Position Papers of the Federated Conference on Computer Science and Information Systems pp. 211–218
DOI: 10.15439/2016F585
ACSIS, Vol. 9. ISSN 2300-5963

Megamodel-based Management of Dynamic Tool Integration in Complex Software Systems

El Hadji Bassirou TOURE, Ibrahima FALL, Alassane BAH, Mamadou Samba CAMARA
Institut de Recherche pour le Développement (IRD),
École Supérieure Polytechnique (ESP),
Université Cheikh Anta Diop de Dakar (UCAD), Sénégal
Email: {bassirou.toure, ibrahima.fall}@esp.sn
{alassane.bah, mamadou.camara}@ucad.edu.sn

***Abstract*—The development of complex software systems is more and more based on the composition and integration of autonomous component systems. This can be done either statically (proactive approach) at development-time or dynamically through a reactive approach in which a new composite system can possibly be created on-demand and/or at run-time from existing systems. With the aim of constructing and managing such complex and reactive software systems, we propose a megamodel-based environment supporting dynamic tool integration. Such an environment must therefore be consistent at any time (i.e. before, during and after an integration) and should also have to exhibit some self-* properties (such self management, self-healing and self-configuration). In order to meet these challenges we propose the use of Hoare's Axiomatic Semantics and some inference rules to maintain the integrity of the megamodel and its components. For that we have defined a formal-safe execution as well as an execution semantic for each operation likely to modify the megamodel contents.**

***Index Terms*—Dynamic Tool Integration, Complex System, MDE, Megamodeling, Axiomatic Semantics, Verification.**

I. INTRODUCTION

NOWADAYS software systems, such *Component-Based Software Engineering (CBSE)*, are generally composite systems which are becoming more and more complex. These systems are usually built on top of a generic platform by plugging into it many different software components and tools.

Such platforms are often extended or specialized for a given domain by respectively adding or removing one or several software components. This can be done either statically in a *proactive approach* at development-time, or dynamically at run-time through a *reactive approach*. We will present in the following some advantages and drawbacks for both proactive and reactive integration approach. A comparative study of the two approaches are extensively developed in [10] to which we refer the reader for further information.

Proactive approach

A proactive integration is an approach in which application designers implement a new application manually by designing correspondence or composition rules describing the interaction patterns between its components. Such an approach is set up when the architecture is still under development, namely at design or development time.

Limits

It complicates the problem of ensuring consistency in the software systems and is severely limited in flexibility;

Advantages

It supports more powerful integration methods and ensures that the adaptation will not produce anomalous behavior. This is due the fact that a *proactive approach* try to identify the potential effect of making a change even before the change has actually been made. It might also prohibit certain changes that would otherwise lead to unexpected behavior [11].

Reactive approach

In a dynamic (or reactive) integration a new composite system can possibly be created on-demand and/or at runtime from existing systems, considering the user preferences and the context to personalize the system maintenance process.

Limits

It is difficult to use traditional testing and formal verification techniques to check safety and other correctness properties.

Advantages

Dynamic integration allows software components (module behavior) and their interactions to be changed while modules are executing. Such a reactive approach starts to work at the moment the actual changes are being made, and typically try to resolve potential inconsistencies interactively. Moreover it may also enable a number of useful applications that could not be envisioned during development time. Therefore, such a kind of integration is suitable for end-user applications where available components are dynamic and users needs may be varying frequently.

Abstraction plays an important role in component-based programming, it is taken into account through encapsulation which ensures information hiding and independence between components [20]. To provide such an abstraction while building complex software systems, Model-Driven Engineering (MDE) seems to be a preferential solution. In fact, it is a "*recent*" Software Engineering (SE) discipline which promotes *models* as first class entities in the software system development and maintenance. In the MDE field, a *model* is defined as an artifact which consists of model elements, conforms to a specific *metamodel* and represents a given view of a system. A metamodel can be defined as a language that describes the various kinds of contained model elements and the way they are arranged, related and constrained [1]. However a system is often represented by various kinds of *interrelated models*.

Moreover, the architecture of a software system defines such a system in terms of components and interactions between them. MDE provides the concept of a megamodel as a building block for modeling in the large [3], thus 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*, *and so on*. Indeed, megamodeling offers the possibility to handle models (and metamodels) as first-class entities, to specify relationships between them and to navigate among them. Furthermore, by keeping track of all the heterogeneous modeling artifacts (models, metamodels, DSL, etc.) within a megamodel, all of them are treating as models. This results in a homogeneous infrastructure which enables the management of complex modeling artifacts [2].

Our purpose is to use *megamodels* for representing the components of an architecture and the interactions between them. Thereby to develop an integration management approach based on operations defined in the megamodel. As already stated a *megamodel* refers to a model that have models as its elements and that captures the interconnections between multiple models(*component models*) in the form of model operations, generally represented as model transformations (*global operation models*) [1]. All of these *models* will then be represented in a single runtime megamodel which will be handled as an execution environment or more simply as a (mega-)program which is updated with each new *(global) operation execution* [6]. The megamodel is therefore subjected to frequent dynamic changes which consist of either *adding* or *removing components*. However such changes must not violate the integrity of the megamodel and its constituent components. It is therefore necessary for one to be able to add or remove components while maintaining some kind of integrity of the entire system represented by a megamodel.

