Toward an Architecture for Model Composition Techniques

Kleinner Farias, Lucian José Gonçales,
Murillo Scholl, Maurício Veronez
PIPCA, University of Vale do Rio dos Sinos (Unisinos)
São Leopoldo, RS, Brazil
kleinnerfarias@unisinos.br,
lucianjosegoncales@gmail.com,
murillosholl@hotmail.com, veronez@unisinos.br

Toacy Oliveira
PESC/COPPE, Federal University of Rio de Janeiro (UFRJ)
Rio de Janeiro, RJ, Brazil
toacy@cos.ufrj.br

***Abstract*—Academia and industry are increasingly concerned with producing general-purpose model composition techniques to support many software engineering activities, e.g., evolving UML design models or reconciling conflicting models. However, the current techniques fail to provide flexible and reusable architectures, a comprehensive understanding of the critical composition activities, and guidelines about how developers can use and extend them. These limitations are one of the main reasons why state-of-the-art techniques are often unable to aid the development of new composition tools. To overcome these shortcomings, this paper, therefore, proposes a flexible, component-based architecture for aiding the development of composition techniques. Moreover, an intelligible composition workflow is proposed to help developers to improve the understanding of crucial composition activities and their relationships. Our preliminary evaluation indicated that the proposed architecture could support composition tools for UML class, sequence, and component diagrams.**

Keywords: model composition, architecture, UML

I. INTRODUCTION

Researchers and practitioners recognize the importance of model composition in many software engineering activities [1][2][8], e.g., evolving design models to add new features and reconciling multi-view models developed in parallel by different software development teams [5][7]. In collaborative software development, for example, separate virtual teams may concurrently work on a partial model of the overall architecture to allow developers to concentrate more effectively on parts of the architecture relevant to them. At some point, it is necessary to bring these models together to generate a "big picture" view of the overall architecture. Unfortunately, this composition task is considered as an error-prone, time-consuming task [1][8]. In [8], the authors highlight that the model comparison and merging task are tedious, time-consuming, and error-prone. In [1], Mens reinforces that software merging continues to be *"a time-consuming, complicated, and error-prone process because many interconnected elements are involved and merging depends on both the syntax and semantics of these elements."*

For this reason, there has been a significant body of research into defining model composition techniques in the areas of software modeling [9], synthesis of feature models [12], and software product lines [11]. In fact, the high number of conventional, general-purpose composition techniques created in the last decade attests this importance, e.g., Kompose [13], IBM Rational Software Architect (IBM RSA) [4], MATA [3] and Epsilon [5].

Model composition can be briefly defined as an operation where a set of tasks should be performed over two input models, M_A and M_B, in order to produce an output-intended model, M_{AB}. While M_A represents the base model, M_B consists of the delta model having all increments that should be inserted into M_A to transform it into M_{AB}. Existing composition techniques usually produce an output composed model (M_{CM}) that often does not match the output intended model (M_{AB}), i.e. $M_{CM} \neq M_{AB}$. Because the elements of the input models usually conflict with each other in some way, and these techniques end up being unable to deal with all contradicting changes properly.

The problem is that the general-purpose feature of composition techniques hinders coping with a set of particular composition cases. Unfortunately, they fail to provide flexible, reusable architectures, a comprehensive understanding of the chief composition activities, or even provide guidelines about how developers can use and extend them. The limitations can be explained for two principal reasons as follows: (1) composition techniques are not structured with design-for-change principles upfront, being rigid to support modern composition strategies. Typically, developers are commonly forced to go through the source code to locate the component to-be changed or even create new architectural components to implement upcoming features. An incorrect modification of such components can jeopardize the implementation of new features, and (2) they rely on generic representation, i.e., usually graph, rather than on the semantics of constructs of OO design modeling languages, e.g., UML [9]. Since the current multi-view UML diagrams demand different but complementary ways to be integrated, generic approaches tend to produce output models with inconsistencies. Consequently, they fail to provide a systematic and flexible way to derive composition techniques for a particular purpose, or even provide guidelines about how developers can evolve them.

