Supporting the Composition of UML Component Diagrams

Guilherme Ermel
University of Vale do Rio dos Sinos
São Leopoldo, Brazil
gui_ermel@hotmail.com

Kleinner Farias
University of Vale do Rio dos Sinos
São Leopoldo, Brazil
kleinnerfarias@unisinos.br

Lucian José Gonçales
University of Vale do Rio dos Sinos
São Leopoldo, Brazil
lucianj@edu.unisinos.br

Vinicius Bischoff
University of Vale do Rio dos Sinos
São Leopoldo, Brazil
viniciusbischof@edu.unisinos.br

ABSTRACT

Fast-changing business environments have become enterprise information systems more heterogeneous and complex. This extreme uncertainty leads to continuous development and integration of architecturally relevant components developed in parallel. In this context, the proper composition of such components is critical to reduce the development effort. However, the current composition tools are still considered imprecise and inflexible for this purpose. This article, therefore, proposes MoCoTo, a model composition tool to support the integration of UML component diagrams. It exploits equivalence relationships between the UML component elements to improve integration precision and accuracy. Developers and system analysts can benefit from using MoCoTo when evolving or maintaining architectural models of enterprise information systems. MoCoTo was implemented as an Eclipse platform plug-in. The tool was used to support the composition of architectural components in three realistic evolution scenarios of a Software Product Line. Our preliminary results indicated that MoCoTo was able to integrate architectural models represented with UML component diagrams. The metrics used to evaluate the effectiveness of the proposed tool (i.e., precision, recall and F-measure) presented values higher than 0.6 in all evaluation scenarios.

CCS CONCEPTS

• **Information systems** → *Information systems applications*; *Enterprise applications*;

KEYWORDS

Software Modeling, UML, Software Components, Model Composition, Empirical Studies

ACM Reference Format:
Guilherme Ermel, Kleinner Farias, Lucian José Gonçales, and Vinicius Bischoff. 2018. Supporting the Composition of UML Component Diagrams. In *Proceedings of Simpósio Brasileiro de Sistemas de Informação (SBSI'2018),* Jennifer B. Sartor, Theo D'Hondt, and Wolfgang De Meuter (Eds.). ACM, New York, NY, USA, Article 4, 8 pages. https://doi.org/10.475/123_4

SBSI'2018, June 2018, Caxias do Sul, Rio Grande do Sul Brazil

ACM ISBN 123-4567-24-567/08/06.
https://doi.org/10.475/123_4

1 INTRODUCTION

Nowadays, fast-changing business environments, pressures of shorter time-to-market, and rapidly advancing technologies are ever-present characteristics found in the most development projects of enterprise systems. These characteristics cause constant changes in architectural components [2]. Software developers represent such architectural components using the UML (Unified Modeling Language) component diagram [16], for example.

In the context of collaborative software modeling, developers create and change architectural components (represented using UML (Section 2.1)) in parallel to allow them to concentrate on specific parts of the components relevant to them. In [4], the authors highlight that software modeling has increasingly become a collaborative task. But, at some point, software developers need to bring together the parts of such architectural components created concurrently so that a consolidated view of the software architecture may be created.

In this sense, model composition plays a central role, e.g., evolving architecturally relevant software components to add new features, or even reconciling components changed in parallel. The term *model composition* can be seen as a set of activities to be carried out over two input model, M_A and M_B, to produce an output-desired component, M_{AB}. In practice, an output-composed model with inconsistencies may be produced, M_{CM}, instead of M_{AB}, due to conflict problems. The input models M_A and M_B have often conflicting parts, which are typically resolved improperly by software developers, given the problem at hand. Previous studies (e.g., [4, 14]) highlight that these conflicts are common in real-world settings, and software developers end up investing some extra effort to detect and resolve inconsistencies found in M_{CM} to transform it into M_{AB}.

The problem is that, even though several approaches have been proposed to represent and combine UML component diagrams automatically (e.g., Epsilon and IBM Rational Software Architect (IBM RSA)), they are not effective enough to produce an output-desired component diagram without requiring extra effort. The current composition approaches consider a low number of properties of UML components (e.g., name), rather than all properties defined in the UML metamodel (Section 2.1) to integrate two or more software components. Moreover, they were not built to combine architecturally relevant software components upfront. Instead, they are focused on generic design diagrams (e.g., domain models), overlooking specific elements of UML component diagrams. In [10], the

authors report that the current approaches, such as Astha, Epsilon and IBM Rational Software Architect (IBM RSA), have not shown to be effective to combine design models. In fact, usually M_{CM} and M_{AB} do not match (i.e., $M_{CM} \neq M_{AB}$).

To overcome these shortcomings, this article, therefore, proposes MoCoTo, a tool-supported approach for the composition of UML diagram components. It exploits equivalence relationships between the UML component elements to improve the precision and accuracy of the compositions. Developers and system analysts can benefit from using MoCoTo when evolving or maintaining architectural models of enterprise information systems. MoCoTo was implemented as an Eclipse platform plug-in. The tool was used to support the composition of architectural components in three realistic evolution scenarios of a Software Product Line. Our preliminary results indicated that MoCoTo was able to integrate architectural models represented using UML component diagrams. The metrics used to evaluate the effectiveness of the proposed tool (i.e., precision, recall and F-measure) presented values higher than 0.6 in all evaluation scenarios.

