Analyzing the Effort of Composing Design Models of Large-Scale Software in Industrial Case Studies

Kleinner Farias[1], Alessandro Garcia[2], Jon Whittle[3], and Carlos Lucena[2]

[1] PIPCA, University of Vale do Rio dos Sinos (Unisinos), São Leopoldo, RS, Brazil
kleinnerfarias@unisinos.br
[2] OPUS Research Group/LES, Informatics Department, PUC-Rio, RJ, Brazil
{afgarcia,lucena}@inf.puc-rio.br
[3] School of Computing and Communications, Lancaster University, UK
whittle@comp.lancs.ac.uk

Abstract. The importance of model composition in model-centric software development is well recognized by researchers and practitioners. However, little is known about the critical factors influencing the effort that developers invest to combine design models, detect and resolve inconsistencies in practice. This paper, therefore, reports on five industrial case studies where the model composition was used to evolve and reconcile large-scale design models. These studies aim at: (1) gathering empirical evidence about the extent of composition effort when realizing different categories of changes, and (2) identifying and analyzing their influential factors. A series of 297 evolution scenarios was performed on the target systems, leading to more than 2 million compositions of model elements. Our findings suggest that: the inconsistency resolution effort is much higher than the upfront effort to apply the composition technique and detect inconsistencies; the developer's reputation significantly influences the resolution of conflicting changes; and the evolutions dominated by additions required less effort.

Keywords: Model composition effort, empirical studies, effort measurement.

1 Introduction

Model composition plays a central role in many software engineering activities, e.g. reconciling models developed in parallel by different development teams [11][18][33], and evolving models to add new features [14][15][32]. In collaborative software development [30], for example, separate development teams may concurrently work on a partial model of an overall design model to allow them to concentrate more effectively on parts of the model relevant to them. However, at some point, it is necessary to bring these models together to generate a "big picture" view of the overall design model. So, there has been a significant body of research into defining model composition techniques in the area of governance and management of enterprise design models [9], software configuration management [11], and the composition of software product lines [25][28].

A. Moreira et al. (Eds.): MODELS 2013, LNCS 8107, pp. 639–655, 2013.

Consequently, both academia and industry are increasingly concerned in developing effective techniques for composing design models (e.g. [3-8][10-17]). Unfortunately, both commercial and academic model composition techniques suffer from composition conflict problems [10][11][12]. That is, models to-be composed conflict with each other and developers are usually unable to deal with the conflicting changes. Hence, these conflicts may be transformed into inconsistencies in the output composed model [24][26].

The current composition techniques cannot automatically resolve these inconsistencies [24][27][29]. The reason is that the inconsistency resolution relies on an understanding of what the models actually mean. This semantic information is typically not included in any formal way in the design models. Consequently, developers must invest some effort to manually detect and resolve these inconsistencies. The problem is that high effort compromises the potential benefits of using model composition techniques, such as gains in productivity. To date, however, nothing has been done to *quantify* the composition effort and *characterize* the factors that can influence the developers' effort in practice. Hence, developers cannot adopt or assess model composition based on practical, evidence-based knowledge from experimental studies.

The goal of this paper, therefore, is to report on five industrial exploratory case studies that aimed at (1) providing empirical evidence about model composition effort, and (2) describing the influential factors that affected the developers' effort. These studies were performed in the context of using model composition to evolve design models of five large-scale software systems. During 56 weeks, 297 evolution scenarios were performed, leading to 2.288.393 compositions between modules, classes, interfaces, and their relationships. We draw the conclusions from quantitative and qualitative investigations including the use of metrics, interviews, and observational studies. We investigate the composition phenomena in their context, stressing the use of multiple sources of evidence, and making clear the boundary between the identified phenomenon and its context.

The remainder of the paper is organized as follows. Section 2 introduces the main concepts used throughout the paper. Section 3 presents the empirical methodology. Section 4 discusses the study results. Section 5 contrasts our study with related work. Finally, Section 6 presents some concluding remarks and future work.

2 Background

2.1 Model Composition Tasks and Effort

The term model composition refers to a set of activities that should be performed over two (or more) input models, M_A and M_B, in order to produce an output intended model, M_{AB}. M_A is the base model while M_B is the delta model that has the needed changes to transform M_A into M_{AB}. Developers use composition algorithms to produce M_{AB}. These algorithms are responsible for defining the model composition semantics. In practice, these algorithms are unable to generate M_{AB} in all cases due to some influential factors (Section 4.2). Consequently, an output composed model, M_{CM}, is produced instead of M_{AB}.

We use M_{CM} and M_{AB} to differentiate between the output composed model, which has inconsistencies and the model desired by developers, respectively. In practice, these models do not often match ($M_{CM} \neq M_{AB}$) because the input models, M_A and M_B, have some conflicting changes. However, usually it is not always possible to deal with all conflicts properly given the problem at hand [12][32][33]. The problem is that syntactic and semantic information should be considered, but they are rarely represented in a formal way. Rather, they are represented in natural language. Consequently, some conflicting changes are transformed into inconsistencies in M_{CM}.

With this in mind, the model composition effort can be defined, as the effort required to produce M_{AB} from M_A and M_B. Fig. 1 states the effort equation. The equation makes it explicit that the composition effort is based on the effort to perform three key composition tasks such as: (i) $f(M_A,M_B)$: the effort to apply a model composition technique; (ii) $diff(M_{CM},M_{AB})$: the effort to detect inconsistencies in the composed model; (iii) $g(M_{CM})$: the effort to resolve inconsistencies i.e., the effort to transform M_{CM} into the intended model (M_{AB}). Note that if M_{CM} is equal to M_{AB}, then $diff(M_{CM},M_{AB}) = 0$ and $g(M_{CM}) = 0$. Otherwise, $diff(M_{CM},M_{AB}) > 0$ and $g(M_{CM}) > 0$. These variables are counted in minutes in our study.