To overcome these shortcomings, this paper, therefore, proposes a flexible, component-based Architecture for aiding the development of Model Composition Tools, hereafter called

(DOI reference number: 10.18293/SEKE2015-107)

MoCoTo-Arch, and a model composition workflow for helping developers to improve the understanding of the crucial composition activities and their relationships. Our preliminary evaluation indicated that the proposed architecture could support the development of composition tools for UML class, sequence, and component diagrams.

The remainder of the paper is organized as follows. Section II contrasts this work with the current literature. Section III presents the MoCoTo architecture. Section IV describes the composition tool developed using the MoCoTo-Arch. Finally, Section V presents some concluding remarks and future work.

II. RELATED WORK

The last few years, some techniques have been proposed, including MATA [3], a tool based on graph transformations for composing aspects, IBM RSA [4], a robust software modeling and model composition tool, and Epsilon [5], an Eclipse Plugin consists of a family of languages for composing models and other vast functions. Although some works provide programming languages to express composition logic [5], little is known about the flexibility and capacity of the current techniques to support new composition strategies. This lack hinders the understanding about how such techniques can evolve to support the composition of new design models, until then not supported.

Many works aim at studying the proactive detection and earlier resolution of composition conflicts. In [2], Brun *et al*. proposes Crystal, an approach to help developers identify and resolve conflicts early. The authors highlight the ever-present occurrence of composition conflicts, more than would be expected, e.g. overlapping textual changes and their subsequent build and test failures. Likewise, Sarma [6] comes up with Palantír, a workspace-aware approach for detecting and resolving contradicting changes in early stage. Although these two approaches are interesting studies, they neither propose a flexible, design-for-change approach nor provide a comprehensible workflow to leverage the understanding of the inherent model composition activities and its relationships. Still, they overlook the challenging considering the synthesis of heterogeneous design models, thus not leading to broader generalizations of their findings at the modeling level.

On the other hand, in [7], the authors discuss the problem of tolerating conflicts and transforming them in an object of enhancement in collaborative software development. The purpose is to maintain all the conflicting changes in the resulting model. For this, they propose annotations, relating the conflicts as well as the developers involved in a further resolution. In [8], the authors introduce an approach to find similarities between business process models. For this, they define metrics to match the input model elements, and use typography and synonym dictionary.

III. MOCOTO ARCHITECTURE

We present the MoCoTo's built-in model composition process by identifying the phases, the artifacts generated, and the main activities required to transform the input models, M_A and M_B, into an intended output composed model, M_{AB}. Moreover, it details the most relevant characteristics related to design and implementation issues, including feature model elicited, components that implement such features, and the architectural design.

A. Model Composition Process

Figure 2 shows the proposed model composition process. It is represented as an intelligible workflow, thus allowing developers to understand the inherent activities of a composition process in terms of phases, its artifacts, activities and the flow among each other.

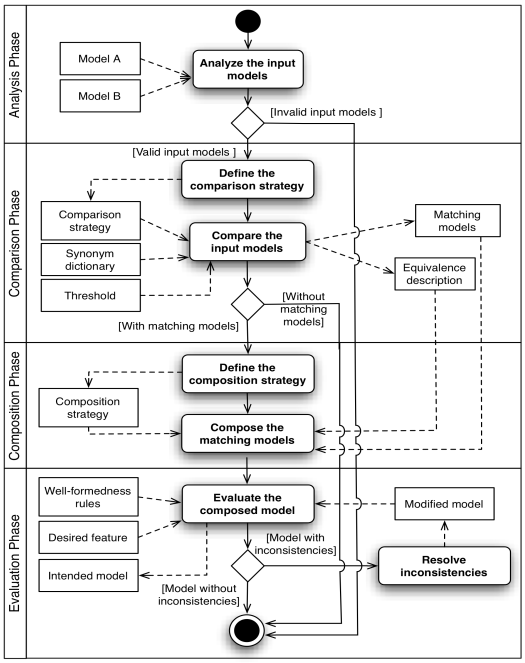

Figure 1. The proposed model composition process.

