6 Effort on the Resolution of Inconsistency

The goal of this Chapter is to evaluate the effects of model stability and design modeling language on the inconsistency resolution effort. For this, two studies are realized. The first study (Section 6.1) is an exploratory study that analyzes and reports the effects of model stability on the effort required to resolve inconsistencies, and its impact on the inconsistency rate. These inconsistencies emerged when three well-known composition algorithms (such as *override*, *merge,* and *union*) were applied in evolution scenarios of three software product lines. The results, supported by statistical tests, show that model stability was an effective indicator of severe inconsistencies and high resolution effort of inconsistency.

The second exploratory study (Section 6.2) reports the impact of modeling language on the inconsistency rate and the resolution effort. More specifically, it investigates whether aspect-orientation reduces the resolution effort as improved modularization may help developers to better restructure the model. Similar to the previous study, it uses model composition to express the evolution of design models along six releases of a software product line. The composition algorithms (i.e., override, merge, and union algorithms) were also applied. The AO and non-AO composed models produced were compared in terms of their inconsistency rate and effort to solve the identified inconsistencies. The findings reveal specific scenarios where aspect-orientation properties, such as obliviousness and quantification, result in a lower (or higher) resolution effort.

6.1. Effect of Model Stability on Inconsistency Resolution

As previously mentioned, 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

(Clarke, 2001) 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., $M_{CM} \neq M_{AB}$). Since, M_A and MB conflict with each other in some way, producing some 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 an error-prone task (France & Rumpe, 2007). 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 pointing 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 (Norris & Letkman, 2011). 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}. 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 (Kelly, 2006; Molesini et al., 2009). 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) have different level of abstraction and are characterized by alternative features. 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 it is expected (or not) i.e., $M_{CM} \approx M_{AB}$. If the assigned value to a characteristic is close to 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., (Clarke & Walker, 2001; Kompose, 2010; Nejati et al., 2007). That is, little 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 (Mens, 2002).

This section, 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: the use of model composition when adding new features to a set of models for three realistic software product lines. Software product lines (SPLs) commonly involve model composition activities (Jayaraman et al., 2007; Thaker et al., 2007; Apel et al., 2009) 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 (Clarke & Walker, 2001), namely override, merge and union, were employed to evolve the SPL design models 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. That is, in each new release, models for the new feature are composed with the models for the existing features. 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. 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 study as a first step in a more ambitious agenda to empirically assess model stability.

The remainder of the chapter is organized as follows. Section 6.1.1 describes the main concepts and knowledge that are going to be used and discussed throughout the Chapter. Section 6.1.2 presents the study methodology. Section 6.1.3 discusses the study results. Section 6.1.4 compares this work with others, presenting the main differences and commonalities. Section 6.1.5 highlights some threats to validity. Finally, Section 6.1.6 presents some concluding remarks and future work.

6.1.1. Background

This Section presents the fundamental concepts to a correct understanding of the contributions presented in this Chapter. To this end, the concepts of model stability, composition heuristics, and model inconsistency will be discussed.

6.1.1.1. Model Stability

According to (Kelly, 2006), a design characteristic of software is stable if, when compared to other, the differences in the metric associated with that characteristic are regarded 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 (Kelly, 2006), 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 percent. This choice is based on previous empirical studies (Kelly, 2006 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 from the comparison of measures of each model characteristic calculated with a suite of metrics described in Chapter 3 and Table 27.

We adopt the definition of stability from (Kelly, 2006) (and its threshold) due to 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 were 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 (Farias et al., 2008a; Medeiros et al., 2010; Guimarães et al., 2010; Kelly, 2006; Farias, 2011) 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 model. The stability quantification method is presented later in Section 6.1.2.4.

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.

Table 27: Metrics used

6.1.1.2. Composition Heuristics

As previously mentioned in Section 2.4, composition heuristics rely on two key activities: *matching* and combining the input model elements (Farias et al., 2010a; Farias et al., 2010b; Clarke, 2001, Reddy et al., 2006). Usually they are used to *modify*, *remove*, and *add* features to an existing design model. This work focuses on three state-of-practice composition heuristics: override, merge, and union (Clarke & Walker, 2001; Clarke & Walker, 2005). 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., (Farias et al., 2010a). In the following, we briefly define these three heuristics, and assume M_A and M_B as the input two 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* (Oliveira et al., 2009a; Oliveira et al., 2009b; Reddy et al., 2005).

The design models used are typical UML class and component diagrams, which have been widely used to represent software architecture in mainstream software development (Ambler, 2005; Fowler, 2003; Dennis et al., 2007; Lüders et al., 2000). In Figure 17, 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 in Section 6.1.2.3. The composition heuristics considered in our study were override, merge, and union. These heuristics were previously discussed in Section 2.4.1. Figure 17 shows two input models and two composed models produced following the override and merge heuristics, respectively. Figure 18 shows the intended model and the composed model produced following the union heuristic.

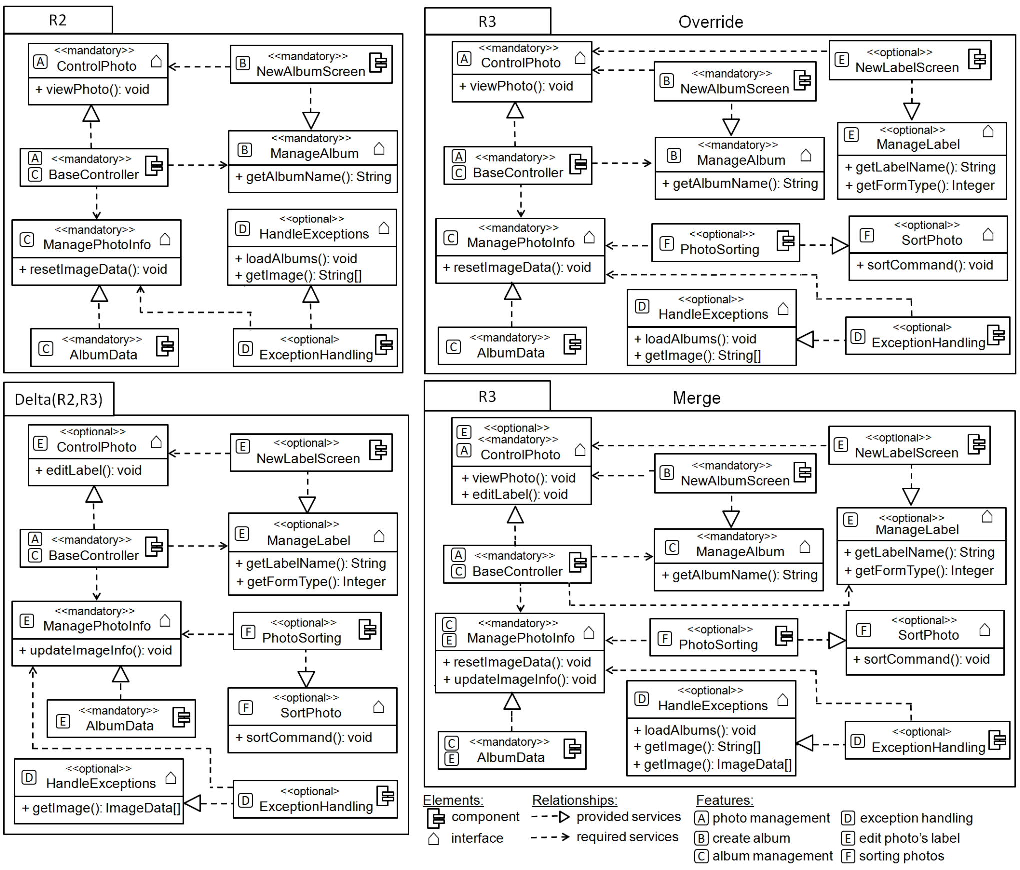

Figure 17: Example of composition of the Mobile Media product line

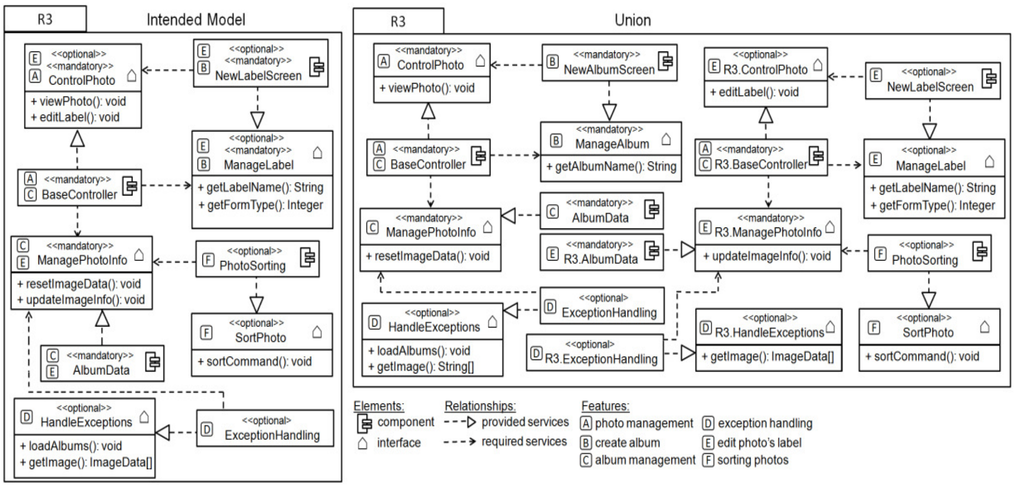

Figure 18: The intended and composed model produced following the union heuristic