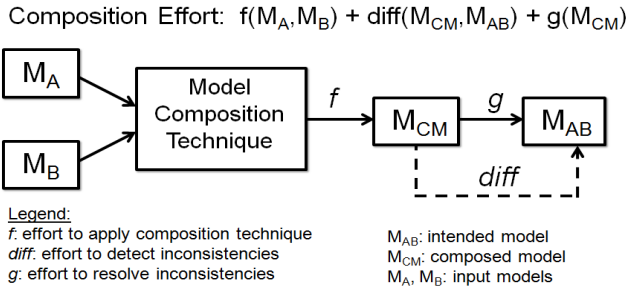

Fig. 1. Model composition effort: an equation

2.2 Composition Conflict and Inconsistency

Composition conflicts arise when contradicting values are assigned to model element's properties. Usually these contractions happen when teamwork members edit such properties in parallel and they are not aware of the changes. Two types of properties can be affected: *syntactic* and *semantic* properties. While the syntactic properties are defined in the modeling language's metamodel [36], the developers are ought to specify the (static and behavioral) semantic properties. Developers should determine which contradicting values assigned to these properties will remain. For example, a developer should define if a class A will be concrete (i.e. *A.isAbstarct* = false) or abstract (i.e. *A.isAbstarct* = true). The output intended class A will be produced, if and only if, this decision is done correctly; otherwise, the output composed class A will be inconsistent. In practical terms, these inconsistencies are unexpected values attributed to model element's properties e.g., A.isAbstract = false instead of the expected value true. Two broad categories of inconsistencies are usually present in output models of our study, namely syntactic and semantic inconsistencies.

Syntactic inconsistencies emerged when any output composed model elements did not conform to the rules defined in the modeling language's metamodel. For example, a package UML cannot have UML classes with the same name. Another example would be all relationship should have the client and supplier defined. Semantic inconsistencies emerged when the meaning of the composed model elements does not match with the meaning of the elements of the intended model. For instance, an inconsistency occurs when functionalities found in M_{CM} are not found in M_{AB}, or when model elements assume a meaning that is no longer expected or valid. The presence of both types of inconsistencies affects the correctness of the composed model.

3 Study Methodology

3.1 Objective and Research Questions

This study aims at gathering knowledge about the values that the composition effort's variables (Fig. 1) can assume in real-world settings. As these variables may be affected by some influential factors, this work also attempts to reveal and characterize these factors. With these aims in mind, we formulate two research questions:

- RQ1: What is the effort to compose design models?
- RQ2: What are the factors that affect composition effort?

3.2 Context and Case Studies

As previously mentioned, during 56 weeks, 297 evolution scenarios were performed leading to 2.288.393 compositions between modules, classes, interfaces, and relationships. All five cases differ in terms of their size, number of participants, and application domain. We present a brief description of the five systems used as follows:

1. *System AL (SysAL)*: controls and manages the importation and exportation of products.
2. *System Band (SysBand)*: a logistics system that manages the flow of goods.
3. *System GR (SysGR)*: supports weather forecast and controls environmental catastrophes.
4. *System Mar (SysMar)*: simulates the extraction of oil from deep ocean areas.
5. *System PR (SysPR)*: a logistics system for refineries.

They were chosen based on some reasons presented in the following. First, they are characterized as typical, revelatory [2], and encompassed UML class and sequence diagrams, use case specifications, architectural diagrams, glossary of domain terms, and business rules. Still, they are representative of complex software systems, which were initially unknown by the developers. This characterizes a typical situation where maintainers are not the initial developers of the system.

Second, the subjects used IBM Rational Software Architect (RSA) [16], a robust modeling tool to create and compose design models. The IBM RSA was used due to: (1) the implementation robustness of its composition algorithms; (2) the tight integration with the Eclipse IDE; and (3) the tool had been already adopted in previous successful projects. Additionally, all cases used a bug tracking system, i.e., JIRA [37],

with which it was possible to coordinate the developers' tasks, specifically during the creation of the design models and review of the models.

Finally, industrial case studies avoid one of the main criticisms of case studies in software engineering regarding the degree of realism of the studies. Thus, we believe that the collected data are representative of developers with industrial skills.

3.3 Subjects

In total, 12 subjects were recruited based on convenience [2]. Table 1 describes the subjects' background. We analyzed the level of *theoretical knowledge* and practical experience of these subjects. The subjects had, on average, 120 hours of courses (lecture and laboratory) considering theoretical issues about software engineering, including object-oriented programming, software architecture, and software modeling using UML. This can be seen, in part, as an intensive UML-specific training. The subjects also had a considerable practical experience, which was acquired from previous software development projects. The data show that the subjects fulfil the requirements in terms of age, education, and experience. The knowledge and experience sharing help subjects solve the composition problems more properly. All subjects were familiar with IBM RSA. Therefore, we are confident that the subjects had the required training, theoretical knowledge and practical experience about model composition to get rid of any threat to the vitality of our findings.