The remainder of the paper is organized as follows. Section 2 provides the main concepts and knowledge that are going to be used and discussed throughout the article. Section 3 presents the proposed approach. Section 4 discusses the implementation and evaluation aspects, describing technologies used to develop the proposed approach and metrics applied to assess it. Section 5 contrasts this study with the current literature. Finally, Section 6 presents some conclusions and future works.

2 BACKGROUND

This section presents the main concepts related to the understanding of our work. For this, Section 2.1 introduces the UML component diagram. Next, Section 2.2 presents a brief definition of model composition, conflicts, inconsistencies, as well as introduces an example of composition.

2.1 UML Component Diagram

The UML component diagram is one of the most used UML diagram [4], which allows developers to represent modules of software systems and their relationships [16]. Each component has one or more provided and/or required interfaces (potentially exposed via ports), and its internal details are hidden and inaccessible other than as provided by its interfaces. Even though the component may be dependent on other elements in terms of interfaces that are required, a component is encapsulated and its dependencies are designed such that it can be treated as independently as possible. A component can be seen as a *modular unit* with well-defined interfaces that is replaceable within its environment [16, 17].

Figure 1 shows the *Order* component, with two provided interfaces (i.e., *Person* and *Invoice*) and two required interfaces (i.e., *Item* and *Tracking*). The provided interface defines the behavior that an architectural component offers to the environment, while the required interface specifies the behavior that the behavior needs to work.

The UML component diagram is defined following a metamodeling approach. This means that each component diagram is produced based on the instance of the UML metamodel. The UML metamodel defines a set of constructs, which can be used to define software systems of arbitrary size and complexity [16]. Any composition approach (Section 3) should thus manipulate each attribute in the UML metamodel for properly combining the input models, M_A and M_B.

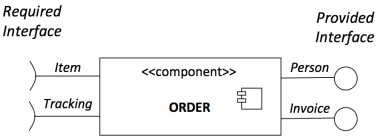

Figure 1: Example of a component [17].

2.2 Model Composition, Conflicts and Inconsistencies

Researchers and practitioners recognize the importance of the model composition in many software engineering activities, e.g., during the evolution of design models to add new features and consolidate models developed in parallel by different software development teams [8, 9].

Composition. As already mentioned in Section 1, the term *composition of UML component diagrams* can be defined as a set of activities that should be performed over two input models, M_A and M_B, so that an output-desired model, M_{AB}, can be generated. To realize the model composition in practice, software developers often make use of composition heuristics (e.g., override and union described in [5]) to produce M_{AB}, which matching the input model elements of M_A and M_B by automatically guessing their semantics and then bring the similar elements together to create a big picture view of the overall design model.

Composition conflicts. It can be defined as conflicting changes assigned to the model elements of M_A and M_B. They are usually detected when software developers seek to fuse the overlapping parts of M_A and M_B.

Inconsistencies. Inconsistencies are unexpected values assigned to the properties (or characteristics) of the design models [7]. For example, *Researcher.isAbstract* = false represents an inconsistency as the expected value is true. Note that when the conflicts are incorrectly resolved, they are converted into inconsistencies in M_{CM}.

The proposed approach (presented in Section 3) addresses the difficulty of developers to overcome two main problems. First, software developers have no support to indicate which model elements should be combined, given a large number of architectural components. Second, developers tend to be unable to understand the accuracy of the composition to be performed upfront, as well as how much effort the derivation of M_{AB} from M_{CM} can demand, given the problem at hand.

3 THE PROPOSED APPROACH

To better explain the proposed approach, we propose a model composition process (Section 3.1) and describe the main features supported by the approach, as well as show the variability of features supported (Section 3.2). Finally, we describe how each feature is mapped into the modules of proposed approach (Section 3.3).

3.1 Model Composition Process

Figure 2 shows the model composition process used to combine two input models, M_A and M_B. We have adopted this process since earlier works [11, 15] have mentioned its utility for composition in general terms. Before producing an output-intended model, the input diagrams go through four stages: (1) analysis, (2) comparison, (3) composition, and (4) evaluation steps. These steps are better specified as follows:

1) Analysis: this step prevents the tools processing incompatible, and inconsistent input models. For this, the approach checks whether (1) the tool supports the type of file of the input models, (2) both input diagrams are of the same type, and (3) the input models do not have any invalid characters. The composition process will finish if the tool does not comply with one of these criteria. If these input models check these three criteria, the input models follows directly to the comparison phase.

2) Comparison: the input diagrams are validated and then the composition approach verifies the equivalence between the elements of the input models. For this, this step receives as inputs, such as valid models from the previous step, an algorithm for computing string equivalence (LCS common substring [6]), the synonym dictionary, the matching strategy, and the threshold. This step is important to determine which elements are equivalent, identifying overlapping parts of the input model elements.

