Softw Syst Model
DOI 10.1007/s10270-012-0308-2

REGULAR PAPER

Effects of stability on model composition effort: an exploratory study

Kleinner Farias · Alessandro Garcia · Carlos Lucena

Received: 4 July 2011 / Revised: 17 August 2012 / Accepted: 7 November 2012

Abstract Model composition plays a central role in many software engineering activities, e.g., evolving design models to add new features. To support these activities, developers usually rely on model composition heuristics. The problem is that the models to-be-composed usually conflict with each other in several ways and such composition heuristics might be unable to properly deal with all emerging conflicts. Hence, the composed model may bear some syntactic and semantic inconsistencies that should be resolved. As a result, the production of the intended model is an error-prone and effort-consuming task. It is often the case that developers end up examining all parts of the output composed model instead of prioritizing the most critical ones, i.e., those that are likely to be inconsistent with the intended model. Unfortunately, little is known about indicators that help developers (1) to identify which model is more likely to exhibit inconsistencies, and (2) to understand which composed models require more effort to be invested. It is often claimed that software systems remaining stable over time tends to have a lower number of defects and require less effort to be fixed than unstable systems. However, little is known about the effects of software stability in the context of model evolution when supported by composition heuristics. This paper, therefore, presents an exploratory study analyzing stability as an indicator of inconsistency rate and resolution effort on model composition activities. Our findings are derived from 180 compositions performed to evolve design models of three software product lines. Our initial results, supported by statistical tests, also indicate which types of changes led to lower inconsistency rate and lower resolution effort.

Keywords Model composition · Software development effort · Design stability

Communicated by Prof. Lionel Briand.

K. Farias (✉) · A. Garcia · C. Lucena
OPUS Research Group, LES, Informatics Department,
Pontifical Catholic University of Rio de Janeiro,
Rio de Janeiro, RJ, Brazil
e-mail: kleinner@gmail.com; kfarias@inf.puc-rio.br

A. Garcia
e-mail: afgarcia@inf.puc-rio.br

C. Lucena
e-mail: lucena@inf.puc-rio.br

1 Introduction

Model composition plays a central role in many software engineering activities [18], e.g., evolving design models to add new features and reconciling multiple models developed in parallel by different software development teams [28,38]. The composition of design models can be defined as a set of activities that should be performed over two input models, M_A and M_B, in order to produce an output intended model, M_{AB}. To put the model composition in practice, software developers usually make use of composition heuristics [9] to produce M_{AB}. These heuristics match the 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.

The problem is that, in practice, the output composed model (M_{CM}) and the intended model (M_{AB}) often do not match (i.e., $M_{CM} \neq M_{AB}$) because M_A and M_B conflict with each other in some way [18]. Hence, these conflicts are converted into syntactic and semantics inconsistencies in M_{CM}. Consequently, software developers should be able to anticipate composed models that are likely to exhibit inconsistencies and transform them into M_{AB}. In fact, it is well known that the derivation of M_{AB} from M_{CM} is considered

Published online: 09 January 2013

an error-prone task [18,35]. The developers do not even have practical information or guidance to plan this task. Their inability is due to two main problems.

First, developers do not have any indicator to reveal which M_{CM} should be reviewed (or not), given a sequence of output composed models produced by the software development team. Hence, they have no means to identify or prioritize parts of design models that are likely to have a higher density of inconsistencies. They are often forced to go through all output models produced or assume an overoptimistic position, i.e., all output composed models produced is a M_{AB}. In both cases, the inadequate identification of an inconsistent M_{CM} can even compromise the evolution of the existing design model (M_A) as some composition inconsistencies can affect further model compositions.

Second, model managers are unable to grasp how much effort the derivation of M_{AB} from M_{CM} can demand, given the problem at hand [35]. Hence, they end up not designating the most qualified developers for resolving the most critical effort-consuming cases where severe semantic inconsistencies are commonly found. Instead, unqualified developers end up being allocated to deal with these cases. In short, model managers have no idea about which M_{CM} will demand more effort to be transformed into a M_{AB} [35]. If the effort to resolve these inconsistencies is high, then the potential benefits of using composition heuristics (e.g., gains in productivity) may be compromised.

The literature in software evolution highlights that software remaining stable over time tends to have a lower number of flaws and require less effort to be fixed than its counterpart [21,32]. However, little is known whether the benefits of stability are also found in the context of the evolution of design models supported by composition heuristics. This is by no means obvious for us because the software artifacts (code and models) can have different level of abstraction. In fact, design model has a set of characteristics (defined in language metamodel expressing it) that are manipulated by composition heuristics and can assume values close to what is expected (or not), i.e. $M_{CM} \approx M_{AB}$. If the assigned value of a characteristic is close to the one found in the intended model, then the composed model is considered stable concerning that characteristic. For example, if the difference between the coupling of the composed model and the intended model is small, then they can be considered stable considering coupling.

Although researchers recognize software stability as a good indicator to address the two problems described above in the context of software evolution, most of the current research on model composition is focused on building new model composition heuristics (e.g., [9,25,34,44]). That is, nothing has been done to evaluate stability as an indicator of the presence of semantic inconsistencies and of the effort that, on average, developers should spend to derive M_{AB} from M_{CM}. Today, the identification of critical M_{CM} and the effort estimation to produce M_{AB} are based on the evangelists' feedback that often diverge [28].

This paper, therefore, presents an initial exploratory study analyzing stability as an indicator of composition inconsistencies and resolution effort. More specifically, we are concerned with understanding the effects of the model stability on the inconsistency rate and inconsistency resolution effort. We study a particular facet of model composition in this paper: the use of model composition in adding new features to models for three realistic software product lines. Software product lines (SPLs) commonly involve model composition activities [20,43] and, while we 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 SPL model composition. Three well-established composition heuristics [9], namely *override*, *merge,* and *union*, were employed to evolve the SPL design models [1,20] along eighteen releases. SPLs are chosen because designers need to maximize the modularization of features allowing the specification of the compositions. The use of composition is required to accommodate new variabilities and variants (mandatory and optional features) that may be required when SPLs evolve. We analyze if stability is a good indicator of high inconsistency rate and resolution effort.

Our findings are derived from 180 compositions performed to evolve design models of three software product lines. Our results, supported by statistical tests, show that stable models tend to manifest a lower inconsistency rate and require a lower resolution effort than their counterparts. In other words, this means that there is significant evidence that the higher the model stability, the lower the model composition effort.

In addition, we discuss scenarios where the use of the composition heuristics became either costly or prohibitive. In these scenarios, developers need to invest some extra effort to derive M_{AB} from M_{CM}. Additionally, we discuss the main factors that contributed to the stable models outnumber the unstable one in terms of inconsistency rate and inconsistency resolution effort. For example, our findings show that the highest inconsistency rates are observed when severe evolution scenarios are implemented, and when inconsistency propagation happens from model elements implementing optional features to ones implementing mandatory features (Sect. 4.1.3). We also notice that the higher instability in the model elements of the SPL design models realizing optional features, the higher the resolution effort. To the best of our knowledge, our results are the first to investigate the potential advantages of model stability in realistic scenarios of model composition. We therefore see this paper as a first step in a more ambitious agenda to assess model stability empirically.

The remainder of the paper is organized as follows: Sect. 2 describes the main concepts and knowledge that are going to be used and discussed throughout the paper. Section 3

presents the study methodology. Section 4 discusses the study results. Section 5 compares this work with others, presenting the main differences and commonalities. Section 6 points out some threats to validity. Finally, Sect. 7 presents some concluding remarks and future work.

2 Background

2.1 Model composition effort

To produce an output intended model (M_{AB}), a set of activities is performed over M_A and M_B. M_A is the current design model, while M_B is the model expressing the evolution (delta model), for example, the upcoming changes being added. M_B is inserted into the M_A using some composition heuristics, which are responsible for defining the semantics of the composition and specify how M_A and M_B should be manipulated in order to produce M_{AB}. We will use the terms composed model (M_{CM}) and intended model (M_{AB}) to differentiate between the output model produced by a composition heuristic and one is desired by the developers. As previously mentioned, usually M_{CM} and M_{AB} do not match ($M_{CM} \neq M_{AB}$) because the input models conflict with each other in some way. The higher the number of inconsistencies in M_{CM}, the more distant it is from the intended model. This may mean a high effort to be spent to derive M_{AB} from M_{CM} (or not). Once M_{CM} has been produced, the next step is to measure the effort to transform M_{CM} into M_{AB}, i.e., the effort to resolve inconsistencies. If M_{CM} is equal to M_{AB}, then this implies that the design characteristics of M_{CM} keep stable over composition. Therefore, the inconsistency resolution effort is equal to zero. Otherwise, the effort is higher than zero.

The composition effort can be understood by using the equation defined in Fig. 1. The equation gives an overview of how composition effort can be measured and what part we focus our study on. The equation makes it explicit that the model composition effort includes: (1) the effort to apply a composition heuristic: f(M_A, M_B); (2) the effort to detect undesirable inconsistencies in the output model: diff(M_{CM}, M_{AB}); and (3) the effort to resolve inconsistencies: $g(M_{CM})$. Once M_{CM} has been produced, the next step is to measure the effort to transform M_{CM} into M_{AB}. If M_{CM} is equal to M_{AB}, then diff(M_{CM}, M_{AB}) and $g(M_{CM})$ are equal to zero. Otherwise, diff(M_{CM}, M_{AB}) and $g(M_{CM})$ are higher than zero. This study focuses specifically on evaluating the effort to inconsistency resolution (i.e., $g(M_{CM})$) rather than inconsistency detection and algorithm application.

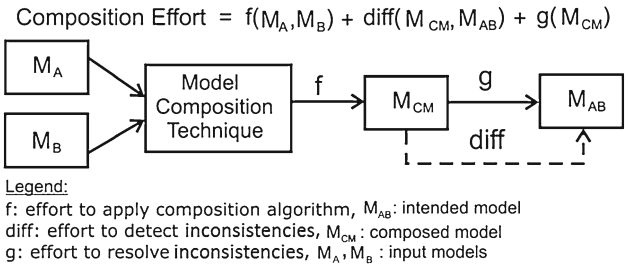


Fig. 1 Model composition effort: an equation

2.2 Model stability

According to [21,30], a design characteristic of software is stable if, when compared to other, the differences in the indicator associated with that characteristic are considered, in the context, to be small. In a similar way in the context of model composition, M_{CM} can be considered stable if its design characteristics have a low variation concerning the characteristics of M_{AB}. In [21], Kelly studies stability from a retrospective view, i.e., comparing the current version to previous ones. In our study, we compare the current model and the intended model.

We define low variation as being equal to (or less than) 20 %. This choice is based on previous empirical studies [21] on software stability that has demonstrated the usefulness of this threshold. For example, if the measure of a particular characteristic (e.g., coupling and cohesion) of the M_{CM} is equal to 9, and the measure of the M_{AB} is equal to 11. So M_{CM} is considered stable concerning M_{AB} (because 9 is 18 % lower than 11) with respect to the measure under analysis. Following this stability threshold, we can systematically identify weather (or not) M_{CM} keeps stable considering M_{AB}, given an evolution scenario. Note that threshold is used more as a reference value rather than a final decision maker. The results of this study can regulate it, for example. The differences between the models are computed comparing the measures obtained by a set of metrics (Table 1) [12].

We adopt the definition of stability from [21] (and its threshold) because of some reasons. First, it defines and validates the quantification method of stability in practice. This method is used to examine software systems that have been actively maintained and used over a long term. Second, the quantification method of stability has demonstrated to be effective to flag evolutions that have jeopardized the system design.

