On the Use of Graph Transformations for Model Composition Traceability

Y. Laghouaouta*, A. Anwar†, M. Nassar*, J.-M. Bruel‡
*IMS-SIME, ENSIAS, Mohamed Vth Souissi, Rabat, Morocco
y.laghouaouta@um5s.net.ma,nassar@ensias.ma
†Siweb, EMI, Mohamed Vth Agdal University, Rabat, Morocco
anwar@emi.ac.ma
‡IRIT/Université de Toulouse, France
bruel@irit.fr

Abstract—**The model composition provides support to build systems based on a set of less complex sub-models. This operation allows managing complexity while supporting the modularity and reusability tasks. Due to the increase number of the involving models, their composition becomes a tedious task. For that, the need for maintaining traceability information is raised to help managing the composition operation. We propose in this work a graph-based model transformations approach, which aims to keep track of the model composition operation. Our objective is to capture traces in an automatic and reusable manner. Finally, a composition scenario is given to demonstrate the feasibility of our proposal.**

***Keywords*—model traceability, model composition, Aspect-Oriented Modeling, graph transformations.**

I. Introduction

Large and heterogeneous systems are too complex to be described using a single model. Actually, the model composition provides support to build systems based on different models. Each one refers to a specific concern, perspective, point of view or component. Such modularization allows managing the system complexity by reasoning about less complex sub-models.

Despite of the model composition benefits (validation of involving models, models synchronization); this operation is a laborious and error prone activity. A traceability mechanism handles this task. It helps designer to comprehend the exact effects of the composition and reveals the interactions among involving models. Besides, such information provides means to validate the composition and assist the propagation of changes during the evolution of the system.

In this context, this paper proposes a traceability management approach for the model composition operation. Within our proposal, we consider the traceability management as a crosscutting concern. The weaving of the traces generation is specified using graph transformations [1]. The Incorporated structure generates the trace model as an regular target model of the specification we wish trace. Such an extra output has manifold application in the model composition feild:

- To validate the composition: trace links provide a detailed view of the flow of execution. Indeed, they represent relationships between source model elements and their target equivalents. Through these links, we can verify the consistency and the completeness of the model composition.
- To support co-evolution of models: the trace model specifies how source artifacts participate in the production of the composed model. Those links are useful to analyze the impact of changing sub-models during the evolution of the system.
- To optimize the composition chain: As a part of a model composition chain, the restriction to source artifacts of a given stage of the overall chain confuses its management. Hence, the use of the trace model can broaden its scope, through the reuse of previous valuable links.

The remained of this paper is structured as follows: Section 2 presents a review on the traceability management and a discussion of the need of a traceability mechanism addressing the model composition operation. In section 3 we propose an overview of our approach. Thereafter, in section 4 we implement our traceability aspect for the ATL language; while in section 5 we demonstrate the applicability of our proposal using a merging scenario. Finally Section 6 summaries this paper and represents future works.

II. Background and motivation

The IEEE Standard Glossary of software Engineering Terminology [2] defines traceability as:

the degree to which relationship can be established between two or more products of the development process, especially products having a predecessor-successor or master-subordinate relationship to one other; for example, the degree to which the requirements and design of a given software component match

The definition of traceability is tied to the abstraction level of the managed artifacts and the traceability scenario. Indeed, the above definition is closely related to the requirements traceability field. As regard to model transformations, traceability is defined as:

- *Any relationship that exists between artifacts involved in the software engineering life cycle* [3].
- *The runtime footprint of model transformation .Essentially, trace links provide this kind of information*

by associating source and target model element with respect of the execution of a certain model transformation [4].

Essentially, traceability refers to the ability to manage links between elements handled by a model driven development operation. It provides a view of the changes that have occurred in these model elements and reveals the complexity of logical relations [5] existing among them.

Traces in the model transformation traceability field can either be generated naturally with an internal traceability tool (implicit traceability) or produced through an external dedicated support (explicit traceability). In the former case, the generation task does not require an additional effort; however, the generation process is fixed and cannot be configured to produce the required traces with respect to a given traceability scenario. In addition; traceability metamodel are often simple to allow an advanced post configuration. As to the implicit traceability; even if there is a need for encoding the traces generation; but manifold configuration tracks are offered. Indeed, both the traceability metamodel and the application scope can be chosen freely by the developer.

Besides, the Center of Excellence for Software Traceability [6] stated some grand challenges to guide the traceability management. The first one is the purpose of generating traces. Indeed, the traceability concern has to fit the users intension. In addition, the authors argue that traces have to be configurable, portable and generated with a scalable manner. In order to satisfy those requirements, we propose a traceability approach for the model composition operation. Actually, we have found in the literature various traceability solutions that address the model transformation task; however, we have not encountered a specific approach concerning the composition operation. Therefore, adopting such solutions does not deal with the purpose challenge. Indeed, the composition has particular intensions and the traceability support has to be aware of them. On the other hand, we believe that existing solutions do not provide means to express configurable traces, since they are designed in such a way that disregards the composition process. Our objective is to gather the benefits of the existing model transformation traceability solutions while focusing on the model composition task. In fact, we have conducted an analysis of the main traceability approaches. Accordingly we have set the above traceability requirements:

- The traceability data has to be stored in a separate model in order to not pollute the managed models. Besides, this trace model has to be expressed using a generic metamodel to reduce effort to achieve traceability. This mechanism allows the reusability of traces.
- The traceability metamodel must provide an extensibility mechanism for express-ing highly configurable traces with respect to the traceability purpose and the spe-cification of contributing models.
- Regarding the scalability challenge, we generate traces with an automatic mechan-ism. Besides, the generation process must be configurable with respect to the tra-ceability scenario.

Finally, we have taken into account the fact that our proposal must support a visualization system for expressing the trace model in a user friendly representation.

III. Overview of our traceability approach

In this section we provide an overview of the way traces are captured and structured. We aim through our approach keep track of the model composition operation. For that, we propose to generate the trace model as being an additional model of the specification to trace (see Figure 1). We consider the traceability concern as being a crosscutting concern. Indeed, we define a traceability aspect that automatically weaves the traces generation pattern. Traces are conforming to a generic traceability metamodel that deals with the configuration and portability challenges.