3) Composition: this step aims at integrating the input models based on the outputs produced in the previous step, i.e., similarity matrix, list of equivalent elements, and equivalent elements. Moreover, it makes use of well-established composition strategies (i.e., override, merge, and union) to combine the elements of the input models (M_A, M_B). Therefore, while the matching elements are unified, the no matching ones are added to the resulting model.

4) Evaluation: this step aims at evaluating if there are inconsistencies in the output composed model M_{CM}. The tool receives as input the well-formedness rules, the composed model M_{CM}, and the features defined by the user. The composition process finishes when the well-formedness rules meet the requirements defined by the user in the composed model M_{CM}, and then, the M_{CM} is stored. Otherwise, the composition mechanism applies some transformation rules on M_{CM} to try the overcome the inconsistencies, and then M_{CM} is stored.

After presenting the composition process, the next step is to describe the main features of the proposed composition tool.

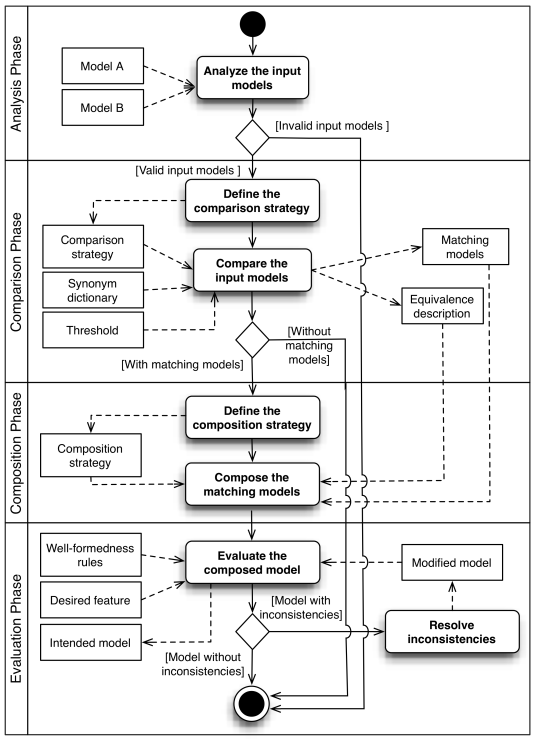

Figure 2: The general view of the model composition process.

3.2 Feature Diagram

Given that MoCoTo tool was designed as a Software Product Line, the proposed feature diagram, shown in Figure 3, can visually demonstrate the units of program construction, so-called *features* [12] — i.e., fine-grained increments in the context of program functions. We have used feature diagram to represent all characteristics supported by the MoCoTo tool. Thus, we can present a compact representation of all its functions of the tool in terms of features. A software in an SPL can be identified by a unique and legal combination of features. Thus, we can produce different MoCoTo tools based on the combination of the features available in Figure 3. Using this diagram it is possible to represent the variability of the MoCoTo tools, as well as upcoming features can be inserted into the current supported feature settings.

Figure 3 presents an excerpt of the feature diagram representing the overview of the main characteristics supported by the proposed architecture. The architecture is formed by the essential features for putting in practice the activities of the model composition process defined in the previous Section. In practice, these features correspond to its main steps of the model composition, as previously described, including Analysis, Comparison, Composition and Evaluation. In addition, the persistence is a concern that spreads over the phases of the model composition process (like a crosscutting concern) as it always need to persist models and files in each phase.

This means that, for example, the analysis feature implements the first phase in the composition process. The notations used to represent Figure 3 may be divided into two main groups, namely basic representation feature (e.g., MoCoTo, Analysis and Comparison) and their relationships. Relationships between a parent feature and its child features shown in Figure 3 can be categorized as: *Mandatory*, child feature is required (represent by a filled circle); *Optional*, child feature is optional (represent by an empty circle); and *Or*, at least one of the sub-features must be selected (represent by a filled bow). Therefore, all versions of the MoCoTo tool instances must have the following features Analysis, Comparison, Composition, Persistence, and Evaluation . The UML profile format and UML format can be present or not in MoCoTo tool instance, being called optional features (represent by an empty circle).

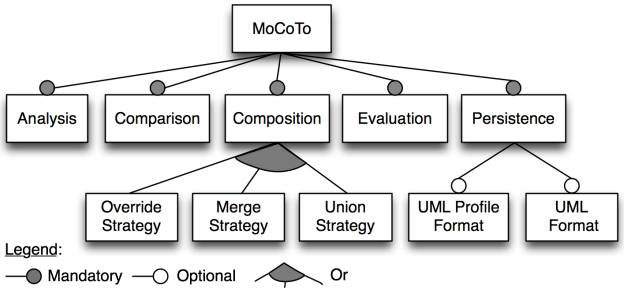

Figure 3: Features of the MoCoTo composition tool.

3.3 Architectural Design Model

As previously mentioned, each feature presented in Figure 3 was mapped into a module, which implements, in practice, each feature of the MoCoTo tool. We have chosen the UML component diagram to represent these modules, which are displayed in Figure 4. This mapping is identified on the small squares located on the left or bottom sides of the components. For example, the letter *M* in the side of Composition component (Figure 4) means that it implements the composition feature (Figure 3).