6.1.1.3. Model Inconsistency

Inconsistencies emerge in the composed model when its properties assume values other than those expected, as previously defined in Section 3. These values can affect the syntactic and semantic properties of the model elements. Usually the undesired values come from conflicting changes that were incorrectly realized (Samar et al., 2011). 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 (IBM RSA, 2011). For example, this robust tool is able to detect the violation of well-formedness rules defined in the UML metamodel specification (OMG, 2011). In order to improve our inconsistency analysis, we also considered the types of inconsistencies shown in Table 28, which were checked by using the SDMetrics tool (Wust, 2011). In particular, these inconsistencies were used because their effectiveness has been demonstrated in previous works (Farias et al., 2008a; Farias et al., 2010a; Farias et al., 2012d). In addition, both syntactic and semantic inconsistencies were manually identified 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 28) 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 in the resulting composed model. For example, the *AlbumData* does not provide the service "Update Image Information" because the method *updateImageInfo():void* is not present in the *ManagePhotoInfoInterface*. Hence, the *PhotoSorting* component is unable to provide the service "*SortingPhotos*." This means that the feature "*SortingPhoto*" (feature 'F' in Figure 17) – a critical feature of the software product line – is not correctly realized. On the other hand, this problem is not present in Figure 17 (merge), in which the *AlbumData* implement two features (C, model management, and E, edit photo's label). We defer further discussion about the examples and the quantification of these types of inconsistencies to Section 6.1.2.4.

Metric	Description
NFCon	The number of functionality inconsistencies.
NCCon	The number of model elements that are not compliance 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.

Table 28: The inconsistencies used in our case study

6.1.2. 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 (Section 6.1.2.1). Next, the study hypotheses are systematically stated from these research questions (Section 6.1.2.2). The product lines used in our studies are also discussed in detail as well as their evolutionary changes (Section 6.1.2.3). Then, the variables and quantification methods considered are precisely described (Section 6.1.2.4). Finally, the method used to produce the releases of the target architectures is carefully discussed (Section 6.1.2.5). All these methodological steps were based on practical guidelines of empirical studies (Wohlin et al., 2000; Basili, 2007; Kitchenham et al., 2008; Kitchenham, 2006; Shadish et al., 2006).

6.1.2.1. 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 (Basili, 1994) 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 on inconsistency rate and the inconsistency resolution effort. Therefore, we address research question RQ4, as stated in Section 1.3:

- **RQ4:** What is the impact of design characteristics on the inconsistency rate and inconsistency resolution effort?

Considering the quality notions defined in Chapter 3, we study whether the syntactic and semantic quality notions of a model affects the effort and resolution quality notions. We refine the research question into two research questions. Thus, we focus on the following two research questions:

- **RQ4.1:** What is the effect of stability on the *inconsistency rate*?
- **RQ4.2:** What is the effect of stability on the *developers' effort*?

6.1.2.2. Hypothesis Formulation

First Hypotheses: Effect of Stability on Inconsistency Rate (RQ5.1). 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, $H_{1\text{-}0}$:

Stable design models have similar or higher inconsistency rate than unstable design models.

$H_{1\text{-}0}$: Rate(stable design models) $\geq$ Rate(unstable design models).

Alternative Hypothesis 1, $H_{1\text{-}1}$:

Stable design models have a lower inconsistency rate than unstable design models.

$H_{1\text{-}1}$: Rate(stable design models) < Rate(unstable design models)

By testing the 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}.

Second Hypothesis: Effect of Stability on Developer Effort (RQ5.2). 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. 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, H_{2-0}:

Stable models require similar or higher effort to resolve inconsistencies than unstable models.

H_{2-0}: Effort(stable models) $\geq$ Effort(unstable models).

Alternative Hypothesis 2, H_{2-1}:

Stable models tend to require a lower inconsistency resolution effort than unstable ones.

H_{2-1}: Effort(stable models) $<$ Effort(unstable models).

By testing the 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 mangers to allocate qualified developers to overcome the composition inconsistencies in M_{CM}.

6.1.2.3. Target Cases: Evolving Product-Line Design Models

Model Composition for Expressing SPL Evolution. We have applied the composition heuristics to evolve design models of three realistic SPLs for a set of evolution scenarios (Table 29). 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. The first target case is a product-line called MobileMedia, 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. Figure 17 and Figure 18 show a practical example of the use of composition to evolve this SPL.

The second SPL, called Shogi Game, is a board game, whose purpose is to allow users to move, customize pieces, save, and load game. All the movements of the pieces are governed by a set of well-defined rules. The last SPL, called Checkers Game, is a board game played on an eight by eight-squared board with twelve pieces on each side. The purpose of Checkers is to essentially move and capture diagonally forwards.

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 were produced 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 (Table 28). 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 properly. These 36 releases were produced from the evolution scenarios described in Table 29. Secondly, these releases were available for our investigation and had a considerable quantity of structural changes in the evolution scenarios.

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 in a reliable fashion. Moreover, 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. The SPL designs were 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. . In total, eight developers worked during the development of the SPLs used in our

	Release	Description
Mobile Media	R1	MobilePhoto core (Figueiredo et al, 2008)
	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 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 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

Table 29: Descriptions of the evolution scenarios

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), and one from Federal University of Pernambuco (Brazil).

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 were 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 29.

6.1.2.4. 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 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 inconsistency rate is defined from multiple inconsistency metrics (Oliveira, 2008a).

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 developer to evolve software in real-world settings (Mens, 2002). 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 were 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 the 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}.

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

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}

Table 27 defines the metrics used to quantify the design characteristics of the models, while Formula 1 shows how the Distance is computed. The Stability can assume two possible values: 1, indicating that M_{CM} and M_{AB} are *stable*, and 0, 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 $\text{Stability}(M_{CM},M_{AB}) = 0$. 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 $\text{Distance}(M_{CM},M_{AB}) > 0.2$), then $\text{Stability}(M_{CM},M_{AB}) = 0$. 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. Formula 2 shows how the measure Stability is computed.

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

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.

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

As the Distance(M_{CM},M_{AB}) is equal to 0.2, then we can consider that M_{CM} is equal to 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.

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

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

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 NInter(M_{CM}) = 15, NInter(M_{AB}) = 17, DIT(M_{CM}) = 11, and DIT(M_{AB}) = 13, the Distance is calculated as follows.

$$\text{Distance}(M_{\text{CM}}, M_{AB}) = \frac{|NInter(M_{CM}) - NInter(M_{AB})|}{NInter(M_{AB})} = \frac{|15 - 17|}{17} = 0.11$$

$$\text{Distance}(M_{\text{CM}}, M_{AB}) = \frac{|\text{DIT}(M_{CM}) - \text{DIT}(M_{AB})|}{\text{DIT}(M_{AB})} = \frac{|11 - 13|}{13} = 0.15$$

In both cases, M_{CM} is stable as 0.11 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}.

$$Stability(x,y)_{overall} = 1 - \frac{\sum_{k=0}^{j-1}(Stability_k)}{j} \qquad (3)$$

Legend:

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

$$Stability(x,y)_{overall} = 1 - \frac{\sum_{k=0}^{4}(Stability(x,y))}{5}$$

$$\sum_{k=0}^{4}\big(Stability(x,y)\big) = \frac{|NClass(M_{CM}) - NClass(M_{AB})|}{NClass(M_{AB})}$$

$$+\frac{|DepOut(M_{CM}) - DepOut(M_{AB})|}{DepOut(M_{AB})} + \frac{|NAttr(M_{CM}) - NAttr(M_{AB})|}{NAttr(M_{AB})}$$

$$+\frac{|NInter(M_{CM}) - NInter(M_{AB})|}{NInter(M_{AB})} + \frac{|DIT(M_{CM}) - DIT(M_{AB})|}{DIT(M_{AB})}$$

$= 0.2 + 0.14 + 0.22 + 0.11 + 0.11$ (applying the Formula 2)

$= 1 + 1 + 0 + 1 + 1 = 4$

Then,

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

6.1.2.5. Evaluation Procedures

a. *Target Model Versions and Releases*

To test the study hypotheses, we have used the releases described in Table 29. 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) 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 to help us to identify scenarios where the SPL design models succumb (or not). For example, to produce the release 3 (R3) of the Mobile Media, 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 previously. A practical example about how these models are produced can be seen in Figure 17 and Figure 18.