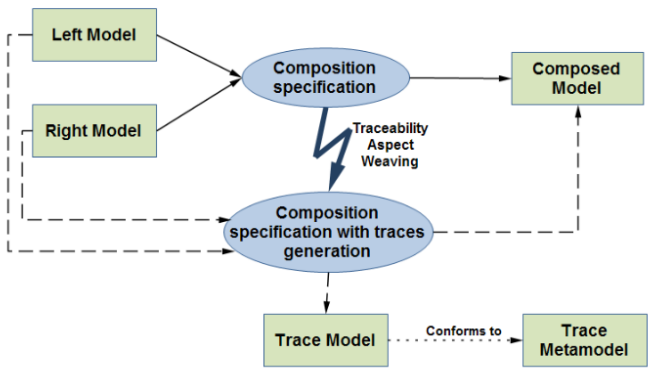
Left Model
Right Model
Composition specification
Composed Model
Traceability Aspect Weaving
Composition specification with traces generation
Trace Model
Conforms to
Trace Metamodel

Fig. 1: Traceability generation process

A. Traces structuring

We illustrate our traceability metamodel in Figure 2. We aim through this formalism expressing the composition relationship kinds in a trivial manner. The composition operation has a particular process [7]. It consists of detecting matching elements that describe the same concept in the left and right models. These pairs of elements are merged while other elements are eventually transformed into the target model. Hence, the structuring of traces must take account of these specificities in order to express purposed links. Accordingly, we have set two types of trace links: Merging and transformation links.

- *TraceLink*: it generalizes relationship between the source and target artifacts han-dled by the model composition operation.
- *MergingLink*: connects the left and right model elements to the merged one.
- *TransformationLink*: expresses a transition from a source model element (belonging to the left or right model) to their target equivalents.
- *ModelElement*: This concept refers the linked model elements.
- *Context*: it allows expressing of semantically rich traces through the assignment of further information to a sub-set of traces. This extensibility mechanism

is based on the definition of the relevant context attributes.

- *ContextAttribute*: represents the additional information to be assigned to a sub-set of trace links, such as: the intension of capturing those traces, the rule that generates them
- *TraceModel*: it is the root element which contains all the generated trace links and contexts.

Besides, we represent the rule invocation through parent-child relations among trace links. This structuring provides a multi-scales character to the trace model. Hence, the user can examine traces at different granularity degree.

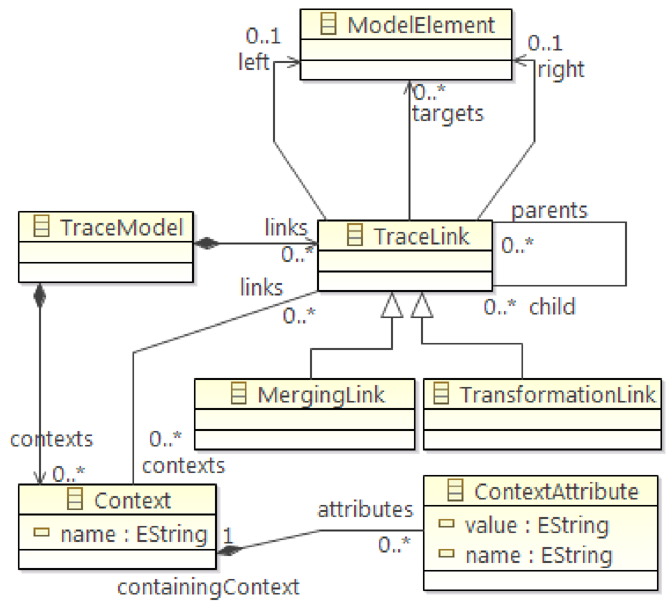

Fig. 2: Composition traceability metamodel.

B. Traces generation

Aspect Oriented Modeling (AOM) [8] applies aspect oriented programming [9] in the context of model driven engineering. It focuses on modularizing and composing crosscutting concerns during the design phase of a software system. Aspect oriented approaches aim to separate crosscutting concerns (security, persistence...) from the core concerns of a system. In AOM, both the aspects that encapsulate the crosscutting structures and the base model they crosscut are models. An aspect is defined principally by:

- A pointcut: it is a predicate over a model used to determine the places where the aspect should be applied (joinpoints).
- An advice: It is the new structure that replaces the relevant jointpoints.

Within our approach, we adopt aspect orientation to generate traces. This mechanism deals with the generation task while supporting the portability and scalability challenges. Since the traceability concern is encapsulated in a reusable aspect that automatically weaves the traces generations patterns. Besides, AOM allows abstracting the specification through its corresponding model. Hence, we can keep track of the composition regardless its concrete syntax nature (textual or graphical).

The implementation of the traceability aspect is based on graph transformations. A graph rewriting rule consists of two parts, a left-hand side (LHS) and a right-hand side (RHS). A rule is applied by substituting the objects of the left-hand side with the objects of the right-hand side, only if the pattern of the left-hand side can be matched to a given graph [10]. In what follows, the traceability aspect corresponds to a set of graph transformation rules. The LHS parts determine the places where the aspect should be applied (**joinpoints**) and the RHS parts define the new structures that replace the relevant joinpoints (**advice**).

The graph transformation unit depicted in Figure 3 presents an overview of the weaving process. The TraceModelDec rule allows declaring the trace model to be an additional model of the specification to trace. Thereafter, the second rule (a loop subunit) is applied to trace the rules. The tracing of a rule consists of providing it with the behavior it needs to produce a trace link that relates the source elements with the composed ones. Basically, through the declaration of an additional output element that refers the traceability link and the assignment of the traceability data (left, right and targets reference) to it. Subsequently, the rule TraceLinksNesting weaves the patterns responsible of the imbrications of trace links with respect to the rule calls. The last rule deletes all the intermediate information we use to perform the weaving process. The next section illustrates the implementation of this weaving process for the ATL language.

