintenance). However, whatever the type of maintenance involved, any megamodel change is done through the execution of GOM instances.

Such a situation leads mainly to two problems.

1. The problem of preserving megamodel consistency after changes, amounts to ensuring safe executions of GOMs by defining a formal semantics for each GOM instance. This problem has been addressed in previous works (Toure and al., 2018).

2. Considering the megamodel as a mega-program raises the issue of the language in which such a mega-program is written. The goal of this paper is to propose a domain-specific Language for the Management and the Evolution of Megamodels (LAMEME) and its axiomatic semantics. The design of LAMEME will enable us to check the correctness of GOM instances and thus to maintain the megamodel quality and/or integrity after changes.

Such concrete and practical tools, to which Barbero refers to, resulted in a kind of a special MDE model called a megamodel which is intended to provide a macroscopic representation of a complex system (Bezivin and al., 2004), (Bezivin and al., 2005b), (Vignaga A., 2009).

Subsequently a megamodel has to hide fine-grained details that obscure understanding and focus on the "big picture", i.e. system structure, interactions between models, assignment of models as parameters or results for model transformations, etc. Indeed, the notions of megamodelling (Bezivin and al., 2004) and modeling-in-the-large (Bezivin and al., 2005b) are introduced by analogy to respectively megaprogramming (Boehm and Scherlis., 1992) and programming-in-the-large (DeRemer and Kron., 1976) to address the development of large-scale systems from a modeling perspective called Global Model Management (Vignaga A., 2009).

Thus the task of manipulating the megamodel is considered as a programming-in-the-large one (DeRemer and Kron., 1976), where the megamodel acts as the mega-program which is manipulated through mega-modules represented by GOMs (Toure and al., 2018). Such megamodel manipulations coincide roughly with the needs of maintenance activities. Indeed nearly all systems particularly the complex ones will need maintenance over their lifetime because something in the system needs fixing (corrective maintenance), or external changes force a change to the system (adaptive maintenance), or something can be improved (perfective maintenance). However, whatever the type of maintenance involved, any megamodel change is done through the execution of GOM instances.

Such a situation leads mainly to two problems.

1. The problem of preserving megamodel consistency after changes, amounts to ensuring safe executions of GOMs by defining a formal semantic for each GOM instance. This problem has been addressed in previous works (Toure and al., 2018).

2. Considering the megamodel as a mega-program raises the issue of the language in which such a mega-program is written. The goal of this paper is to propose a domain-specific Language for the Management and the Evolution of Megamodels (LAMEME) and its axiomatic semantics. The design of LAMEME will enable us to check the correctness of GOM instances and thus to maintain the megamodel quality and/or integrity after changes.

The remainder of the paper is organized as follows. We first, in section II, present the megamodel structure and behavior. In section III, we present the syntax of LAMEME and describe its semantics (operational and axiomatic semantics). Section IV presents some related works and section V concludes the paper.

2 MEGAMODELLING

2.1 A Megamodel: A Kind of SCM

Typically SCM consist of two parts which collaborate in a harmonious way (Estublier, 2000). The first part is about the management of a repository of components. Indeed there is a need for storing the different components of a software product and all their versions safely. The second part of SCM concerns the process control and support. In fact, traditionally, change control is an integral part of an SCM product.

Otherwise, on the one hand, megamodels have often been used as a register for software architecture components and their interconnections (Bezivin and al., 2004), (Kling and al., 2011). On the other hand, megamodels have also to take under consideration the dynamic aspect of a system i.e the way a system change (Seibel A., 2009), (Favre and Nguyen., 2005). Thus in a MDE context, a megamodel plays the role of a SCM. Therefore a megamodel results from a Repository Model (RM) which represents its static aspect and an Execution Environment Model (EE) to represent the operations performed in the megamodel.

2.2 The Repository Model

The RM is the place wherein Component Models (CMs) and their relations known as Global Links (GLs), and GOMs are represented, stored and manipulated. A component is any reusable software unit. It can be as simple as a class diagram or as complex as a whole system. We use models of such components in order to manage all of them in a homogeneous way without considering their nature and their internal details.