Model Releases and Composition Specification. The releases in Table 29 were in particular 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. During the design of all releases, a main concern was to maximize good modeling practices in addition to the design-for-change principles. For example, assume that the mean of the coupling measure of M_{CM} and M_{AB} is equal to 9 and 11, respectively. So M_{CM} is stable regarding M_{AB} (because 9 is 18% lower than 11). Following this stability threshold, we can systematically identify if the M_{CM} keeps stable over the evolution scenarios.

b. Execution and Analysis Phases

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 (IBM RSA, 2011; Norris & Letkeman, 2011). 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 (Oliveira et al., 2008; Farias et al., 2010a; Farias et al., 2010b; Medeiros et al., 2010; Guimaraes et al., 2010; Farias, 2011a, Farias et al., 2011b) 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 (Wust, 2011). In addition, we also check both syntactic and semantic inconsistencies manually because some metrics e.g., "the number of non-meaningful model elements" depend 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 Section 6.1.2.4. 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 in Appendix A.

6.1.3. Results

This section reports and analyzes the data set obtained from the experimental procedures described in the previous section. The findings of this work are derived from both the numerical processing of this data set and the graphical representation of interesting aspects of the gathered results. Then, Section 7.1.3.1 elaborates on the gathered data in order to test the first hypothesis (H1). Lastly, Section 7.1.3.2 discusses the collected data related to the second hypothesis (H2).

6.1.3.1. H1: Stability and Inconsistency Rate

c. 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. The understanding of these statistics is a key step 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 composition scenarios 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. From this bunch of evolution scenarios, we are confident that the collected data are representative to be analyzed using descriptive statistics.

Table 30 shows descriptive statistics about the collected data regarding inconsistency rate. Figure 19 depicts the box-plot 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 observed only harmful effects in the absence of stability. The main outstanding finding is that inconsistency rate in stable design model is lower than in unstable design model. This result is supported by some observations described as follows

Variables	Groups	N	Min	25th	Median	75th	Max	St. Dev.
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, St. Dev.: Standard Deviation

Table 30: Descriptive statistics of the inconsistency rate

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 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. If, for example, one SPL developer has to work with an unstable model, then he or she will certainly have to handle 91.9 percent more inconsistencies, compared the medians 0.31 (stable) and 3.86 (unstable). In fact, stable models tend to have just 8.1 percent of the inconsistencies that are found in unstable models, compared 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, with a standard deviation equals 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. According to (Wohlin et al., 2000; Basili, 2007), 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 latter, they should 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.

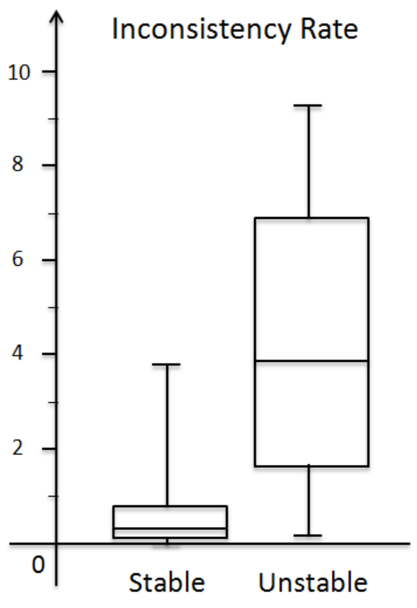

Figure 19: Box-plot of inconsistencies

d. *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.

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 ($H_{1\text{-}0}$) can be rejected. That is, there is sufficient evidence to say that the difference between the inconsistency rates of stable and unstable models are statically significant. Table 31 depicts that the mean rank of inconsistency rate for unstable models are higher than that of stable models. As Mann-Whitney test (Wohlin, 2000) 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.

Variable	**Groups**	**N**	**Mean Rank**	**Rank Sum**	**SC**	**t-value***	***p***
Resolution effort	Stable	78	46,99	3665	- 0,698	- 13	< 0.001
	Unstable	102	123,77	12625			

*with 178 degree of freedom, SC: Spearman's Correlation

Table 31: Mann-whitney test and Spearman's correlation analysis

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 31). 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; i.e., is neither dependent on, causes nor influences the other. 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 31, 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 stability of product-line architectures decreases the inconsistency rate in their models increases.

Therefore, the results suggest that, on average, stable models have *significantly* lower inconsistency rate than unstable design models. Therefore, we are confident that the results confirm a strong 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.

e. Discussion

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., (Kelly, 2006; Kemerer & Slaughter, 1999; Eman et al., 2002; Perry, 1998; Berzins, 1994, Yang et al., 1992) 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 occurred mainly when composition heuristics accommodate unanticipated, severe changes from M_A to M_B. The most complicate 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 from a set of ever-present evolving change categories, such as *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 previously defined. In Figure 17 and Figure 18, 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 a little 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 in our target cases was the refinement of the MVC architecture style of the MobileMedia SPL in the third evolution scenario. In practical terms, the central architectural component, *BaseController*, was 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 design rigidness to accommodate four new specific controllers (by refining the previous general one) contributed significantly to the instability of the output composed model. 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) encompassed the required interface *ControlPhoto* (release 3) of the *AlbumListScreen* component. This interface was decomposed into two new required interfaces *ControlAlbum* and *ControlPhotoList* (release 4), thereby characterizing a relationship 1:2. For this particular case, the name-based model comparison should be able to “recognize” that *ControlAlbum* and *ControlPhotoList* are equivalent to *ControlPhoto*. However, in the output model (release 4), the *AlbumListScreen* component provides duplicate services to the environment giving rise to a severe inconsistency.

Inconsistency Propagation. After addressing the hypotheses and knowing that instabilities have a detrimental effect on the density of inconsistencies, we analyze whether the local where they arise (i.e., architectural elements realizing mandatory, alternative 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 the model elements implementing mandatory features than in alternative 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 alternative and optional features, the higher the inconsistency propagation to mandatory features. However, the propagation in the inverse order (i.e., from alternative and optional to mandatory features) seems to be less common. In Figure 17 (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 emerging 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.

6.1.3.2.
H2: Stability and Resolution Effort

a. 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 32 provides the descriptive statistics of sampled inconsistency resolution effort in stable and unstable model groups. Figure 20 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 6) is much lower than that one of unstable models (equals to 111).

Variables	Groups	N	Min	25th	Median	75th	Max	St. Dev.
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, St. Dev.: Standard Deviation

Table 32: Descriptive statistics of the resolution effort

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 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.

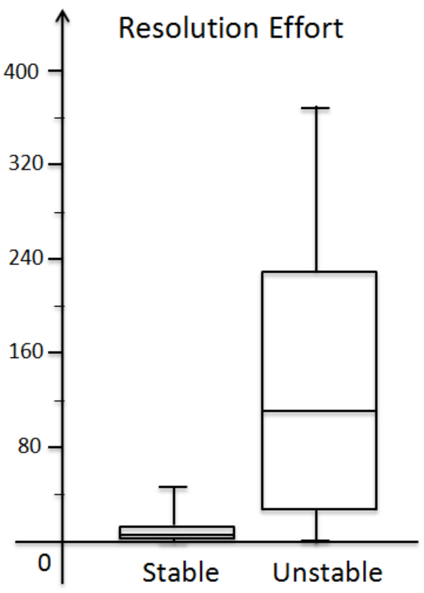

Figure 20: Box-plot of resolution effort in relation to the intended model

b. Hypothesis Testing

Given the difference between the mean and median described in the descriptive statistical 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.