In order to meet these challenges we consider a megamodel as a program and use techniques for proving program correctness. These techniques are known as Hoare's axiomatic semantics and are used with some inference rules for checking the megamodel's consistency by defining, for each *global operation* likely to modify the megamodel contents, a formal and safe execution as well as an execution semantic which denotes the observable behavior of a program as it is executed.

The rest of the paper is organized as follows. Section II presents the problem statements. Section III is reserved for related works in which we present some papers that use megamodeling techniques. Section IV presents our approach of megamodel management of dynamic tool integration. Section V is reserved for the use of axiomatic semantics in order to provide a mean to check the megamodel's consistency. Section VI presents an example in which we illustrate the presented approach. Section VII concludes the paper and gives its future works.

II. Problem Statements

In the MDE vision, software development and management processes involve the creation and use of many related modeling artifacts which are becoming increasingly important. As a consequence, there is the need for efficient mechanisms to manage this constantly growing number of models which is due to many reasons, as those that follow [21]:

- Each viewpoint of the software is represented using a model with respect to the most adapted formalism (metamodel).
- Complex models need to be decomposed into smaller ones with different levels of abstraction.
- On behalf of the "*separation of concerns*", different models are created for different purposes.

Many aspects of model-management have been considered in the literature in which the concept of megamodels ([1], [5], [6], [15], etc.) and macromodels [21] have been proposed. However all of them, except in [15], focus on the management of *development models* whereas it could be very interesting to extend the use of models such as macromodels and megamodels for the management of *runtime artifacts* through the use of *runtime models*. Such runtime artifacts may include *component creation and destruction*, *exceptions/errors*, *operation inputs and output*, *components invocation operations*, *dynamic artifact types*, *dynamic component names*, and *so on*.

Otherwise, causal connection between models and represented systems means that each time one read the model, he gets the information representing the current system state, and similarly, each time one writes the model, the information he writes makes the proper system change [16]. Using models at runtime for specifying runtime artifacts have two main requirements. 1. the model as interrogated should provide up-to-date and exact information about the system. 2. if the model is causally connected, then adaptations can be made at the model level rather than at the system level.

The importance of the use of models at runtime has been extensively discussed in [20]. In that paper, several problems for which runtime models could be useful have been proposed without giving any piste about how to implement a solution. Among those problems, one can cite the support of semantic integration of software components represented through runtime models. For example, suppose that we have a user who expresses a request in order to merge two models into one. To achieve this, most of the model-based approaches such in [12], [13] and [14], use development models by putting the system offline, at first. Then they specify a set of correspondence or composition rules generally using the respective metamodels of the two input models. Finally they restart the system to apply changes in the system. Another solution which does not necessitate of putting down the system consists of describing directly the merging operation by reasoning on runtime models. In this case, when implementing such an approach, one has to recognize that the solution will never stay constant. That is, it could happen a situation where new models are introduced in the running system, and perhaps, some models to be removed from it. This continuous change necessitates the modeler to focus on how the system will react to those frequent changes and therefore how it will evolve

over time. An important challenge here is how to realize such a solution while maintaining some kind of integrity of the entire system. To achieve this, the approach should therefore implement a reactive environment to face changes in the running system. In order to satisfy users requests for example, such an environment should have to exhibit some self-* properties as self-management, self-healing and self-configuration.

The presented model-based approaches ([12], [13] and [14]) solve the problem of integration of software components by using the first solution through the use of development models, therefore such approaches can be considered as proactive integration. But we found no approach carrying out this problem using runtime models.

III. Related Works

In this section, some approaches of model management are presented. Such approaches use some specific models such as megamodels or macromodels to manage changes in a complex modeling environment. Megamodels and macromodels based-approaches provide a framework for an efficient creation, storage, access and execution of large amounts of modeling artifacts and their interconnections [5]. Indeed each of these approaches provides a mechanism to represent and control changes that occur in the megamodel through various techniques.

In [1], Bezivin and al. are experimenting through their *AMMP* (*Atlas Model Management Platform*) environment the need to consider separately the activities of modeling in the small and modeling in the large. A *megamodel* is considered as *a kind of registry that can be seen as a model which elements are models or refer to models*. Thus a megamodel will help the *AMMP* platform to know the available tools or services. Authors use megamodels (which provide a global view on models) for the support of model-driven software development by using it for model management. Megamodels are also applied to facilitate traceability between models and their elements.

In [4], authors present *MoScript* which is a megamodel agnostic platform and a textual DSL for accessing, querying and manipulating modeling artifacts represented in a megamodel. Several modeling tasks are performed using different kinds of operations which involve *operations without side effects* and *operations with side effects*. *Operations without side effects* (*QueryOp*, *TransformOp*, *ProjectOp*, *StateCheckOp*) are those that do not modify the megamodel contents. *Operations with side effects* (*SaveOp*, *RemoveOp*, *RegisterOp*) are operations that are likely to modify the megamodel contents. MoScript also allows to write queries that retrieve models from a repository or that register newly produced models back to the repository.

