

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

El Hadji Bassirou Toure^{1,2,3}, Ibrahima Fall^{1,2,3}, Alassane Bah^{1,2,3}, Mamadou S. Camara^{1,2,3},
Mandicou Ba^{1,2,3} and Ahmad Fall^{1,2}

¹*École Supérieure Polytechnique, ESP, Dakar, Senegal*

²*Université Cheikh Anta Diop de Dakar, UCAD University, Dakar, Senegal*

³*Institut de Recherche pour le Développement, IRD Institute, Dakar, Senegal*

Keywords: Software Configuration Management, MDE, Megamodeling, DSML, Design By Contract.

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 in *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 (Murer and Bonati., 2010). Such systems are inherently hard to man-

age 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 and 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 megamodeling (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 : $\mathcal{C} = \langle M_e, Op, M_s \rangle$, where :

- M_e is the list of *source CMs* \mathcal{C} ;
- Op is an instance of a GOM (e.g : merge, match);
- M_s 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_e , a specific instance of the GOM Op and will give as output a new variant of the CMs in the list M_s . 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 :

$\langle \mathcal{C} = \{ms_d, map_d, mt_d\}, merge_t^s, merge_d \rangle$ where
 $ms_d \in \text{input}(\mathcal{C}) \Rightarrow \exists m \in \text{revList}(s_model) : m == ms_d.$
 $mt_d \in \text{input}(\mathcal{C}) \Rightarrow \exists m \in \text{revList}(t_model) : m == mt_d.$
 $map_d \in \text{input}(\mathcal{C}) \Rightarrow \exists m \in \text{revList}(map_t^s) : m == map_t^s.$
 $merge_d \in \text{output}(\mathcal{C}).$

map_d is a mapping between ms_d and mt_d . Such a mapping is a morphism that identifies combinations of objects in the input models that are equal.

$merge_t^s$ is a generic instance (a function definition) of the merge GOM. Thus an application of such a function is done by defining specific instances of the merge GOM.

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

```

1 context com.mega.test {
2   use metamodel com.mega.init.KM3
3   def metamodel Kermeta { ... }
4   def megamodel megaWork {
5     def interface IGeneral { ... }
6     def interface IKM3 { ... }
7     def component model KM3Model:KM3 implements IKM3 {
8       properties (
9         service "data",
10        type "descriptive",
11        level "terminal",
12      )
13    }

```

Listing 1: CMs definition.

```

1 def component model KermetaModel:Kermeta implements
   IGeneral{
2   properties (
3     service "data",
4     type "descriptive",
5     level "terminal",
6     language "KermetaLanguage",
7     otherProperties "An executable model"
8   )
9 }
10 def component model UMLModel : com.mega.init.UML2
   implements IGeneral {
11   properties (
12     service "data",
13     type "descriptive",

```

```

14   level "terminal",
15   language "UML2.3",
16   otherProperties "An executable model"
17 )
18 }

```

Listing 2: CM definition: variant.

```

1 def component model KM3ExtModel:KM3 extends KM3Model
   implements IKM3 {
2   properties (
3     author "Max Payne",
4     language "KM3Language"
5   )
6 }

```

Listing 3: CM definition : revision

```

1 def configuration model matching_KM3_Kermeta::Match {
2   ...
3   @input from: repository KM3Model
4   @input from: repository KermetaModel
5   @output KM3_Kermeta_MatchModel
6 }

```

Listing 4: Configuration model definition.

```

1 megaprogram matchingModel {
2   let KM3 m1
3   let Kermeta m2
4   m2 = exec(model::query(with (name = "KermetaModel") and (
5     author = "John Doe") and Version > 0))
6   m1 = exec(model::queryLast(with (ReferenceModel = "KM3")
7     and (name="KM3ExtendedModel")))
8   let configuration matchGOM_23122018 = exec(configuration
9     ::load(with name = "matching_KM3_Kermeta"))
10  let any m3 = call matchGOM_23122018(m1, m2)
11  m3 = exec(model::save(with name = "map_model_m3"))
12 }
13 matchingModel.execute()

```

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 is what has been done at *Line 7 - 10*. The *Line 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 $\langle \text{attribute}, \text{value} \rangle$ in the *properties* section (Line 8 - 11).