Mann-Whitney test. As the dataset does not respect the assumption of normality, we use the non-parametric Mann-Whitney test was used as the main statistical test as well as it was done in the first hypothesis. However, the Mann-Whitney test was only applied to the effort measures needed to transform the composed model into the intended model. 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 33 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.

Variable	Groups	N	Mean Rank	Rank Sum	SC	t-value*	*p*
Resolution effort	Stable	78	46,99	3665	- 0,698	- 13	< 0.001
	Unstable	102	123,77	12625			

*with 178 degree of freedom

Table 33: Mann-whitney test and Spearman's correlation analysis

Correlation Analysis. As the gathered data do not follow a normal distribution, we cannot apply the Pearson's correlation analysis. An alternative way was to apply the Spearman's correlation (SC) test to measure the strength of the linear relationship (the correlation coefficient) between stability and inconsistency resolution effort. Table 33 provides the results of the Spearman's correlation test. The low p-value < 0.001 indicates that the correlation significantly departs from zero. Remember 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.

Hence, 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.

c. Discussion

The Effect of Instability on Resolution Effort. We have observed that the higher instability in optional and alternative features, the higher the resolution effort. This increased effort is due to instabilities in optional features cause inconsistencies in model elements implementing mandatory features. In practice, inconsistencies in architectural elements realizing optional features tend to affect the structure of model elements realizing mandatory features. The reason is that some relationships are (or not) introduced between architectural elements realizing mandatory and optional features during the composition. These undesired dependences favor the inconsistency propagation. Consequently, developers must invest some additional effort to resolve the inconsistencies. The effort is to restructure the composed model. That is, instability in optional features tends to jeopardize some properties of the architectural elements realizing the mandatory features, which requires some unexpected effort. That is, it is required to resolve a cascading chain of inconsistencies, and usually this process should be applied recursively until all inconsistencies have been resolved. This is typically the case scenario when inconsistencies of operations with earlier operation, the heuristic can therefore remove the earlier operation and add the new one, or vice-versa.

We have identified that this higher effort to resolve inconsistencies is due to the syntax-based composition heuristics being unable to deal with occurring semantic conflicts between the model elements of mandatory and optional features. As a result, inconsistencies are formed. In Figure 17, 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 systematic semantic alignment between these two domain terms. Consequently, the heuristics would be able to properly match *R2.NewAlbumScreen* and *R3.NewLabelScreen.*

Still in Figure 17, 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 (Mens, 2002), the current detection of behavioral inconsistency is just based on 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

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 models 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 has 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 Figure 17). 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 cyclic *inconsistency propagation.* Understanding this type of phenomenon, designer can examine upfront and more precisely the design models in order to localize undetected cyclic dependence between the model elements.

Another striking 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.

d. Some Additional Considerations

Quantification Method. We are aware that there are pros and cons in studying either an overall indicator or a single metric of design stability. In (Kelly, 2006), she defines a single metric of design stability and then uses this method as an indicator of good practices of design. This study is performed in retrospective i.e., analyzing software artifacts that evolved over a long term. On the other hand, this thesis has a different goal that is to evaluate whether the "most severe instabilities" may be related to model composition effort. We conjecture that the most severe instability can be identified considering a greater number of design characteristics. This will be also analyzed during the empirical studies.

If we consider only one single design characteristic, we will have at least two problems: (i) first, we will potentially ignore severe instabilities that affected other design characteristics, and (ii) second, we will end up artificially concluding those variations of a single characteristic (e.g., high number of methods or high number of attributes) always represents severe design instabilities. Then, we opted for following a strategy, commonly adopted nowadays e.g., (Marinescu, 2004; Lanza & Marinescu, 2006), to detect significant design problems through a combination of multiple measures rather than a single metric.

Effectiveness of the Threshold. As previously mentioned in Chapter 2, we have also analyzed whether the threshold defined in (Kelly, 2006) is also valid in the context of this study. To this end, we analyze whether the threshold (0.2) jeopardizes the results (or not). More specifically, we study whether small differences around a threshold of 0.2 can produce different results. After a careful analysis of the collected data, we conclude that our conjecture stated in Section 2.6.1 is confirmed. That is, the threshold of 0.2 was effective for the purpose of this study. The main reason is that the threshold did not harm the identification of severe cases of inconsistency rate and resolution effort. This can be confirmed by analyzing, for instance, the data in Table 30: the inconsistency rates of the stable group and instable group are significantly different considering the median (0.31 against 3.86); the same pattern of significant difference applies to the other cases (25th and 75th columns). Again, the same pattern is observed in Table 32 for resolution effort. This means that the threshold considered (0.2) can clearly separate the composed models into groups of stable and unstable models; since, their measures concentrate in the opposite extremes. This confirms that we are able to consistently implement our strategy of studying the impact of models with the most severe instabilities (i.e., ones where more than 20% of the design characteristics varied considerably) rather than analyzing the different degrees of instabilities.

6.1.4. Limitations of Related Work

To the best of our knowledge, our results are the first to empirically investigate the relation between quality notions and model composition effort in a broader context. In (Farias et al., 2011b), we initially investigated the research questions addressed in this Chapter, but they were evaluated in a smaller scope. This work, 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 composition 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 Nejati et al., 2007; Clarke, 2001; Reddy, et al., 2005; Lange & Chaudron, 2006a; OMG, 2011; Kompose, 2011; Norris & Letkeman, 2011; Whittle & Jayaraman, 2010; France et al., 2007; Fleury et al., 2007) have been created, but also some previous works (Farias et al., 2011b; Nagappan et al., 2010) have demonstrated that stability is a good predictor of defects (Nagappan et al., 2010) and the presence of good designs (Kelly, 2006). 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 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 (France & Rumpe, 2007; Dingel et al., 2008; Farias et al., 2008; Molesini et al., 2009; Mens, 2002; Berzins, 1994; France et al., 2006; Dzidek et al., 2008).

The current model composition literature does not provide any support to perform empirical studies in model composition effort (France & Rumpe, 2007; Farias et al. 2010a), or even to evaluate the effects of model stability on composition effort. In (France & Rumpe, 2007), the authors highlight the need empirical studies in model composition to provide insights about how deal with ever-present problems such as conflicts and inconsistencies in real world settings. In (Mens, 2002), Mens also reveals the need of 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) (Dingel et al., 2008; OMG, 2011).

There are some specific metrics available in the literature for supporting the evaluation of model composition specifications. For instance, Chitchyan and colleagues (Chitchyan et al., 2009) 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. Boucke and colleagues (Boucke et al., 2006) 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 (Lange & Chaudron, 2006a; Lange & Chaudron, 2006b; Nugroho et al., 2008)) cannot be directly applied to our context.

Although we have proposed a metric suite for quantifying inconsistencies in UML class diagrams (Farias et al., 2008a) and then applied these metrics to evaluate the composition of aspect-oriented models and UML class diagrams (Farias et al., 2010a), nothing has been done to understand the effects of model stability on the developers' effort. We therefore see this study as a first step in a more ambitious agenda to support empirically the assessment of model composition techniques in general.

Finally, some previous works investigate the effect of using UML diagrams and its profiles with different purposes. In (Briand et al., 2005), Briand looked into the formality of UML models and its relation with model quality and comprehensibility. In particular, Briand and colleagues investigated the impact of using OCL (Object Constraint Language (OMG, 2011)) on defect detection, comprehension, and impact analysis of changes in UML models. In (Ricca et al., 2010), Ricca 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.

6.1.5. 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.

6.1.5.1. 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 (Brewer, 2000; Shadish et al., 2002). 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 study. 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 (Shadish et al., 2002). That is, rather than just the independent variable, an unknown third variable unexpectedly affects the dependent variable. To avoid c*onfounding* 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 are automatically applied and are algorithms explicitly and independently defined by others. 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.

6.1.5.2. Statistical Conclusion Validity

We evaluated the statistical conclusion validity checking if the independent and dependent variables (Section 6.1.2.4) were submitted to suitable statistical methods. These methods are useful to analyze whether (or not) the research variables covary (Cook et al., 1979; Shadish et al., 2006). The evaluation is concerned on two related statistical inferences: (1) whether the presumed cause and effect covary, and (2) how strongly they covary (Cook et al., 1979; Shadish et al., 2006). 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 (Shadish et al., 2006).

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, nonparametric statistics were used (Section 6.1.2.1 and Section 6.1.2.2.). Hence, we are confident that the test statistics were applied correctly; as the assumptions of the test statistics 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 (Wohlin et al., 2000). 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.