Third, many releases of the system under study was considered. This is a fundamental requirement to test the usefulness of the method. As such, all these factors provided a solid foundation for our study. These metrics were used because previous works [21] have already observed the effectiveness of these indicators for the quantification of software stability. Knowing the stability in relation to the intended model it is possible to identify evolution scenarios, where composition heuristics are able to accommodate upcoming changes effectively and the effort spent to obtain the intended

Table 1 Metrics used in our study

Type	Metric	Description
Size	NClass	The number of classes
	NAttr	The number of attributes
	NOps	The number of operations
	NInter	The number of interfaces
	NOI	The number of operations in each interface
Inheritance	DIT	The depth of the class in the inheritance hierarchy
	InhOps	The number of operations inherited
	InhAttr	The number of attributes inherited
Coupling	DepOut	The number of elements on which a class depends
	DepIn	The number of elements that depend on this class

model. The stability quantification method is presented later in Sect. 3.4.

2.3 Composition heuristics

Composition heuristics rely on two key activities: matching and combining the input model elements [14]. Usually they are used to modify, remove, and add features to an existing design model. This paper focuses on three composition heuristics: override, merge, and union [9]. These heuristics were chosen because they have been applied to a wide range of model composition scenarios such as model evolution, ontology merge, and conceptual model composition. In addition, they have been recognized as effective heuristics in evolving product-line architectures (e.g., [4]). In the following, we briefly define these three heuristics, and assume M_A and M_B as the two input models. The input model elements are corresponding if they can be identified as equivalent in a matching process. Matching can be achieved using any kind of standard heuristics, such as *match-by-name* [9]. The design models used are typical UML class and component diagrams [37] (see Fig. 2), which have been widely used to represent software architecture in mainstream software development [26]. In Fig. 2, for example, $R2$ diagram plays the role of the base model (M_A) and $Delta(R2, R3)$ diagram plays the role of the delta model (M_B). The components $R2.BaseController$ and $Delta(R2, R3).BaseController$ are considered as equivalent. We defer further considerations about the design models used in our study to Sect. 3.3. The composition heuristics considered in our study are discussed in the following paragraphs.

Override For all pairs of corresponding elements in M_A and M_B, M_A's elements should override M_B's corresponding elements. The model elements that do not match remain unchanged. They are just inserted into the output model. For example, Fig. 2 shows an example where the output composed model, $R3$, is produced following this heuristic applied to $R2$ and $Delta(R2, R3)$.

Merge For all corresponding elements in M_A and M_B, the elements should be combined. The combination depends on the element type. Elements in M_A and M_B that are not equivalent remain unchanged and are inserted into the output model directly (see Fig. 2).

Union For all elements in the M_A and M_B that are corresponding elements, they should be manipulated in order to preserve their distinguished identification; it means that they should coexist in the output models with different identifiers; elements in the M_A and M_B that are not involved in a correspondence match remain unchanged and they are inserted into the output model, M_{AB}. For example, the $Delta(R2, R3).BaseController$ has its name modified to $R3.BaseController$ (see Fig. 3).

2.4 Inconsistencies

Inconsistencies emerge in the composed model when its properties assume values other than those would be expected. These values can affect the syntactic and semantic properties of the model elements. Usually such undesired values come from conflicting changes that were incorrectly realized. We can identify two broad categories of inconsistencies: (i) *syntactic inconsistencies*, which arise when the composed model elements do not conform to the modeling language's metamodel; and (ii) *semantic inconsistencies*, which mean that static and behavioral semantics of the composed model elements do not match those of the intended model elements.

In our study, we take into account syntactic inconsistencies that were identified by the IBM Rational Software Architecture's model validation mechanism [35]. For example, this robust tool is able to detect the violation of well-formedness rules defined in the UML metamodel specification [37]. In order to improve our inconsistency analysis, we also considered the types of inconsistencies shown in Table 2 [12],

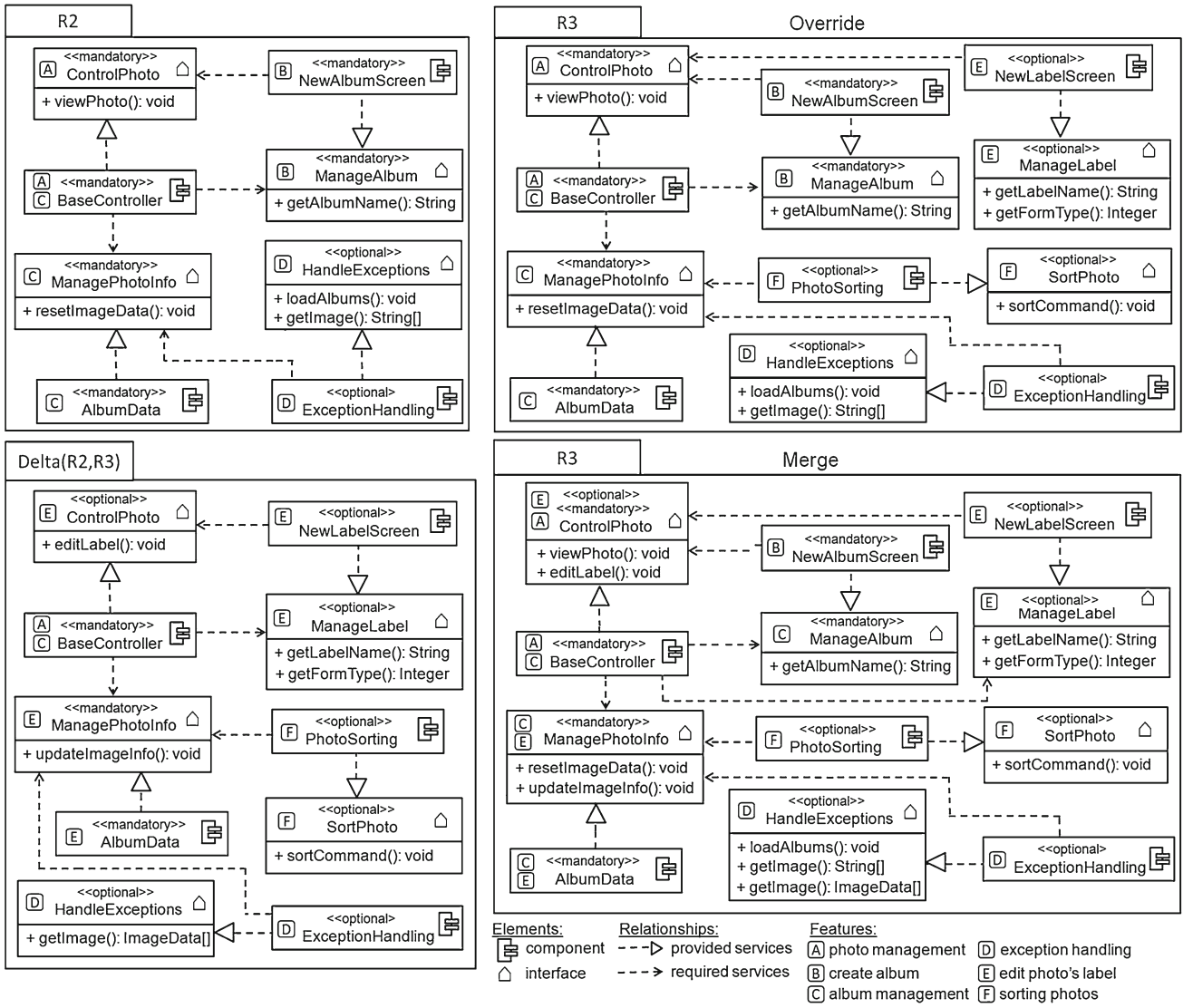


Fig. 2 Practical examples of model composition of the Mobile Media product line

which were checked by using the SDMetrics tool [46]. In particular, these inconsistencies were used because their effectiveness has been demonstrated in previous works [13–16]. In addition, both syntactic and semantic inconsistencies are manually reviewed as well. All these procedures were followed in order to improve our confidence that a representative set of inconsistencies were tackled by our study. Many instances of these inconsistency types (Table 2) were found in our study. For example, the static property of a model element, *isAbstract*, assumes the value *true* rather than *false*. The result is an abstract class where a concrete class was being expected. Another typical inconsistency considered in our study was when a model element provides (or requires) an unexpected functionality or even requires a functionality that does not exist.

The absence of this functionality can affect other design model elements responsible for implementing other functionalities, thereby propagating an undesirable ripple effect between the model elements of M_{CM}. In Fig. 3 (override), for example, the *AlbumData* does not provide the service "update image information" (from the feature "edit photo's label") because the method $updateImageInfo()$*:void* is not present in the *ManagePhotoInfoInterface*. Hence, the *PhotoSorting* component is unable to provide the service "sorting photos." This means that the feature "sorting photo" (feature 'F' in Fig. 2)—a critical feature of the software product line—is not correctly realized. On the other hand, this problem is not present in Fig. 2 (merge), in which the *AlbumData* implements two features (C, model management, and E, edit photo's label). We defer further discussion about the

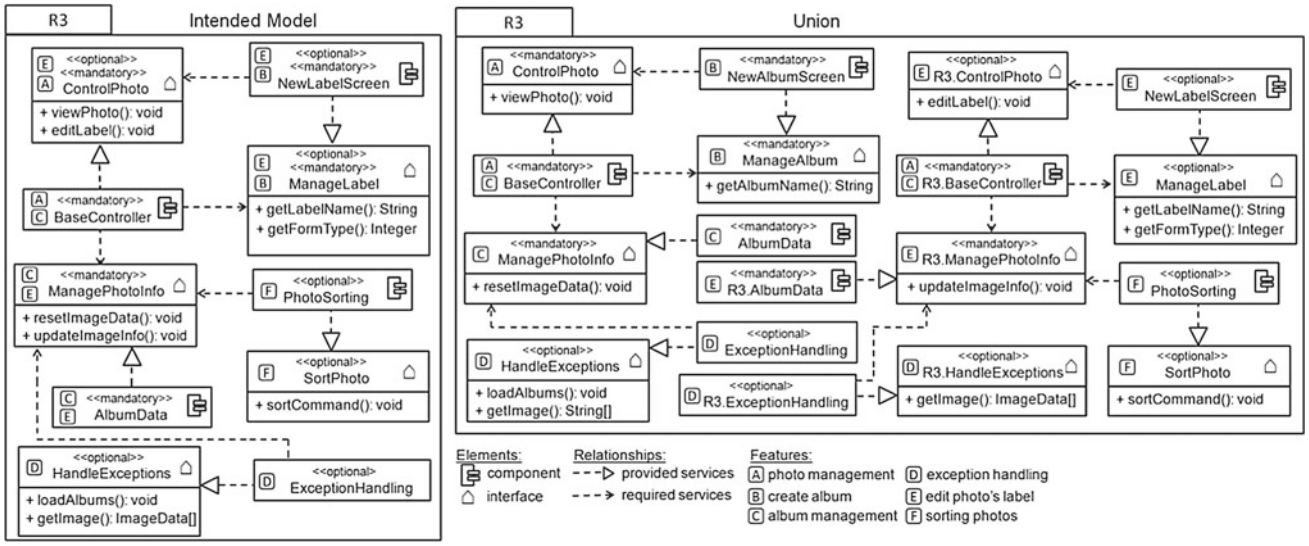


Fig. 3 The intended model (*left*) and composed model (*right*) produced following the union heuristic

Table 2 The inconsistencies used in our case study

Metric	Description
NFCon	The number of functionality inconsistencies
NCCon	The number of model elements that are not compliant with the intended model
NDRCOn	The number of dangling reference inconsistencies
NASCon	The number of abstract syntax inconsistencies
NUMECon	The number of non-meaningful model elements
NBFCon	The number of behavioral feature inconsistencies

examples and the quantification of these types of inconsistencies to Sect. 3.4.