A GL between two CMs may be viewed as summarizing a set of interactions between that two CMs which are said to be dependent each other (Bezivin and al., 2005b). Among others GLs we can distinguish the conformsTo relation between a CM and its reference model, or the realizedBy relation between a CM and its interface, etc. GLs are very useful for changes propagation management because if a CM is modified then all CMs which depend on it are likely to be targeted by this change.

A CM always represents a realization of another CM which represents its interface. On the one hand a
CM can be realized throughout more than one CM, each of them conforming to a specific metamodel. Such realizations of a given component are called variants of that component. On the other hand, each time a CM is changed a new revision of that CM is created. A variant (resp. revision) of a given CM is called a logical (resp. historical) version of that CM. At the same time, for a given CM, several variants may exist but only one revision of that CM is stored in the RM. Typically it is the last created. A variant may be derived in several successive revisions. The existence of different variants can be independent from the notion of "time". In the RM, variants correspond to significant changes such as different metamodels, whereas revisions result from addition of new functionality, error corrections, refinements, etc.

2.3 Execution Environment

The Execution Environment controls the dynamic aspect of the megamodel by enabling GOMs to be loaded from the RM and to be performed. Indeed a GOM can only be applied to CMs already contained in the RM, and its results are new CMs which are automatically added in or old CMs may be removed from the RM. However the details of CMs are abstracted away and the implementation details of GOMs considered not relevant. For example, we can use a GOM such as merge to compose two CMs. In this case, we are only interested in the impacts of that operation in the megamodel and, potentially, to its effects in other CMs but not on how such a merge GOM is defined. So, focus is made in the product and not in the process of GOMs in the megamodel.

The goal of a SCM is the control of the evolution of complex systems. A megamodel being a SCM is modified throughout the execution of GOM instances. So we use a kind of a complex model, namely a configuration model to describe an instance of a GOM, the list of CMs as parameters and the CM as the result of such a GOM. The parameters of a configuration model are called "source CMs" and the result is called "derived CM". A configuration model \( \mathcal{C} \) is defined by the tuple : 
\[ \mathcal{C} = \langle M_c, O p, M_t \rangle, \]
where :
- \( M_c \) is the list of source CMs \( \mathcal{C} \);
- \( O p \) is an instance of a GOM (e.g. merge, match);
- \( M_t \) is the derived CM of \( \mathcal{C} \).

An instance of a given configuration model can be defined and will take as input a revision of each CM in the list \( M_c \), a specific instance of the GOM \( O p \) and will give as output a new variant of the CMs in the list \( M_t \). For example, a configuration model for a GOM instance which have to merge two CMs representing respectively the data model of two components \( s \) and \( t \), and a mapping between that two CMs. Such an operation produces a CM representing a merge of \( s \) and \( t \). Such a configuration model is formally described below :
\[ < \mathcal{C} = \{ m_{sd}, m_{td}, m_{td}, \text{merge} \}, \text{merge}_d, \text{merge}_d > \]  
where 
\[ m_{sd} \in \text{input}(\mathcal{C}) \Rightarrow \exists m \in \text{revList}(s_{\text{model}}) : m = m_{sd}. \]
\[ m_{td} \in \text{input}(\mathcal{C}) \Rightarrow \exists m \in \text{revList}(t_{\text{model}}) : m = m_{td}. \]
\[ m_{map} \in \text{input}(\mathcal{C}) \Rightarrow \exists m \in \text{revList}(\text{map}_r) : m = \text{map}_r; \]
\[ \text{merge}_d \in \text{output}(\mathcal{C}). \]
\[ \text{map}_d \] is a mapping between \( m_{sd} \) and \( m_{td} \). Such a mapping is a morphism that identifies combinations of objects in the input models that are equal. 

3 A LANGUAGE-ORIENTED MEGAMODEL MANAGEMENT

3.1 Overview

In this section, we present a textual domain-specific modeling language (DSML), called LAMEME. It is about a programming language or an executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, the particular problems of megamodel evolution and maintenance. LAMEME can be considered as an interpreted functional language because GOM instances (functions) and input CMs (parameters) passing are the primary means of accomplishing the megamodel changes.