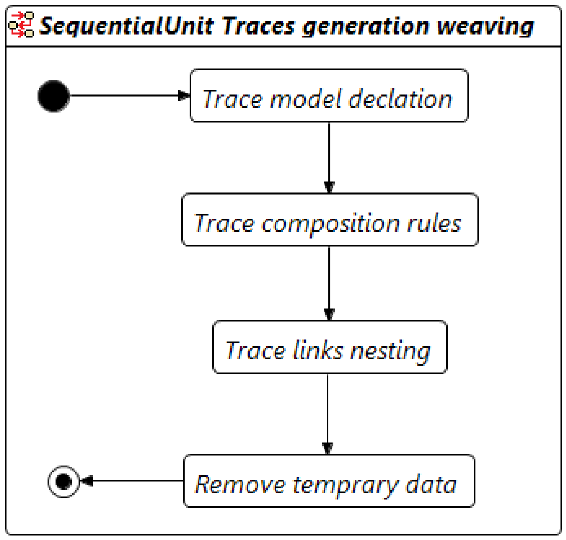

Fig. 3: Traces generation weaving unit.

IV. Implementation of the Traces Generation Process

In this section we implement the traceability weaving process for the ATL language [11]. ATL is a model transformation language that provides tools to compute a set of targets models from source models. However, it does not support means to express the composition scenario in a native manner. We have experienced the applicability of our solution to trace composition oriented approaches like EML [12] and amar et al. [13]. By choosing the ATL language, we aim to reveal the generic character of our proposal and the way we master the specificities of a given language using AOM.

A. *Traceability management within ATL*

Jouault [14] proposed a traceability approach to overcome the limitation of the implicit traceability in ATL. It addresses two configuration dimensions: the range and the format. The range refers to the possibility to select a subset of elements to trace, while the format corresponds to the traces structuring. The generation of trace links is based on a High-Order Transformation (HOT) named TraceAdder. This HOT automatically inserts the traces generation code to any existing ATL program.

Yie and Wagelaar [15] presented a solution that enlarges the limited access to the implicit traceability links handled by the resolveTemp operation. They proposed two mechanisms to provide this rich access. The first one consists of an extension of the ATL virtual machine in order to provide the selection of the target elements by type and copying the implicit transient links into an external trace model. Besides, they present another mechanism that allows generating the trace model on demand through byte code adaptation.

Although the proposal of the aforementioned approaches is structured around the configuration issue, they use a very simple traceability metamodels which are not amenable to generate highly configurable trace models. Actually, the authors have not proposed way for adding customized tracing information or for adapting their generation process to other traceability metamodel mean to express valuable and interesting traces depending on the traceability context (the composition scenario in our case).

B. *Applying our traceability approach to trace ATL specifications*

We have chosen the ATL language as an example of transformation language used to specify the merging scenarios in a non native way. Indeed, ATL is not a dedicated composition language and does not categorize rules on merging and transformation rules. It used matched rules to specify the transformation behavior in a declarative way; while lazy and called rules allow defining imperative transformations.

We have proposed to employ graph transformations to perform the weaving of the traceability aspect. For that, we use the ATLCopy transformation to generate the corresponding model of an ATL concrete specification. Thereafter, a specific graph transformation unit is applied on the resulting model to weave the traces generation patterns. In what follows, we explained the main graph transformation rules which constitute this unit. The Henshin project [16] is used for specifying and implementing these rules. Note that its representation of a rule does not explicit the description of the left and right hand sides. Instead, it is based on the following stereotypes to depict the rule application semantic:

- preserve: all elements (edges or nodes) labeled with this stereotype must be sought to enable the rule application. Furthermore, those elements will be copied in the resulting graph.
- create: it is used to describe the new elements to be added to the graph.
- delete: it references the elements to be removed from the graph. require: this stereotype allows the expression of conditions are required for applying the rule.
- forbid: by contrast, the presence of such a pattern prohibits the rule application.

1) Trace model declaration: The purpose of the rule depicted in Figure 4 is to declare the trace model trace as an extra output of the module to trace. This model is conforming to our generic traceability metamodel (cf. section 3.1) which corresponds to the other created node.

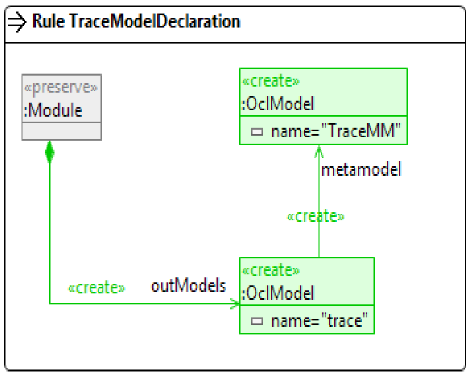

Fig. 4: Trace model declaration rule

2) Trace ATL rules: We have presented that ATL does not categorize rules on merging and transformation rules. Therefore, the user assistance is required to resolve the rule category in a composition scenario task. For that, we have augmented the rule concept in the ATL abstract syntax with the type attribute. This attribute admits two values, Merge and Transform, they are assigned by the end user depending on the intension of defining the rule. Note that we have experienced an automatic resolution of the rule category based on the number of input patterns. Indeed, we considered rules with two input elements as simulating the merging behavior and the others as transformation rules. However, this hypothesis does not support a realistic selection, since a rule with two inputs may encapsulates a transformation behavior. Hence, we argue that even if this annotation mechanism is time consuming but it allows generating trusted [6] traces.