3 Study methodology

This section presents the main decisions underlying the experimental design of our exploratory study. To begin with, the objective and research questions are presented (Sect. 3.1). Next, the study hypotheses are systematically stated from these research questions (Sect. 3.2). The product lines used in our studies are also discussed in detail as well as their evolutionary changes (Sect. 3.3). Then, the variables and quantification methods considered are precisely described (Sect. 3.4). Finally, the method used to produce the releases of the target architectures is carefully discussed (Sect. 3.5). All these methodological steps were based on practical guidelines on empirical studies [42,45].

3.1 Objective and research questions

This study essentially attempts to evaluate the effects of model stability on two variables: the *inconsistency rate* and *inconsistency resolution effort*. These effects are investigated from concrete scenarios involving design model compositions so that practical knowledge can be generated. In addition, some influential factors are also considered into precisely revealing how they can affect these variables. With this in mind, the objective of this study is stated based on the GQM template [3] as follows:

> **analyze** the stability of design models
> **for the purpose of** investigating its effect
> **with respect to** inconsistency rate and resolution effort
> **from the perspective of** developers
> **in the context of** evolving design models with composition heuristics.

In particular, this study aims at revealing the stability effects while evolving composed design models (Sect. 3.3) on inconsistency rate and the inconsistency resolution effort. Thus, we focus on the following two research questions:

- **RQ1:** What is the effect of stability on the *inconsistency rate*?
- **RQ2:** What is the effect of stability on the *developers' effort*?

3.2 Hypothesis formulation

3.2.1 First hypotheses: effect of stability on inconsistency rate

In the first hypothesis, we speculate that a high variation of the design characteristics of the design models may lead to a higher incidence of inconsistencies; since, it increases the chance for an incorrect manipulation of the design characteristic by the composition heuristics. In fact, modifications from severe evolutions may lead the composition heuristics to be ineffective or even prohibitive. In addition, these inconsistencies may also propagate. As a higher incidence of changes is found in unstable models, we hypothesize that unstable models tend to have a higher (or equal to) inconsistency rate than stable models. The first hypothesis evaluates whether the inconsistency rate in unstable models is significantly higher (or equal to) than in stable models. Thus, our hypotheses are summarized as follows:

Null Hypothesis 1, $\mathbf{H}_{1-0}$:
Stable design models have similar or higher inconsistency rate than unstable design models.
$\mathbf{H}_{1-0}$: Rate(stable design models) $\geq$ Rate(unstable design models).
Alternative Hypothesis 1, $\mathbf{H}_{1-1}$:
Stable design models have a lower inconsistency rate than unstable design models.
$\mathbf{H}_{1-1}$: Rate(stable design models) $<$ Rate(unstable design models).

By testing this first hypothesis, we evaluate if stability is a good indicator to identify the most critical M_{CM} in term of inconsistency rate from a sequence of M_{CM} produced from multiple software development teams. Hence, developers can then review the design models having a higher density of composition inconsistencies. We believe that this strategy is a more effective one than going through all M_{CM} produced or assuming an overoptimistic position where all M_{CM} produced is a M_{AB}.

3.2.2 Second hypothesis: effect of stability on developer effort

As previously mentioned, developers tend to invest different quantity of effort to derive M_{AB} from M_{CM}. Today, model managers are unable to grasp how much effort this transformation can demand. This variation is because developers need to resolve different types of problems in a composed model, from a simple renaming of elements to complex modifications in the structure of the composed model. In fact, the structure of the composed models may be affected in different ways during the composition, e.g., creating unexpected interdependences between the model elements. Even worse, these modifications in the structure of the model may cause ripple effects, i.e., inconsistency propagation between the model elements. The introduction of one inconsistency can often lead to multiple other inconsistencies because of a "knock-on" effect. An example would be the inconsistency whereby a client component is missing an important operation in the interface of a server component (see example in Sect. 2.4). This semantic inconsistency leads to a "knock-on" syntactic inconsistency if another component requires the operation. In the worst case, there may be long chains of inconsistencies all derived from a single inconsistency. Given a composed model at hand, developers need to know if they will invest little or too much effort to transform M_{CM} into M_{AB}, given the problem at hand. Based on this knowledge, they will be able to prioritize the review of the output composed models and to better comprehend the effort to be invested, e.g., reviewing the models that require higher effort first and those requiring less effort after. With this in mind, we are interested in understanding the possible difference of effort to resolve inconsistencies in stable and unstable design models. The expectation is that stable models require a lower developers' effort to produce the output intended model. This expectation is based on the speculation that unstable models may demand more restructuring modifications than stable models; hence, requiring more effort. This leads to the second null and alternative hypotheses as follows:

Null Hypothesis 2, $\mathbf{H}_{2-0}$:
Stable models require similar or higher effort to resolve inconsistencies than unstable models.
$\mathbf{H}_{2-0}$: Effort(stable models) $\geq$ Effort(unstable models).
Alternative Hypothesis 2, $\mathbf{H}_{2-1}$:
Stable models tend to require a lower inconsistency resolution effort than unstable ones.
$\mathbf{H}_{2-1}$: Effort(stable models) $<$ Effort(unstable models).

By testing this first hypothesis, we evaluate if stability is a useful indicator to identify the most critical effort-consuming cases in which severe semantic inconsistencies in architectural components are more often. This knowledge helps model managers to allocate qualified developers to overcome the composition inconsistencies in M_{CM}.

3.3 Target cases: evolving product-line design models

Model Composition for Expressing SPL Evolution We apply the composition heuristics (Sect. 2.3) to evolve design models of three realistic SPLs for a set of evolution scenarios (Table 3). That is, the compositions are defined to generate the new releases of the SPL design models. These three SPLs are described below and soon after the evolution scenarios are presented.

Table 3 Descriptions of the SPL releases

Releases	Descriptions
Mobile Media	
R1	MobilePhoto core [17]
R2	Exception handling included
R3	New feature added to count the number of times a photo has been viewed and sorting photos by highest viewing frequency. New feature added to edit the photo's label
R4	New feature added to allow users to specify and view their favorite photos
R5	New feature to allow users to keep multiple copies of photos
R6	New feature to send photo to other users by SMS
Checkers Game	
R1	Checkers Game core
R2	New feature to indicate the movable pieces
R3	New feature to indicate possible movements
R4	New feature to allow the users to save and load the game
R5	New feature added to customize the pieces
R6	New feature added to log the game
Shogi Game	
R1	Shogi Game core
R2	New feature to customize pictures
R3	New feature to customize pieces
R4	New feature to indicate the piece movement
R5	New feature to indicate the movable pieces
R6	New feature to allow the users to save and load the game

The transition from one release to another one represents an evolution scenario

The first target case is a product-line called MobileMedia [17], whose purpose is to support the manipulation of photos, music, and videos on mobile devices. The last release of its design model consists of a UML component diagram with more than 50 component elements. Figures 2 and 3 show a practical example of the use of composition to evolve this SPL.

The second SPL, called Shogi Game [12], is a two-player board game, whose purpose is to allow users to move, customize pieces, save, and load the game. All these pieces' movements are governed by a set of well-defined rules. The last SPL, called Checkers Game [12], is a draughts board game played on an eight by eight-squared board with 12 pieces on each side. The purpose of Checkers is to essentially move and capture diagonally forwards. In [12], it is possible to find a fine-grained description about their characteristics and details about their evolutions.

The reason for selecting these SPLs in our evaluation is manifold. Firstly, the models are well designed. Next, 12 releases of Mobile Media's architectural models are considered by independent developers using the model composition heuristics. These releases are produced from five evolution scenarios. Note that an evolution is the production of a release from another one, e.g., from R1 to R2 (see Table 3). In addition, 12 releases of Shogi's and Checkers' architectural models were available as well. In both cases, six releases were produced from five evolution scenarios. Together the 36 releases provide a wide range of SPL evolution scenarios to enable us to investigate our hypotheses in detail. These 36 releases were produced from the 18 evolution scenarios described in Table 3. Moreover, these releases were available for our investigation and had a considerable quantity of structural changes in the evolution scenarios. Table 3 describes the evolution scenarios. Each scenario represents the addition of a feature. All evolution scenarios were obtained from the addition of optional features, totaling 15 optional features.

Another reason to choose these SPLs is that the original developers are available to help us to validate the identified list of syntactic and semantic inconsistencies. In total, eight developers worked during the development of the SPLs used in our study being three developers from the Lancaster University (UK), two from the Pontifical Catholic University of Rio de Janeiro (Brazil), two from University of São Paulo (Brazil), one from Federal University of Pernambuco (Brazil). These are fundamental requirements to test our hypotheses (Sect. 3.2) in a reliable fashion. Equally important, each SPL has more than one hundred modules and their architecture models are the main artifact to reason about change requests and derive new products. Moreover,

the SPL designs are produced by the original developers without any of the model composition heuristics under assessment in mind. It helped to avoid any bias and entailed natural software development scenarios.

Finally, these SPLs have a number of other relevant characteristics for our study, such as: (i) proper documentation of the driving requirements; and (ii) different types of changes are realized in each release, including refinements over time of the architecture style employed. After describing the SPLs employed in our empirical studies, the evolution scenarios suffered by them are explained in Table 3.

3.4 Measured variables and quantification method

First dependent variable The dependent variable of hypothesis 1 is the inconsistency rate. It quantifies the amount of composition inconsistencies (Sect. 2.4) divided by the total number of elements in the composed model. That is, it allows computing the density of composition inconsistencies in the output composed models. This metric makes it possible to assess the difference between the inconsistency rate of stable models and unstable models (H1). It is important to point out that the inconsistency rate is defined from multiple inconsistency metrics (see Table 2).

Second dependent variable The dependent variable of the hypothesis 2 is the inconsistency resolution effort, $g(M_{CM})$—that is, the number of operations (creations, removals, and updates) required to transform the composed model into the intended model. We compute these operations because they represent the main operations performed by developers to evolve software in realistic settings [28]. Thus, this computation represents an estimation of the inconsistency resolution effort. The collected measures of inconsistency rate are used to assess if the composed model has inconsistencies after the composition heuristic is applied (diff(M_{CM}, M_{AB}) > 0). Then, a set of removals, updates, and creations are performed to resolve the inconsistencies. As a result, the intended model is produced and the inconsistency resolution effort is computed.

Independent variable The independent variable of hypotheses 1 and 2 is the Stability (S) of the output composed model (M_{CM}) with respect to the output intended model (M_{AB}). The Stability is defined in terms of the Distance (D) between the measures of the design characteristics of M_{CM} and M_{AB}. Table 1 defines the method used to quantify the design characteristics of the models, while Formula 1 shows how the Distance is computed.

$$\text{Distance}(x, y) = \frac{|\text{Metric}(x) - \text{Metric}(y)|}{\text{Metric}(y)} \quad (1)$$

where Metric are the indicators defined in Table 1, X is the output composed model, M_{CM}, Y is the output intended model, M_{AB}.