Because of space constraints the abstract syntax of LAMEME (the metamodel of the proposed megamodel), representing its model is not presented in this paper (Toure and al., 2017). Otherwise the concrete syntax of LAMEME is highlighted through some simple examples. So, this paper mainly focus on the semantics of LAMEME which is more relevant for the proposed approach.

Indeed to define the semantics of LAMEME, we use Hoare's axiomatic semantics or in its modern form, the Design By Contract method (DBC) (Meyer, 1992). The principal idea behind such an approach is that a GOM instance being executed and the EE have a "contract" with each other. The EE must guarantee certain conditions (preconditions) before calling an operation defined by the specific GOM instance, and in return the specific GOM instance guarantees
certain properties (postconditions) that will hold after the call. The use of DBC against Hoare-Logic is that it makes these contracts executable.

In LAMEME, the contracts also called specifications are defined using annotations and are translated into executable code by the interpreter. Thus, any violation of the specification that occurs while a GOM instance is executed can be detected immediately. A specification defines the oracle for a particular GOM execution. An oracle is a mechanism for determining behavioral correctness of GOM execution by examining its output. Indeed a GOM execution requires an oracle to determine whether its output is correct or not. Often such an oracle is a human, but the process can be time-consuming, tedious and error-prone. Whereas if the specification is mathematical (or logical), such as in LAMEME through Hoare logic, it is possible to derive a software oracle from it.

Such a software oracle is called a Runtime Assertion Checker (RAC) (Cheon and Leavens., 2002), that we have used in LAMEME in order to determine if a given GOM execution is safe for the megamodel.

### 3.2 Syntax

#### 3.2.1 Listings

The Listing 1. starts with the definition of an execution context (Line 1) that can be simply considered as a namespace. Then, the KM3 metamodel is imported from the execution context com.mega.init (Line 2). After the elaboration phase (imports and metamodels definition), the initialization phase starts with the definition of a megamodel instance megaWork (Line 6). Before the definition of a CM you must first determine the interface to be realized by that CM. This interface is called the "megaprogram" (Line 7).

Listing 1: CMs definition.
```plaintext
context com.mega.test {
  use metamodel com.mega.init.KM3
  def metamodel Kermeta { ... }
  def megamodel megaWork { ... }
  def interface IGeneral { ... }
  def interface IKM3 { ... }
  def component model KM3Model:KM3 implements IKM3 {
    properties {
      service "data",
      type "descriptive",
      level "terminal",
    }
  }
  def component model KM3ExtModel:KM3 extends KM3Model {
    properties {
      service "data",
      type "descriptive",
      level "terminal",
      language "KermetaLanguage",
    } otherProperties "An executable model"
  }
  def component model UMLModel:com.mega.init.UML2 implements IGeneral {
    properties {
      service "data",
      type "descriptive",
    }
  }
}
```

Listing 2: CM definition: variant.
```plaintext
def component model KM3ExtModel:KM3 extends KM3Model {
  properties {
    author = "Max Payne",
    language = "KermetaLanguage"
  }
}
```

Listing 3: CM definition: revision.
```plaintext
def configuration model matching KM3.Kermeta: Match {
  ...}
def configuration model matching KM3.Kermeta: Kermeta {
  ...}
```

Listing 4: Configuration model definition.
```plaintext
megaprogram matchingModel {
  let KM3 m1
  let Kermeta m2
  m1 = exec(model::query(with (name = "KermetaModel") and (author = "John Doe") and (Version > 0)))
  m2 = exec(model::query(with (name = "KermetaModel") and (name = "KM3ExtendedModel")))
  let configuration matchGOM,23122018 = exec(configuration:
    load(with name = "matching KM3.Kermeta"))
  let any m3 = call matchGOM,23122018(m1, m2)
  m3 = exec(model::save(with name = "map_model.m3"))
}
```

Listing 5: Megaprogram definition.