In practical terms, to make feasible the implementation of well-modularized components we are concerned on building: (1) self-contained components encapsulating the state and behavior of a set of executable elements, which are responsible for the implementation of one feature; and (2) well-defined interfaces. Therefore, to add a new strategy for composing two input models, the new Composition component must implement the provided interface, *CompositionStrategy*.

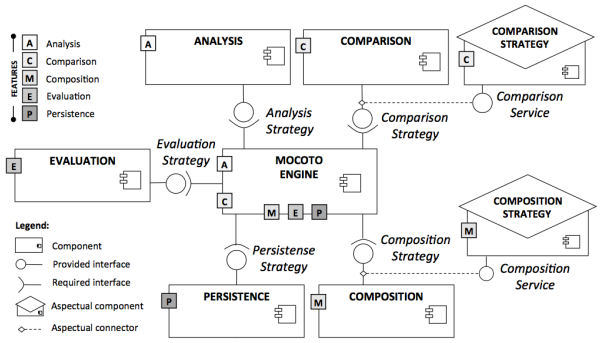

Figure 4: The architectural design of the MoCoTo tool.

4 EVALUATION

This section aims at presenting the implementation aspects of the proposed approach, as well as evaluating it in realistic composition scenarios. First, we present a target case used to evaluate the proposed approach (Section 4.1). After, we described the evaluation procedures (Section 4.2). Then, we describe three usage scenarios (Section 4.3). Finally, we present some additional discussions and limitations of the proposed approach.

Table 1: Description of the metrics applied.

Metric	Description
#Comp	The number of component present.
#IntProv	The number of provided interfaces.
#IntReq	The number of required Interfaces.
#Operat	The number of operations of the interfaces.

4.1 Target Case

We apply the proposed approach to support the evolution of the component diagrams of a software product line, called *Mobile Media SPL* [18, 19]. The diagrams used in this study are adapted from [18]. The Mobile Media SPL supports the manipulation of photos, music, and videos on mobile devices. The choice of the Mobile Media SPL as the target case can be explained by some reasons. First, it went through the four types of change commonly found in software design, including addition, removal, modification and derivation (as described in Section 2.2). Second, the Mobile Media's architectural models are well structured and elaborated by independent developers. Third, the original developers were available to help us to detect and evaluate the composition problems, including syntactic and semantic inconsistencies.

Finally, three evolution scenarios of Mobile Media SPL were considered in our study. In each scenario the focal point was to support the evolution of the product line Mobile Media [19]. We discuss the evaluation method used in our study in the following section.

4.2 Method

Evaluation procedures. The proposed approach was used to realize the three evolution scenarios of the Mobile Media SPL (Section 4.3). The output composed models were evaluated collaboratively by the authors. For this, the following steps were performed. First, the authors examined each composition and extracted information related to likely composition problems, e.g., inconsistencies. After, the observations collected were analyzed and discussed so that a consensus between the authors might be obtained. In addition, the original developers of the Mobile Media SPL might also validate the compositions realized. For this, two evaluation cycles were held.

Evolution scenarios and metrics. With this in mind, the MoCoTo tool was evaluated in three evolution scenarios. Thus, for each step there is a reference model, M_A, and a delta model, M_B, that represents what was developed in that cycle. From these diagrams three experiments were drawn up. In each scenario the proposed approach was used to compose two models, M_A and M_B (e.g., Figure 5), producing a new model M_{CM}. The tool was used with a same configuration, in terms of strategies used, for all composition scenarios. The SDMetrics tool was applied to M_A, M_B, M_{CM} and M_{AB} to compute the indicators: Precision, Recall and F-measure. Table 1 presents the metrics computed.

After presenting the metrics used and explaining the reasons to choose the Mobile Media SPL as the target case, we describe carefully the three case studies.

Table 2: The results obtained in the first experiment.

Metric	M_A	M_B	M_{AB}	M_{CM}	$M_{CM} \cap M_{AB}$	Precision	Recall	F-Measure
#Comp	8	3	8	8	8	1	1	1
#IntProv	7	2	7	7	7	1	1	1
#IntReq	12	2	12	12	12	1	1	1
#Operat	37	20	37	37	37	1	1	1

4.3 Case studies

We evaluate the proposed approach in the context of three evolution scenarios of the Mobile Media SPL (Section 4.1). By doing so, the MoCoTo tool was used to perform three composition scenarios described as follows.

4.3.1 First Composition Scenario. Figure 5 represents the first target composition case supported (i.e., M_A and M_B), while Figure 6 introduces the result of the composition (i.e., M_{AB}). We used the MoCoTo tool to realize the composition expressed through a merge relationship in Figure 5. Thus, the model elements in M_B (i.e., the components *BaseController*, *ImageAcessor*, and *AlgumData*) are accommodated into the diagram M_A. In particular, this composition represents the case in which the elements present in M_B are already in M_A, so the resulting composed model becomes equal to M_A. After performing the composition, the metrics were applied. The collected results are shown in Table 2. This evaluation was performed to show the high precision (described in Table 2) of the tool in case where M_B is a subset of M_A.