The Stability can assume two possible values: one, indicating that M_{CM} and M_{AB} are *stable*, and zero, indicating that M_{CM} and M_{AB} are *unstable*. M_{CM} is stable concerning M_{AB} if the distance between M_{CM} and M_{AB} (considering a particular design characteristic) assumes a value equal (or lower than) to 0.2. That is, if $0 \leq \text{Distance}(M_{CM}, M_{AB}) \leq 0.2$), then Stability($M_{CM}$, M_{AB}) = 0. On the other hand, M_{CM} is *unstable* if the distance between M_{CM} and M_{AB} (regarding a specific design characteristic) assumes a value higher than 0.2. That is, if Distance(M_{CM}, M_{AB}) > 0.2), then Stability (M_{CM}, M_{AB}) = 0. We use this threshold to point out the most severe unstable models. For example, we check if architectural problems happen even in cases where the output composed models are considered stable. In addition, we also analyze the models that are closer to the threshold in Sect.4. Formula 2 shows how the measure Stability is computed.

$$\text{Stability}(x, y) = \begin{cases} 1, & \text{if } 0 \leq \text{Distance}(x, y) \leq 0.2 \\ 0, & \text{if Distance}(x, y) > 0.2 \end{cases} \quad (2)$$

For example, M_{CM} and M_{AB} have the number of classes equals to 8 and 10, respectively (i.e., NClass = 8 and NClass = 10). To check the stability of M_{CM} regarding this metric, we calculate the distance between M_{CM} and M_{AB} considering the metric NClass as described below.

$$\text{Distance}(M_{CM}, M_{AB}) = \frac{|\text{NClass}(M_{CM}) - \text{NClass}(M_{AB})|}{\text{NClass}(M_{AB})} = \frac{|8 - 10|}{10} = 0.2$$

As the Distance(M_{CM}, M_{AB}) = 0.2, then we can consider that M_{CM} = 1. Therefore, M_{CM} is stable considering M_{AB} in terms of the number of classes. Elaborating on the previous example, we can now consider two design characteristics: the number of classes (NClass), the afferent coupling (DepOut), and the number of attributes (NAttr). Assuming DepOut (M_{CM}) = 12, DepOut(M_{AB}) = 14, NAttr (M_{CM}) = 6, and NAttr(M_{AB}) = 7, the Distance is calculated as follows:

$$\text{Distance}(M_{CM}, M_{AB}) = \frac{|\text{DepOut}(M_{CM}) - \text{DepOut}(M_{AB})|}{\text{DepOut}(M_{AB})} = \frac{|12 - 14|}{14} = 0.14$$

$$\text{Distance}(M_{CM}, M_{AB}) = \frac{|\text{NAttr}(M_{CM}) - \text{NAttr}(M_{AB})|}{\text{NAttr}(M_{AB})} = \frac{|7 - 9|}{9} = 0.22$$

Therefore, M_{CM} is stable concerning M_{AB} in terms of NClass and DepOut. However, M_{CM} is unstable in terms of NAttr. In this example, we evaluate the stability of M_{CM} considering three design characteristics, which was stable in two

cases. As developers can consider various design characteristics to determine the stability of the M_{CM}, we define the Formula 3 that calculates the overall stability of M_{CM} with respect to M_{AB}. Refining the previous example, we evaluate the stability of M_{CM} considering two additional design characteristics: the number of interfaces (NInter) and the depth of the class in the inheritance hierarchy (DIT). Supposing that $\mathrm{NInter}(M_{\mathrm{CM}}) = 15$, $\mathrm{NInter}(M_{\mathrm{AB}}) = 17$, $\mathrm{DIT}(M_{\mathrm{CM}}) = 11$, and $\mathrm{DIT}(M_{\mathrm{AB}}) = 13$, the Distance is calculated as follows:

$$\begin{aligned}\mathrm{Distance}(M_{\mathrm{CM}}, M_{AB}) &= \frac{|\mathrm{NInter}\,(M_{CM}) - \mathrm{NInter}\,(M_{AB})|}{\mathrm{NInter}\,(M_{AB})}\\ &= \frac{|15-17|}{17} = 0.11\\ \mathrm{Distance}(M_{\mathrm{CM}}, M_{AB}) &= \frac{|\mathrm{DIT}\,(M_{CM}) - \mathrm{DIT}\,(M_{AB})|}{\mathrm{DIT}\,(M_{AB})}\\ &= \frac{|11-13|}{13} = 0.15\end{aligned}$$

In both cases, M_{CM} is stable as the values 0.1 and 0.15 are ≥ 0 and ≤ 0.2. Investigating this overall stability, we are able to understand how far the measures of the design characteristics of M_{CM} in relation to M_{AB} are. The overall stability of M_{CM} in terms of NClass, DepOut, NAttr, NInter, and DIT is calculated as follows: As the overall stability is equal to 0.2, we can consider that M_{CM} is stable considering M_{AB}.

$$\mathrm{Stability}(x, y)_{\mathrm{overall}} = 1 - \frac{\sum_{k=0}^{j-1}\left(\mathrm{Stability}_k\right)}{j} \tag{3}$$

Legend: j: number of metrics used (e.g., 10 metrics in case of Table 1).

$$\begin{aligned}&\mathrm{Stability}(x, y)_{\mathrm{overall}} = 1 - \frac{\sum_{k=0}^{4}(\mathrm{Stability}(x, y))}{5}\\ &\sum_{k=0}^{4}(\mathrm{Stability}\,(x, y)) = \frac{|\mathrm{NClass}\,(M_{CM}) - \mathrm{NClass}\,(M_{AB})|}{\mathrm{NClass}\,(M_{AB})}\\ &\quad + \frac{|\mathrm{DepOut}\,(M_{CM}) - \mathrm{DepOut}\,(M_{AB})|}{\mathrm{DepOut}\,(M_{AB})}\\ &\quad + \frac{|\mathrm{NAttr}\,(M_{CM}) - \mathrm{NAttr}\,(M_{AB})|}{\mathrm{NAttr}\,(M_{AB})}\\ &\quad + \frac{|NInter(M_{CM}) - \mathrm{NInter}(M_{AB})|}{\mathrm{NInter}(M_{AB})} + \frac{|\mathrm{DIT}(M_{CM}) - \mathrm{DIT}(M_{AB})|}{\mathrm{DIT}(M_{AB})}\\ &= 0.2+0.14+0.22+0.11+0.11 \text{ (applying the Formula 2)}\\ &= 1+1+0+1+1 = 4\end{aligned}$$

Then,

$$\mathrm{Stability}(x, y)_{\mathrm{overall}} = 1 - \frac{4}{5} = 1 - 0.8 = 0.2.$$

3.5 Evaluation procedures

3.5.1 Target model versions and releases

To test the study hypotheses, we use the releases described in Table 3. Our key concern is to investigate these hypotheses considering a larger number of realistic SPL releases as possible in order to avoid bias of specific evolution scenarios.

Deriving SPL model releases For each release of the three product-line architectures, we have applied each of the composition heuristics [override, merge, and union (Sect. 2.3)] to compose two input models in order to produce a new release model. That is, each release was produced using the three algorithms. Similar compositions were performed using the override, merge, and union heuristics. This has helped us to identify scenarios where the SPL design models succumb (or not). For example, to produce the release three (R3) of the Mobile Media (Table 3), the developers combine R3 with a delta model that represents the model elements that should be inserted into R3 in order to transform it into R4. For this, the developers use the composition heuristics described in Sect. 2.3. A practical example about how these models are produced can be seen in Figs. 2 and 3.

Model releases and composition specification In Table 3, the releases were selected because visible and structural modifications in the architectural design were carried out to add new features. For each new release, the previous release was changed in order to accommodate the new features. To implement a new evolution scenario, a composition heuristic can remove, add, or update the entities present in the previous model release. Throughout the design of all releases, a main concern was to use good modeling practices in addition to the design-for-change principles. For example, assuming that the mean of the coupling measure of M_{CM} and $M_{\mathrm{AB}} = 9$ and 11, respectively. So M_{CM} is stable regarding M_{AB} (because nine is 18 % lower than 11). Following this stability threshold, we can systematically identify if the M_{CM} keeps stable over the evolution scenarios.

3.5.2 Execution and analysis phase

Model definition stage This step is a pivotal activity to define the input models and to express the model evolution as a model composition. The evolution has two models: the base model, M_{A}, the current release, and the delta model, M_{B}, which represents the changes that should be inserted into M_{A} to transform it into M_{CM}, as previously discussed. Considering the product-line design models used in the case studies, M_{B} represents the new design elements realizing the new feature. Then, a composition relationship is specified between

M_A and M_B so that the composed model can be produced, M_{CM}.

Composition and measurement stage In total, 180 compositions were performed, being 60 in the Mobile Media, 60 in the Shogi Game and 60 in the Checkers Game. The compositions were performed manually using the IBM RSA [19,35]. The result of this phase was a document of composition descriptions, including the gathered data from the application of our metrics suite and all design models created. We used a well-validated suite of inconsistency metrics applied in previous work [14] focused on quantifying syntactic and semantic inconsistencies. The syntactic inconsistencies were quantified using the IBM RSA's model validation mechanism. The semantic inconsistencies were quantified using the SDMetrics tool [46]. In addition, we also check both syntactic and semantic inconsistencies manually because some metrics, e.g., "the number of non-meaningful model elements" depends on the meaning of the model elements and the current modeling tools are unable to compute this metric.

The identification of the inconsistencies was performed in three review cycles in order to avoid false positives and false negatives. We also consulted the developers as needed, such as checking and confirming specific cases of semantic inconsistencies. On the other hand, the well-formedness (syntactic and semantic) rules defined in the UML metamodel were automatically checked by the IBM RAS's model validation mechanism.

Effort assessment stage The goal of the third phase was to assess the effort to resolve the inconsistencies using the quantification method described in Sect. 3.4. The composition heuristics were used to generate the evolved models, so that we could evaluate the effect of stability on the model composition effort. In order to support a detailed data analysis, the assessment phase was further decomposed in two main stages. The first stage is concerned with pinpointing the inconsistency rates produced by the compositions (H1). The second stage aims at assessing the effort to resolve a set of previously identified inconsistencies (H2). All measurement results and the raw data are available at [12].

4 Result analysis

This section analyzes the data set obtained from the experimental procedures described in Sect. 3. Our findings are derived from both the numerical processing of this data set and the graphical representation of interesting aspects of the gathered results. Section 4.1 elaborates on the gathered data in order to test the first hypothesis (H1). Section 4.2 discusses the collected data related to the second hypothesis (H2).

4.1 H1: Stability and inconsistency rate

4.1.1 Descriptive statistics

This section describes aspects of the collected data with respect to the impact of stability on the inconsistency rate. For this, descriptive statistics are carefully computed and discussed. Understanding these statistics are key steps to know the data distribution and grasp the main trends. To go about this direction, not only the main trend was calculated using the two most used statistics to discover trends (mean and median); the dispersion of the data around them was also computed mainly making use of the standard deviation. Note that these statistics are calculated from 180 compositions, i.e., with 60 compositions applied to the evolution of MobileMedia SPL, 60 compositions applied to the Shogi SPL, and 60 compositions applied to the Checkers SPL.

Table 4 shows descriptive statistics about the collected data regarding inconsistency rate. Figure 4 depicts the boxplot of the collected data. By having carried out a thorough analysis of this statistic, we can observe the positive effects of high level of stability on the inconsistency rate. In fact, we observe only harmful effects in the absence of stability. The main outstanding finding is that inconsistency rate in the stable design model is lower than in the unstable design model. This result is supported by some observations described as follows (see Fig. 4):

First, the median of inconsistency rate in stable models is considerably lower than in unstable models. That is, a mean of 0.31 in relation to the intended model instead of 3.86 presented by unstable models. This means, for example, that stable SPL models can present no inconsistencies in some cases. On the other hand, unstable models probably hold a higher inconsistency rate than that presented by stable models. This comprises normally 3.86 inconsistencies in relation to the intended model. This implies, for example, that if the output composed model is unstable, then there is a high probability of having inconsistencies in these models.

Stable models have a favorable impact on the inconsistency rate. More importantly, its absence has harmful consequences for the number of inconsistencies. These negative effects are evidenced by the significant difference between the number of inconsistencies in stable and unstable models. In fact, stable models tend to have just 8.1 % of the inconsistencies that are found in unstable models, compared with the medians 0.31 (stable) and 3.86 (unstable). One of the main reasons is because *inconsistency propagations* are found in unstable models more frequently. This means that developers must check all model elements so that they can identify and manipulate the composed model so that the intended model can be obtained.