The Listing 2. shows the creation of two CMs which are variants each other. Indeed the CMs *KermetaModel* and *UMLModel* implement both the same interface *IGeneral* but, the fact that the two CMs conform to two distinct metamodels results they are variants 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 conform to the same metamodel.

The Listing 4. highlights the definition of a configuration model (*matching_KM3_Kermeta*) representing 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 instantiate 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 exist 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 represents a mapping between *KM3Model* and *KermetaModel* (line 8). Line 10 corresponds to the *matchingModel* 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 understand, 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.

```

def configuration model matching_KM3_Kermeta : Match {
  metadata {
    namespace "Lama.Dsl"
    date "19/02/2018"
    description "The match GOM takes two models as input and returns
    a mapping between them. The mapping identifies contributions
    of objects in the input models that are equal."
    version "1"
  }
  @input Nonconformity KM3Model
  @input Nonconformity KermetaModel
  @input KM3_Kermeta_MatchModel
}

megaprogram matchingModel {
  let GOM def
  let Kermeta m2
  m1 = nonconformity.queryWith (name = "KermetaModel") and (author = "Lama.Dsl") and (version = "1")
  m2 = nonconformity.queryWith (name = "KermetaModel" = "197" and (author = "19/02/2018"))
  let configuration matchGOM_23122018 = nonconformity.configuration.with (name = "matching_KM3_Kermeta")
  let m3 = nonconformity.configuration.with (name = "map_model.m3")
}

matchingModel.execute()

```

Execution Engine (LAMEME) 11

```

Starting LAMEME's Interpreter ... done
> Megaprogram "matchingModel"
> Loading configuration model -> matching_KM3_Kermeta : Match(GOM)
> Input -> m1 > KM3Model
> Output -> m2 > Kermeta_MatchModel
> Loading component model "map_model.m3"
>

```

Figure 1: LAMEME's Operational Semantic.

Despite these advantages, they are rarely used directly for program verification, because proofs tend to be low-level, as they work directly with the corresponding 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 language as a set of abstract proof rules. Hoare logic, often identified with axiomatic semantics, was proposed 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 Language), (Leavens and al., 1998) which is a powerful DBC tool for Java. As such, it allows one to specify 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 parameters, 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 input 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 specifications 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.

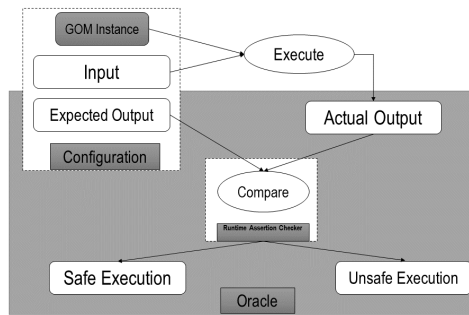


Figure 2: Decision Procedures for Oracles.