1) Analysis Phase: the prime goal is to analyze the input models adequately as a basis for assuring the composition of compatible input models as well as preventing input models with inconsistencies. This phase should attend to the *Lifecycle Analysis Milestone* criteria answering: are the input models of the same type? Do the input models have inconsistencies? If the input models do not attend to this milestone, the composition process can be cancelled or repeated after the input models are redesigned to comply with milestone criteria.

2) Comparison Phase: the chief goal is to systematically compare the input models for determining the similarity between their elements, thereby mitigating mistaken equivalence relationships, including false-positive and false-negative ones. The MoCoTo architecture supports a range of matching strategies (but not limited to), including default, partial and complete one [10], to alleviate the more severe risk items. The inputs of this phase are: ngram algorithm, synonym dictionary, matching strategies, matching rules, and threshold. Hence, producing the following outputs: (1) the similarity matrix, specifying the degree of equivalence (ranging from 0 to 1) between the input model elements; (2) the matching elements, a description of the elements of M_A and M_B being considered equivalent; (3) the no-matching elements, a description of the elements of M_A and M_B being considered no equivalent. Two input elements are considered similar when

the degree of similarity between them is equal or higher than 0.8, the threshold used. This threshold is based on previous studies [10] on model comparison, which have demonstrated its usefulness.

3) *Composition Phase*: The master goal is to carefully bring together the matching and no-matching elements for producing an output intended model, M_{AB}. For this, the proposed composition technique takes into account the similarity matrix, as well as the description of the matching and no-matching elements of the input models. In addition, it uses a range of well-established composition strategies (but not limited to), including *override*, *merge,* and *union* [15], to accommodate the elements from M_B into M_A, thereby alleviating the more severe risk items. The MoCoTo's built-in composition strategies integrate the matching elements while the no matching ones are just inserted into the M_{AB}. Thus, M_{AB} represents the matching and no-matching elements, all blended systematically.

4) *Evaluation Phase*: the master goal is to evaluate if the output model produced in the previous phase matches the output intended one, i.e. $M_{CM} = M_{AB}$. If $M_{CM} \neq M_{AB}$, then M_{CM} needs to be manipulated so that the inconsistencies can be resolved. For this, the tool checks if the output model is in compliance with well-formedness rules defined in the UML metamodel and meets a set of desired features specified by the user. If the model has inconsistencies, then some transformation rules can be applied to transform M_{CM} into M_{AB}. This phase end producing the output intended model. After detailing the composition process, the next Section focuses on describing the design and implementation issues required to put the process in practice.

B. MoCoTo architecture feature model

The MoCoTo architecture was proposed due to several reasons and requirements identified in previous works [10][14][15]. First, our experience with model composition has indicated the increasing need for reusable architecture to support and guide the development of new composition tools. Second, it is representative of the model composition domain, since its design decomposes the key concerns into well-modularized features. Third, it assures the derivation of different products by defining several variability points related to heterogeneous strategies related to analyzing, comparing, and composing the input models. Lastly, it allows evaluating the models generated and persisting the results. Thus, the proposed architecture provides a set of pivotal features, including analysis of the input models, comparison of the input models, composition of the equivalent input model elements, persistence of the output model generated, and evaluation of the output model.

Figure 3 shows a simplified view of MoCoTo-Arch's feature model. Thus, to develop composition tools developers should firstly implement the *mandatory features*, including analysis, comparison, composition, persistence, and evaluation. Besides identifying a set of core functionalities, the mandatory features seamlessly specify their dependencies in an easy-to-understand manner. An ever-present concern throughout the MoCoTo architecture was to assure the mandatory features comply with the model composition process described in Figure 3, for example, the *analysis* feature implements the first phase and the *persistence* feature provides the functionality required to persist the output-composed model generated at the end of the model composition process. The *optional features* are the types of file format that the output-composed model can be persisted, including UML and UML profile format. The *or features* are represented by the composition strategies, and the comparison strategies, the latter are not shown in the feature model for space constraints. Thus, one (or more) comparison and composition strategy should be selected when a composition tool is derived from the MoCoTo-Arch.