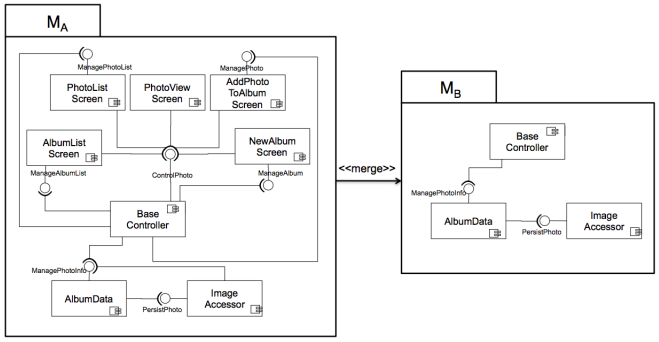

Figure 5: First usage scenario to evaluate the tool.

There is no addition of new elements and interfaces in Figure 5. After the composition of the models M_A and M_B, we can check that there were no change in the model; hence, the metrics for M_{AB} and M_{CM} are equal. We might observe that given that the metric measures in M_{AB} and M_{CM} are equal to $M_{CM} \cap M_{AB}$, the precision and recall as well as their F-measure indicators are equivalent in all cases. This result indicates that the tool worked properly in case where the model elements found in M_B overlap ones found in M_A.

4.3.2 Second Composition Scenario. The second scenario, shown in Figure 7, displays a case in which the model elements need to be accommodated into M_A, not necessarily overlapping model elements found in M_B. Figure 7 presents the input models, while Figure 8 depicts the output desired model. Two new components

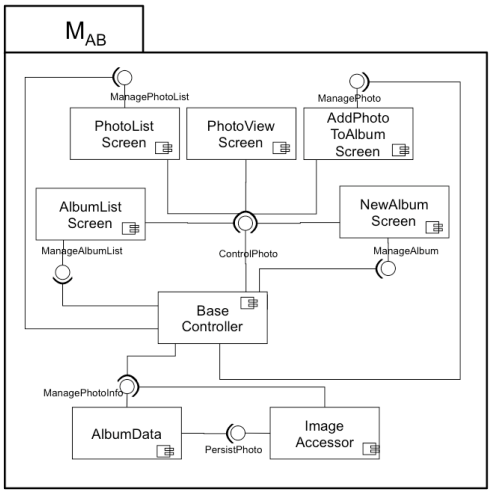

Figure 6: The desired model of the first usage scenario.

can be observed in M_B (i.e., *NewLabelScreen* and *PhotoController*) and their provided and required interfaces. In addition, some operations were added and removed at this stage of development (which are not shown in the illustration). This more realistic scenario allowed us to evaluate the tool in scenarios where it was required to support the integration of overlapping model elements, as well as non-overlapping ones, i.e., the latter will accommodate into the M_A without requiring more severe changes.

Table 3 shows the results obtained after combining the diagrams presented in Figure 7. The lower values of the Precision and F-Measure (compared to the previous experiment) can be explained for the fact the tool is not able to remove undesired model elements found in M_A. A careful analysis of the results has pointed out that the reduction of the precision and F-Measure was strictly motivated for more severe evolution scenarios, in which model elements have some properties modified; for instance, the component *NewAlbumScreen* (Figure 7 in M_A) has its name modified to *NewLabelScreen* in Release 2 (Figure 7 in M_B). The precision demonstrates the correctness of the composition process, we can note that the metrics ranges from 0.89 to 0.96, being the average of the precision equal to 0.91. This means that the proposed tool reached a high-precision rate.

Decomposing the value of the precision metric, we might see that the number of components obtained was 0.9, the number of provided interfaces was 0.92, the number of required interfaces is 0.89, the number of operations was 0.96. Regarding the recall values, found also in Table 3, we might observe a high value, measuring

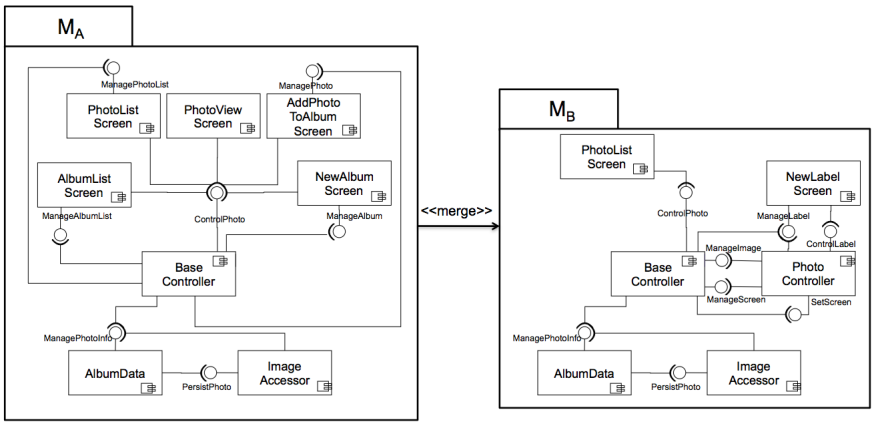

Figure 7: Second usage scenario to evaluate the tool.