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 us-

ing 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_t* CM which will be the new data model of *s* and subsequently *s_merge_t* will represent the data of the megamodel.

```

1 context com.mega.example {
2   ...
3   def configuration model Merge_t.s :: Merge {
4     /* @ ASSIGN m1 as input IMPLIES m1 as s_datamodel->
5       revision
6     * @ ASSIGN m2 as input IMPLIES m2 as t_datamodel->
7       revision
8     * @ ASSIGN m as input IMPLIES m as s_match.t->revision
9     * @ INITIALLY (service, level , language,  otherProperties )
10      IN m1
11     * @ INITIALLY (service, level , language,  otherProperties )
12      IN m2
13     * @ RESULTS m3 as output IMPLIES m3 as s_merge.t->
14       revision
15     * @ REQUIRES (P, (m1.service EQUALS m2.service) AND
16     * (m1.level EQUALS m2.level) AND (m1.language EQUALS
17     * m2.language) AND
18     * (m as mapping) AND (m EQUALS m1 MATCH m2))
19     * @ ENSURES (Q, (m3.service EQUALS m1.service) AND
20     * (m3.level EQUALS m1.level) AND (m3.language EQUALS
21     * m1.language))*/
22     @input from: repository t_datamodel
23     @input from: repository s_datamodel
24     @output s_merge_t
25   }
26   megaprogram exampleMerge{
27     let UML2 t_m1 = null
28     let List <UML2> results = exec(interface::query(
29       realisation , with name = "IDataModel.s")
30     while( results != null ){
31       let any s_m1 = exec(model::queryLast(with
32         ReferenceModel = "UML2"
33         and RealizedInterface = "IDataModel.s"))
34       if ( exec(model::exists (with (name = "s_datamodel") and
35         Version = "3.X" ))){
36         t_m1 = exec(model::new( realisation , with
37           ReferenceModel = "UML2"
38           and RealizedInterface = "IDataModel.t" and Version
39           = "1"))
40       }
41     }
42     let configuration mergeInstance = exec( configuration ::
43       load(with name = "Merge_t.s"))
44     let UML2 merge_st = call mergeInstance(s_m1, t_m1)
45     merge_st = exec(model::save(with name = "s_merge.t"))
46   }
47   exampleMerge.execute()
48 }

```

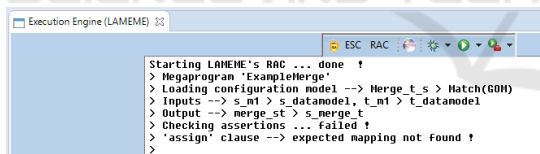
Listing 6: Illustrative example.

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 *t_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 *Merge.t_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 *Merge.t_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.



```

Execution Engine (LAMEME)
ESC RAC
Starting LAMEME's RAC ... done !
> Megaprogram 'ExampleMerge'
> Loading configuration model --> Merge.t_s > Match(GOM)
> Inputs --> s_m1 > s_datamodel, t_m1 > t_datamodel
> Output --> merge_st > s_merge.t
> Checking assertions ... Failed !
> 'assign' clause --> expected mapping not found !

```

Figure 3: An unsafe execution of the generic Merge instance : *Merge.t_s*.

To fix it, we have to redefine the configuration model *Merge.t_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 *Merge.t_s* was obtained as the result of the generic *Match* instance. In other words, the third parameter of the generic instance *Merge.t_s* should have to be a specific instance of the generic instance *Match.t_s*.

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

```

1  def configuration model Merge.t_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  }

```

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

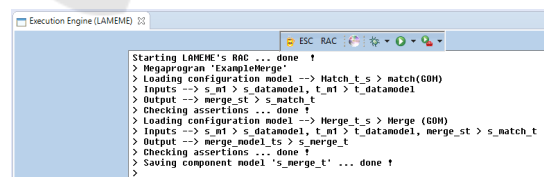
In Listing 8, the *exampleMerge* megaprogram is re-executed but using the generic instance *Merge.t_s* with three parameters. Indeed the *Merge.t_s* is redefined by adding a third parameter representing a mapping of the two input models which uncovers how the two source models "correspond" to each other. The *new* Merge instance *Merge.t_s* does not produce no longer any inconsistency in the megamodel, so such an execution is safe for the megamodel as illustrated in Figure. 4.

```

1  megaprogram exampleMerge{
2      ...
3      let configuration matchInstance = exec( configuration ::
4          load(with name = "Match.t_s"))
5      let any merge_st = call matchInstance (s_m1, t_m1)
6      let configuration mergeInstance = exec( configuration ::
7          load(with name = "Merge.t_s"))
8      let UML2 merge_model_ts = call mergeInstance(s_m1, t_m1
9          , merge_st)
10     merge_model_ts = exec(model::save(with name = "s_merge.t
11         "))