The Figure 5 describes the graph transformation that allows tracing an ATL rule which encapsulates the merging behavior. Keeping track of such a rule consists of declaring the traceability link that captures the relationship between the contributing model elements and the target one as being an extra output. Indeed, this graph transformation seeks an ATL rule annotated with the type Merge. Subsequently, it creates a new SimpleOutPatternElement node of type MergingLink that refers to the traceability link. Note that the tracing of

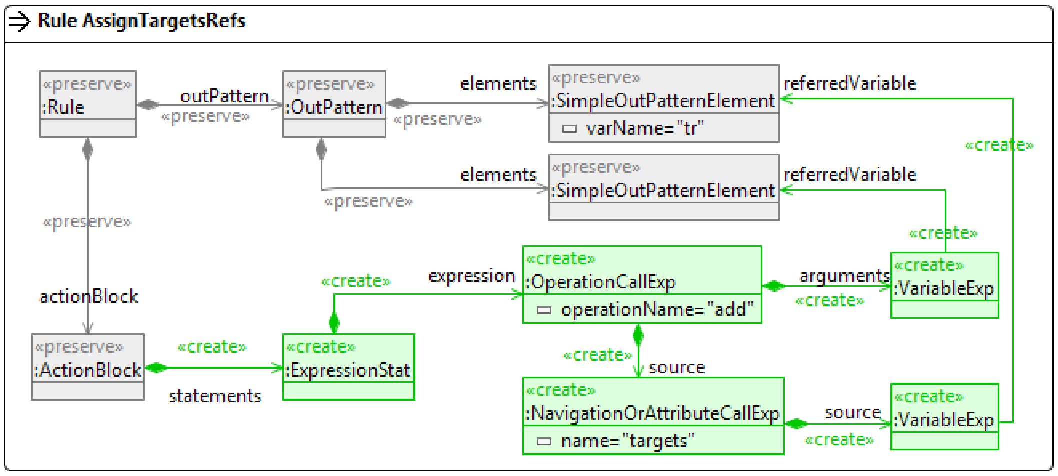

Fig. 6: Assignment of the targets references

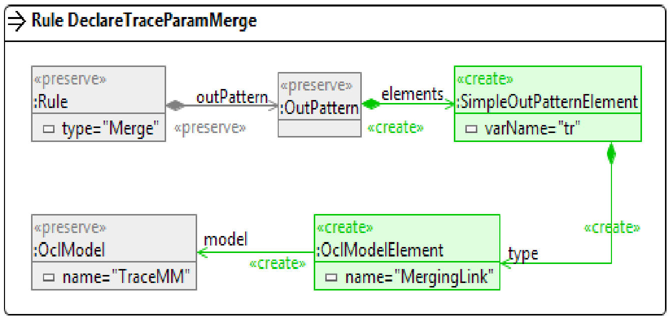

Fig. 5: Trace a merge rule

transformation rules is performed using a similar manner. In addition, we designed additional rules to connect trace links to their relevant traced elements. These graph transformations look for the InPattern elements, parameters and OutPattern elements, then each one is assigned as a left, right or targets reference of the link generated by the current rule (which is connected to the selected element). As an example, the rule depicted in Figure 6 allows the assignment of the targets reference value. It adds the selected OutPattern element as further target of the trace link (referenced by the tr element) through the created ExpressionStat node.

3) Trace links nesting: During the execution of the resulting ATL module, the application of the pervious rules allow the generation of trace links while producing the target model elements. The structuring of these links will be closely modeled on the rules invocation sequence. Basically, we catch every call of a rule; then the traceability element created by the calling rule will be assigned as being a parent of the link generated by the called one. Furthermore, the ATL language defines two mechanisms of calling rules: an implicit call of matched rules through the use of the resolveTemp operation and an explicit call of lazy and called rules.

In the former case, the rule responsible of nesting traces (see Figure 7) searches for a call of the resolveTemp operation. This ATL operation returns a target equivalent of a given source element. Given that it results from the tracing of all matched rules, the production of additional outputs corresponding to the traceability data; those links can be returned as being a target equivalent of the element to resolve element. Essentially, the MatchedLinksNesting rule adds two statements. The first one copies the original call of the resolveTemp operation. The other statement assigns the selected trace equivalent to be a parent of the trace link corresponding to the resolved source element. This filtering is based on the second parameter of resolveTemp that encodes the name the of target pattern element. Regarding the explicit call, we allow the called and lazy rules to access the traceability element generated by the calling rule and assign it as parent of their trace link. This parent must be passed as parameter in the call expression. The rule depicted in Figure 8 inserts the code responsible of this kind of nesting. This graph transformation seeks a call of an operation which corresponds to a called rule name. Thereafter, it augments this call with a new parameter value referencing the parent link. The definition of the called rule has to be modified to take account of the newly passed value. For that, we add a parameter trp (resp. an InPattern element in the case of lazy rules) of type TraceLink and create the ExpressionStat node to connect this parameter to the trace link generated by the rule that have been matched. Note that the same rule may be called several times. The definition of the called rule has to be changed once. Actually, we use two rules to allow the explicit nesting. The first one adds the parameter to the rule call expression and annotates the called rule. Thereafter, the second one browses all the annotated rules and changes their definition.

4) Context assignment: The use of generic purpose traceability metamodel proves advantages to support the reusability task. However, traceability data has to bring interesting information with respect to the traceability scenario. For that, the generic traceability metamodel has to be augmented with an extensibility mechanism to allow defining relevant expressiveness data regarding the traceability point of view and the models to compose specifications. The context and contextAttribute concepts deal with this task. Basically, the definition