6.1.5.3. Constructs 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 (Farias et al. 2010a; Kelly, 2006; Medeiros et al., 2010; Guimaraes et al.; 2010). Moreover, the concept of stability used in our study is well known in the literature (Kelly, 2006) 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 (Section 6.1.2.1) and hypotheses (Section 6.1.2.2). 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 (Wohlin et al., 2000; Basili et al., 1999; Kitchenham et al., 2008; Kitchenham et al., 2006).

Execution of the Compositions. Another threat that we have controlled is if by using manual composition threats validity since we might unintentionally avoids 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 don't think the manual composition would be a noticeable problem to the study for many reasons, including: (i) 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, and (ii) 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.

6.1.5.4. External Validity

External validity refers to the validity of the obtained results in other broader contexts (Mitchell & Jolley, 2001). 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 it in a large variety of places, with different people, and at different times, we made use of the theory of proximal similarity (proposed by Donald T. Campbell (Campbell & Russo, 1998)) 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 (Section 7.1.1.2) to evolve UML design models such as UML class and component diagrams. Second, developers should also be able to apply the inconsistency metrics described previously and use some robust software modeling tool e.g., IBM RSA (Norris & Letkeman, 2011; IBM RSA, 2011).

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 (Dingle et al., 2008; Chitchyan et al., 2009; Farias et al., 2010a; Whittle & Jayaraman, 2010; Briand et al., 2005; Clarke & Walker, 2001; Norris & Letkeman, 2011). 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 (Farias et al., 2010b). 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.

6.1.6. Concluding Remarks

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. Our study, 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 chapter.

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 statistical 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 varieties of our stability metrics. We also wish to improve understanding if design models with superior stability have some gain (or not): (i) when produced from other 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 evaluation 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.

6.2. Impact of Design Language on Inconsistency Resolution Effort

This section aims at evaluating the impact of design modeling languages such as AO and non-AO modeling on the inconsistency resolution effort. The hypothesis investigated is that aspect-orientation may alleviate the effort of inconsistency resolution to some extent. Aspect-orientation provides an improved modularity and that more effective modularization may help developers to deal with the inconsistencies, thus minimizing the resolution effort. However, it is by no means obvious that this hypothesis holds. It may be, for instance, that inconsistencies in aspect-oriented models have a detrimental effect on the resolution effort because inconsistencies aspectual elements may require the developers to examine all points in the model crosscut by the aspects.

With this in mind, the goal of this section is to report on an exploratory empirical study that aimed at providing evidence to support or refute this hypothesis. To this end, we again make use of model composition to add new features to a set of models in a software product line, called Mobile Media.

We investigate this hypothesis in the context of SPLs evolution because they commonly involve model composition activities (Jayaraman et al., 2007; Thaker et al., 2007) 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. We show the results for model compositions of six releases of an SPL. In each release, models for the new feature are composed with the models for existing features. For each release, we analyze both the quantity and nature of the composition inconsistencies. Furthermore, we compare two versions of the SPL models — one which uses aspect-oriented modeling and one which does not.

The results show that higher inconsistency rates were observed in the presence of aspects when they had a higher degree of quantification. On the other hand, this problem did not entail more effort on inconsistency resolution. We also found that higher degree of obliviousness tended to yield compositions of AO composed models that are closer to the intended compositions. To the best of our knowledge, our results are the first to empirically investigate the potential advantages of aspects during modeling phase. Despite a wide variety of technical approaches to AOM e.g., MATA (Whittle & Jayaraman, 2010) and Kompose (Kompose, 2011), to-date there has been almost no empirical evaluation of AOM. We therefore see this study as a first step in a more ambitious agenda to empirically assess aspect-oriented modeling.

The remainder of the study is organized as follows. Section 6.2.1 introduces the main concepts and knowledge that are going to be used and discussed throughout this section. Section 6.2.2 we present the methodology. Section 6.2.3 discusses the composition analysis effort. Section 6.2.4 contrasts this work with others, highlighting the commonalities and differences. Section 6.2.5 analyzes the threats to validity. Finally, Section 6.2.6 presents some concluding remarks and future work.

6.2.1. Aspect-Oriented Modeling for Architectural Models

Model composition applies both to development with and without aspect-oriented modeling (Clarke & Walker, 2005). This study compares the inconsistency resolution effort in both cases. AOM languages aim at improving separation of concerns by supporting the modular representation of concerns that cut across multiple software modules. Crosscutting concerns are represented by a new model element, called *aspect*. The goal of AOM is, therefore, to provide software developers with the means to express aspects and crosscutting relationships in their models. There are AOM languages for modeling aspects at many levels of abstraction, ranging from use cases and architectural design to detailed designs. As far as the solution space is concerned, aspects are usually first expressed in architectural models.

Figure 21 is an illustrative example of the architectural AOM language (Garcia et al., 2009) used in this study (Section 6.2.3). We chose this AOM language because: (i) we selected architectural models as our focus due to the availability of existing industrial models; (ii) the AOM language has been widely used in other contexts (such as modularization of crosscutting concerns (Sant'Anna, 2008)) and is therefore mature (Garcia et al., 2009).

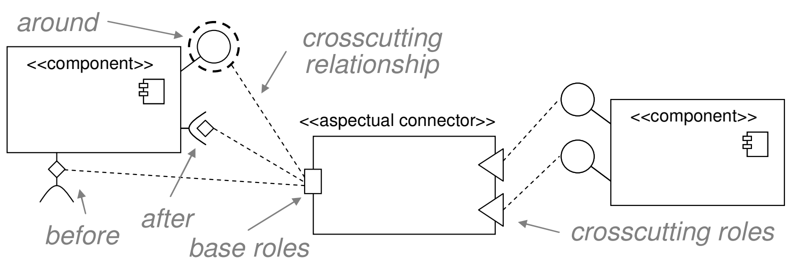

Figure 21: AOM language for architectural models

The notation supports the visual symmetric representation of aspect-oriented software architectures. The target modeling approach consists of an extension of the UML's component diagram (OMG, 2011). In order to put the composition in practice, we should consider the properties of model elements defined in the UML metamodel specification in this diagram. Thus, the properties of the model elements considered were component (name, provided interface, and required interface), interface (name, operation, and attribute), operation (name, return type, and parameters), attribute (name and type), relationship (source and target), crosscutting relationship, and join-points. Therefore, the composition algorithms are fine-grained due to take into account these properties in each composition.

The notation provides explicit elements for expressing different forms of component-aspect collaborations, which are represented by aspectual connectors. Aspectual connectors are illustrated by rectangles in Figure 21. They define which components, interfaces or specific operations are affected by a component modularizing a crosscutting concern. Aspectual connectors are associated with crosscutting relationships represented by dashed arrows. The notation also supports the visual modeling of specific pointcut designators (e.g., advising all the provided interfaces) and sequencing operators (after, before, and around). For the sake of simplicity in this study, only aspectual connectors and crosscutting relationships will be represented in the models of our case study; all the other visual details have been omitted from here on.

6.2.2. Study Methodology

This section describes the study definition, the target application, the evaluation method used for computing model composition effort, and the other study procedures in our exploratory study.

6.2.2.1. Objective and Research Questions

This study attempts to evaluate the impacts of aspect-oriented modeling on two variables: the *inconsistency rate* and *inconsistency propagation*. These effects are evaluated from evolution scenarios considering compositions of architectural models. Additionally, some scenarios are described in which the influence of AO models on effort is precisely described. With this in mind, the objective of this study is stated based on the GQM template (Basili et al., 1994) as follows:

Analyze design modeling techniques
for the purpose of investigating their effects
with respect to inconsistency rate and inconsistency propagation
from the perspective of developers
in the context of evolution of architectural models

Specially, this study aims at discovering the inconsistency rate, resolution effort, and revealing scenarios where these inconsistencies propagate, affecting multiple model elements. Therefore, we address research question RQ3, as stated in Chapter 1:

- **RQ3:** What is the effect of design decomposition techniques in particular with respect to misinterpretation, inconsistency rate, inconsistency detection effort, and inconsistency resolution effort?

Regarding the quality notions defined in Chapter 3, we study whether the syntactic and semantic quality of a design model affects the effort and resolution quality notions. We refine RQ4 into two more research questions. Thus, we focus on the following research questions:

- **RQ3.4:** Does the composition of AO models produce a higher inconsistency rate than non-AO models?
- **RQ3.5:** What is the impact of AO modeling on the way inconsistencies propagate in the output model?

These research questions were investigated considering the inconsistencies described in Section 5.1.2 and Section 6.1.1.3.