In [21], a *macromodel* is defined as *a model consisting of elements denoting models and links denoting intended relationships between these models with their internal details abstracted away*. For an efficient management of models, the author considers different kinds of modeling artifacts (*models and relationships*) which act on different layers (*orders of hierarchy*. Applications of the use of these different models could be twice : the consistency checking between constituent models and the inference of relations from other relations.

In [6], the author considers a *megamodel* as *a program in which the declaration and definition of models within a megamodel as statements of a model-based programming language*. Then the execution of a simple program composed of a sequence of such statements manipulates the contents of a megamodel. The important contribution of handling a megamodel as being a program is that it enables the prevention of typing errors during the execution of such programs. An typing error is for example the application of a function on arguments for which it was not defined.

In [15] authors propose the use of megamodels for the management of models at runtime. A runtime model provides a viewpoint, on a running software system, that is usually used for managing the system. Authors present a set of models, which have to be managed at runtime, as *Reflection models*, *Implementation models*, *Evaluation models*, *Change models*, *Monitoring models* and *Execution models*. In order to manage these models authors also propose a set of operations such *Update*, *Effects*, *Check for failures*, *Failure analysis rules*, *Repair strategies*. They propose the use of megamodels at runtime for both *navigation* and *automation*.

IV. Our approach : Use of MDE techniques for dynamic tool integration

Our motivation here is to use MDE techniques, particularly runtime models, in order to set up a megamodel-based environment supporting dynamic tool integration in a complex software system. In fact, our management of changes must ensure the consistency, correctness, and any desired properties of runtime change in the megamodel. In other words, our proposition for supporting dynamic integration should :

- Use a *megamodel* as a basis for model changes and management and which represents the dynamic architectural model of the global system;
- Provide inference rules for reasoning about and preventing changes from violating the *megamodel*'s integrity ;
- Help identify what models are likely to be targeted/modified, for example, by the insertion of a new model into the running system.

In this section we present at first the problem of tool integration, then we show how a tool is represented through various kinds of models and their interconnections in the megamodel. Finally we present our megamodel management and its constituent components.

A. Tool integration

A problem for tool integration is the general lack of standards for representing tools and their relationships. This has slowed the creation of fully integrated environments. In fact, one important barrier in current software engineering environments is the difficulty of integrating tools that address different aspects of the development process. As mentioned

in [8], integration is not a property of a single tool, but of its relationships with other elements in the environment. The key notion is the relationships between tools but also the properties of these relationships. These relationships can be represented through dimensions or types; a very consistent work on this purpose was attributed to Wassermann in [7] whose dimensions of tool integration are quoted by almost the totality of the other authors. The *platform dimension* for tool integration is concerned with the framework services that are commonly used by tools. *Presentation integration* deals with user interaction and *data integration* with data interchange. *Control integration* is concerned with interoperability between tools, and finally *process integration* refers to the different roles played by tools during a whole software process.

B. Tools as integrated models

In component-based programing a software system is conceived as a collection of several tools. Each tool contains some components which could encapsulate some services and provides or requires some operations from other tools. The components are always reached through the operations of the tool. They are said to be encapsulated in the tool. For each tool the problem is how to offer services, each one representing an integration dimension, to other tools and if there is the need how to use their services, in order to achieve a common goal. For that, it is necessary that each time a tool is added into the software system, it has to be registered as its supported services. Such a situation supposes to have a common view of what are a tool, its implemented services, operations it offers or requires etc. With respect to these requirements, we assume, as in [1], that *tools* will be handled as *models*.

Each tool consists of component models which could encapsulate some services. A service may correspond to any element of the Wasserman's integration dimension [7]. It can then be *data*, a *function*, a *process element*, *platform element* or *presentation element*. A tool may also provide or require some operations from other tools. Indeed, given two different tools, the interactions between two corresponding models come in the form of model operations (*global operation models*). A *global operation model* for example may also have input and output *parameters*, each *parameter* being itself considered as a *component model*. Thus a megamodel consists of *component models* and *global operation models*. A *component model* encapsulates artifacts which represent *services* of a tool. A *global operation model* can be seen as a type of an operation between component models. It therefore represents a model of future interactions (*global operation instances*) that connect some *component instances*. In other words, a *global operation model* defines one or more interaction rules, can be instantiated on *component instances* and allows to dynamically establish links between components. A *global operation model* can only be applied to component models already contained in the megamodel, and its results are *new component models* which are automatically added in the megamodel.

Manipulating a megamodel is then like programming where the megamodel acts as an environment, or more simply as a (mega-) program. It can be updated with each new *global operation execution*. However considering a megamodel as a program has already been proposed in [6]. In such contexts, the megamodel is subjected to frequent and dynamic changes which can due to the execution of operations (instructions) on components such the *introduction of new components*; the *recreation of failed components*; the *modifications of component interconnections*; the*change component operating parameters*, etc. A *global operation models* corresponds to a *set of operations* which are executed in response to changes related to the underlying system state changes. Indeed, after each execution of a global operation model, the megamodel has to be updated. Then, each component has the possibility/responsibility to update the megamodel through the execution of *global operation instances*. However the corresponding changes have not to violate the integrity of the megamodel which have to stay consistent.