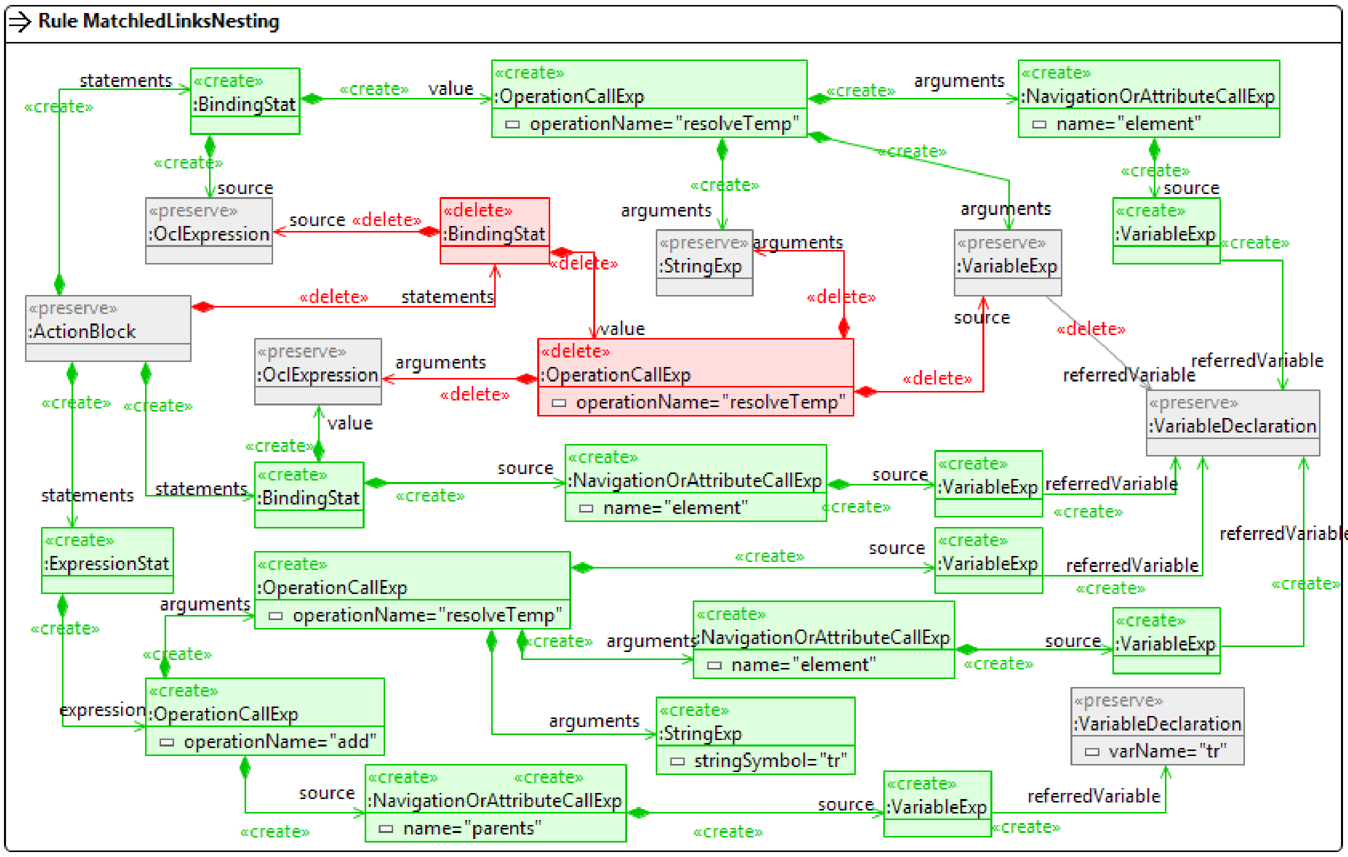

Fig. 7: Trace links nesting for implicit calls

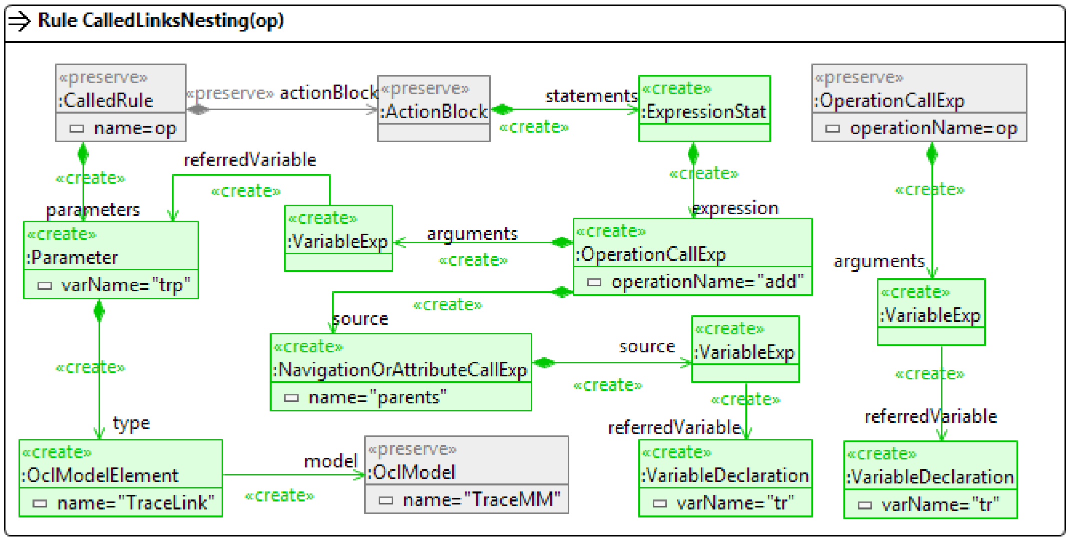

Fig. 8: Trace links nesting for explicit calls

of a context attribute is tied to the additional information to be appended to a specific sub-set of traces, while the context concept is a well thought out combination of attributes.

In order to assign further information to trace links, the user has to provide the struc-turing of data to be appended using the proposed extensibility mechanism. Thereafter, all the possible contexts have to be generated. We define two ATL rules that support this task: the createAttribute rule (see listing 1) that allows the production of a context attribute with respect to a given combination of an attribute name and value, thereafter, it connects the created attribute to its relevant context which is produced by the createContext rule (see listing 2). Note that the possible values of context attribute are either given by the user or automatically resolved through the definition of a specific graph transformation. In either case, the context name is automatically assigned depending on the aggregated context attributes and can be viewed as a context key. The createContext rule is defined as unique lazy to prohibit the creation of multiple contexts with the same key.