Another interesting finding is that the inconsistencies tend to be quite close to the central tendency in stable models,

Table 4 Descriptive statistics of the inconsistency rate

Variables	Groups	*N*	Min	25th	Median	75th	Max	SD
Inconsistency rate	Stable	78	0	0.11	0.31	0.78	3.86	0.84
	Unstable	102	0.17	1.64	3.86	6.88	9.21	2.63

N number of composed models, SD standard deviation

Table 5 Mann–Whitney test and Spearman's correlation analysis

Variable	Groups	*N*	Mean rank	Rank sum	SC	*t* value*	*p*
Inconsistency rate	Stable	78	49.04	3,825	−0.71	−13.56	< 0.001
	Unstable	102	122.21	12,465			

* With 178 degree of freedom

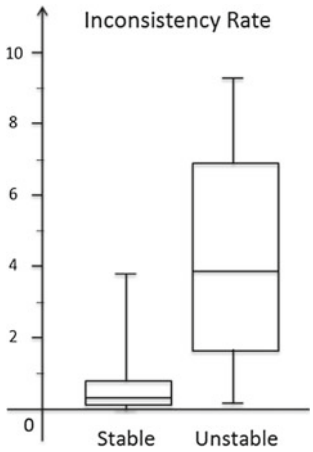


Fig. 4 *Box-plot* of inconsistency rate

with a standard deviation equal to 0.84. On the other hand, in unstable models, these inconsistencies tend to spread out over a large range of values. This is represented by a high value of the standard deviation that is equal to 2.63. It is important to point out that to draw out valid conclusions from the collected data it is necessary to analyze and possibly remove outliers from the data. Outliers are extreme values assumed by the inconsistency measures that may influence the study's conclusions. To analyze the threat of these outliers to the collected data, we made use of box-plots (Fig. 4). According to [45], it is necessary to verify whether the outliers are caused by an extraordinary exception (unlikely to happen again), or whether the cause of the outlier can be expected to happen again. Considering the first case, the outliers must be removed, and in the last case, they must not be removed. In our study, some outliers were identified; however, they were not extraordinary exceptions since they could happen again. Consequently, they were left in the collected data set, as they do not affect the results.

4.1.2 Hypothesis testing

We performed a statistical test to evaluate whether in fact the difference between the inconsistency rates of stable and unstable models are *statistically significant*. As we hypothesize that stable models tend to exert a lower inconsistency rate than unstable models, the test of the mean difference between stable and unstable groups will be performed as one-tailed test. In the analyses, we considered significance level at 0.05 level ($p \leq 0.05$) to indicate a true significance.

a. Mann–Whitney test

As the collected data violated the assumption of normality, the non-parametric Mann–Whitney test was used as the main statistical test. The results produced are $U' = 7.21, U = 744, z = 9.33$ and $p < 0.001$. The p value is lower than z and 0.05. Therefore, the null hypothesis of no difference between the rates of inconsistency in stable and unstable models (H_{1-0}) can be rejected. That is, there is sufficient evidence to say that the difference between the inconsistency rates of stable and unstable models are statistically significant.

Table 5 depicts that the mean rank of inconsistency rate for unstable models are higher than that of stable models. As the Mann–Whitney test [45] relies on ranking scores from lowest to highest, the group with the lowest mean rank is the one that contains the largest amount of lower inconsistency rate. Likewise, the group with the highest mean rank is the group that contains the largest amount of higher inconsistency rate. Hence, the collected data confirm that unstable models tend to have a higher inconsistency rate than the stable design models.

b. Correlation

To examine the strength of the relationship (the correlation coefficient) between stability and inconsistency rate, the Spearman's correlation (SC) test was applied (see Table 5). Pearson's correlation is not used because the data sets are not normally distributed. Note that this statistic test assumes that both variables are independent. The correlation coefficient takes on values between −1 and 1. Values close to 1 or −1 indicate a strong relationship between the stability and inconsistency rate. A value close to zero indicates a weak or non-existent relationship.

As can be seen in Table 5, the *t* test of significance of the relationship has a low *p* value, indicating that the correlation

is significantly different from zero. Spearman's correlation analysis resulted in a negative and significant correlation (SC $= -0.71$). The negative value indicates an inverse relationship. That is, as one variable increases, the other decreases. Hence, composition inconsistencies tend to manifest more often in unstable models than stable models. The above correlation suggests that whereas the model stability of the output composed model decreases the inconsistency rate in their models increases.

Therefore, the results suggest that, on average, stable models tend to have a *significantly* lower inconsistency rate than unstable design models. Therefore, we believe that the results confirm the indication of correlation between stability and inconsistency rate. Consequently, the null hypothesis (H_{1-0}) can be rejected and the alternative hypothesis (H_{1-1}) confirmed.

4.1.3 Discussion

a. The effect of severe evolution categories

After discussing how the dataset is grouped, grasping the main trends, and studying the relevance of the outliers, the main conclusion is that stable models tend to present a lower inconsistency rate than unstable models. This finding can be seen as the first step to overcome the lack of practical knowledge about the effects of the model stability on the inconsistency rate in realistic scenarios of model evolution supported by composition heuristics. Some previous studies (e.g., [21,38]) also check similar insights on the code level. These studies report a positive association between low variation of coupling and size with stability.

We have noticed that although the input design models (M_A and M_B) are well structured, they are the target of widely scoped inconsistencies in certain model composition scenarios. These widely scoped inconsistencies are motivated by unexpected modifications in specific design characteristics of the design models such as coupling and cohesion. These scenarios mainly occurred when composition heuristics accommodate unanticipated, severe changes from M_A to M_B. The most challenging changes observed are those related to the refinement of the MVC (Model-View-Controller) architecture design of the SPLs used in this study.

Another observation is that the composition heuristics (override, merge, and union) are not effective to accommodate these changes from M_A to M_B. The main reason is that the heuristics are unable to "restructure" the design models in such way that these changes do not harm static or behavioral aspects of the design models. These harmful changes usually emerge with a set of ever-present evolving change categories, such as a *modification* of the model properties and *derivation* of new model elements (e.g., components or classes) from other existing ones.

In the first category, *modification*, model elements have some properties affected. This is typically the case when a new operation conflicts with an operation defined previously. In Fig. 2, for example, the operation $getImage()$ in the interface *R2.HandleException* had its return type, $String[]$, conflicting with the return type, $ImageData[]$ of the interface $Delta(R2, R3).HandleException$. Another example is the component *ManageAlbum* that had its name modified to *ManageLabel* to express semantic alterations in the concepts used to realize the error-handling feature. Only one of the names and return types can be accepted, but the two modifications cannot be combined. Both cases are scenarios in which the heuristics are unable to correctly pick out what element must be renamed and what return type must be considered. The problem is that detection and decision of these inconsistencies demand a thorough understanding of: (i) what the design model elements actually mean as well as the domain terms "Album" and "Label"; and (ii) the expected semantics of the modified method. In addition, semantic information is typically not included in any formal way so that the heuristics can infer the most appropriated choice. Consequently, the new model elements responsible for implementing the added features are presented with overlapping semantic values and unexpected behaviors. Interestingly, this has been the case where existing optional as well as alternative features are involved in the change.

In the second category, *derivation*, the changes are more severe. Architectural elements are refined and/or moved in the model to accommodate the new changes. Differently from the previous category, the affected architectural elements are usually mandatory features because this kind of evolution in software product lines is mainly required to facilitate the additions of new variabilities or variants later in the project. Unfortunately, in this context of more widely scoped changes, the heuristic-based composition heuristics have demonstrated to be ineffective.

A concrete example of this inability is the refinement of the MVC architecture style of the MobileMedia SPL in the third evolution scenario. In practical terms, the central architectural component, *BaseController*, is broken into other controllers such as *PhotoListController*, *AudioController, VideoController* and *LabelController* to support a better manipulation of the upcoming media like photo, audio, video and the label attached to them. This is partially due to the name-based model comparison policy in the heuristics, which are unable to recognize more intricate equivalence relationships between the model elements. Indeed, this comparison strategy is very restrictive whenever there is a correspondence relationship 1:N between elements in the two input models. That is, it is unable to match the upcoming four controllers with the previous one, *BaseController*.

A practical example of this category of relationship (1:N) involves the required interface *ControlPhoto* (release

three) of the *AlbumListScreen* component. This interface was decomposed into two new required interfaces *ControlAlbum* and *ControlPhotoList* (release four), thereby characterizing a relationship 1:2. In 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 model (release four), the *AlbumListScreen* component provides duplicate services to the environment giving rise to a severe inconsistency.

b. Inconsistency propagation

After addressing the hypotheses and knowing that instabilities have a detrimental effect on the density of inconsistencies, we analyze whether the location where they arise (i.e., architectural elements realizing mandatory, or optional features) can cause some unknown side effects. Some interesting findings were found, which is properly discussed as follows: To begin with, instability problems are more harmful when they take place in design model elements realizing mandatory features. This can be explained by some reasons.

First, the *inconsistency propagation* is often higher in model elements implementing mandatory features than in alternatives (or optional features). When inconsistencies arise in elements realizing optional and alternative features they also tend to naturally cascade to elements realizing mandatory features. Consequently, the mandatory features end up being the target of inconsistency propagation.

Based on the knowledge that mandatory features tend to be more vulnerable to ripple effects of inconsistencies, developers must structure product-line architectures in such a way that inconsistencies can keep precisely "confined" in the model elements where they appear. Otherwise, the quality of the products extracted from the SPL can be compromised as the core elements of the SPL can suffer from problems caused by incorrect feature compositions. The higher the number of inconsistencies, the higher the chance of them to continue in the same output model, even after an inspection process performed by a designer. Consequently, the extraction of certain products can become error-prone or even prohibitive.

The second interesting insight is that the higher the instability in optional features, the higher the inconsistency propagation toward mandatory features. However, the propagation in the inverse order (i.e., from alternative and optional to mandatory features) seems to be less common. In Fig. 2 (override), a practical example can be seen. The instability in mandatory features, "album and photo management," compromises the optional feature, "edit photo's label." The *NewLabelScreen* component (optional feature) has its two services i.e., $getLabelName()$ and $getFormType()$ (specified in the interface $ManageLabel()$ compromised. The reason is that the required service $editLabel()$ cannot be provided by the *BaseController* (mandatory feature). Thus, the "edit photo' label" feature can no longer be provided due to problems in the mandatory feature "album and photo management." For example, in the fourth evolution scenario of the Checkers Game, the optional feature, *Customize Pieces*, is correctly glued to the R4 using the *override* heuristic so that the new release, R5, can be generated. The problem is that the inconsistencies that emerge in the architectural component, *Command,* are propagated to the architectural elements *CustomizePieces* and *GameManager*. Thus, the mandatory feature "piece management" implemented by the Command is affecting the optional feature "customize pieces" implemented by the components *CustomizePieces* and *GameManager.*

Although the optional feature, *Customize Pieces*, has been correctly attached to the base architecture, the composed models will not have the expected functionality related to the customization of pieces.

4.2 H2: Stability and resolution effort

4.2.1 Descriptive statistics

This section discusses interesting aspects of the collected data concerning the impact of stability on the developers' effort. The knowledge derived from them helps to understand the effects of model stability on the inconsistency resolution effort. In a similar way to the previous section, we calculate the main trend and the data dispersion.

Table 6 provides the descriptive statistics of sampled inconsistency resolution effort in stable and unstable model groups. Figure 5 graphically depicts the collected data by using box-plot. To begin with our discussion, we first compare the median values of the inconsistency resolution effort of the both stable and unstable groups. We can observe that the median of the stable models (equals to six) is much lower than that one of unstable models (equals to 111).