C. MoCoTo architectural components

Figure 4 shows the components that are responsible for implementing the feature model as well as relates them with the features depicted in Figure 3. The small squares located on the left or bottom sides of the components represent this feature-component mapping. For instance, the `C` on the top of the `Comparison` component (Figure 4) indicates that this component contributes to the implementation of the comparison feature. This *design-for-features* is supported by the component-based development, a systematic feature-component mapping and aspect-oriented programming.

This method of decomposing components based on the features allows creating autonomous, well-modularized design elements within a model composition tool, thereby promoting the reuse of previously elicited feature and constructed components. Each component was designed to: (1) be a self-contained module that encapsulates the state and behavior of a set of executable elements, which are responsible for the implementation of one (or more) feature; (2) present emergent behaviors resulting from the interaction of its executable elements, i.e., one or more classes that realize the expected functionalities of the features; and (3) have well-defined

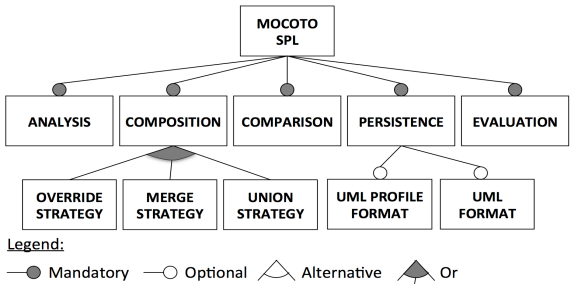

Figure 3. A simplified MoCoTo-Arch feature model.

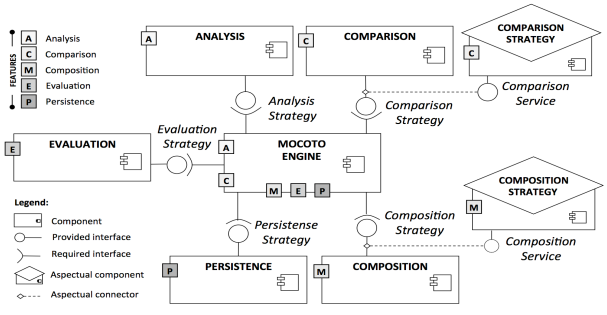

Figure 4. The MoCoTo architectural components.

interfaces, including the provided and required ones. For example, to provide the behavior of matching two input models, the `Comparison` component implements the provided interface, `ComparisonStrategy`. If new components are inserted, then they should implement this interface only. Moreover, Figure 4 focuses on presenting the components as a coherent group of elements implementing one (or more) feature. Each component can be seen as a building block that plays a crucial role within the model composition process.

D. MoCoTo multilayered architecture layers

The logical, multilayered architecture enables us to support a well-modularized design, thereby putting the heterogeneous, crosscutting concerns, previously described in Figure 2, in shape. The architecture is composed by five layers: (1) *Presentation layer* represents the topmost tier of the application gathering the input data required to perform the functionalities and putting out the results to the compositions; *Application layer* encompasses MoCoTo's engine and its operators. It is responsible for orchestrating, along with its operators, the composition process as a whole. As an orchestrator, it plays a pivotal role by providing the principal main entry point, coordinating incoming composition requests, transforming the requests into commands for the operators, and rendering views; (3) *Variability layer* implements the variation points. For this, aspectual components weave the behaviors (or advices) from design elements (from the business logic layer) to the operators (in the application layer). Aspectual components augment the operators with additional or alternative behaviors, i.e. strategies and their rules; (4) *Business Logic layer* defines a family of algorithms that implement the MoCoTo features. These algorithms analyze the input models, seek to find the commonalities and differences between the input models, integrate the commonalities, and then evaluate the output models, and (5) *Infrastructure layer* accommodates the concerns related to handling exception, data access, persistence and logging, which are key crosscutting functionalities to put the composition process in practice.