Table 1. Descriptive statistics: subjects' background

Variables	Mean	SD	Min	25th	Med	75th	Max
Age	25.3	4.47	21	22	24.5	27	38
Degree	2.16	1.06	1	1	2	3	4
Graduation year	2006.4	4.8	1992	2005.25	2006.5	2010	2010
Years of study at university	5.75	2.8	3	3	5	7.5	12
YOEW UML	1	1.4	1	1.25	3	4.75	5
YOEW Java	4.5	1.84	2	2.5	4	6.75	7
Used IBM RSA (1 or 0)	1	1	1	1	1	1	1
YOEW software development	5	3.6	2	2.25	4.5	5.75	16
Hours of software modeling	98.33	40.38	60	60	90	120	180
Hours of OO programming	156.66	89	80	80	130	225	360
Hours of software design	130	53.85	80	80	120	190	220

Degree: 1 = Student, 2 = Bachelors, 3 = Masters, 4 = PhD, YOEW = Years of experience with, Med: Median, SD = Standard Deviation, 25th = lower quartile, 75th = upper quartile

3.4 Study Design

The study design is characterized as a *holistic case study* [1][2], where contemporary phenomena of model composition are studied as a whole in their real-life contexts. Five industrial case studies were performed to investigate RQ1 and RQ2. The subjects were *randomly* and *equally* distributed to the five studies, following a within-subjects design [1]. The study had a set of activities that were organized in three phases. In each study, the subjects used the IBM Rational Sofwtare Architect to create and combine the design models. Fig. 2 shows through an experimental process how the three phases were organized. The activities are further described as follows.

Firstly, the issues are created and submitted to JIRA, an issue tracking system. After opening an issue, the developers may perform three activities, including the creation of design models, detection and resolution of inconsistencies.

Training. All subjects received training to ensure they acquired the needed familiarity with the model composition technique.

Apply Composition Technique. The models used in our study were UML class and sequence diagrams. Table 2 shows some metrics about the models used. The subjects create UML class and sequence diagrams using IBM RSA. Both diagrams were elaborated regarding the specifications of use cases and following the best modeling practices. Thus, the participants composed M_A and M_B taking into account the use case specifications. Note that M_B (delta model) represented the changes to be submitted to the repository. The measure of application effort (time in minutes) was collected during this activity. In addition, the composed model, video and audio records represent the outputs of this activity. The video and audio records were later used during the qualitative analyses (Section 4). It is important to point out that a participant (subject x) that produced an M_{CM} was discouraged from detecting inconsistencies in it to avoid bias; thus, another participant (subject n-x) was responsible for detecting and resolving the inconsistencies in M_{CM} in order to produce M_{AB}.

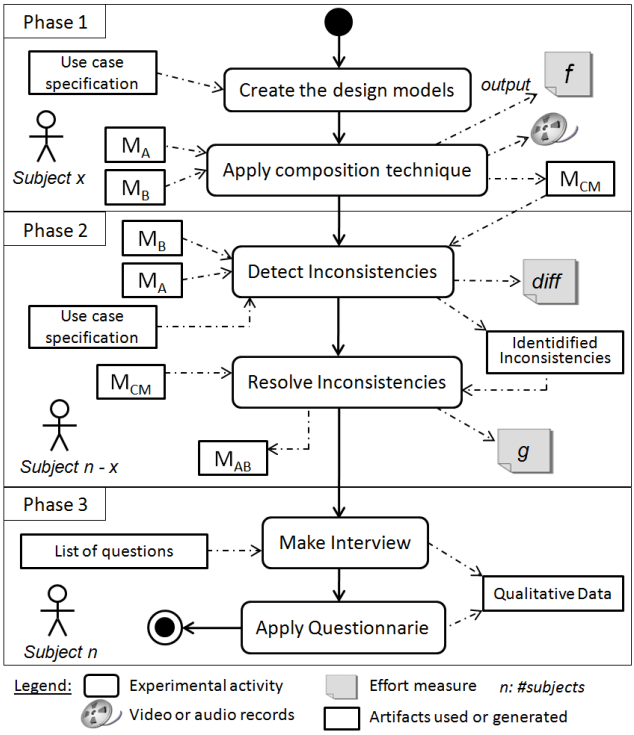

Fig. 2. The experimental process

Detect Inconsistencies. Subjects reviewed M_{CM} for detecting inconsistencies. To this end, they checked if M_{CM} had the changes described in the use case specification. They used the IBM RSA's model validation mechanism to identify syntactic inconsistencies. As a result of this activity, we have the measure of detection effort (time in minutes), and video and audio records.

Resolve Inconsistencies. The subjects resolved the inconsistencies localized in order to produce M_{AB}. In practical terms, they added, removed, or modified some existing model elements to solve them. The resolution effort was also measured (time in minutes) and the video and audios were recorded. After addressing the model inconsistencies, the developers submitted the intended model to the repository. Thus, the compositions were executed in two moments: after the original creation of the models and after resolving the inconsistencies. All model versions were registered using a version control system, thereby allowing a systematic historical analysis of the compositions, M_{CM}.

Make Interview and Answer Questionnaire. Some interviews were conducted with the purpose of collecting qualitative data. The subjects also filled out a questionnaire. These procedures allowed us to collect information about their background (i.e., their academic background and work experience) and apply some inquisitive questions.

Table 2. The collected measures of the design models used

Metrics	SysAL	SysBand	SysGR	SysMar	SysPR
#classes	316	892	1394	2828	1173
#attributes	1732	3349	8424	9689	3808
#operations	3479	7590	10608	23722	9111
#interfaces	18	83	143	223	93
#packages	34	166	175	345	187
#afferent coupling of the packages	278	1147	1632	4044	2329
#efferent coupling of the packages	235	996	1278	2723	1451
#abstractness of the packages.	9.58	50.45	36.9	66.5	51.9
#weeks	6	15	8	17	10
#developers	3	7	2	7	4
#evolutions scenarios	6	95	55	64	77

#: the number of or degree of all, Sys: system

4 Study Results

This section presents the study results about the composition effort variables (RQ1) and explains the factors that we found to influence the composition effort in our study (RQ2).