This superiority of the unstable models is also observed in the mean and standard deviation, which represent the main trend and dispersion measures, respectively. The gathered results, therefore, indicate that stable models claim less resolution effort than unstable models. This means that developers tend to perform a lower amount of tasks (creations, removals, and modifications) to transform the composed model into the intended model. Although we have observed some outliers, e.g., the maximum value (368) registered in

Table 6 Descriptive statistics of the resolution effort (min)

Variables	Groups	*N*	Min	25th	Median	75th	Max	SD
Resolution effort	Stable	78	0	3,50	6	13	46	10.29
	Unstable	102	4	27	111	229.25	368	106.7

N number of composed models, *SD* standard deviation

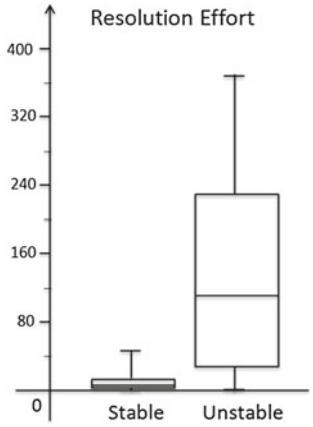


Fig. 5 *Box-plot* of resolution effort in relation to the intended model

unstable models, they are not an extraordinary exception as they could happen again. Consequently, they were left in the collected data set, as they do not tamper the results.

4.2.2 Hypothesis testing

Given the difference between the mean and median described in the descriptive analysis, statistical tests are applied to assess whether in fact the difference in effort to fix unstable model and stable model is statistically significant. We conjecture that stable models tend to require a lower inconsistency resolution effort than unstable models. Hence, a one-tailed test is performed to test the significance of the mean difference between stable and unstable groups. Again, in the analyses we considered significance level at 0.05 level ($p \leq 0.05$) to indicate a true significance.

a. Mann–Whitney test

As the dataset does not respect the assumption of normality, we use the non-parametric Mann–Whitney test as the main statistical test. The results of the Mann–Whitney test produced are $U' = 7.372, U = 584, z = 9.79$ and $p < 0.001$. The p value is lower than z and 0.05; therefore, the null hypothesis can be rejected. In other words, there exists a difference between the efforts required to resolve inconsistencies in stable and unstable model groups. In fact, there is substantial evidence pointing out the difference between the median measures of the two groups.

Table 7 shows that the difference between the mean ranks is significant. The mean of rank in stable models consists of about 38 of the mean rank in unstable models. As the Mann–Whitney test relies on ranking scores from lowest to highest, the group with the lowest mean rank is the one that requires the highest incidence of lowest effort. Likewise, the group with the highest mean rank is the group that contains the largest occurrence of higher effort needed. Hence, the collected data show that unstable models that are not stable tend to have higher effort than the stable models.

b. Correlation Analysis

As the gathered data do not follow a normal distribution, we apply the Spearman's correlation test. Table 7 provides the results of the Spearman's correlation test. The low p value $<$ 0.001 indicates that the correlation significantly departs from zero. Recall that Spearman's correlation value close to 1 or -1 indicates a strong relationship between the stability and effort. On the other hand, a value close to 0 indicates a weak or non-existent relationship. The results (SC $= -0.698$) suggest that there is a negative and significant correlation between the two variables. This implies that whereas the stability increases the effort to resolve inconsistency decreases. Consequently, stable models required much lesser effort to be transformed into the intended model than unstable models. Based on such results, we can reject the null hypothesis (H_{2-0}), and accept the alternative hypothesis (H_{2-1}): stable models tend to require lower effort to resolve composition inconsistency than unstable models.

4.2.3 Discussion

a. The effect of instability on resolution effort

In Sect. 4.1, we discuss that the inconsistencies in the model elements realizing optional features tend to propagate to ones realizing mandatory features. Inconsistencies in elements realizing optional features tend to affect the structure of model elements realizing mandatory features. The reason is that some relationships are often introduced between elements realizing mandatory and optional features during the composition. Considering the resolution effort, we have observed that the higher instability in optional features, the higher the resolution effort. Developers need to resolve a cascading chain of inconsistencies, and usually this process should be recursively applied until all inconsistencies have been resolved. This resolution is more effort consuming because widely scoped changes are required to tame such ripple effects. The required effort is to restructure the composed model.

Table 7 Mann–Whitney test and Spearman's correlation analysis

Variable	Groups	*N*	Mean rank	Rank sum	SC	*t* value*	*p*
Resolution effort	Stable	78	46,99	3,665	−0.698	−13	< 0.001
	Unstable	102	123,77	12,625			

* With 178 degree of freedom

We have identified that this superior effort to resolve inconsistencies is due to the syntactic-based composition heuristics are unable to deal with occurring semantic conflicts between the model elements of mandatory and optional features. As a result, inconsistencies are formed. In Fig. 3, for example, the component *BaseController* requires services from a component *NewALbumScreen* that provides just one mandatory feature "create album" rather than from a component that provides two features: "create album" and "edit photo's label." This is because releases R2 and R3 use different component names (*R2.NewAlbumScreen* and *R3.NewLabelScreen*) for the same purpose. That is, they implement the mandatory feature *Create Album* in components with contracting names.

A syntax-based composition is unable to foresee these kinds of semantic inconsistencies, or even indicate any problem in *BaseController* as the component remains syntactically correct. From R2 to R3, the domain term *Album* was replaced by *Label*. However, the purely syntactical, match-by-name mechanism is unable to catch and incorporate this simple semantic change into the composition heuristic. To overcome this, a semantic-based approach would be required to allow, for example, a semantic alignment between these two domain terms. Consequently, the heuristics would be able to properly match *R2.NewAlbumScreen* and *R3.NewLabelScreen.*

Still in Fig. 2, the architectural model R3, which was produced following *merge heuristic*, contains a second facet of semantic problem: *behavioral inconsistency*. The component *ExceptionHandling* provides two services with the same purpose, $getImage():String[]$ and $getImage():ImageData[]$. However, they have different semantic values. This contrasting characteristic is emphasized by the different return types, $String[]$ and $ImageData[]$. However, in this case, the inconsistency got confined in the optional feature rather than propagating to model elements implementing mandatory features. To resolve the problem, the method $getImage():String[]$ should be removed. In total, only one operation is performed. Thus, these inconsistencies can be only pinpointed by resorting to sophisticated semantics-based composition, which relies on the action semantics of the model elements. According to [28], the current detection of behavioral inconsistency is based on the complex mathematical, program slicing, and program dependence graphs. Unfortunately, none of them is able to systematically compare behavioral aspects of components neither realizing two features nor even composing them properly. Even worse, the composition techniques would be unable to match, for example, *ManageAlbum* and *ManageLabel* interface.

b. The effect of multiple concerns on resolution effort

Another finding is that the higher the number of features implemented by a model element, the higher the resolution effort. We have observed that model elements realizing multiple features tend to require more inconsistency resolution effort than those realizing just one feature. The reason is that the model elements realizing multiple features tend to receive a higher number of upcoming changes to-be accommodated by the composition heuristics than ones realizing a single feature. These model elements become more vulnerable to the unpredictable effects of the severe evolution categories. This means that developers tend to invest more effort to resolve all possible inconsistencies.

In fact, a higher number of inconsistencies have been observed in 'multiple-featured' components rather than in 'single-featured' components. As developers cannot foresee or even precisely identify all ripple effects of these inconsistencies through other model elements, the absence of stability can be used as a good indicator of inconsistency. Let us consider the *BaseController*, the central controller in MobileMedia architecture that implements two features (see Fig. 2). The collected data show that the *BaseController* was modified in almost all evolution scenarios because it is a pivotal architectural component in the *model-view-control* architectural style of the SPL MobileMedia. Unfortunately, the changes cannot be properly realized in all cases. In addition, we observe that *BaseController*'s inconsistencies affect other four components, namely *NewLabelScreen*, *AlbumListScreen*, *PhotoListScreen*, *PhotoViewScreen*, and *AddPhotoToAlbumScreen*. All these affected components require the provided services by the *BaseController*.

Moreover, we notice that the *BaseController* had a higher likelihood to receive inconsistencies from other model elements than any other components. The reason is that it also depends on many other components to provide the services of the multiple features. For example, *BaseController* can be harmed by inconsistencies arising from the components*ManageAlbum, ManagePhotoInfo,* and *ControlPhoto.* This means that, at some point, *BaseController* can no longer provide its services because it was probably affected by inconsistencies located in these components.

It is interesting to note that *NewAlbumScreen* is also affected by an inconsistency that emerged from *AlbumData*, as it requires the service (*viewPhoto*) provided by the *BaseController* in the interface, *ControlPhoto* that cannot be accessed. The main reason is that the service, *resetImageData*(), specified in the interface *ManagePhotoInfo* can no longer be provided by the component *AlbumData*, compromising the serviced offered in the interface *ControlPhoto*. Since *BaseController* is not able to correctly provide all services defined in the provided interface *ControlPhoto*, it is also re-affected by an inconsistency that previously arose from it. This happens because *NewAlbumScreen* does not provide the services described in the interface *ManageAlbum*. This phenomenon represents the *cyclic inconsistency propagation*. Understanding this type of phenomenon,

the software designer can examine upfront and more precisely the design models in order to localize undetected cyclic dependence between the model elements.

Another observation is that optional features are also harmed by this propagation on the mandatory features. For example, the *PhotoSorting* component (realizing optional feature "sorting photos") is unable to provide the service, $sortCommand()$, specified in the interface *SoftPhoto*. This is due to the absence of the required service, $resetImageData()$ from the *ManagePhotoInfo* interface, which the mandatory feature "album management." In practical terms, it indicates that undesired effects in features can be due to some unexpected instabilities in the mandatory features. In collaborative software development, for example, this is a typical problem because the model elements implementing different features are developed in parallel, but they rarely prepared upfront to-be composed. Hence, developers should invest some considerable effort to properly promote the composition.

5 Related work

To the best of our knowledge, our results are the first to investigate empirically the relation between quality attributes and model composition effort in a broader context. In [13], we initially investigated the research questions addressed in this paper, but they were evaluated in a smaller scope. This paper, therefore, represents an extension of the results obtained previously. The main extensions can be described as follows: (1) two more case studies were performed, i.e., the evolution studies with the Shogi and Checkers SPLs. This implies that the number of compositions jumped from 60 to 180; (2) new lessons learned were obtained from a broader study; and (3) the size of the sample data was higher than the previously found; hence, the hypotheses might be better tested.

We have observed not only a wide variety of model composition techniques [9,25] have been created, but also some previous works [13,33] have demonstrated that stability is a good predictor of defects [33] and the presence of good designs [21]. However, none of them has directly investigated the impact of stability on model composition effort.

The lack of empirical evidence hinders the understanding of the side effects peculiar to stability on developers' effort. Consequently, developers in industrial projects have to rely solely on feedback from experts to determine "the goodness" of the input models and their compositions. In fact, according to several recent observations [9,18,28], the state of the practice in model quality assessment indicates that modeling is still in the craftsmanship era and this problem is even more accentuated in the context of model composition.

The current model composition literature does not provide any support to perform empirical studies considering model composition effort [18,28], or even to evaluate the effects of model stability on composition effort. In [18], the authors highlight the need empirical studies in model composition to provides insights about how deal with ever-present problems such as conflicts and inconsistencies in real world settings. In [28], Mens also reveals the need for more "experimental researches on the validation and scalability of syntactic and semantic merge approaches, not only regarding conflict detection, but also regarding the amount of time and effort required to resolve the conflicts." Without empirical studies, researchers and developers are left without any insight about how to evaluate model composition in practice. For example, there is no metric, indicator, or criterion available to assess the UML models that are merged through, for instance, the UML built-in composition mechanism (i.e., package merge) [11,37].