C. The megamodel management

The first step in creating any composite service is to locate the service components (or tools) that provide the functionality that is to be placed in the new service [19]. To facilitate this process in our approach, all *component models* must be stored in a component directory, namely the megamodel, which can be accessed and managed at runtime. Each *component model* should be named and typed, and has also to specify what information (services) it represents in its corresponding integration dimension [7] represented by the model. *Component models* should also provide a description of its provided *operations* (through *global operation models*), its required *operations* (from other components), as well as the *input* and *output* services of all of these operations. Figure 1 represents the metamodel of the megamodel with respect to such considerations. Hereafter we present its main concepts.

- ***Model*** : Specifies that an entity is a model;
- ***ModelLevel*** : Allows knowing if it is a terminal model or a reference model ;
- ***ModelType*** : Allows giving the type of a model, to know its metamodel ;
- ***ModelDimension*** : Allows knowing the information represented in the model (data, process, ...);
- ***GlobalOperation*** : Supports operation on component models as :
  - ***Load*** : Registers a *component model* in the *Megamodel* if it is not already registered.
  - ***Extract*** : Allows the extraction of *component models* from the *Megamodel*.
  - ***Refine***: Allows to represent the refinement operation, i.e. transforming a model in a given dimension into another ;
  - ***Change2Ref***: Transforms a model in a given metamodel to another.
  - ***Match***: Takes as input two models of the same kind and performs especially well for detecting the differences between two versions of a *component model*

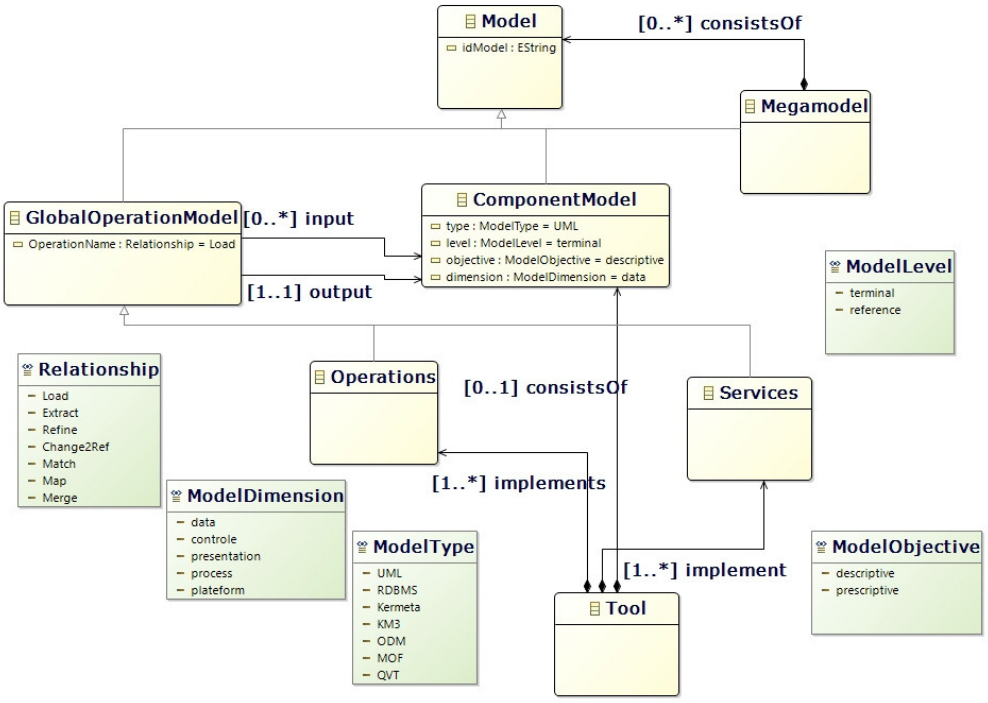

Fig. 1. Megamodel's Metamodel

- ***Map*** : Uncovers how two models "*correspond*" to each other. It takes two models as input and returns a morphism (which will be considered as a model) between them.
- ***Merge***: Merges two models into one based on a mapping (*Map*) between them.