#### 3.2.2 Listing Descriptions

The Listing 1. starts with the definition of an execution context (Line 1) that can be simply considered as a namespace. Then, the KM3 metamodel is imported from the execution context com.mega.init (Line 2). After the elaboration phase (imports and metamodels definition), the initialization phase starts with the definition of a megamodel instance megaWork (Line 6). Before the definition of a CM you must first determine the interface to be realized by that CM. This interface is called the "megaprogram" (Line 7). The Listing 13 states that the KM3Model conforms to the metamodel KM3 and realizes the IKM3 interface. The first step in creating any composite system is to locate the CMs that provide data or functionality that is to be placed in the new system. For example, each CM should be
named, typed, and must specify what information it represents (a data element, a process element, etc). Such characteristics are represented through a set of couples \(<\text{attribute}, \text{value}>\) in the properties section (Line 8 - 11).

The Listing 2. shows the creation of two CMs which are variants each other. Indeed the CMs Ker-
metaModel and UMLModel implement both the same interface IGeral but, the fact that the two CMs conform to two distinct metamodels results they are vari-
ants one another.

The Listing 3. shows the creation of a new revision (KM3ExtModel) for the CM KM3. By contrast with variants definition, two revisions have to con-
form to the same metamodel.

The Listing 4. highlights the definition of a configuration model (matching KM3 Kermeta) repre-
senting a generic instance of the Match GOM. Such a generic instance has as sources the KM3Model and the KermetaModel and as derived the KM3 Kermeta MatchModel. It is possible to instant-
tiate the generic instance matching KM3 Kermeta to produce a specific instance of the Match GOM. Such a specific instance will take as input a revision of the KM3Model and a revision of the KermetaModel and will produce as output a mapping between that two CMs.

The Listing 5. starts the construction phase by defining the matchingModel megaprogram which starts with the declaration of two CMs m1 and m2 which conform respectively to KM3 and Kermeta metamodels. To m2 is assigned the result of the search of any revision of the KermetaModel (there existst only one). The result of the search of the last created revision (KM3ExtendedModel) of the KM3Model is assigned to m1. A specific instance matchGOM_23122018 of matching KM3 Kermeta is created (line 6). A generic model m3 is created to which is assigned the result of the application of matchGOM_23122018 to m1 and m2 (line 7). Finally m3 is saved in the RM as map model.m3 which re-
presents a mapping between KM3Model and Kermeta-
Model (line 8). Line 10 corresponds to the matching-
Model execution launch.

3.3 Semantics

Describing syntax is easier than describing semantics, partly because a concise and universally accepted notation is available for syntax description, but none has yet been developed for semantics.

Operational semantics are easy to define and under-
stand, similarly to implementing an interpreter (Figure. 1). They require little formal training, scale up well, and, being executable, can be tested. Thus, operational semantics are typically used as trusted reference models for the defined languages.

Despite these advantages, they are rarely used directly for program verification, because proofs tend to be low-level, as they work directly with the correspond-
ing transition system. Moreover the real problem with an operational semantics is the interpreter itself: it is represented as an algorithm. If the algorithm is simple and written in an elegant notation, the interpreter can give insight into the language.

Otherwise, Hoare or dynamic logics allow higher level reasoning at the cost of (re)defining the lan-
guage as a set of abstract proof rules. Hoare logic, often identified with axiomatic semantics, was pro-
posed more than forty years ago (Hoare, 1969) as a rigorous means to reason about program correctness.

In order to define an axiomatic semantic for LAMEME programs, GOMs in this case, we use a framework similar to JML (Java Modeling Lan-
guage), (Leavens and al., 1998) which is a powerful DBC tool for Java. As such, it allows one to spec-
ify both the syntactic interface of Java code and its behavior.

GOMs specifications, as in JML, are written in special annotation comments, which start with an at-
sign (@). We use a requires clause to specify the GOM’s preconditions (a requires clause consists of an assign clause to specify the GOM’s parameters, an initially clause to specify the properties of pa-
rameters, etc.) and an ensures clause to specify the GOM’s postcondition. A contract is typically written by specifying a GOM’s pre and postconditions. A GOM’s precondition says what must be true about in-
put and output CMs to call it. A GOM’s postcondition says what must be true about input and output CMs when it terminates. Unlike simple comments, formal speci-
fications in LAMEME are machine checkable, and so can help with debugging. That is, checking the specification can help isolate errors before they prop-
agate too far.