Table 3: The results obtained in the second experiment.

Metric	M_A	M_B	M_{AB}	M_{CM}	$M_{CM} \cap M_{AB}$	Precision	Recall	F-Measure
#Comp	8	6	9	10	9	0.9	1	0.95
#IntProv	7	8	11	12	11	0.92	1	0.96
#IntReq	12	11	17	19	17	0.89	1	0.94
#Operat	37	41	47	49	47	0.96	1	0.98

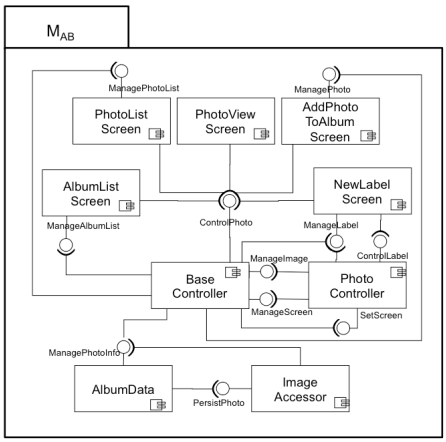

Figure 8: The desired model from the second usage scenario.

1 for the four metrics. Remember that F-Measure aims at analyzing the accuracy in the results extracted during the composition process. Thus, the F-measure obtained was, on average, equal to 0.95, which is distributed as follows: the number of components obtained was 0.95; the number of provided interfaces produced was 0.96; the number of required interfaces was 0.94; and the number of operations was 0.98. Therefore, based on the collected measures for the Precision, Recall and F-measure, we can conclude that the MoCoTo tool was also effective to support the evolution found in the second usage scenario.

4.3.3 Third Composition Scenario. The third usage scenario presented a higher number of changes, including addition, removal, modification and derivation of model elements, compared to previous usage scenarios. Figure 9 presents two UML component diagrams, while Figure 10 displays the desired model. As previously mentioned, the first model, M_A, represents the model receiving a set of new model elements from the second model, M_B. For this, we have used the MoCoTo tool to accommodate the elements from M_B into M_A.

In M_A, the *BaseController* component has the *ControlPhoto* interface, however, in the development cycle this interface has been moved to *PhotoController* component and therefore appears on M_B as shown in Figure 9. Thus, the components that used this interface in M_A should point to the new interface in *PhotoController*. However, the tool does not identify that the *ControlPhoto* interface in M_A is equivalent to one in M_B, as this could just be a new interface with the same name and methods (as well as several *HandleCommand* interfaces in M_B).

Even though the third usage scenario presented more severe changes, leading to more critical changes, the results indicated that the MoCoTo was able to combine M_A and M_B, producing the values of the metrics Precision, Recall and F-Measure higher than 0.6. Table 4 shows the results obtained.

An interesting observation can be seen in the first metric, i.e., the number of components. Where 14 components were created, but only 10 were expected. This occurred because the M_A.*BaseController* and M_B.*PhotoController* did not obtain a similarity equals or greater than the required threshold defined (i.e., 0.7). Thus, M_A.*BaseController*, M_B.. *BaseController*, M_B.*PhotoController*, M_B.*PhotoController* were generated. Thus, in M_{CM} two things can be observed, the first one

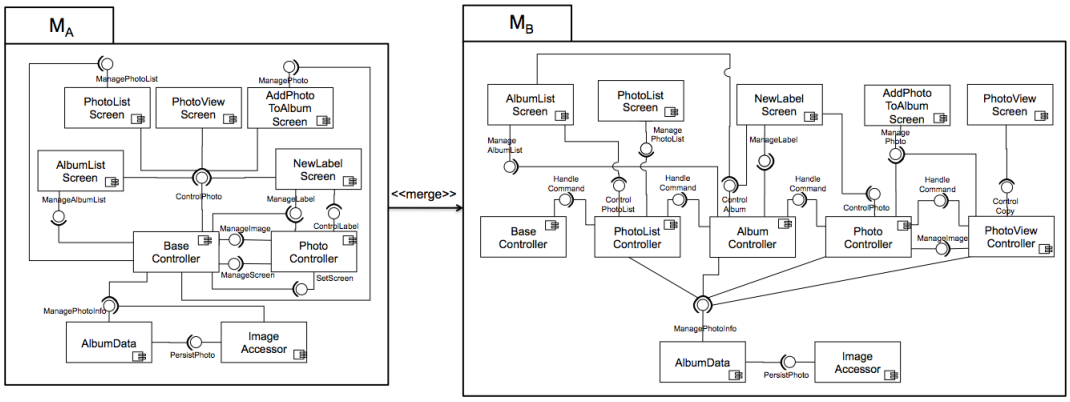

Figure 9: Third usage scenario to evaluate the tool.