V. Using Hoare's Axiomatics for Proving Megamodel's Consistency

A. Hoare Triples To Define Proof System of Axiomatic Semantics

In Hoare logic a program is considered a transformer of states. A state here represents the values of all the variables of a program. The execution of a program has the effect, if it ends, to transform an *initial state* to a *final state*. The specification of a program will be developing properties on these two states. *Axiomatic semantics* provides a style appropriate to the proof of programs. Thus, to prove a program, it just needed to specify the means of assertions written using *logical formulas* and then establish that the program meets its specification. That is to say that the specification and the program comply with the rules of an axiomatic semantics defining the valid executions of each statement in the source language. The technique used to prove an annotated program by assertions reduces it into a set of *logical formulas* called *verification conditions*, no longer refer to the instructions of the program. Prove a program thus reduces to *checking the validity* of logical formulas [9]. Properties of axiomatic semantics are expressed in general as expressions of Hoare logic: *{p}S{q}* where *p* and *q* are properties expressed in predicate logic, *p* supposed to be verified by memory before execution of the program *S*, and *q* to be checked after execution of *S* on the same machine that have checked *p*. Starting from the fact that one have to define a set of assertions that define as *pre-conditions* and *post-condition* rules of equilibrium of the system then the formalism of the Hoare logic is the most appropriate to represent the development of the possible states of the program, but also to make easier correction system. Considering the triple "*(x+1 > 0){ x := x + 1 }(x > 0)*". Such assertion means : *if " x+1>0"* is true before the execution of *" x := x+1 "*, *then* after his execution, the condition *" x>0"* becomes true. Or for the condition *" x>0"* is true after execution of *" x:=x+1"*, It is necessary that the condition *"x+1 > 0"* is verified, before performing the allocation.

B. Hoare's axiomatic semantics for proving megamodels consistency

As already said, the megamodel acts as an environment which is updated with each new *global operation execution*. A *global operation model* consists of a set of *pre-conditions* P, a *sequence of operations* Seq , and a set of *post-conditions* Q.

$$\{P\}Seq\{Q\}$$

The *pre-conditions* define the set of states of the system from which the *sequence of operations* can be invoked. Each *operation* is to be invoked in the order that it appears in the sequence, with the specified actual parameters and has possibly a *side effect* which corresponds to the effect of that operation

on other components. Indeed, each operation targets by default one or more *component models* or other *global operation models* in the megamodel. The *post-conditions* define the set of states that satisfy the desired result after executing the sequence of operations. The state of the megamodel is modeled by a set of assertions. And hence, for each *global operation models* to maintain this balance, it is necessary to know how its statements and flow effect affect correctness of models based on techniques available for proving programs consistency. To achieve, this we have defined as in our previous works [9], an *execution semantics* for each *global operation* in the megamodel. Execution semantics of a global operation in the megamodel denote its observable behavior (i.e. its effect on the state of the megamodel and components) as it is executed.

With this intention, it is enough that all the Hoare's triplets are valid and for this reason, it is necessary to formally define an execution semantic for each instance *Seq* of a *global operation model*.

VI. Example : Dynamic Tool Integration

A. Defining semantic execution for global operations

Suppose that we have a megamodel $\mathbf{M}$ which represents the overall structure of a software system $\mathbf{S}$ in terms of its constituent components (*component models*) and their interconnections (*global operation models*). Given that we have a new tool $\mathbf{T}$ to plug in the system $\mathbf{S}$. The tool $\mathbf{T}$ is then represented through multiple models, each one representing an integration dimension. $\mathbf{T}$ is said to be well integrated in $\mathbf{S}$ if for each dimension, the model representing a given dimension is well integrated with the model of $\mathbf{S}$ representing that dimension. To integrate two models we use the global operator called *Merge*. Our goal is not to say how to integrate two models (for that see [18]) but how the environment must react to the integration of new tools.

The algorithm (*see algorithm 1*) only gives the framework for implementing an environment supporting dynamic integration: it does not show how to implement an integration. When implementing an integration approach one has to recognize that the solution will never stay constant: new tools will be added, and, perhaps tools will be removed. This continuous change necessitates that the designer places emphasis on how the megamodel will evolve over time. Hoare's axiomatic semantics allow us to fix this problem by proposing formal safe and semantics execution for all the global operation instances in the megamodel. The execution semantics of a *global operation* is the effect of that operation on the state of the megamodel and will be defined by a rule.

In order to show how we use the axiomatic semantics for checking the megamodel's integrity, we consider an execution of the above algorithm. We assume that we have a system $\mathbf{S}$ which will be extended by plugging into it a tool $\mathbf{T}$. Given that the treatment is the same for all of the dimensions, we will consider in this example that $\mathbf{T}$ will be integrated to $\mathbf{S}$ according to the *data dimension*.

- $\mathbf{M}$ is the megamodel representing the system $\mathbf{S}$.

Algorithm 1 : Integrating new tool in the megamodel

1.**Input** : Software System $\mathbf{S}$, Tool $\mathbf{T}$, Megamodel $\mathbf{M}$
2.**Output**:$\mathbf{M}$ representing $\mathbf{S}$ in which $\mathbf{T}$ has been plugged
3.**BEGIN**
4. **For** *each dimension* d_i
5. **Begin**
6. **Let** ms_i *representing the dimension* d_i *of S*
7. $ms_i \leftarrow$ **Extract** (S, d_i)
. /* *Extract the model representing the dimension* d_i *in* $\mathbf{S}$ */
8. **Load** (M, ms_i)
. /* *Load the model representing the dimension* d_i *in* $\mathbf{M}$ */
9. **Let** mt_i *representing the dimension* d_i *of T*
10. $mt_i \leftarrow$ **Extract** (T, d_i)
. /* *Extract the model representing the dimension* d_i *in* $\mathbf{T}$ */
11. **Load** (M, mt_i)
. /* *Extract the model representing the dimension* d_i *in* $\mathbf{M}$ */
11. **If** (mt_i **AND** ms_i *have different metamodels*) **Then**
12. $mt_i \leftarrow$ **Change2Ref**(ms_i)
. /* *Transform a model in a given metamodel to another* */
13. **Let** map_i *be a morphism model defined as follows*
14. $map_i \leftarrow$ **Map**(ms_i, mt_i);
15. **Load** (M, map_i);
16. $ms_i \leftarrow$ **Merge**(ms_i, map_i, mt_i);
17. **Load** (M, ms_i);
18. **End**
19. **Return** M
20.**END.**