4.1 RQ1: Composition Effort Analysis

Application Effort. Table 3 shows a descriptive statistics about the application effort. The results indicate that effort to compose models was, on average, 3.17 minutes and 4.43 minutes in SysBand and SysMar projects, respectively. Given the complexity and

the size of the design models in question, these central tendency measures are in fact low values. For example, a developer spent just around 4 minutes to submit the most complex evolving changes to the repository in the SysMar project. In addition, the median measures follow these trends: 3 minutes and 3.12 minutes into the SysBand and Marlin project, respectively. Thus, these measures imply *that the required effort to apply the semi-automated composition technique is low even for large-scale models*. Consequently, it is possible to advocate model composition as appropriate to support collaborative software modeling in which resources and time are usually tight.

In general, we observed that there was no significant variation on developers' application effort. *Developers' effort tends to be similar rather than spreading out over a large range of values*. There were a few exceptions as we are going to discuss below. With 1.55 and 1.58 minutes, the standard deviation measures indicate that in the majority of the model composition sessions the developers spent an effort near 3.17 minutes or 4.43 minutes. These results can help developers to better estimate the effort by establishing thresholds, and check if the effort spent by developers is an expected value (or not).

Table 3. Descriptive statistics for application effort

Cases	N	Mean	SD	Min	25th	Med	75th	Max
SysMar	40	4.73	4.52	0.25	2	3.2	6.79	22
SysBand	69	3.29	1.93	0.83	2	3	4	14.2

N = number of compositions, SD = standard deviation, Min = minimum,
25th = first quartile; Med = median, 75th: third quartile, Max: maximum.

Fig. 3 distributes the collected sample in six effort ranges. These ranges in the histogram systematically group the cases of application effort. The axis-y of the histogram represents the number of compositions, while the axis-x captures the ranges of effort. The main feature is that: *the presence of a distribution pattern of the application effort through the ranges of effort*. The three low-effort categories (i.e., $t < 2$, $2 \leq t < 4$, and $4 \leq t < 6$) represent the most likely ranges of effort that developers invest to compose the input models. The number of cases falling into these categories is equal to 29 (in SysMar) and 64 (in SysBand), representing 72.5% and 92.75% of the composition cases, respectively.

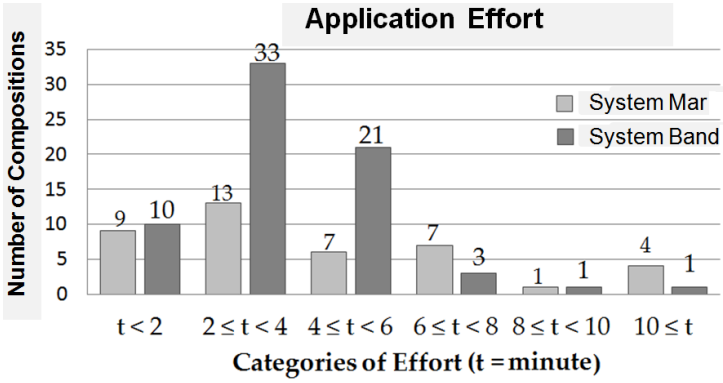

Fig. 3. Histogram of the application effort measures

On the other hand, the number of cases in the high-effort categories (i.e., $6 \leq t < 8$, $8 \leq t < 10$ and $10 \leq t$) is equal to 12 (in Marlin) and 5 (in SysBand), comprising 17.39 % and 12.5% of the cases respectively. The number of composition cases in the low-effort categories outnumbers the amount of cases in the high-effort categories, comprising more than 70% and 90% of the cases in the SysMar and SysBand projects, respectively. On the other hand, the number of cases in the high-effort categories was by around 30% (in Marlin) and 7.25 % (in SysBand). In practice, these results mean that developers spent less than 6 minutes in 85.32% of the full set of composition cases, and only 14.68% of the cases required more than 6 minutes.

Detection Effort. Table 4 shows a descriptive statistics about the effort spent to detect inconsistencies. A careful analysis indicated that some interesting features were observed. First, *the most experienced developers spent 23.2% less effort to detect inconsistencies than less experienced developers*. This observation was derived from the comparison of the medians in the SysMar and SysBand cases. This observation is also confirmed by the means' values. In this case, the most experienced developers invested 38.57% less effort to detect inconsistencies than less experienced developers.

Second, we also found that *the higher the number of teamwork members, the higher the effort to localize inconsistencies*. Comparing the number of teamwork members of the projects, we could observe that the developers of the SysMar and SysBand projects, both with 7 developers, invested a higher amount of effort to detect inconsistencies than the developers of the SysGR and SysPR systems (with 2 and 4 developers, respectively). For example, the developers spent 49.46% more effort (by about 3.45) to detect inconsistencies in the SysMar project than in SysGR project, by taking the medians 6.55 and 3.31 into account. This observation was also reinforced when we compare the SysMar and SysPR projects. That is, SysMar's developers spent 64.27% more effort (by about 4.21) to localize the inconsistencies; this difference is observed by comparing the medians 6.55 and 2.34, respectively. Therefore, the projects with a higher number of developers had to invest the double of effort to localize the inconsistencies.

Third, *the higher the number of inconsistencies in behavioral models, the higher the effort to detect inconsistencies*. Even though certain projects (e.g., System A) had a lower number of developers, a number of inconsistencies were concentrated on behavioral models, i.e. sequence diagrams in our case. The key problem highlighted by developers was that the behavioral models require an additional effort to go through the execution flows. An association in a structural model (e.g., class diagram) represents essentially one relationship between two classes. On the other hand, in a sequence diagram, which represents the interaction between the instances of these classes, the counterpart of the simple association is represented by n interactions (i.e. several messages exchanged between the objects). The problem is that developers must check each interaction.