There are some specific metrics available in the literature for supporting the evaluation of model composition specifications. For instance, Chitchyan et al. [8] have defined some metrics, such as scaffolding and mobility, to quantify quality attributes of compositions between two or more requirements artifacts. However, their metrics are targeted at evaluating the reusability and stability of explicit descriptions of model composition specifications. In other words, their work is not targeted at evaluating model composition heuristics. Boucké et al. [4] also propose a number of metrics for evaluating the complexity and reuse of explicitly defined compositions of architectural models. Their work is not focused on heuristic-based model composition as well. Instead, we have focused on analyzing the impact of stability on the effort to resolve emerging inconsistencies in output models. Therefore, existing metrics (such as those described in [36]) cannot be directly applied to our context.

Although we have proposed a metric suite for quantifying inconsistencies in UML class diagrams and then applied these metrics to evaluate the composition of aspect-oriented models and UML class diagrams [14], nothing has been done to understand the effects of model stability on the developers' effort. Some previous works investigated the effect of using UML diagrams and its profiles with different purposes. In [6], Briand et al. looked into the formality of UML models and its relation with model quality and comprehensibility. In particular, Briand et al. investigated the impact of using OCL (Object Constraint Language [37]) on defect detection, comprehension, and impact analysis of changes in UML models. In [40], Filippo et al. carried out a series of four experiments to assess how developer's experience and ability influence Web application comprehension tasks supported by UML stereotypes. Although they have found that the use of UML models provide real benefits for typical software engineering activities, none has investigated the peculiarities of UML models in the context of model composition. Finally, we therefore see this paper as a first step in a more ambitious agenda to support

empirically the assessment of model composition techniques in general.

6 Threats to validity

Our exploratory study has obviously a number of threats to validity that range from internal, construct, statistical conclusion validity threats to external threats. This section discusses how these threats were minimized and offers suggestions for improvements in future study.

6.1 Internal validity

Inferences between our independent variable (stability) and the dependent variables (inconsistency rate and composition effort) are internally valid if a causal relation involving these two variables is demonstrated [5,41]. Our study met the internal validity because: (1) the *temporal precedence* criterion was met, i.e., the instability of design models preceded the inconsistencies and composition effort; (2) the covariation was observed, i.e., instability of design models varied accordingly to both inconsistencies and composition effort; and (3) there is no clear extra cause for the detected covariation. Our study satisfied all these three requirements for internal validity.

The internal validity can be also supported by other means. First, the detailed analysis of concrete examples demonstrating how the instabilities were constantly the main drivers of inconsistencies presented in this paper. Second, our concerns throughout the study to make sure that the observed values in the inconsistency rates and composition effort were confidently caused by the stability of the design models. However, some threats were also identified, which are explicitly discussed below.

First, due to the exploratory nature of our study, we cannot state that the internal validity of our findings is comparable to the more explicit manipulation of independent variables in controlled experiments. This exceeding control employed to deal with some factors (i.e., with random selection, experimental groups, and safeguards against confounding factors) was not used because it would significantly jeopardize the external validity of the findings.

Second, another threat to the internal validity is related to the imperfections governing the measurements of inconsistency rate and resolution effort. As the measures were partially calculated in a manual fashion, there was the risk that collected data would not be always reliable. Hence, this could lead to inconsistent results. However, we have mitigated this risk by establishing measurement guidelines, two-round data reviews with the actual developers of the SPL design models, and by engaging them in discussions in cases of doubts related to, for instance, the semantic inconsistencies.

Next, usually the *confounding* variable is seen as the major threat to the internal validity [41]. That is, rather than just the independent variable, an unknown third variable unexpectedly affects the dependent variable. To avoid *confounding* variables in our study, a pilot study was carried out to make sure that the inconsistency rate and composition effort were not affected by any existing variable other than stability. During this pilot study, we tried to identify which other variables could affect the inconsistency rate and resolution effort such as the size of the models.

Another concern was to deal with the *experimenter bias.* That is, the experimenters inadvertently affect the results by unconsciously realizing experimental tasks differently that would be expected. To minimize the possibility of experimenter bias, the evaluation tasks were performed by developers, which that know neither the purpose of the study nor the variables involved. For example, developers created the input design models of the SPLs without being aware of the experimental purpose of the study. In addition, the composition heuristics can be automatically applied. Consequently, the study results can be more confidently applied to realistic development settings without suffering influences from experimenters.

Finally, the randomization of the subjects was not performed because it would require simple task simple software engineering task. Hence, this would undermine the objective of this study (Sect. 3.1).

6.2 Statistical conclusion validity

We evaluated the statistical conclusion validity checking if the independent and dependent variables (Sect. 3.4) were submitted to suitable statistical methods. These methods are useful to analyze whether (or not) the research variables covary [10]. The evaluation is concerned on two related statistical inferences: (1) whether the presumed cause and effect covary, and (2) how strongly they covary [10].

Considering the first inferences, we may improperly conclude that there is a causal relation between the variables when, in fact, they do not. We may also incorrectly state that the causal relation does not exist when, in fact, it exists. With respect to the second inference, we may incorrectly define the magnitude of covariation and the degree of confidence that the estimate warrants [7,45].

Covariance of cause and effect We eliminated the threats to the causal relation between the research variables studying the normal distribution of the collected sample. Thus, it was possible to verify if parametric or non-parametric statistical methods could be used (or not). For this purpose, we used the Kolmogorov–Smirnov test to determine how likely the collected sample was normally distributed. As the dataset did not assume a normal distribution, non-parametric statistics

were used (Sects. 4.1 and 4.2). Hence, we are confident that the test statistics were applied correctly, as the assumptions of the statistical test were not violated.

Statistical significance Based on the significance level at 0.05 level ($p \leq 0.05$), Mann–Whitney test was used to evaluate our formulated hypotheses. The results collected from this test indicated $p < 0.001$. This shows sufficient evidence to say that the difference between the inconsistency rates (and composition effort) of stable and unstable models are statically significant. The correlation between the independent and dependent variables is also evaluated. For this, Spearman's correlation test was used. The low collected p value (<0.001) indicated that there is a significant correlation between the inconsistency rate and stability as well as composition effort and stability.

In addition, we followed some general guidelines to improve conclusion validity [39,45]. First, a high number of compositions were performed to increase the sample size, hence improving the statistical power. Second, experienced developers used more realistic design models of SPLs, state-of-practice composition heuristics, and robust software modeling tool. These improvements reduced "errors" that could obscure the causal relationship between the variable under study. Consequently, it brought a better reliability for our results.

6.3 Construct validity

Construct validity concerns the degree to which inferences are warranted from the observed cause and effect operations included in our study to the constructs that these instances might represent. That is, it answers the question: "Are we actually measuring what we think we are measuring?" With this in mind, we evaluated (1) whether the quantification method is correct, (2) whether the quantification was accurately done, and (3) whether the manual composition threats the validity.

Quantification method All variables of this study were quantified using a suite of metrics, which was previously defined and independently validated [14,21]. Moreover, the concept of stability used in our study is well known in the literature [21] and its quantification method was reused from previous work. The inconsistencies were quantified automatically using the IBM RSA's model validation mechanisms and manually by the developers through several cycles of measurements and reviews. In practice, the developers' effort is computed by "time spent." However, the "time spent" is a reliable metric when used in controlled experiments. Unfortunately, controlled experiments require that the software engineering tasks are simple; hence, it harms the objective of our investigation (Sect. 3.1) and hypotheses (Sect. 3.2). Moreover, we have observed in the examples of recovering models that, in fact, the "time spent" is actually greater for unstable models than stable models, independently of the type of inconsistencies. In addition, the number of syntactic and semantic inconsistencies was always higher in unstable models than stable models.

Correctness of the quantification Developers worked together to assure that the study does not suffer from construct validity problems with respect to the correctness of the compositions and application of the suite of metrics. We checked if the collected data were in line with the objective and hypotheses of our study. It is important to emphasize that just one facet of composition effort was studied: the effort to evolve well-structured design models using composition heuristics. The quantification procedures were carefully planned and followed well-known quantification guidelines [3,23,24,45].

Execution of the compositions Another threat that we have controlled is if the use of manual composition might unintentionally avoid conflicts. We have observed that the manual composition helps to minimize problems that are directly related to model composition tools. There are some tools to compose design models, such as IBM Rational Software Architect. However, the use of these tools to compose the models was not included in our study for several reasons. First, the nature of the compositions would require that developers understood the resources/details of the tools. Second, even though the use of these tools might intentionally reduce (or exacerbate) the generation of specific categories of inconsistencies in the output composed models, it was not our goal to evaluate particular tools. Therefore, we believe that by using a model composition tool would impose more severe threats to the validity of our experimental results. Finally, and more importantly, we do not think the manual composition would be a noticeable problem in the study for two reasons. First, even if the conflicts were unconsciously avoided, we deeply believe that the heuristics should be used as "rules of thumb" (guidelines) even if tool support is somehow available. Second, we have reviewed the produced models, at least, three times in order to ensure that conflicts were injected accordingly; in the case they still made their way to the models used in our analysis, they should be minimal.

6.4 External validity

External validity refers to the validity of the obtained results in other broader contexts [31]. That is, to what extent the results of this study can be generalized to other realities, for instance, with different UML design models, with different developers and using different composition heuristics. Thus, we analyzed whether the causal relationships investigated in this study could be held over variations in people, treatments, and other settings.

As this study was not replicated in a large variety of places, with different people, and at different times, we made use of the theory of proximal similarity (proposed by Campbell [7]) to identify the degree of generalization of the results. The goal is to define criteria that can be used to identify similar contexts where the results of this study can be applied. Two criteria are shown as follows: First, developers should be able to make use of composition heuristics (Sect. 2.3) to evolve UML design models such as UML class and component diagrams. Second, developers should also be able to apply the inconsistency metrics described in Table 2 and use some robust software modeling tool (e.g., IBM RSA [19]).

Given that these criteria can be seen as ever-present characteristics in mainstream software development, we conclude that the results of our study can be generalized to other people, places, or times that are more similar to these requirements. Some characteristics of this study contributed strongly to its external validity as follows: First, the reported exploratory study is realistic and, in particular, when compared to previously reported case studies and controlled experiments on composing design models [6,14]. Second, experienced developers used: (1) state-of-practice composition heuristics to evolve three realistic design models of software product lines; (2) industrial software modeling tool (i.e., IBM RSA) to create and validate the design models; and (3) metrics that were validated in previous works [14]. Next, the design models used were planned with the *design-for-change principles* upfront. Finally, this work investigates only one facet of model composition: the use of model composition heuristics in adding new features to a set of design models for three realistic software product lines.

7 Conclusions and future work

Model composition plays a pivotal role in many software engineering activities, e.g., evolving SPL design models to add new features. Hence, software designers are naturally concerned with the quality of the composed models.This paper, therefore, represents a first exploratory study to empirically evaluate the impact of stability on model composition effort. More specifically, the focus was on investigating whether the presence of stable models reduces (or not) the inconsistency rate and composition effort. In our study, model composition was exclusively used to express the evolution of design models along eighteen releases of three SPL design models. Three state-of-practice composition heuristics have been applied, and all were discussed in detail throughout this paper.

The main finding was that the model stability is a good indicator of composition inconsistencies and resolution effort. More specifically, we found that stable models tend to minimize the inconsistency rate and alleviate the model composition effort. This observation was derived from statistic analysis of the collected empirical data that have shown a significant correlation between the independent variable (stability) and the dependent variables (inconsistency rate and effort). Moreover, our results also revealed that instability in design models would be caused by a set of factors as follows: First, SPL design models are not able to support all upcoming changes, mainly unanticipated incremental changes. Next, the state-of-practice composition heuristics are unable to semantically match simple changes in the input model elements, mainly when changes take place in crosscutting requirements. Finally, design models implementing crosscutting requirements tend to cause a higher number of inconsistencies than the ones modularizing their requirements more effectively. The main consequence is that the evolution of the design models using composition heuristics can even become prohibitive given the effort required to produce the intended model.