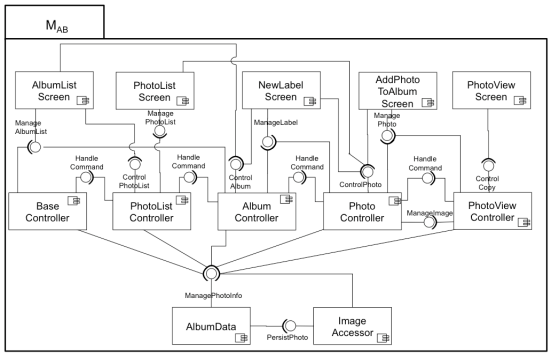

Figure 10: The desired model from the third usage scenario.

is the same characteristic already described in scenario 2, where the tool does not remove components or interfaces. The other feature is that it does not consider a similarity among interfaces of different components. In this sense, the tool does not enable that an interface can be provided by another component in M_{CM}.

Table 4 shows the results obtained in the third experiment. We can observe that the values obtained from 0.64 to 0.76, the average of the precision metrics analyzed was 0.71. The Precision considering the number of components was 0.71, the number of provided interfaces was 0.75, the number of required Interfaces was 0.64, and the number of operations was 0.76.

The recall verifies if all the elements were generated, the results extracted show on average 92.75% efficiency distributed as follows: 83% of the desired components were obtained; 100% of the provided interfaces were produced; 88% of the required Interfaces were produced; 100% of the operations (i.e., method) were produced. Finally, f-measure, whose objective is to analyze the accuracy to the results extracted during the composition process, obtained an average of 80.75% distributed as follows: the number of components obtained was 77%; the number of provided interfaces was 86%; the number of required interfaces was 74%; the number of operations was 86%. Therefore, the collected results indicate that the MoCoTo tool was effective to support the evolution of architectural components throughout compositions.

4.4 Discussion and Study Limitations

Although the proposed tool has obtained good results in our initial empirical studies, we do no claim any generalization of the results. We highlight that the MoToCo tool might require more composition effort in more severe composition scenarios where widely-scoped architecture evolutions often happen, such as the refinement of the MVC (Model-View-Controller) architecture style of the MobileMedia SPL. In part, this can be explained for the fact the name-based model comparison used cannot be able to recognize more intricate equivalence relationships between the model elements of M_A and M_B. We have observed that the comparison strategy is rather restrictive whenever there is a 1:N correspondence relationship between elements of M_A and M_B.

An example of the 1:N relationship category encompassed the required interface *ControlPhoto* of the *AlbumListScreen* component. This interface was decomposed into two new required interfaces *ControlAlbum* and *ControlPhotoList*, thereby characterizing an 1:2 relationship. For this particular case, the name-based model comparison should be able to recognize that *ControlAlbum* and *ControlPhotoList* are equivalent to *ControlPhoto*. However, in the output-composed model produced the *AlbumListScreen* component, providing duplicated services to the environment. According to UML metamodel [16], this duplication can be characterized as a model inconsistency. The presence of inconsistencies is the responsible for decreasing the values of the metrics Precision, Recall and F-measure.

Our idea initially would be that developers could provide a set of well-formedness rules of the design models themselves (e.g., the type relationship between components). However, these rules cannot yet be entered into the tool nor evaluated. Developer should do this manually. Hence, the transformation rules cannot be applied to M_{CM} to try the overcome the inconsistencies.

Finally, we have studied one facet supported by the MoCoTo tool in this study: the use of model composition in adding new features to architectural components of an SPL. Even though SPLs commonly involve model composition activities and, while we

Table 4: The collected results in the third experiment.

Metric	M_A	M_B	M_{AB}	M_{CM}	$M_{CM} \cap M_{AB}$	Precision	Recall	F-Measure
#Comp	9	12	12	14	10	0.71	0.83	0.77
#IntProv	11	15	15	20	15	0.75	1	0.86
#IntReq	17	20	26	36	23	0.64	0.88	0.74
#Operat	47	56	56	74	56	0.76	1	0.86

believe the kinds of model composition in SPLs are representative of the broader issues, we make no claims about the generality of our results beyond model composition of SPL of enterprise systems. We show the results for model compositions of three releases of an SPL. In each release, models for the new features are composed with the models for existing features. We analyze, for each release, how close the output composed model produced is from the desired model (Section 4).

5 RELATED WORK

Literature reports developers use compositions techniques to combine UML class diagrams frequently [3]. However, there is a lack of studies concerned on supporting architecturally relevant software components, as well as demonstrating their precision.

In [1], the authors proposed a solution for comparing two input class diagrams. For this, the authors made use of a greedy algorithm to limit the search-scope for equivalent models. Thus, this technique compared two input UML class diagrams, instead of comparing and composing UML component diagrams. In [13], an approach was developed for a similar purpose, namely GaMMa Meta-model Matching tool. However, the tool has a different purpose, i.e., comparing metamodels rather than composing component diagrams.

Finally, the current approaches failed in providing tool support for merging architectural components. Rather, they focused on solving the composition of traditional diagrams such as UML class diagrams. To sum up, this study is the first work to propose a technique for combining UML component diagrams and running a pilot study to evaluate the tool's precision through realistic scenarios.