6.2.2.2. Hypotheses Formulation

Aspect-oriented modeling has been a topic of research for at least ten years (Clarke & Walker, 2005; Clarke & Banaissad, 2005). However, there is currently very limited knowledge as to how aspects, when incorporated in input models, affect the model composition effort. In particular, there is no understanding if the composition of aspect-oriented models affects the emergence of inconsistencies in the output composed models.

First Hypothesis: Impact of Aspect on Inconsistency Rate. Our first null hypothesis assumes that the inconsistency rate in output AO composed models is equal or higher than in output non-AO composed models. As aspect orientation tends to improve the modularization of design models, the alternative hypothesis states that the inconsistency rate in AO models is lower than in non-AO models. This would lead to the following null and alternative hypotheses:

Null Hypothesis 1, H_{1-0}: The inconsistency rate (Rate) in AO models is equal or higher than in non-AO models.

H_{1-0}: Rate(AO) $\geq$ Rate(non-AO).

Alternative Hypothesis 1, H_{1-1}: The inconsistency rate (Rate) in AO models is lower than in non-AO models.

H_{1-1}: Rate(AO) $<$ Rate(non-AO).

Given that inconsistency tends to propagate in a composed model (Farias et al., 2010a). That is, 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 composed component is missing an important operation. 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. Studying such propagation effects is important because propagation directly affects the effort in resolving inconsistencies e.g., a propagation chain of length *n* may be actually fixed by resolving a single inconsistency rather than the expected *n* inconsistencies. Thus, we are interested in understanding the possible inconsistency propagation patterns in AO and non-AO models (RQ4.5). Similar to the previous hypothesis, it is assumed that inconsistency equally spread through output (non-)AO models. This leads to the second null and alternative hypotheses as follows:

Null Hypothesis 2, $H_{2\text{-}0}$: The inconsistency propagation in AO models is equal or higher than in non-AO models.

$H_{2\text{-}0}$: Prop(AO) $\geq$ Prop(non-AO).

Alternative Hypothesis 2, $H_{2\text{-}1}$: The inconsistency propagation in AO models is lower than in non-AO models.

$H_{2\text{-}1}$: Prop(AO) $<$ Prop(non-AO).

To test the hypotheses, metrics were used to quantify inconsistency rate, the propagation, and the effort to resolve the inconsistencies when they spread through model elements. Aforementioned, these metrics are presented in Chapter 3. The metrics were applied to both non-AO and AO models of an evolving software product line described in the next section.

6.2.2.3. Case Study: Evolving an SPL

Model composition can be applied in different contexts and with different purposes. We have selected a particular scenario to test our study hypotheses: the use of model composition to express the evolution of software product line (SPL) architecture.

Model Composition for Expressing SPL Evolution. Model compositions were defined to generate the new releases of the SPL architecture model. That is, the composition algorithms (override, merge, and union) were used to define how each architecture model (M_A) of an SPL release and the new model increments (M_B) were going to be combined to generate the new architecture SPL release (M_{AB}). The first input model (M_A) represents the current architecture of an SPL release, while the second input model (M_B) represents the delta capturing the modifications to the base model (M_A). The output model (M_{AB}) generated by the application of the composition algorithm represents the next SPL release.

MobileMedia: the Target SPL. A product line, called Mobile Media (Figueiredo et al., 2008), of 6 kLOC was selected to be the target case of the evaluation. The purpose of the MobileMedia SPL is to manipulate photos, music, and videos on mobile devices. In (Figueiredo et al., 2008), it is possible to find a fine-grained description about its characteristics and how its evolution happened. The reasons for selecting this system in the evaluation are described as follows. First, the developers of the MobileMedia SPL are the responsible for creating its architecture design models. Second, two versions of the same product line and the respective architectural models were available for our investigation: an AO version and a non-AO version. This is a fundamental requirement to test the hypotheses (Section 6.2.2.2). Third, the last release of the architectural design has more than one hundred modules, and its architectural models are the main artifact to reason about change requests and derive new products. Fourth, the architectural models were produced by the original developers, which do not have any of the model composition algorithms under assessment in mind, thereby avoiding any bias and entailing a more natural software development scenario. Fifth, the architectural models (M_A) and the increment models (M_B) were conceived with the modularity and changeability as key drivers. Sixth, we had available seven fully documented evolution scenarios, which could be expressed with model compositions (examples are given later).

Finally, Mobile Media met a number of other equally-important requirements, such as: (1) proper documentation of the driving requirements; (2) the system evolved for more than three years, and the more recent releases have more than 100 modules; (3) different types of change were realized in each release, including refinements of the architecture style employed, (4) the system has been successfully used in other studies involving empirical evaluation of OO and AO implementations (Figueiredo et al., 2008), and (5) the original developers were available to help us with the production and analysis of the composed models and the intended models. As such, all these factors provided a solid foundation for our study.

6.2.2.4.
Quantifying Inconsistency Rate and Resolution Effort

The goal is to quantify: (i) the number of inconsistencies, and (ii) the activities required to transform the output composed model into an output intended model. The analysis of the results relies on an inconsistency measure, called inconsistency rate (Rate), to quantify the amount of composition inconsistencies divided by the total number of elements in the output model. That is, inconsistency rate allows computing the density of composition inconsistencies in the output composed models. Using this metric, we may quantify the inconsistency rate in AO and non-AO models, and analyze the differences between them (H1). Note that the inconsistency rate is defined from multiple inconsistencies, which can be found in Section 6.1.1.3.

The resolution effort consists of the number of operations that should be performed to transform an output composed model into an output intended model. We compute the number of *creations*, *removals*, and *modifications* needed to realize this transformation. That is, this computation represents an estimation of the resolution effort ($g(M_{CM})$). After we collect the $g(M_{CM})$ measure, we performed an inspection of the output model to check if there was any occurrence of inconsistency propagation. This enabled us to check if the presence of aspects in the input models had any impact on the way composition inconsistencies were propagated (H2). In order to come up with a suitable characterization of the measures of the compositions and the MobileMedia SPL releases, we defined a basic formalism for the metric space of composition effort as follows.

A metric space is a set M equipped with a real-valued function CE(w,s) defined for all w, s ϵ M. Let M = $\{R_{i,x,y}$, i = 1,…,n; x = override, merge; y = left, right$\}$, where:

- *n* is a finite natural number representing the model release;
- *left* and *right* represent the direction of the composition relationship in the override algorithm.

For example, $R_{3,merge,right}$ represents the Release 3 that was produced by merging: Release 2 $+_{merge}$ Delta(Release 2, Release 3) → Release 3. Delta(Release 2, Release 3) represents the model elements that should be merged with Release 2 to transform it into Release 3, as previously discussed. In practical terms, the Delta represents the evolution to be inserted into the previous release. On the other hand, $R_{3,merge,left}$ would be Delta(Release 2,Release3) $+_{merge}$ Release 2 → Release 3 (the inverse order can also be represented with an asterisk). Therefore, the reader should note that the order of override-based composition might produce different output composite models (Dingel et al., 2008). Each model of a $R_{i,x,y}$ can be characterized by observing its syntactic and semantic properties. If we have a high inconsistency rate in an evolution scenario, then this implies a higher effort to resolve inconsistencies.

6.2.2.5. Evaluation Procedures

Once the case study was selected (Section 6.2.2.3) and the inconsistency resolution metrics were defined (Section 6.2.2.4), we needed to undergo a number of specific evaluation procedures. They are discussed in the following.

a. Target Model Versions and Releases

We have used both non-AO and AO versions of the Mobile Media models in order to test the study hypotheses (Section 6.2.2.2). These two model versions of the same system enabled us to identify if the presence of aspects in the input models had positive or negative effects on the quality of the output model.

Deriving AO and non-AO Model Releases. For each release of Mobile Media, we have applied each of the composition algorithms described in Section 2.3. That is, we have used the merge algorithm to compose two input AO models in order to produce a new AO release model; similarly, we applied the merge strategy to compose two input non-AO models in order to produce the next non-AO release model. We performed similar compositions with override and union algorithms. The goal was to identify if the outcomes, in terms of inconsistency rate and propagation (hypotheses), were the same or different. All the releases of the non-AO and AO versions realized exactly the same SPL features and variability points. They also underwent the same evolution scenarios, ranging from changes in heterogeneous mobile platforms and additions of many alternative and optional features (Figueiredo et al., 2008). Non-AO models were represented by conventional UML component models, while AO models were represented using the AOM language described in Section 6.2.1.