Another finding is that *the higher the distribution of inconsistencies in different models, the higher the effort to identify them*. In the case studies, the systems were strongly decomposed in different concerns. These concerns were called "conceptual areas" by the developers. This unit of modularization brings together application domain concerns in a same package. The biggest problem arises when the inconsistencies in a conceptual area give rise to several inconsistencies, and hence affecting many other model elements located in other conceptual areas, thereby leading to

ripple effects. This propagation is inevitable as there are usually some relationships between these units of modularization. Hence, developers often had to identify inconsistencies in the model elements of the conceptual areas they have from limited to none knowledge. Note that during the case studies the developers created diagrams related to a specific concern of the system (specified in use cases), and these diagrams were grouped in a conceptual area (similar to a package). Thus, the lack of knowledge about the model elements in the unknown conceptual area led developers to invest an extra effort to detect and resolve the inconsistencies.

Table 4. Descriptive statistics for detection effort

Cases	N	Mean	SD	Min	25th	Med	75th	Max
SysMar	63	7.57	5.1	0.54	2.45	6.55	12.49	16.54
SysBand	86	4.65	2.39	0.36	2.37	5.03	6.38	9.21
SysGR	24	3.66	1.52	1.32	2.67	3.31	4.16	7.39
SysPR	44	2.91	1.75	1.04	1.39	2.34	4.12	7.15
System A	6	12.37	4.2	5.26	8.25	13.15	16.36	17.37

N = number of compositions, SD = standard deviation, Min = minimum,
25th = first quartile; Med = median, 75th: third quartile, Max: maximum.

Resolution Effort (g). Table 5 shows a descriptive statistics of the inconsistency resolution effort. A key finding is that *the developers invest more effort to resolve inconsistencies than to both apply the model composition technique and detect the inconsistencies*. This can be explained based on several observations. First, in the SysMar project, for example, the teamwork members spent 64.91% more effort resolving inconsistencies than applying the model composition technique. This difference comprises the comparison between the medians 3.2 (application) and 9.12 (resolution). This difference becomes more explicit when we consider the values of the mean. This evidence is reinforced by the SysBand project. The resolution of inconsistencies consumes almost three times more effort than the application of the composition technique, if we compare the medians 3.2 (application) and 9.12 (resolution). The difference between the application and resolution effort becomes higher when we consider the value of the mean, i.e. jumping significantly their values from 64.91% to 88.40% (in SysMar) and from 80.31% to 88.35% (in SysBand).

Second, in SysMar project, the inconsistency resolution consumed 28.17% more effort than the inconsistency detection. This comprises the difference between the medians 6.55 and 9.12. The results in the SysBand project followed the same trend. Developers spent 66.99 percent more effort with inconsistency resolution than with inconsistency detection, when compared with the medians 5.03 and 15.24. Considering the mean, this difference of effort becomes more evident, leaping abruptly from 28.17 percent to 81.44 percent (in SysMar) and from 66.99 percent to 83.42 percent (in SysBand). Analyzing the collected data from the *SysGR* and *SysAL* projects, this observation is also confirmed. For example, the resolution effort is 82.98 percent and 54.96 percent higher than the detection effort in *SysGR* and *SysAL*, respectively. On the other hand, in *Sys*AL project, the resolution and detection effort were practically equal. Therefore, the collected data suggest that teamwork members tend to spend more effort resolving inconsistency rather than applying the model composition technique and detecting inconsistencies.

Table 5. Descriptive statistics for resolution effort

Cases	N	Mean	SD	Min	25th	Med	75th	Max
SysMar	31	40.79	74.79	3.09	4.13	9.12	11.33	246.25
SysBand	8	28.06	28.04	5.55	8.17	15.24	41.44	95.44
SysGR	16	25.86	13.75	5.12	17.70	19.45	42.5	53.33
SysPR	44	2.86	1.92	1.2	2.03	2.33	2.52	10.41
SysAL	5	31.04	12.75	16.21	16.21	29.20	46.8	55.4

N = number of compositions, SD = standard deviation, Min = minimum, 25th = first quartile; Med = median, 75th: third quartile, Max: maximum.

Another finding is that *the experience acquired by the developers did not help to significantly reduce the inconsistency resolution effort*. Although more experienced developers have invested less effort to compose the input models and detect inconsistencies, their additional experience did not help significantly to reduce the inconsistency resolution effort. For example, in SysBand project, more experienced developers spent 40.15 percent more effort to resolve inconsistency than less experienced developers from SysMar project, compared the medians 9.12 and 15.24. The main reason is that most experienced developers tend to be more cautious than less experienced ones, and hence they tend to invest more time analyzing the impact of the resolution of each inconsistency.

4.2 RQ2: Influential Factors on Composition Effort

Some factors influence the effort of composing large-scale design models in real-world settings. This section analyzes the side effects of these factors on the composition effort variables.

The Effects of Conflicting Changes. A careful analysis of the results has pointed out that the production of the intended model is strictly affected by the presence of different types of change categories in the delta model. These changes would be: a*ddition*, model elements are inserted into base model; *removal*, a model element in the base model is removed; m*odification*, a model element has some properties modified; d*erivation*: model elements are refined for accommodating new changes and/or moved to other ones, commonly seen as a 1:N modification. We have also observed that the current composition algorithms are not able to effectively accommodate these changes in the base model, in particular when they occur simultaneously.