- ms_d is the model representing the data dimension of the system $\mathbf{S}$.
- mt_d is the model representing the data dimension of the system $\mathbf{T}$.
- map_d is the morphism (model) representing how ms_d and mt_d "*correspond*" to each other;

We will define an execution semantic for the *global operation* **Merge** enabling to integrate the tool T with the software S represented by the megamodel M according to their data dimension:

Operation : Merge
$merge_d \longleftarrow$ *Merge* (ms_d, map_d, mt_d)
Pre-conditions

$$P \Leftrightarrow \bigcap \begin{cases} P_1 & ::= & ms_d \in M \\ P_2 & ::= & level(ms_d, \text{"}\textit{terminal}\text{"}) \\ P_3 & ::= & type(ms_d, \text{"}\textit{UML}\text{"}) \\ P_4 & ::= & dimension(ms_d, \text{"}\textit{data}\text{"}) \\ P_5 & ::= & mt_d \in M \\ P_6 & ::= & level(mt_d, \text{"}\textit{terminal}\text{"}) \\ P_7 & ::= & type(mt_d, \text{"}\textit{UML}\text{"}) \\ P_8 & ::= & dimension(mt_d, \text{"}\textit{data}\text{"}) \\ P_9 & ::= & map_d \in M \\ P_{10} & ::= & merge_d \notin M \end{cases}$$

Post-conditions

$$Q \Leftrightarrow \bigcap \begin{cases} Q_1 & ::= & ms_d \in M \\ Q_2 & ::= & level(ms_d, \text{"}\textit{terminal}\text{"}) \\ Q_3 & ::= & type(ms_d, \text{"}\textit{UML}\text{"}) \\ Q_4 & ::= & dimension(ms_d, \text{"}\textit{data}\text{"}) \\ Q_5 & ::= & mt_d \in M \\ Q_6 & ::= & level(mt_d, \text{"}\textit{terminal}\text{"}) \\ Q_7 & ::= & type(mt_d, \text{"}\textit{UML}\text{"}) \\ Q_8 & ::= & dimension(mt_d, \text{"}\textit{data}\text{"}) \\ Q_9 & ::= & map_d \in M \\ Q_{10} & ::= & merge_d \in M \end{cases}$$

We have the triplet:

$$\{\text{P}\}\ \text{Merge}\ (ms_d,\ map_d,\ mt_d)\ \{\text{Q}\}$$

In the same manner, we define an execution semantics for each *global operation* defined in the megamodel. However, carrying out a *global operation* may require the execution of other *global operations* for standardizing the input models (ripple effect). This means that the output of a *global operation* may correspond to the inputs of another.

B. Inference rules for safe executions of global operations

The operation *Merge* takes as inputs two models representing the same dimension, in the contrary case, it would be necessary to call the *global operation model Refine*. Then if the two input models have not the same metamodel then we can also use the global operation *Change2Ref* enabling to transform a model specified in a given metamodel to another metamodel. Before applying the *Merge* operation, we have to earlier call the global operation *Map* with the two input models. Indeed the third parameter (*map*) to *Merge* is a morphism that describes elements of ms_d and mt_d that are equivalent and should be "*merged*" into a single element map_d in M. Once that all the models as inputs were described in the sound formalism, then one can apply the operation of merging. One thus realizes that it could exist a logical precedence between two or more *global operations* defined in the megamodel. In order to take into account these considerations we have to set up, in addition to axioms, a *deductive system* which permits the deduction of new theorem from one or more axioms or theorems already proved. A *rule of inference* takes the form "*If* $\vdash$ *X and* $\vdash$ *Y then* $\vdash$ *Z*", i.e. if assertions of the form *X* and *Y* have been proved as theorems, then *Z* also is thereby proved as a theorem.

For that we will use two rules of inference presented in [17], namely the *rule of consequence* and the *rule of composition*. After that, we present an example in which these two rules are applied.

(i) Rules of consequence :

If $\vdash$ *{P}S{R} and* $\vdash$ *R* $\supset$ *Q then* $\vdash$ *{P}S{Q}*
If $\vdash$ *{P}S{R} and* $\vdash$ *P* $\subset$ *Q then* $\vdash$ *{Q}S{R}*

These rules state that if the execution of a global operation ensures the truth of the assertion Q, then it also ensures the truth of every assertion logically implied by Q.