As future work, we will replicate the study in other contexts (e.g., evolution of statecharts) to check whether (or not) our findings can be extended to different evolution scenarios of design models supported by composition heuristics. We also consider exploring different variants of the stability metrics. We also wish to better understand if design models with superior stability have some gain (or not): (i) when produced from another composition heuristics, and (ii) on the effort localizing the inconsistencies. It would be useful if, for example, intelligent recommendation systems could help the developers to indicate the best heuristic to-be applied to a given evolution scenario or even recommending how the input model should be restructured to prevent inconsistencies. Finally, we hope that the issues outlined throughout the paper encourage other researchers to replicate our study in the future under different circumstances and that this work represents a first step in a more ambitious agenda on better supporting model composition tasks.

References

1. Apel, S., Janda, F., Trujillo, S., Kästner, C.: Model superimposition in software product lines. In: International Conference on Model Transformation (ICMT), vol. 5563 (LNCS), pp. 4–19, Springer, Berlin (2009)
2. Asklund, U.: Identifying inconsistencies during structural merge. In: Proceedings of the Nordic Workshop Programming Environment Research, pp. 86–96 (1994)
3. Basili, V., Caldiera, G., Rombach, H.: The goal question metric paradigm. In: Encyclopedia of Software Engineering, vol. 2, pp. 528–532. Wiley, Hoboken (1994)

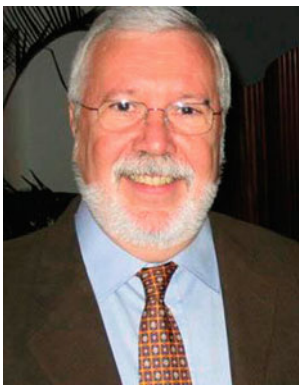
4. Boucké, N., Weyns, D., Holvoet, T.: Experiences with Theme/UML for architectural design in multiagent systems. In: MASSAA'06, pp. 87–110 (2006)
5. Brewer, M.: Research design and issues of validity. In: Handbook of Research Methods in Social and Personality Psychology, Cambridge University Press, Cambridge (2000)
6. Briand, L., Labiche, Y., Di Penta, M., BondocL, H.: An experimental investigation of formality in UML-based development. IEEE Trans. Softw. Eng. **31**(10), 833–849 (2005)
7. Campbell, D., Russo, M.: Social Experimentation. SAGE Classics, Beverly Hills (1998)
8. Chitchyan, R., Greenwood, P., Sampaio, A., Rashid, A., Garcia, A., Silva, L.: Semantic vs. syntactic compositions in aspect-oriented requirements engineering: an empirical study. In: International Conference on Aspect-Oriented Software, Development (AOSD'09), pp. 36–48 (2009)
9. Clarke, S., Walker, R.: Composition patterns: an approach to designing reusable aspects. In: 23rd International Conference on Software Engineering (ICSE'01), pp. 5–14, Toronto (2001)
10. Cook, T., Campbell, D., Day, A.: Quasi-Experimentation: Design & Analysis Issues for Field Settings. Houghton Mifflin, Boston (1979)
11. Dingel, J., Diskin, Z., Zito, A.: Understanding and improving UML package merge. J. SoSym **7**(4), 443–467 (2008)
12. Effects of stability on model composition effort: an exploratory study. http://www.les.inf.puc-rio.br/opus/sosym2012 (2012)
13. Farias, K., Garcia, A., Lucena, C.: Evaluating the effects of stability on model composition effort: an exploratory study. In : VIII Experimental Software Engineering Latin American Workshop collocated at XIV Iberoamerican Conference on Software Engineering, Rio de Janeiro (2011)
14. Farias, K., Garcia, A., Whittle, J.: Assessing the impact of aspects on model composition effort. In: AOSD'10, pp. 73–84, Saint Malo (2010)
15. Farias, K., Garcia, A., Lucena, C.: Evaluating the impact of aspects on inconsistency detection effort: a controlled experiment. In: 15th International Conference on Model-Driven Engineering Languages and Systems (MODELS'12), pp. 219–234, Innsbruck (2012)
16. Farias, K., Garcia, A., Whittle, J., Chavez, C., Lucena, C.: Evaluating the effort of composing design models: a controlled experiment. In: 15th International Conference on Model-Driven Engineering Languages and Systems (MODELS'12), pp. 676–691, Innsbruck (2012)
17. Figueiredo, et al.: Evolving software product lines with aspects: an empirical study on design stability. In: International Conference on Software Engineering (ICSE'08), pp. 261–270, Leipzig (2008)
18. France, R., Rumpe, B.: Model-driven development of complex software: a research roadmap. In: Future of Software Engineering at ICSE'07, pp. 37–54, Minneapolis (2007)
19. IBM Rational Software Architecture (IBM RSA). http://www.ibm.com/developerworks/rational/products/rsa/ (2011)
20. Jayaraman, P., Whittle, J., Elkhodary, A., Gomaa, H.: Model composition in product lines and feature interaction detection using critical pair analysis. In: International Conference on Model Driven Engineering Languages and Systems (MODELS), pp. 151–165, Nashville (2007)
21. Kelly, D.: A study of design characteristics in evolving software using stability as a criterion. IEEE Trans. Softw. Eng. **32**(5), 315–329 (2006)
22. Kemerer, C., Slaughter, S.: An empirical approach to studying software evolution. IEEE Trans. Softw. Eng. **25**(4), 493–509 (1999)
23. Kitchenham, B., Al-Kilidar, 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. Emp. Softw. Eng. **13**(1), 97–112 (2008)
24. Kitchenham, B.: Empirical Paradigm—the role of experiments, pp. 25–32. Empirical Software Engineering, Issues (2006)
25. Kompose: a generic model composition tool. http://www.kermeta.org/kompose (2010)
26. Larman, C.: Applying UML and patterns: an introduction to object-oriented analysis and design and iterative development, 3rd edn. Prentice Hall (2004). ISBN 0131489062
27. Martin, R.: Agile software development, principles, patterns, and practices, 1st edn. Prentice Hall (2002). ISBN 0135974445
28. Mens, T.: A state-of-the-art survey on software merging. IEEE Trans. Softw. Eng. **28**(5), 449–562 (2002)
29. Menzies, T., Chen, Z., Hihn, J., Lum, K.: Selecting best practices for effort estimation. IEEE Trans. Softw. Eng. (TSE) **32**(11), 883–895 (2006)
30. Meyer, B.: Object-oriented software construction, 1st edn. Prentice-Meyer Hall, Englewood Cliffs (1988)
31. Mitchell, M., Jolley, J.: Research design explained, 4th edn. Harcourt, New York (2001)
32. Molesini, A., Garcia, G., Chavez, C., Batista, T.: Stability assessment of aspect-oriented software architectures: a quantitative study. J. Syst. Softw. **38**(5), 711–722 (2009)
33. Nagappan, N., Zeller, A., Zimmermann, T., Herzig, K., Murphy, B.: Change bursts as defect predictors. In: 21st International Symposium on Software Reliability Engineering, pp. 309–318, San Jose (2010)
34. Nejati, S., Sabetzadeh, M., Chechik, M., Easterbrook, S., Zave, P.: Matching and merging of statecharts specifications. In: International Conference on Software Engineering (ICSE'07), pp. 54–64, Minneapolis, EUA (2007)
35. Norris, N., Letkeman, K.: Governing and managing enterprise models: Part 1. Introduction and concepts. IBM Developer Works. http://www.ibm.com/developerworks/rational/library/09/0113_letkeman-norris (2011)
36. Nugroho, A., Flaton, B., Chaudron, M.: Empirical analysis of the relation between level of detail in UML models and defect density. In: International Conference on Model Driven Engineering Languages and Systems (MoDELS'08), pp. 600–614, Toulouse (2008)
37. OMG.: Unified modeling language: infrastructure version 2.2. Object Management Group (2008)
38. Perry, D., Siya, P., Votta, L.: Parallel changes in large scale software development: an observational case study. In: International Conference on, Software Engineering (ICSE'98), pp. 251–260 (1998)
39. Research method knowledge base: improving conclusion validity. http://www.socialresearchmethods.net/kb/concimp.php(2011)
40. Ricca, F., Penta, M., Torchiano, M., Tonella, P., Ceccato, M.: How developers' experience and ability influence web application comprehension tasks supported by UML stereotypes: a series of four experiments. IEEE Trans. Softw. Eng. **96**(1), 96–118 (2010)
41. Shadish, W., Cook, T., Campbell, D.: Experimental and quasi-experimental designs for generalized causal inference. Houghton Mifflin, Boston (2002)
42. Sjøberg, D., Anda, B., Arisholm, E., Dybå, T., Jørgensen, M., Karahasanovic, A., Koren, E., Vokác, M.: Conducting realistic experiments in software engineering. In: 1st International Symposium on, Empirical Software Engineering, pp. 17–26 (2002)
43. Thaker, S., Batory, D., Kitchin, D., Cook, W.: Safe composition of product lines. In: 6th International Conference on Generative Programming and Component Engineering (GPCE'07), pp. 95–104, Salzburg (2007)
44. Whittle, J., Jayaraman, P.: Synthesizing hierarchical state machines from expressive scenario descriptions. ACM Trans. Softw. Eng. Methodol. (TOSEM'10) **19**(3), 1–45 (2010)

45. Wohlin, C., Runeson, P., Höst, M., Ohlsson, M., Regnell, B., Wesslén, A.: Experimentation in software engineering: an introduction. Kluwer Academic Publishers, Norwell (2000)
46. Wust, J.: The software design metrics tool for the UML. http://www.sdmetrics.com

Author Biographies

Kleinner Farias is an associate member of the OPUS Researcher Group at the Pontifical Catholic University of Rio de Janeiro (PUC-Rio), Brazil. He received his PhD in Computer Science from PUC-Rio in 2012. He received his Masters degree in Computer Science from the Pontifical Catholic University of Rio Grande do Sul in 2008. He completed his undergraduate studies in Computer Science at the Federal University of Alagoas and in Information Technology at the Federal Institute of Alagoas in 2006. His current research interests include software modeling, empirical evaluation of model composition techniques, model-driven software development, software metrics and software product lines.

Alessandro Garcia is an Assistant Professor in the Informatics Department at the Pontifical Catholic University of Rio de Janeiro (PUC-Rio), where he leads the Opus research group. He received his PhD in Computer Science from PUC-Rio in 2004. His current research interests include empirical evaluation of advanced modularity techniques, software metrics, software architecture, exception handling, and software product lines. He has been serving as a Program Committee member of premier international conferences on software engineering, such as ICSE, AOSD, FSE, MODELS and SPLC. He received many awards and distinctions, including Best Dissertation Award (Computer Brazilian Society, 2000), Best Researcher Award (Lancaster University, 2006), Distinguished Young Scholar (PUC-Rio, 2009), and Young Scientist Fellowship (FAPERJ, 2010).



Carlos Lucena is a Full Professor of Computer Science at the Pontifical Catholic University of Rio de Janeiro (PUC-Rio) since 1982 and an Adjunct Professor of Computer Science and a Senior Research Associate of the Computer Systems Group at the University of Waterloo, which he has visited on a regular basis since 1975. He completed his undergraduate studies in Economics and Mathematics between 1962 and 1965 at PUC-Rio and received his Masters degree from the University of Waterloo (1969), Canada, and his PhD from the University of California in Los Angeles (1974). His current research focuses on agent-oriented software engineering, multi-agent applications, autonomic computing and software reuse.