Developers and researchers recognize that software should adhere to the Open-Closed principle [31] as the evolutions become more straightforward. This principle states "software should be open for extensions, but closed for modifications." However, this observation did not occur in all the cases as modifications and derivations of model elements happened as well. In our study, the open-closed principle was more closely adhered by the evolutions dominated by additions rather than any other one. In this case, developers invested low effort compared to other cases. This suggests that the closer to the *Open-Closed* principle the change, the lower the composition effort.

On the other hand, evolution scenarios that do not follow the Open-Closed principle required more effort to produce the intended model, M_{AB}. This finding was identified when the change categories simultaneously occur in the delta model; hence,

compromising the composition for some extent. This extra effort was due to the incapability of the matching algorithm to identify the similarities between the input model elements given the presence of widely scoped changes. In the SysMar project, for example, the composition techniques were not able to execute the compositions by about 17 percent (11/64) of the evolution scenarios. This required developers to recreate the models manually. In the SysBand project, by about 10 percent (10/95) of the composition cases did not produce an output model as well; or the composed model produced had to be thrown away due to the high amount of inconsistencies.

In particular, we also observed that the refinement (1:N) of model elements in the delta model caused more severe problems. This problematic scenario was noticed during the refinement of some classes belonging to the MVC (Model-View-Controller) architecture style into a set of more specialized ones. In both cases, the name-based, structural model comparison was unable to recognize the 1:N composition relations between the input model elements. However, we have observed these conflicts do not only happen when developers perform modifications, removals, or refinements in parallel, but also when developers insert new model elements. This finding was noted from the fact that although evolutions following the Open-Closed principle had reduced the developers' effort, they still caused too frequent undetected inconsistencies.

Conflict Management. The detection of all possible semantic conflicts between two versions of a model is an undecidable problem [10]; as many false positive conflicts can appear. To alleviate this problem, some previous works recommend to reduce the size of the delta model to minimize the number of conflicts [11]. However, this approach does not ameliorate in fact the complexity of the changes. The problem is not the number of conflicts that the size of the delta can cause, but the complexity of the conflicts. To alleviate the effort to tame the conflicts, we narrowed down the scope of the conflicts. For this, the delta model now represented one or two functionalities of a particular use case. Hence, the conflicts became more manageable and reasonable. The compositions had a smaller scope.

On the other hand, sometimes the presence of more widely scope changes was inevitable in the delta model. This was, for example, the case when the models (e.g., class and sequence diagrams) were reviewed and meliorated for assuring quality issues. Unfortunately, this led to decrease the precision of the compositions due to the presence of non-trivial compositions. It is known that the domain independent composition algorithms cannot rely on the detailed semantics of the models being composed or on the meaning of changes. Instead of being able to identify all possible conflicts, the algorithms detect as many conflicts as possible, assuming an approximate approach. Consequently, developers need to deal with many false positive conflicts.

In practice, we noted that if the composition generates many conflicts, developers prefer throwing the models away (and investing more effort to recreate it after) to resolving all conflicts. Although the composition algorithm detects the conflicting changes created by developers in parallel, developers are unable to understand and proactively resolve these conflicts generated from non-trivial compositions. This can be explained by two reasons. First, the complexity of the conflicts affected the model elements. Second, the difficulty of understanding the meaning of the changes performed by other developers. More importantly, developers were unable to foresee the

ripple effects of their actions. This is linked to two very interesting findings. First, developers have a tacit assumption that the models to-be-composed will not conflict with each other, and a common expectation is that little effort must be spent to integrate models. Hence, the developer tends to invest low effort to check whether the composition produced inconsistencies or not. Therefore, we can conclude that the need to throw the model away in order to recreate it after demonstrates the complexity of the problem.

Conflict Resolution and Developer Reputation. We have observed that when two changes in the input models (M_A and M_B) contradict each other, the one created by the more experienced developer tends to remain in the output composed model. In other words, the reputation of the developers influences the resolution of conflicting changes. It is important to recall that a developer can accept and reject the conflicting change of another developer. We observed this finding during the observational study, interviews, and analyzing the change history in the repository. This was particularly observed when novice developers reject the changes performed by them, and accept the ones carried out by senior developers. That is, if a novice developer modifies a design model, and this change conflicts with another one performed by a more experienced developer, the novice tends to consider the change carried out by the latter.

An additional interesting finding was that the effort of taming the conflicting changes tended to be less when the reputations of the developers were particularly opposite, one much high and another one too low. A careful analysis of the changes in the model elements reveals some interesting insights. We have noted that the implementation of the new changes (via M_A) by more experienced developers for encapsulating new evolutions are more oblivious to the modifications being implemented in the delta model. This observation holds for both structural and behavioral models i.e., class and sequence diagrams, respectively. As a consequence, the modifications realized by more experienced developers tended to help novice developers find an answer for the conflicts more quickly, thereby reducing the composition effort. Still, these modifications usually stay unchanged for a longer time, when compared with those realized by novice developers.

Reputation can be seen as the opinion (or a social evaluation) of a member of the development team toward other developer. We have identified two types of reputation: *technical* and *social*. The technical reputation refers to the level of knowledge considering issues related to the technology and tools used in the company such as the composition tool, IDEs, CASE tools, and version control systems. This type of reputation is acquired mainly solving daily problems. On the other hand, the social reputation refers to the position assumed by a member of the development team e.g., senior developer. After interviewing 8 developers, the data collected suggests that the technical reputation caused more influence for resolving conflicts than the social reputation. That is, 75 percent of the developers (6/8) reported that the technical reputation has a higher influence than the social one. We have concluded that the developer reputation indeed affects the way that conflicts are resolved. In particular, the changes performed by the subjects with high reputation tend to remain in the output composed model when ones conflict with other changes implemented by less experienced developers.