If an assertion in a GOM specification does not hold when the execution control reaches it, one knows that something is wrong with either the GOM definition (execution) or the GOM specification Figure 2. Generally, stating what one believes to be true about a program as assertions and checking that program using assertions at runtime is an effective means of increasing the quality of programs.

For such purposes, a runtime assertion checker is used as the decision procedure for oracles which are based on the formal specifications. The way of implementing oracles is to compare the execution output of an instance of a GOM to some pre-calculated, presumably correct, output defined using Hoare’s inference rules. It’s about building expected outputs and comparing them to the execution outputs, the specified behavior of the GOM under execution is monitored to decide whether an execution is safe or not. Execution oracles catch assertion violation exceptions from the execution of an instance of a GOM to decide if the GOM failed to meet its specifications, and hence that its execution is unsafe for the megamodel. An assertion violation will occur, and a specified exception will be thrown to indicate the detail information when some checking failure of specifications is encountered. If an execution of an instance of a GOM satisfies its formal specifications, no such exceptions will be thrown, and that particular execution will be considered as a safe one.

### 3.4 Illustrative Example

In this illustrative example we consider the LAMEME program below defined in Listing 6. In this example we consider that $s$ represents the complex system represented by the runtime megamodel to which will be plugged a component $t$. Such an integration of $t$ into $s$ will be done according to their data dimension. For that, we suppose that the data model of $s$ is represented by the CM $s$.datamodel represented using the UML metamodel. We also consider that the data model of $t$ is represented by the CM $t$.datamodel which also conforms to the UML metamodel. The fusion of $s$.datamodel and $t$.datamodel will produce the $s$._merge CM which will be the new data model of $s$ and subsequently $s$._merge will represent the data of the megamodel.

```plaintext
context com.mega.example {
  def configuration model Merge.t.s :: Merge {
    @ ASSIGN m1 as input IMPLIES m1 as s_revision
    @ ASSIGN m2 as input IMPLIES m2 as t_revision
    @ ASSIGN m as input IMPLIES m as s_match_t.
    @ INITIALLY (service, level, language, otherProperties )
    IN m1
    @ INITIALLY (service, level, language, otherProperties )
    IN m2
    @ RESULTS m3 as output IMPLIES m3 as s_merge.t.
    @ RESULTS m3 as output IMPLIES m3 as s_merge.t.
    @ REQUIRES (P, (m1.service EQUALS m2.service) AND
    (m1.level EQUALS m2.level) AND (m1.language EQUALS
    m2.language) AND
    (m as mapping) AND (m EQUALS m1 MATCH m2))
    @ ENSURES (Q, (m3.service EQUALS m1.service) AND
    (m3.level EQUALS m1.level) AND (m3.language EQUALS
    m1.language))
    @ input from: repository s.datamodel
    @ output s_merge.t
    megaprogram exampleMerge {
      let UML2 t_m1 = null
      let List <UML2> results = exec(interface::query(
        realisation , with = "IDatamodel.s")
      while( results != null )
        let any s_m1 = exec(model::queryLast(with
          ReferenceModel = "UML2"
          and RealizedInterface = "IDatamodel.s")
        if (exec(model::exists(with = "s.datamodel" and
          Version = "3.X" )))
          t_m1 = exec(model::new( realisation , with
            ReferenceModel = "UML2"
            and RealizedInterface = "IDatamodel.t" and
            Version = "1")
        }
        let configuration mergeInstance = exec( configuration ::
          load(with name = "Merge.t.s")
        let UML2 merge_st = call mergeInstance(s_m1, t_m1)
        merge_st = exec(model::save(with name ="s_merge.t")
      }
    }
  }
}
```

Listing 6: Illustrative example.

In Listing 6, we define a configuration model Merge.t.s which is a generic GOM of type Merge that...
takes as input a $s$ datamodel and a $t$ datamodel from the RM and which gives as output the CM $s$.merge.t.