Listing 1: create context attribute for atl

```
rule createAttribute(atName:String, atValue:
    String, cName:String){
        to t:TraceMM!ContextAttribute()
        do{
                t.name<-atName;
                t.value<-atValue;
                t.owningContext<-thisModule.
                    createContext(cName);
        }
}
```

Listing 2: create context for atl

```
unique lazy rule createContext{
from cName:String
to t:TraceMM!Context()
        do{
        t.name<-cName;
        }
}
```

Once all the contexts are generated, a graph transformation rule seeks the declared traceability parameters; thereafter it resolved the key of the context to be assigned to the selected parameter according to its connected contextual information. As a basic example, we propose to structure traces by the generating rule name. For this purpose, the context definition includes one context attribute which references the rule name contextual data. The declaration of contexts is performed using the graph transformation depicted in Figure 9 This rule looks for a rule node, then, it creates the corresponding context through the invocation of the createAttribute rule. The passed aruguments refer respectively to the context attribute name, the attribute value which corresponds to the selected rule name and the context name. The assignContext rule (see Figure 10) allows connecting the traceability parameter to the relevant context by calling the createContext operation , which returns the context identified by the selected rule name .Note that the status attribute has been added to the atl abstract syntax to prohibit multiple application of the graph transformations to the same ATL rule (resp. traceability parameter).

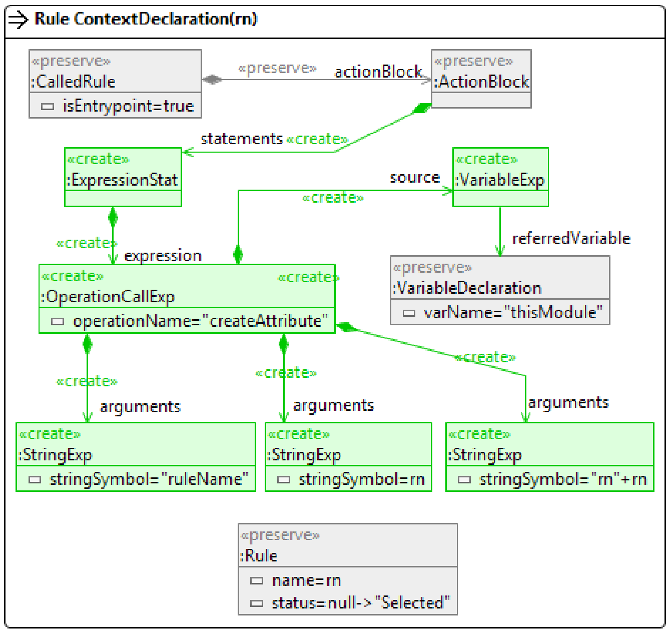

Fig. 9: the context declaration rule

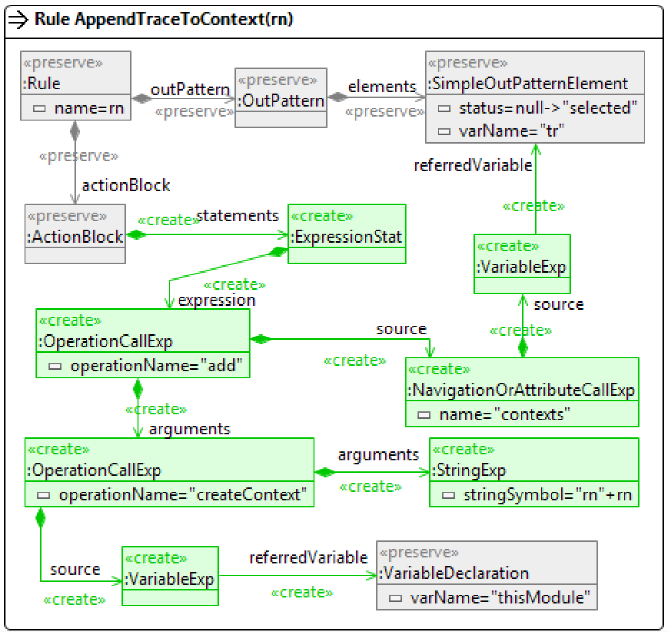

Fig. 10: append the traceability parameter to its context

V. Case Study

In this section, we illustrate the application of our traceability approach. The composition scenario we have chosen is the merging of two UML class diagrams into a VUML class diagram. We first briefly overview the VUML profile and introduce the case study. Thereafter, we present the results we obtain with our traceability framework.

A. *The VUML profile*

The VUML approach was developed to meet the needs of complex systems analysis and design according to various viewpoints [17]. In VUML, a viewpoint represents the perspective from which a given actor interacts with the system. In other words, at the requirements analysis step, a viewpoint expresses requirements and needs of one actor. At the design step, VUML extends the UML language by adding the concept of multiview class which is composed of a base class (shared by all viewpoints), and a set of view classes (extensions of the base class), each view class being specific of a given viewpoint. An actor’s viewpoint on the system is expressed by a set of UML diagrams obtained by focusing on the relation of this actor with the system. A process was defined for VUML [18]. It begins by defining, for each actor, the use case diagram related to its needs, and then scenarii are modeled by means of sequence diagrams. Finally, static class diagrams are defined. Once all viewpoints are modeled, diagrams of the same type are merged into VUML diagrams, in order to express all viewpoints on a unique diagram, while preserving the possibility of selective access. VUMLs semantics is described by a metamodel, a set of wellformed rules expressed in OCL , and a set of textual descriptions in natural language. A multi-target code generator was also developed to produce object code from VUML class diagrams. It was tested with Java as target language [19].

B. *Case study*

In what follows, we illustrate the results of our traceability framework through the composition of two class diagrams that have been extracted from the Course Management System (CMS) [18]. The CMS is used by different users. It allows distant students to apply for courses, access related documentation (slides, web pages, text, etc.), solve/create exercises, communicate with teachers and take exams. It allows teachers to edit their own courses, plan learning experiences and units of work, and record student assessments. The CMS is managed by an administrator whose job consists in recording students and managing resources.

According to the VUML process, the CMS is designed through a set of viewpoints (student, teacher and manager). For each viewpoint, a set of UML diagrams (class diagrams, state machines, sequence diagrams, etc.) are produced .We limit the application of our approach on the composition of structural models (class diagrams). Figure 11 shows an excerpt of the class diagram of the Student.s viewpoint focusing on the Course class, while Figure 12 depicts the class diagram that model the teacher requirements.