5 Related Work

Model composition is a very active research field in many research areas [34][35] such as synthesis of state charts [13][18], weaving of aspect-oriented models [19][20][21], governance and management of enterprise design models [9], software configuration management [30], and composition of software product lines [25][28]. For this reason, several academic and industrial composition techniques have been proposed such as MATA [19], Kompose [23], Epsilon [22], IBM RSA [16], and so on. With this in mind, some observations can be done.

First, these initiatives focus only on proposing the techniques instead of also demonstrate their effectiveness. Consequently, qualitative and quantitative indicators considering these techniques are still incipient. In addition, the situation is accentuated considering effort indicators. This lack hinders mainly the understanding of their side effects. Second, their chief motivation is to provide programming languages to express composition logic. Unfortunately, these approaches do not offer any insights or empirical evidences whether developers might reach the potential benefits claimed by using composition techniques in practice. Although some techniques are interesting approaches, sometimes they are used in practice because of the large number of false positives that they can produce in real-world settings. Nevertheless, the effort required for the user to under-stand and correct composition inconsistencies will ultimately prove to be too great. The current article takes a different approach. It aims to provide a precise assessment of composition effort in real life context, quantifying effort and identifying the influential effort.

Moreover, current works tend to investigate on the proactive detection and earlier resolution of conflicts. Most recently, Brun et al. [33] proposes an approach, namely Crystal, to help developers identify and resolve conflicts early. The key contributions are that conflicts are very common than would be expected, appearing over-lapping textual edits but also as subsequent build and test failures. In a similar way, Sarma et al. [32] proposes a new approach, named Palantír, based on the precept of workspace awareness, to detection and earlier resolution of a larger number of conflicts. Based on two laboratory experiments, the authors confirmed that the use of the Palantír reduced of the number of unresolved conflicts. Although these two approaches are interesting studies, the earlier detection does alleviate the problem of model composition. The problem is the same, but is only reported more quickly. In addition, they appear to be overly restrictive to the code, not leading to broader generalizations at the modeling level. Lastly, they neither make consideration about the effort to compose of the artefacts used nor investigate the research questions in five case studies.

6 Concluding Remarks and Future Work

This paper represented the first in vivo exploratory study to evaluate the effort that developers invest to compose design models (RQ1) and to analyze the factors that affect developers' effort (RQ2). In our study, a best-of-breed model composition technique was applied to evolve industrial design models along 297 evolution scenarios. The works were conducted during 56 weeks producing more than 2 million of compositions of model elements. We investigated the composition effort in this

sample, and analyzed the side effects of key factors that affected the effort of applying the composition technique as well as detecting and resolving inconsistencies.

We summarize the findings related to RQ1 as follows: (1) the application effort measures do not follow an ad hoc distribution and, rather, it assumed a distribution pattern; (2) the application effort tends to reduce as developers become more familiar with technical issues rather than application domain issues; (3) the more experienced developers spend 23.2 percent less effort to detect inconsistencies than less experienced developers; and (4) the more the number of inconsistencies in behavioral models, the higher the effort to detect inconsistencies. Additionally, we also present four findings with respect to RQ2 as follows: (1) the production of the intended model is strictly affected by the presence of different types of change categories in the delta model; (2) the closer to the Open-Closed principle the change, the lower the composition effort. That is, evolutions dominated by additions reduce the composition effort. On the other hand, the refinement (1:N) of model elements in the delta model caused severe composition problems and hence increased the composition effort.

Although we gathered quantitative and qualitative evidence to supporting the aforementioned findings, further empirical studies are still required to check whether they are observed in other contexts and with different subjects. Future investigation points would be to answer some questions such as: (1) Do developers invest much more effort to compose behavioral models (e.g. sequence diagrams) than structural models (e.g. component diagrams)? Are the influential factors in composition effort similar in these two contexts? (2) How different are the findings similar or different with respect to code merge (i.e. implementation-level composition)? (3) Do developers invest more effort to resolve semantic inconsistencies than syntactic ones? It is by no means obvious that, for example, developers invest less effort to resolve inconsistencies related to the well-formedness rules of the language metamodel than to resolve inconsistencies considering the meaning of the model elements. Finally, we hope that the issues outlined throughout the paper encourage other researchers to replicate our study in the future under different circumstances. Moreover, we also hope that this work represents a first step in a more ambitious agenda on better supporting model composition tasks.

References

1. Runeson, P., Höst, M.: Guidelines for Conducting and Reporting Case Study Research in Software Engineering. Empirical Software Engineering 14, 131–164 (2009)
2. Wohlin, C., Runeson, P., Höst, M., Ohlsson, M., Regnell, B., Wesslén, A.: Experimentation Software Engineering - An Introduction. Kluwer Academic Publishers (2000)
3. Kitchenham, B., Al-Khilidar, H., Babar, M., Berry, M., Cox, K., Keung, J., Kurniawati, F., Staples, M., Zhang, H., Zhu, L.: Evaluating Guidelines for Reporting Empirical Software Engineering Studies. Empirical Software Engineering 13(1), 97–12 (2008)
4. Boisvert, R., Tang, P. (eds.): The Architecture of Scientific Software. Kluwer Academic (2001)
5. Kelly, D.: A Study of Design Characteristics in Evolving Software Using Stability as a Criterion. IEEE Transactions on Software Engineering 32(5), 315–329 (2006)
6. Camtasia Studio Pro. (2011), `http://www.techsmith.com/camtasia/`