6 CONCLUSIONS AND FUTURE WORK

This article proposed MoCoTo, a tool-supported composition approach for integration of architectural components. Our work advanced the state of the art by proposing a model composition process, designing a composition approach as an SPL following well-known design-for-change principles, coming up with a strategy-based model composition approach based on a flexible, multi-layer architecture and providing an approach with a seamless integration with the Eclipse platform.

The preliminary results indicated that the proposed tool was effective to support the composition of UML component diagrams. In fact, all three experiments performed presented a high value of the Precision, Recall and F-Measure metrics. Although the MoCoTo tool have shown to be useful, further empirical studies are still required in other contexts.

Upcoming investigations should seek to answer the following questions: (1) do developers invest significantly more effort to combine design models of enterprise information systems using the proposed tool?; (2) how effective is the proposed approach to combine realistic, semantically richer design models (e.g., business process models)?; (3) do developers invest more effort to resolve semantic inconsistencies than syntactic ones using a strategy-based composition technique? Lastly, this work represents a first step in a more ambitious agenda on better supporting the elaboration of more effective composition techniques to support the integration of design models of enterprise information systems.

REFERENCES

[1] M. Al-Khiaty and M. Ahmed. 2014. Similarity assessment of UML class diagrams using simulated annealing. In *Software Engineering and Service Science (ICSESS), 2014 5th IEEE International Conference on*. IEEE, 19–23.

[2] Y. Alotaibi and F. Liu. 2016. Survey of business process management: challenges and solutions. *Enterprise Information Systems* 11, 8 (2016), 1119–1153. https://doi.org/10.1080/17517575.2016.1161238

[3] K. Altmanninger, M. Seidl, and M. Wimmer. 2009. A survey on model versioning approaches. *International Journal of Web Information Systems* 5, 3 (2009), 271–304.

[4] M. Chaudron, W. Heijstek, and A. Nugroho. 2012. How effective is UML modelling? an empirical perspective on costs and benefits. *Software and Systems Modelling* 12 (2012), 571–580.

[5] S. Clarke and R. Walker. 2001. Composition patterns: An approach to designing reusable aspects. In *Proceedings of the 23rd international conference on Software engineering*. IEEE Computer Society, 5–14.

[6] T. H. Cormen. 2009. *Introduction to algorithms*. MIT press.

[7] Weber et al. 2016. Detecting Inconsistencies in Multi-view UML Models. *International Journal of Computer Science and Software Engineering (IJCSSE)* 5, 12 (December 2016), 260–264.

[8] K. Farias. 2012. *Empirical Evaluation of Effort on Composing Design Models*. Ph.D. Dissertation. PUC-Rio, Brazil.

[9] K. Farias, A. Garcia, J. Whittle, C. Chavez, and C. Lucena. 2015. Evaluating the effort of composing design models: a controlled experiment. *Software & Systems Modeling* 14, 4 (2015), 1349–1365.

[10] K. Farias, A. Garcia, J. Whittle, and C. Lucena. 2013. Analyzing the Effort of Composing Design Models of Large-Scale Software in Industrial Case Studies. In *16th International Conference on Model-Driven Engineering Languages and Systems*. Miami, FL, USA, 639–655.

[11] K. Farias, L. Gonçales, M. Scholl, T. Oliveira, and M. Veronez. 2015. Toward an Architecture for Model Composition Techniques. In *27th International Conference on Software Engineering and Knowledge Engineering*. Pittsburgh, USA, 656–659.

[12] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson. 1990. *Feature-oriented domain analysis (FODA) feasibility study*. Technical Report. DTIC Document.

[13] M. Kessentini, A. Ouni, P. Langer, M. Wimmer, and S. Bechikh. 2014. Search-based metamodel matching with structural and syntactic measures. *Journal of Systems and Software* 97 (2014), 1–14.

[14] M. La Rosa, M. Dumas, R. Uba, and R. Dijkman. 2013. Business Process Model Merging: An Approach to Business Process Consolidation. *ACM Trans. Softw. Eng. Methodol.* 22, 2, Article 11 (March 2013), 42 pages. https://doi.org/10.1145/2430545.2430547

[15] K. Oliveira. 2008. *Composição de UML Profiles*. Master's thesis. Pontifícia Universidade Católica do Rio Grande do Sul.

[16] OMG. 2017. Unified Modeling Language: Infrastructure, Version 2.5.1. Available: https://www.omg.org/spec/UML/2.5.1/.

[17] J. Rumbaugh, I. Jacobson, and G. Booch. 2004. *Unified modeling language reference manual, the*. Pearson Higher Education.

[18] C. N. SantAnna. 2018. *On the modularity of aspect-oriented design: A concern-driven measurement approach*. Ph.D. Dissertation. PUC-Rio, Rio de Janeiro, Brazil.

[19] L. Tizzei, M. Dias, C. Rubira, A. Garcia, and J. Lee. 2011. Components Meet Aspects. *Information and Software Technology* 53 (2011), 121–136.