These viewpoint models are composed to produce a VUML model. Figure 13 depicts the VUML class diagram resulting from the composition of the two class diagrams shown above. Classes appearing in both viewpoint models, with the same name and with different properties (attributes, operations, associations, etc), are merged as single multiview classes (Course and Exercise classes). Properties of the class Course that are shared by the two considered viewpoints have been put into the class stereotyped by base; properties that are specific of one viewpoint have been put into classes stereotyped by view.An extract of the ATL specification that allows performing this composition is given in listing

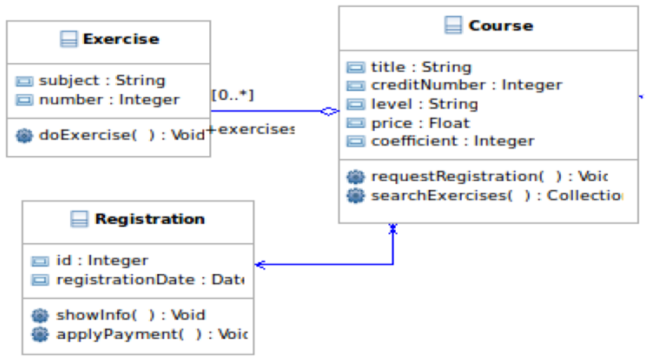

Fig. 11: excerpt of the class diagram of the Student.s viewpointt

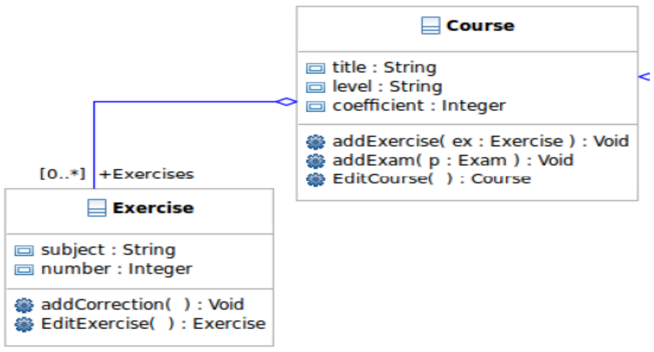

Fig. 12: excerpt of the class diagram of the Teachers viewpointt

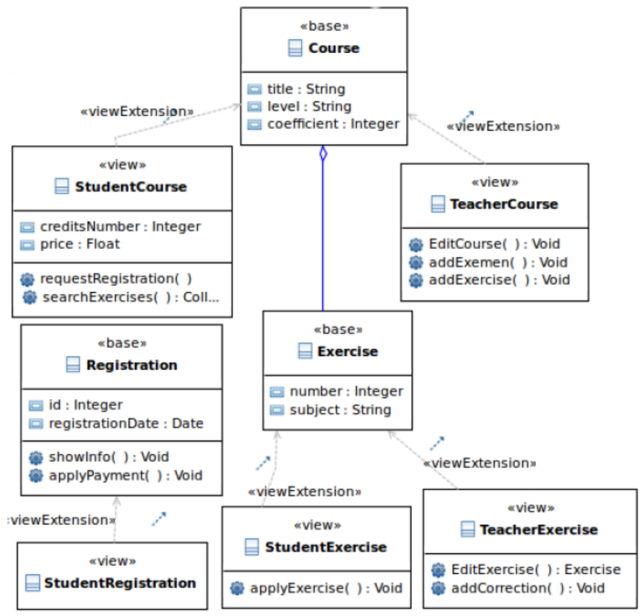

Fig. 13: excerpt of the VUML class diagram of CMS system

VI. Illustrative example

VII. Related works

Jouault [14] presented an approach to trace specifications written in the ATL language. This work was considered as a basis for several approaches addressing the transformation traceability management. The author proposed to generate the trace model in the same way other target models are generated. Actually, the code responsible of producing this extra output is inserted using a High-Order Transformation named TraceAdder. This mechanism deals with the scalability challenge, since it allows automating the generation task. Nevertheless, the definition of this HOT is closely tied to the proposed traceability metamodel which is very simple to express configurable traces. Moreover, the use a practical metamodel requires a laborious and complex modification of the HOT.

In [20], the authors provide a traceability framework for the Kermeta language. The imperative character of such transformations makes them difficult to be analyzed. Indeed, they proposed to manually add the traces generation codes. Their traceability metamodel expanded the metamodel proposed in [14] by structuring traces into steps. This concept allows a basic configuration to the trace model. The manual encoding of the generation concern allows the user to trace its required elements, but to the detriment of scalability.

Aspect oriented programming provides the tool to solve the traceability management issues. Essentially, the traceability concern is encapsulated in a reusable aspect. Its application allows inserting the traces generation code automatically and without polluting the specification. Within this scope, Amar et al. [21] proposed a traceability framework based on AOP for tracing imperative transformations written in JAVA. The framework defines categories of traceable operations and their respective poincuts. Besides, in order to take into account all the operations to trace, the programmer can define new custom categories or restrict the predefined ones. On the other hand, the authors apply the composite design pattern and the link type concept in view of expressing configurable traces.

Grammel and Kastenholz [22] have defined a generic framework to augment transformation support with traceability. The authors do not treat a specific transformation language. They provide a generic interface which involves specific connectors for the transformation supports. The augmentation is based on a generic traceability metamodel extensible with facets to express highly configurable trace models. For generating traces, the authors describe two mechanisms: transforming the implicit trace model to be conforming to their generic metamodel or capturing traces using AOP.

The aforementioned approaches can be used to keep track of the model composition operation. However, the fact that they disregard the composition process and intensions prevents them to express configurable and interesting traces. In this context, our approach takes advantages of the existing solutions while focusing on the composition operation. We proposed to use an AOM approach to generate trace links. This mechanism allows encapsulating the traceability concern in a reusable aspect that automatically weaves the traces generation patterns. On the other hand, the abstraction of the specification to trace through its corresponding model provides support to master the composition languages features such as: the concrete syntax nature (textual, graphical) or the resolution of the rule category (merge or transformation).