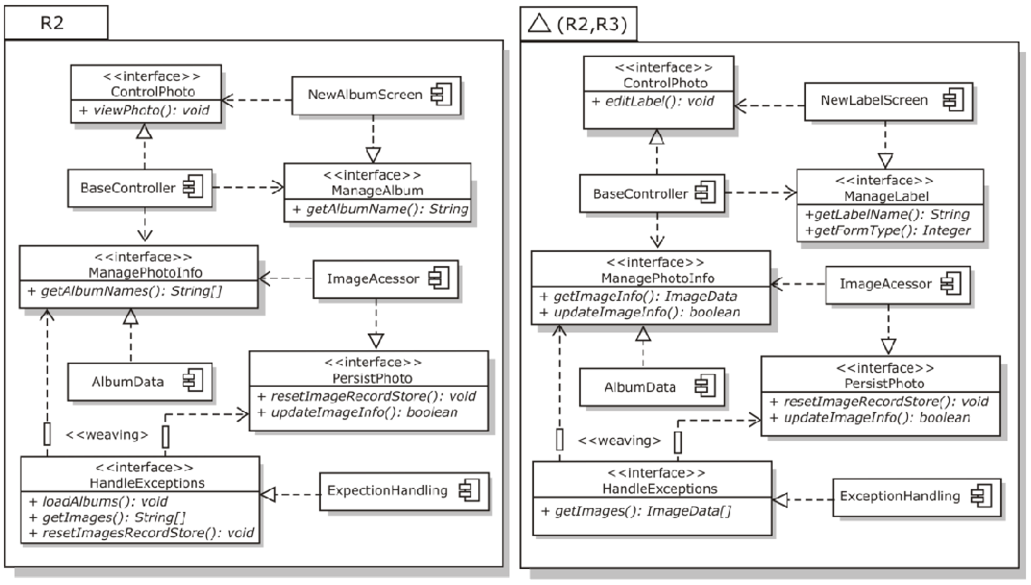
R2
<<interface>> ControlPhoto
+ viewPhoto(): void
NewAlbumScreen
BaseController
<<interface>> ManageAlbum
+ getAlbumName(): String
<<interface>> ManagePhotoInfo
+ getAlbumNames(): String[]
ImageAcessor
AlbumData
<<interface>> PersistPhoto
+ resetImageRecordStore(): void
+ updateImageInfo(): boolean
<<weaving>
<<interface>> HandleExceptions
+ loadAlbums(): void
+ getImages(): String[]
+ resetImagesRecordStore(): void
ExpectionHandling
(R2,R3)
<<interface>> ControlPhoto
+ editLabel(): void
NewLabelScreen
BaseController
<<interface>> ManageLabel
+getLabelName(): String
+getFormType(): Integer
<<interface>> ManagePhotoInfo
+ getImageInfo(): ImageData
+ updateImageInfo(): boolean
ImageAcessor
AlbumData
<<interface>> PersistPhoto
+ resetImageRecordStore(): void
+ updateImageInfo(): boolean
<<weaving>
<<interface>> HandleExceptions
+ getImages(): ImageData[]
ExceptionHandling

Figure 22: The input models: the AO base and AO delta model

In fact, AOM is used in this work to represent the aspect-oriented model releases of the SPL under study. For example, in Figure 22, in addition to have interfaces (e.g., *PersistPhoto*), components (e.g., *ImageAcessor* and *AlbumData*), we also have aspectual components, such as the *ExceptionHandling* aspect.

Moreover, we can also have some relationships: realization (e.g., between the components *BaseController* and *ControlPhoto*), dependency (e.g., between the component *NewAlbumScreen* and the interface *ControlPhoto*), and crosscutting (e.g., between the aspectual component *ExceptionHandling* and the component *PersistPhoto*, in which the service *loadAlbums*(): void is woven into the component). The notation used in this work to express the architectural models has been used in other works (Figueiredo et al., 2008; Garcia et al., 2009) and has shown to be effective for its purpose.

Model Releases and Composition Specification. We considered six releases of MobileMedia (Figueiredo et al., 2008) in this study. They were selected because they were the ones where the changes implied visible modifications in the architectural design. For each new release, the previous release was modified in order to accommodate the features to be modified, inserted, or removed. To implement a new evolution scenario, a model composition specification can remove, add, derive, or modify the entities present in the previous release. During the design of all releases, a main concern was to follow best practices of modeling.

b. Execution and Assessment Phases

The execution and assessment of the study were structured in three main steps, which are described in the following.

Model Refactoring Phase. The model refactoring is a pivotal activity to define the input models and, hence, to express the model evolution as an explicit model composition relationship. To this end, MobileMedia's architectural models were initially refactored to specify the delta itself and to represent the change scenarios as composition relationships. To create the delta model it is necessary to identify the differences between the releases models and then gather them into the input model. To go about this, we took into account an evolution description created by the original modelers involved in a previous study (Figueiredo et al., 2008). These descriptions specify in-depth the modifications needed to realize each evolution scenario (from one release to another). They allowed us to identify how the model elements were changed. For example, in the second evolution description, the Delta(R2,R3) were based on the description such as: the interface *ControlPhoto* — realized by *BaseController* — had the method *edilLabel*(): void added (see Figure 22). Another example would be the change concerning the name of the interface *ManageLabel* to *ManageAlbum*. Thus, all model elements of the Delta(R2,R3) are derived from one evolution description, which ensures that the input model specification is free of bias.

Composition and Measurement Phase. From one release to another, 6 compositions were produced: 3 compositions following override, merge, and union from the current release to delta, and 3 compositions in the inverse direction. We considered 5 evolution scenarios for the non-AO version as well as the AO version of the Mobile Media, totaling 60 compositions. The result of this phase was a document of composition descriptions, including the gathered data from the application of our metrics suite. Figure 22 presents partial input models

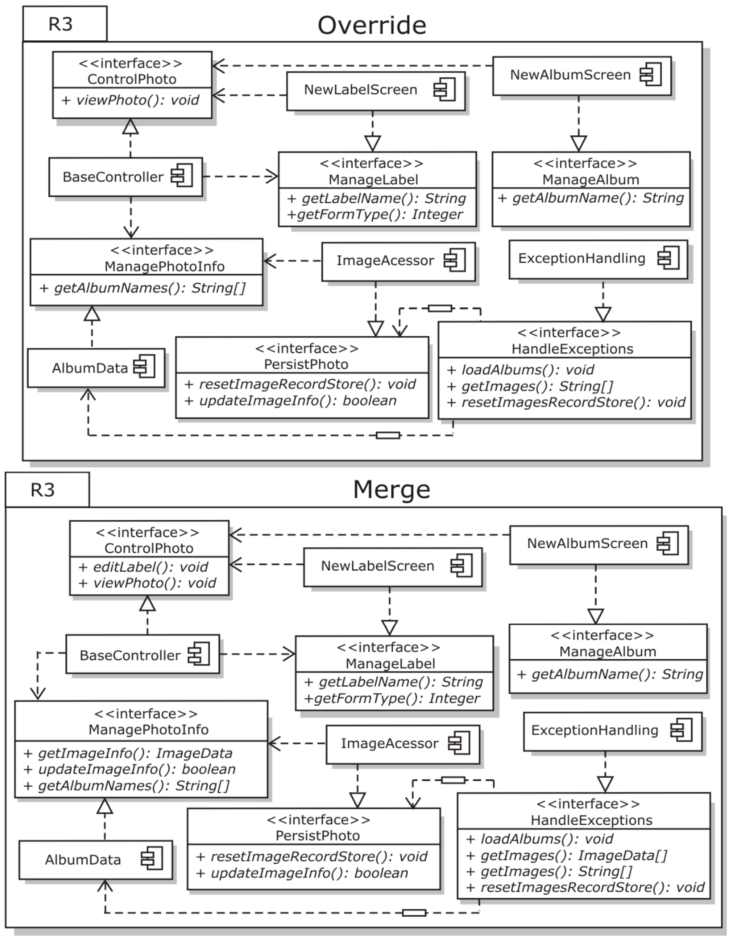
R3
Override
<<interface>> ControlPhoto
+ viewPhoto(): void
NewLabelScreen
NewAlbumScreen
BaseController
<<interface>> ManageLabel
+ getLabelName(): String
+getFormType(): Integer
<<interface>> ManageAlbum
+ getAlbumName(): String
<<interface>> ManagePhotoInfo
+ getAlbumNames(): String[]
ImageAcessor
ExceptionHandling
AlbumData
<<interface>> PersistPhoto
+ resetImageRecordStore(): void
+ updateImageInfo(): boolean
<<interface>> HandleExceptions
+ loadAlbums(): void
+ getImages(): String[]
+ resetImagesRecordStore(): void
R3
Merge
<<interface>> ControlPhoto
+ editLabel(): void
+ viewPhoto(): void
NewLabelScreen
NewAlbumScreen
BaseController
<<interface>> ManageLabel
+ getLabelName(): String
+getFormType(): Integer
<<interface>> ManageAlbum
+ getAlbumName(): String
<<interface>> ManagePhotoInfo
+ getImageInfo(): ImageData
+ updateImageInfo(): boolean
+ getAlbumNames(): String[]
ImageAcessor
ExceptionHandling
AlbumData
<<interface>> PersistPhoto
+ resetImageRecordStore(): void
+ updateImageInfo(): boolean
<<interface>> HandleExceptions
+ loadAlbums(): void
+ getImages(): ImageData[]
+ getImages(): String[]
+ resetImagesRecordStore(): void

Figure 23: Output AO models produced by override and merge algorithms

being used in one of the releases, while Figure 23 and Figure 24 represent examples of composition based on merge, override, and union, respectively.