7. Farias, K.: Analyzing the Effort on Composing Design Models in Industrial Case Studies. In: 10th International Conference on Aspect-Oriented Software Development Companion, Porto de Galinhas, Brazil, pp. 79–80 (2011)
8. Farias, K., Garcia, A., Whittle, J.: Assessing the Impact of Aspects on Model Composition Effort. In: 9th International Conference on Aspect-Oriented Software Development Companion, Saint Malo, France, pp. 73–84 (2010)
9. Norris, N., Letkeman, K.: Governing and Managing Enterprise Models: Part 1. Introduction and Concepts. IBM Developer Works (2011), http://www.ibm.com/developerworks/rational/library/09/0113_letkeman-norris
10. Mens, T.: A State-of-the-Art Survey on Software Merging. IEEE Transactions on Software Engineering 28(5), 449–462 (2002)
11. Perry, D., Siy, H., Votta, L.: Parallel Changes in Large-Scale Software Development: an Observational Case Study. Journal ACM Transactions on Software Engineering and Methodology (TOSEM) 10(3), 308–337 (2001)
12. Keith, E.: Flexible Conflict Detection and Management in Collaborative Applications. In: 10th Annual ACM Symposium on User Interface Software and Technology, pp. 139–148 (1997)
13. Ellis, C., Gibbs, S.: Concurrency Control in Groupware Systems. ACM SIGMOD, 399–407 (1989)
14. Berzins, V.: Software Merge: Semantics of Combining Changes to Programs. Journal ACM Transactions on Programming Languages and Systems 16(6), 1875–1903 (1994)
15. Berzins, V., Dampier, D.: Software merge: Combining Changes to Decompositions. Journal of Systems Integration 6(1-2), 135–150 (1996)
16. IBM Rational Software Architecture (2011), http://www.ibm.com/developerworks/rational/products/rsa/
17. Berzins, V.: On Merging Software Extensions. Acta Informatica 23, 607–619 (1986)
18. Gerth, C., Küster, J.M., Luckey, M., Engels, G.: Precise Detection of Conflicting Change Operations Using Process Model Terms. In: Petriu, D.C., Rouquette, N., Haugen, Ø. (eds.) MODELS 2010, Part II. LNCS, vol. 6395, pp. 93–107. Springer, Heidelberg (2010)
19. Whittle, J., Jayaraman, P., Elkhodary, A., Moreira, A., Araújo, J.: MATA: A unified approach for composing UML aspect models based on graph transformation. In: Katz, S., Ossher, H., France, R., Jézéquel, J.-M. (eds.) Transactions on AOSD VI. LNCS, vol. 5560, pp. 191–237. Springer, Heidelberg (2009)
20. Whittle, J., Jayaraman, P.: Synthesizing Hierarchical State Machines from Expressive Scenario Descriptions. ACM TOSEM 19(3) (January 2010)
21. Klein, J., Hélouët, L., Jézéquel, J.: Semantic-based Weaving of Scenarios. In: 5th AOSD 2006, Bonn, Germany (March 2006)
22. Epsilon Project (2011), http://www.eclipse.org/gmt/epsilon/
23. Kompose: A generic model composition tool (2011), http://www.kermeta.org/kompose
24. Sabetzadeh, M., Nejati, S., Chechik, M., Easterbrook, S.: Reasoning about Consistency in Model Merging. In: 3rd Workshop on Living With Inconsistency in Software Development (September 2010)
25. Jayaraman, P., Whittle, J., Elkhodary, A.M., Gomaa, H.: Model Composition in Product Lines and Feature Interaction Detection Using Critical Pair Analysis. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) MODELS 2007. LNCS, vol. 4735, pp. 151–165. Springer, Heidelberg (2007)

26. Diskin, Z., Xiong, Y., Czarnecki, K.: Specifying Overlaps of Heterogeneous Models for Global Consistency Checking. In: Dingel, J., Solberg, A. (eds.) MODELS 2010. LNCS, vol. 6627, pp. 165–179. Springer, Heidelberg (2011)
27. Egyed, A.: Fixing Inconsistencies in UML Design Models. In: 29th International Conference on Software Engineering, pp. 292–301 (2007)
28. Thaker, S., Batory, D., Kitchin, D., Cook, W.: Safe Composition of Product Lines. In: 6th GPCE 2007, Salzburg, Austria, pp. 95–104 (2007)
29. Egyed, A.: Automatically Detecting and Tracking Inconsistencies in Software Design Models. IEEE Transactions on Software Engineering 37(2), 188–204 (2010)
30. Whitehead, J.: Collaboration in Software Engineering: A Roadmap. In: Future of Software Engineering at ICSE, pp. 214–225 (2007)
31. Meyer, B.: Object-Oriented Software Construction, 1st edn. Prentice-Meyer, Hall, Englewood Cliffs (1988)
32. Sarma, A., Redmiles, D., van Der Hoek, A.: Palantír: Early Detection of Development Conflicts Arising from Parallel Code Changes. IEEE TSE 99(6) (2011)
33. Brun, Y., Holmes, R., Ernst, M., Notkin, D.: Proactive Detection of Collaboration Conflicts. In: 8th SIGSOFT ESEC/FSE, Szeged, Hungary, pp. 168–178 (2011)
34. France, R., Rumpe, B.: Model-Driven Development of Complex Software: A Research Roadmap. In: FuSE at ICSE 2007, 37–54 (2007)
35. Apel, S., Liebig, J., Brandl, B., Lengauer, C., Kästner, C.: Semistructured Merge: Rethinking Merge in Revision Control Systems. In: 8th SIGSOFT ESEC/FSE, pp. 190–200 (2011)
36. OMG, Unified Modeling Language: Infrastructure, version 2.2, Object Management Group (February 2011)
37. JIRA, http://www.atlassian.com/software/jira/overview