As for the configuration challenge, we used a generic traceability metamodel to express reusable trace models. This metamodel was designed with respect to the typical composition process. Indeed, it categorizes traces on two subsets: Merging links and transformation links. This allows expressing the composition relationships kinds in a trivial manner and will guide the reuse of traces. In addition, the context concept provides us with the mechanism to express highly configurable trace links. Actually, it allows the assignment of valuable information to the generated traces depending on the traceability point of view and the managed models specifications.

VIII. Conclusion and future work

The contribution of this paper is a graph transformations based approach to trace the model composition operation. Our objective was to define a purposed traceability approach that generates valuable traces for this particular operation, in an automatic, reusable and configurable manner. For that, we considered the traceability concern as being a crosscutting concern. The weaving of the traces generation patterns is performed using graph transformations rules. This mechanism allows us to master the specificities of model composition approaches. Besides, we use a generic metamodel to express semantically rich traces.

We are currently working on generic framework that allows defining the traceability aspect regardless the composition language. This framework will be augmented with a specialization mechanism to specialize its application for a given language. Furthermore, we are planning for further exploring the possible reuses of traces. Our major intension is to optimize the composition chain by using the previous generated links while executing a given step.

References

[1] G. Rozenberg and H. Ehrig, *Handbook of graph grammars and computing by graph transformation*. World Scientific Singapore, 1997, vol. 1.

[2] J. Radatz, A. Geraci, and F. Katki, "Ieee standard glossary of software engineering terminology," *IEEE Std*, vol. 610121990, p. 121990, 1990.

[3] N. Drivalos, R. F. Paige, K. J. Fernandes, and D. S. Kolovos, "Towards rigorously defined model-to-model traceability," in *ECMDA Traceability Workshop (ECMDA-TW)*, 2008, pp. 17–26.

[4] B. Grammel and K. Voigt, "Foundations for a generic traceability framework in model-driven software engineering," in *In Proceedings of the ECMDA Traceability Workshop*, 2009.

[5] N. Anquetil, B. Grammel, I. Galvao Lourenco da Silva, J. Noppen, S. Shakil Khan, H. Arboleda, A. Rashid, and A. Garcia, "Traceability for model driven, software product line engineering," 2008.

[6] O. Gotel, J. Cleland-Huang, J. H. Hayes, A. Zisman, A. Egyed, P. Grünbacher, A. Dekhtyar, G. Antoniol, and J. Maletic, "The grand challenge of traceability (v1. 0)," in *Software and Systems Traceability*. Springer, 2012, pp. 343–409.

[7] J. Bézivin, S. Bouzitouna, M. D. Del Fabro, M.-P. Gervais, F. Jouault, D. Kolovos, I. Kurtev, and R. F. Paige, "A canonical scheme for model composition," in *Model Driven Architecture–Foundations and Applications*. Springer, 2006, pp. 346–360.

[8] R. France, I. Ray, G. Georg, and S. Ghosh, "Aspect-oriented approach to early design modelling," *IEE Proceedings-Software*, vol. 151, no. 4, pp. 173–185, 2004.

[9] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, *Aspect-oriented programming*. Springer, 1997.

[10] L. Lambers, H. Ehrig, and F. Orejas, "Conflict detection for graph transformation with negative application conditions," in *Graph Transformations*. Springer, 2006, pp. 61–76.

[11] F. Jouault and I. Kurtev, “Transforming models with atl,” in *Satellite Events at the MoDELS 2005 Conference*. Springer, 2006, pp. 128–138.

[12] D. S. Kolovos, R. F. Paige, and F. A. Polack, “Merging models with the epsilon merging language (eml),” in *Model Driven Engineering Languages and Systems*. Springer, 2006, pp. 215–229.

[13] A. Anwar, A. Benelallam, M. Nassar, and B. Coulette, “A graphical specification of model composition with triple graph grammars,” in *Model-Based Methodologies for Pervasive and Embedded Software,LNCS*. Springer-Verlag Berlin Heidelberg, 2013, vol. 7706, pp. 1–18.

[14] F. Jouault, “Loosely coupled traceability for atl,” in *Proceedings of the European Conference on Model Driven Architecture (ECMDA) workshop on traceability, Nuremberg, Germany*, vol. 91. Citeseer, 2005.

[15] A. Yie and D. Wagelaar, “Advanced traceability for atl,” in *1st International Workshop on Model Transformation with ATL*, 2009, pp. 78–87.

[16] T. Arendt, E. Biermann, S. Jurack, C. Krause, and G. Taentzer, “Henshin: advanced concepts and tools for in-place emf model transformations,” in *Model Driven Engineering Languages and Systems*. Springer, 2010, pp. 121–135.

[17] M. Nassar, B. Coulette, X. Crégut, S. Ebersold, and A. Kriouile, “Towards a view based unified modeling language,” in *ICEIS (3)*, 2003, pp. 257–265.

[18] A. Anwar, S. Ebersold, B. Coulette, M. Nassar, and A. Kriouile, “A rule-driven approach for composing viewpoint-oriented models.” *Journal of Object Technology*, vol. 9, no. 2, pp. 89–114, 2010.

[19] M. Nassar, A. Anwar, S. Ebersold, B. Elasri, B. Coulette, and A. Kriouile, “Code generation in vuml profile: A model driven approach,” in *Computer Systems and Applications, 2009. AICCSA 2009. IEEE/ACS International Conference on*. IEEE, 2009, pp. 412–419.

[20] J.-R. Falleri, M. Huchard, C. Nebut *et al.*, “Towards a traceability framework for model transformations in kermeta,” in *ECMDA-TW’06: ECMDA Traceability Workshop*, 2006, pp. 31–40.

[21] B. Amar, H. Leblanc, and B. Coulette, “A traceability engine dedicated to model transformation for software engineering,” in *ECMDA Traceability Workshop (ECMDA-TW)*, 2008, pp. 7–16.

[22] B. Grammel and S. Kastenholz, “A generic traceability framework for facet-based traceability data extraction in model-driven software development,” in *Proceedings of the 6th ECMFA Traceability Workshop*. ACM, 2010, pp. 7–14.