(ii) Rule of composition :

If $\vdash$ *{P}*S_1{Q_1} *and* $\vdash$ {Q_1}S_2{R} *then* $\vdash$ *{P}*(S_1; S_2){R}

This rule states that if the proven result of the first part of a *global operation* is identical to the pre-condition under which the second *global operation* produces its intended result, then both *global operations* will produce the intended result, provided that the pre-condition of the first *global operation* is satisfied. We will use the notation :

$$\frac{\vdash \{P\}S_1\{Q_1\} \ \vdash \{Q_1\}S_2\{R\}}{\vdash \{P\}(S_1; S_2)\{R\}}\ \textit{global operation}$$

We can then define the inference rule for the *Merge global operation* which enables us to carry out the integration of two models.

Considering the previous example. We have to describe an execution of *Merge* between the two models : ms_d and mt_d. However before applying the *Merge* Operation, it must be necessary to invoke at first the Operations : *Change2Ref* and *Map*.

Operation : *Change2Ref*
$mt_d \longleftarrow$ Change2Ref(ms_d)
Pre-conditions

$$P \Leftrightarrow \bigcap \begin{cases} P_1 & ::= & mt_d \in M \\ P_2 & ::= & ms_d \in M \\ P_3 & ::= & type(ms_d)! = type(mt_d) \\ P_4 & ::= & dimension(ms_d) == dimension(mt_d) \end{cases}$$

Post-conditions

$$Q \Leftrightarrow \bigcap \begin{cases} Q_1 & ::= & mt_d \in M \\ Q_2 & ::= & ms_d \in M \\ Q_3 & ::= & type(ms_d) == type(mt_d) \\ Q_4 & ::= & dimension(ms_d) == dimension(mt_d) \end{cases}$$

We have : $A_1 \longleftarrow$ *{P}Change2Ref{Q} (i)*

Operation : *Map*
$map_d \longleftarrow$ Map(ms_d, mt_d)
Pre-conditions

$$M \Leftrightarrow \bigcap \begin{cases} M_1 & ::= & mt_d \in M \\ M_2 & ::= & ms_d \in M \\ M_3 & ::= & map_d \notin M \\ M_4 & ::= & type(ms_d) == type(mt_d) \\ M_5 & ::= & dimension(ms_d) == dimension(mt_d) \end{cases}$$

Post-conditions

$$N \Leftrightarrow \bigcap \begin{cases} N_1 & ::= & mt_d \in M \\ N_2 & ::= & ms_d \in M \\ N_3 & ::= & map_d \in M \\ N_4 & ::= & type(ms_d) == type(mt_d) \\ N_5 & ::= & dimension(ms_d) == dimension(mt_d) \end{cases}$$

We have : $A_2 \longleftarrow$ *{M}Map{N} (ii)*

Using the **rules of consequence** on (i) and (ii) we obtain :

$\vdash$ *{P}Change2Ref{R} and* $\vdash$ *R* $\supset$ *M then* $\vdash$ *{P}MAP{N}*

We have : $A_3 \longleftarrow$ *{P}Map{N} (iii)*

Operation : *Merge*
$merge_d \longleftarrow$ Merge(ms_d, map_d, mt_d)
Pre-conditions

$$I \Leftrightarrow \bigcap \begin{cases} I_1 & ::= & mt_d \in M \\ I_2 & ::= & ms_d \in M \\ I_3 & ::= & map_d \in M \\ I_4 & ::= & merge_d \notin M \\ I_5 & ::= & type(ms_d) == type(mt_d) \\ I_6 & ::= & dimension(ms_d) == dimension(mt_d) \end{cases}$$

Post-conditions

$$R \Leftrightarrow \bigcap \begin{cases} R_1 & ::= & mt_d \in M \\ R_2 & ::= & ms_d \in M \\ R_3 & ::= & map_d \in M \\ R_4 & ::= & merge_d \in M \\ R_5 & ::= & type(ms_d) == type(mt_d) \\ R_6 & ::= & dimension(ms_d) == dimension(mt_d) \end{cases}$$

We have : $A_4 \longleftarrow \{I\}Merge\{R\}$ *(iv)*

Using the **rules of consequence** on (iii) and (iv) we obtain :

$\vdash$ *{P}Map{N} and* $\vdash I \supset N$ *then* $\vdash$ *{P}Merge{R}*

We have : $A_5 \longleftarrow$ *{P}Merge{R} (v)*

More formally, using the **rules of composition** we obtain :

$$\frac{\vdash\{P\}Change2Ref\{Q\} \vdash\{M\}Map\{N\} \vdash\{I\}Merge\{R\}}{\vdash\{P\}Merge\{R\}} \textit{Merge}$$

VII. Conclusion And Future Works

In this paper, we have presented a model-based environment of dynamic tool integration based on a MDE vision. We use a megamodel in order to manage dynamic changes in the software architecture. For that we have considered the megamodel as being a program and accordingly we have proposed an approach (namely Hoare's axiomatic semantics) already used for proving program's correctness which enables to keep the megamodel consistent.