The megaprogram exampleMerge starts with the declaration of a CM $m1$ of type UML2. Then a list of CMs which are realizations of the interface IDataModel.s is assigned to the variable results. A generic CM $s$.m1 is declared to which is assigned the result of the search of the last revision of the realizations of the IDataModel.s and which is conform to UML2. If a CM representing the $s$.datamodel exists in the RM and if such a CM represents the third alternative without considering which revision it represents then the $t$ m1 CM will be assigned the only one revision which represents the realization of the IDataModel.t and which conforms to the UML2 metamodel.

The execution of the exampleMerge megaprogram using the RAC shows that the Merge instance MergeJ.s does not meet its specifications because any generic instance of Merge GOM should have to take three parameters: the two CMs to be merged and a third CM which is a morphism representing the links between the CMs to be merged. In the definition of the specific Merge instance mergeInstance, there is no CM representing the mapping of the two CMs to be merged. The program throws an exception. We say that the execution of the generic Merge instance MergeJ.s is not a safe execution for the megamodel because such an execution is likely to introduce inconsistencies in the megamodel. The RAC causes the program abort Figure 3.

![Figure 3: An unsafe execution of the generic Merge instance: MergeJ.s.](image)

To fix it, we have to redefine the configuration model MergeJ.s by using a third parameter which represents the mapping between the two input CMs. Such a model mapping is similar to that defined in Listing 4.

Indeed the third parameter of the generic instance MergeJ.s was obtained as the result of the generic Match instance. In other words, the third parameter of the generic instance MergeJ.s should have to be a specific instance of the generic instance MatchJ.t.s.

Listing 7 gives the definition of the generic Merge instance with three parameters.

```
7 } }
```

Listing 7: The generic Merge instance with the mapping parameter.

In Listing 8, the exampleMerge megaprogram is re-executed but using the generic instance MergeJ.s with three parameters. Indeed the MergeJ.s is redefined by adding a third parameter representing a mapping of the two input models which uncovers how the two source models "correspond" to each other. The new Merge instance MergeJ.s does not produce any longer any inconsistency in the megamodel, so such an execution is safe for the megamodel as illustrated in Figure 4.

![Figure 4: A safe execution of the Merge instance MergeJ.s.](image)

Listing 8: The sound exampleMerge megaprogram.

```
1 def configuration model MergeJ.s::::Merge {
2 /* */
3 @input from: repository t.datamodel
4 @input from: repository s.datamodel
5 @input from: repository s.match.t
6 @output s.merge.t
7 }
```

4 RELATED WORK

In this section we provide a short overview of the literature related to the tools and/or languages which are developed in the field of Global Model Management (GMM).

MoDisco (Bruneliere and al., 2014), offers a generic and extensible model-driven reverse engineering (MDRE) framework intended to facilitate the
elaboration of MDRE solutions actually deployable within industrial scenarios. The provided description includes its overall underlying approach, architecture, available components and the detail of Model Discovery and Model Understanding.

Neo4EMF (Benelallam and al., 2014), is a tool that can improve the applicability of MDE to large-scale scenarios, where on-demand loading, high-performance access and enterprise-level data-management features are needed.

MoScript (Kling and al., 2011), is a solution for Global Model Management (GMM), based on the notion of a Megamodel. The MoScript language is an OCL-based scripting language for model-based task, based on the metadata contained in a Megamodel and allows for querying a Megamodel.

The AM3 tool (Bezivin and al., 2005a), is an environment for dealing with models or metamodels, together with tools, services and other global entities, when considered as a whole. For each platform, authors suppose that there is an associated megamodel defining the metadata associated to this platform.

5 CONCLUSION AND FUTURE WORK

In this work, we propose LAMEME, a domain-specific language for the management and the evolution of megamodels, and its axiomatic semantics. Our proposal seems at the moment staying at a too high level of abstraction. A difficulty with metamodeling and even more with megamodeling approaches resides in the high level of abstraction they involve. Thus, as future works we are looking at using our solution of complex system maintenance in a real case study in order to fully validated the proposed approach.

ACKNOWLEDGEMENTS

The research in this paper is supported by the Centre d’Excellence Africain en Mathmatiques, Informatique et TIC (CEA-MITIC).

REFERENCES


Toure, E. and al. (2017). Consistency preserving for evolving megamodels through axiomatic semantics. In Intelligent Systems and Computer Vision. IEEE.