IV. CASE STUDY

We evaluate this work by implementing a model composition tool based on the MoCoTo architecture. The tool, so-called MoCoTo, is an Eclipse Plug-in that allows a seamless integration with Eclipse Platform. In addition, it makes use of a range of Eclipse modeling technologies, including EMF, UML2, GEF, UML2 tool, to implement all required activities described in the model composition process discussed earlier. MoCoTo ties together these technologies in such a way that makes it easy to use, even for users with little or no Java or XML coding experience. For example, UML2 API reads and filters information from the tags of files written in XML and transforms it to an abstract data model in which input model elements can be manipulated as objects.

V. CONCLUSIONS AND FUTURE WORK

This paper introduced a flexible, component-based architecture for supporting the development of model composition techniques, and an intelligible model composition workflow for aiding developers to comprehend the crucial composition activities and their relationships more properly. We also reported the MoCoTo tools, a composition tools defined over the MoCoTo-Arch. The preliminary results have indicated that the proposed architecture is able to support the development of composition tools for UML models. Although MoCoTo-Arch has shown to-be useful, further empirical studies are still required, other than case study presented, to check their usefulness to compose other models, including business process models, and with different developers, compared to other composition techniques.

The future investigations should seek to answer some questions such as: (1) do developers invest significantly more effort to develop a new composition technique than derive one from MoCoTo-Arch? (2) How effective is MoCoTo to combine realistic, semantically richer design models? (3) Do developers invest more effort to resolve semantic inconsistencies than syntactic ones using a strategy-based composition technique? (4) How do developers observe the benefits of the composition process? Lastly, this work represents a first step in a more ambitious agenda on better supporting the elaboration of model composition techniques.

ACKNOWLEDGMENT

This work was funded by Universal project – CNPq (grant number 480468/2013-3).

REFERENCES

[1] T. Mens, “A state-of-the-art survey on software merging,” IEEE Trans. Softw. Eng. 28(5), 449–562, 2002.

[2] Y. Brun et al., “Proactive Detection of Collaboration Conflicts,” In: 8th SIGSOFT ESEC/FSE, pp. 168-178, Szeged, Hungary, 2011.

[3] J. Whittle, P. Jayaraman, “Synthesizing hierarchical state machines from expressive scenario descriptions,” ACM TOSEM, 19(3), 1–45, 2010.

[4] IBM Rational Software Architecture (IBM RSA), http://www.ibm.com/developerworks/rational/products/rsa/, 2011.

[5] Kolovos et. al., “The Epsilon Book,” http://eclipse.org/epsilon/doc/book/, 2015.

[6] A. Sarma et al., “Palantír: early detection of development conflicts arising from parallel code changes,” IEEE TSE, vol. 99, no.6, 2011.

[7] K. Wieland *et al*., “Turning conflicts into collaboration - concurrent modeling in the early phases of software development,” CSCW: The Journal of Collaborative Computing, 22 (2013), 2-3; 181 - 240.

[8] M. La Rosa *et al*., “Business process model merging: an approach to business process consolidation,” ACM TOSEM, 22(2): 11, 2013.

[9] OMG, UML: Infrastructure version 2.4, August 2011.

[10] K. Farias *et al*, “A flexible strategy-based model comparison approach: bridging the syntactic and semantic gap,” Journal of Universal Computer Science, 15(11):2225-2253, 2009.

[11] P. Jayaraman, J. Whittle, A. Elkhodary, H. Gomaa, “Model Composition in Product Lines and Feature Interaction Detection using Critical Pair Analysis,” MODEL’7, pages 151-165, 2007.

[12] S. She, U. Ryssel, N. Andersen, A. Wasowski, K. Czarnecki, Efficient synthesis of feature models, Information & Software Technology, 56(9): 1122-1143, 2014.

[13] Fleurey et. al., Kompose : A generic model composition tool, http://www.kermeta.org/kompose/, 2015.

[14] S Clarke, Composition of Object-Oriented Software Design Models, Ph.D. Thesis, Dublin City University, January, 2001.

[15] K. Farias, Empirical Evaluation of Effort on Composing Design Models, PhD thesis, Department of Informatics, PUC-Rio, Rio de Janeiro, RJ, Brazil.