As we look at future works, we will certainly be looking to set up Domain-Specific Modeling Language (DSML) which enables us to automatize the management of the megamodels and whose instructions will be the global operation instances. Therefore, each instruction, which consists of global operation execution, is likely to modify the megamodel structure and semantics. That is why such a DSML should also integrate an inference engine in order to provide a mean to reason about the elements of the megamodels. This is the inference engine that will be used to check the megamodels integrity and its elements through the validation of Hoare's triplets.

Acknowledgment

The research in this paper is supported by the *Centre d'excellence African en Mathématiques, Informatique et TIC (CEA-MITIC)*. We gratefully acknowledge the CEA-MITIC for financial support in that paper.

References

[1] Bezivin, J., Jouault, F., And Valduriez, P. *On the need for megamodels.* In Proceedings of the OOPSLA/GPCE: Best Practices for Model-Driven Software Development workshop, 19th Annual ACM Conference on Object Oriented Programming, Systems, Languages, and Applications.(2004, October).

[2] Iovino, L., Pierantonio, A., And Malavolta, I. *On the Impact Significance of Metamodel Evolution in MDE.* Journal of Object Technology, 11(3), 3-1.(2012).

[3] Bezivin ,J., Jouault, F., Rosenthal, P. And P. Valduriez. *Modeling in the Large and Modeling in the Small.*, In Proc. of European MDA Workshops: Foundations and Applications (MDAFA'05). LNCS 3599/2005, pp. 33-46. Springer, 2005.

[4] Kling, W., Jouault, F., Wagelaar, D., Brambilla, M., And Cabot, J. *MoScript: A DSL for querying and manipulating model repositories.* In Software Language Engineering (pp. 180-200). Springer Berlin Heidelberg.(2011).

[5] Seibel A., Neumann S., Giese H. *Dynamic Hierarchical Mega Models: Comprehensive Traceability and its Efficient Maintenance.* , Software and System Modeling 9(4):493-528, 2009.

[6] Vignaga A. , Jouault F., Bastarrica M. C., Bruneliere H. *Typing in Model Management.* , In R. F. Paige, editor, Second International Conference on Model Transformation. Theory and Practice of Model Transformations, volume 5563 of Lecture Notes in Computer Science, pages 197-212. Springer, 2009.

[7] Wasserman, A. I. *Tool integration in software engineering environments.* In Software Engineering Environments (pp. 137-149). Springer Berlin Heidelberg.(1990).

[8] Thomas, I and Nejmeh, B. A. *Definitions of tool integration for environments.* , IEEE Software, 9(2):29-35, March 1992.

[9] Bousso, M., Sall, O., Thiam, M., Lo, M., Toure, E. H. B. *Ontology Change Estimation Based on Axiomatic Semantic and Entropy Measure.* In Signal Image Technology and Internet Based Systems (SITIS), 2012 Eighth International Conference on (pp. 458-465). (2012, November). IEEE.

[10] Fujii, K., and Suda, T. *Dynamic service composition using semantic information.* In Proceedings of the 2nd international conference on Service oriented computing (pp. 39-48). ACM.(2004, November).

[11] Mens, T., Mens, K., And Wuyts, R. *On the use of declarative meta programming for managing architectural software evolution.* In Proceedings of the ECOOP' 2000 Workshop on Object-Oriented Architectural Evolution, (2000, June).

[12] P. Atzeni, P. Cappellari, P. Bernstein *A multilevel dictionary for model management* , Int. Conf. on Conceptual Modeling (ER), Klagenfurt, Nov. 2005.

[13] T. Reiter, E. Kapsammer, W. Retschitzegger, W. Schwinger *Model Integration Through Mega Operations*, Workshop on Model-driven Web Engineering (MDWE), Sydney, July 2005

[14] G. Straw et al. *Model Composition Directives*, 7th UML Conference, Lisbon, 2004.

[15] Vogel, T., Giese, H. *A language for feedback loops in self-adaptive systems: Executable runtime megamodels.* (2012, June). In Proceedings of the 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (pp. 129-138). IEEE Press.

[16] Song, H., Huang, G., Chauvel, F., Sun, Y.: *Applying MDE Tools at Runtime: Experiments upon Runtime Models.* In: Models@run.time 10. CEUR-WS.org, vol. 641, pp. 25-36 (2010)

[17] Hoare, C.A.R., *An Axiomatic Basis for Computer Programming.* Communications of the ACM 12, 10, 576-583, October 1969.

[18] Pottinger, R. A., Bernstein, P. A. *Merging models based on given correspondences.* In Proceedings of the 29th international conference on Very large data bases-Volume 29 (pp. 862-873). VLDB Endowment (2003, September).

[19] Mennie, D., Pagurek, B. *An architecture to support dynamic composition of service components.* Systems and Computer Engineering. Carleton University, Canada. (2000)

[20] Blair, G., Bencomo, N., France, R. B. *Models@ run. time.* Computer, 42(10), 22-27.(2009).

[21] Salay, R., Mylopoulos, J., Easterbrook, S. *Using macromodels to manage collections of related models.* In Advanced Information Systems Engineering (pp. 141-155). Springer Berlin Heidelberg. (2009, June).