```

Listing 8: The sound *exampleMerge* megaprogram.



```

Execution Engine (LAMEME)
ESC RAC
Starting LAMEME's RAC ... done !
> Megaprogram 'ExampleMerge'
> Loading configuration model --> Match.t_s > Match(GOM)
> Inputs --> s_m1 > s_datamodel, t_m1 > t_datamodel
> Output --> merge_st > s_match_t
> Checking assertions ... done !
> Loading configuration model --> Merge.t_s > Merge(GOM)
> Inputs --> s_m1 > s_datamodel, t_m1 > t_datamodel, merge_st > s_match_t
> Output --> merge_model_ts > s_merge.t
> Checking assertions ... done !
> Saving component model 's_merge.t' ... done !

```

Figure 4: A safe execution of the Merge instance *Merge.t_s*.

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 (Benellallam 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 Mathématiques, Informatique et TIC (CEA-MITIC).

REFERENCES

Barbero and al. (2008). Model driven management of complex systems: Implementing the macroscope's vision. In *Engineering of Computer Based Systems*. IEEE.

Benellallam, A. and al. (2014). Neo4emf, a scalable persistence layer for emf models. In *European Conference on Modelling Foundations and Applications*. Springer.

Bezivin, J. and al. (2004). On the need for megamodels. In *19th Conference on Object Oriented Programming, Systems, Languages, and Applications*. ACM.

Bezivin, J. and al. (2005a). An introduction to the atlas model management architecture. In *Rapport de recherche 5*.

Bezivin, J. and al. (2005b). Modeling in the large and modeling in the small. In *European MDA Workshops: Foundations and Applications (MDAFA'05)*. Springer.

Boehm, B. W. and Scherlis., W. L. (1992). Megaprogramming. In *DARPA Software Technology Conference*.

Bruneliere, H. and al. (2014). Modisco: A model driven reverse engineering framework. In *Information and Software Technology*.

Cheon, Y. and Leavens., G. T. (2002). A runtime assertion checker for the java modeling language (jml).

DeRemer, F. and Kron., H. H. (1976). Programming-in-the-large versus programming-in-the-small. In *Transactions on Software Engineering*. IEEE.

Estublier, J. (2000). Software configuration management: a roadmap. In *Conference on the Future of Software Engineering*. ACM.

Favre, J.-M. (1997). Understanding-in-the-large. In *International Workshop on Program Comprehension*. IEEE.

Favre, J.-M. and NGuyen., T. (2005). Towards a megamodel to model software evolution through transformations. In *Electronic Notes in Theoretical Computer Science*.

Hoare, C. (1969). An axiomatic basis for computer programming. In *Communications of the ACM*. ACM.

Kling, W. and al. (2011). Moscript: A dsl for querying and manipulating model repositories. In *International Conference on Software Language Engineering*. Springer.

Leavens, G. T. and al. (1998). Jml: a java modeling language. In *OOPSLA*.

Meyer, B. (1992). Applying design by contract.

Murer, S. and Bonati., B. (2010). Managed evolution: a strategy for very large information systems. In *Science and Business Media*. Springer.

Rosnay, J. (1975). Le macroscope. vers une vision globale. In *Points Civilisation. Editions du Seuil*.

Schmidt, D. C. (2006). Model-driven engineering. In *Computer IEEE Society*. IEEE.

Seibel A., Neumann S., G. H. (2009). Dynamic hierarchical mega models: Comprehensive traceability and its efficient maintenance. In *Software and System Modeling*.

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

Toure, E. and al. (2018). Megamodel consistency management at runtime. In *Conference Nationale sur la Recherche en Informatique et ses Applications*. Springer.

Vignaga A., a. (2009). Typing in model management. In *Second International Conference on Model Transformation*. Springer.