Figure 24 is the intended result of the composition (or intended model). As well-validated metrics for model composition are not available yet, we used a set of inconsistency metrics defined in our previous work (Farias et al., 2008a). The inconsistencies (and their effects) were identified manually using such inconsistency metrics. The identification of the inconsistencies was performed in 5 review cycles in order to avoid false positives/negatives. We also consulted the Mobile Media developers when needed, such as checking and confirming specific cases of semantic inconsistencies.

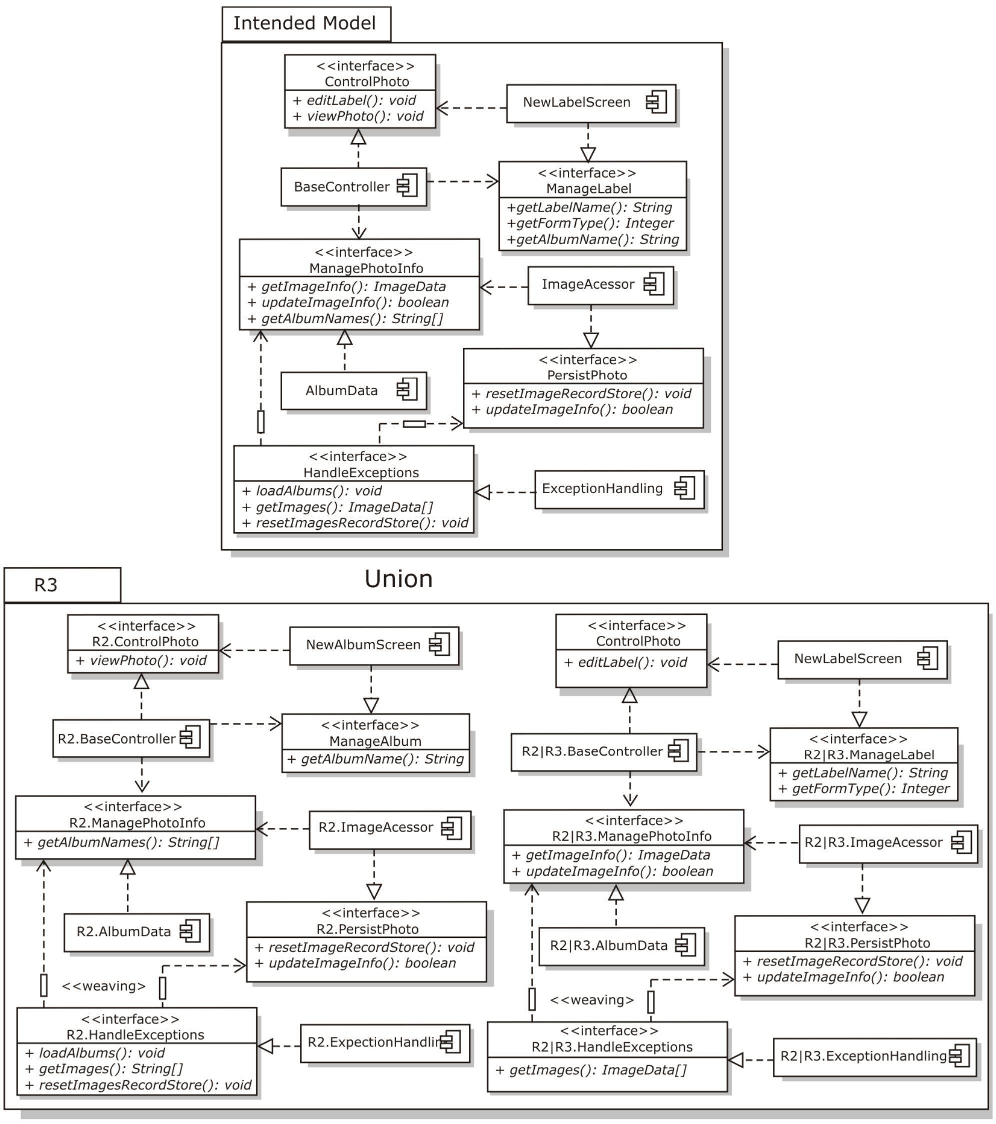
Intended Model
<<interface>> ControlPhoto
+ editLabel(): void
+ viewPhoto(): void
NewLabelScreen
BaseController
<<interface>> ManageLabel
+getLabelName(): String
+getFormType(): Integer
+getAlbumName(): String
<<interface>> ManagePhotoInfo
+ getImageInfo(): ImageData
+ updateImageInfo(): boolean
+ getAlbumNames(): String[]
ImageAcessor
<<interface>> PersistPhoto
+ resetImageRecordStore(): void
+ updateImageInfo(): boolean
AlbumData
<<interface>> HandleExceptions
+ loadAlbums(): void
+ getImages(): ImageData[]
+ resetImagesRecordStore(): void
ExceptionHandling
R3
Union
<<interface>> R2.ControlPhoto
+ viewPhoto(): void
NewAlbumScreen
R2.BaseController
<<interface>> ManageAlbum
+ getAlbumName(): String
<<interface>> R2.ManagePhotoInfo
+ getAlbumNames(): String[]
R2.ImageAcessor
R2.AlbumData
<<interface>> R2.PersistPhoto
+ resetImageRecordStore(): void
+ updateImageInfo(): boolean
<<weaving>>
<<interface>> R2.HandleExceptions
+ loadAlbums(): void
+ getImages(): String[]
+ resetImagesRecordStore(): void
R2.ExpectionHandling
<<interface>> ControlPhoto
+ editLabel(): void
NewLabelScreen
R2|R3.BaseController
<<interface>> R2|R3.ManageLabel
+ getLabelName(): String
+ getFormType(): Integer
<<interface>> R2|R3.ManagePhotoInfo
+ getImageInfo(): ImageData
+ updateImageInfo(): boolean
R2|R3.ImageAcessor
R2|R3.AlbumData
<<interface>> R2|R3.PersistPhoto
+ resetImageRecordStore(): void
+ updateImageInfo(): boolean
<<weaving>>
<<interface>> R2|R3.HandleExceptions
+ getImages(): ImageData[]
R2|R3.ExceptionHandling

Figure 22: AO intended model (from Figure 22) and AO output model produced following the union heuristic

Effort Assessment Phase. The goal of the third phase was to assess the effort to resolve the inconsistencies using the metrics described previously. The composition algorithms were used to generate the evolved models, so that we could assess the impact of aspects 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 (Section 6.2.3.1) is concerned with pinpointing the inconsistency rates produced by composition of either non-AO or AO (H1). The second stage (Section 6.2.3.2) aims at assessing the effort to resolve a set of previously identified inconsistencies and whether (or not) the use of aspect has a higher impact on the way composition inconsistencies are propagated (H2). We analyzed how inconsistency rate differs across the releases in order to detect potential benefits and drawbacks of using AOM in the input models. We have decided to focus the discussions on the merge and override algorithms, because the union algorithm did not present any additional interesting insight. However, all measurement results were considered during the study analysis.

6.2.3. Composition Effort Analysis

This section presents the results collected during the investigation of the RQ3.4 and RQ3.5 to both the AO and non-AO output models realizing each SPL release. Histograms are used to provide an overview of the data gathered in the measurement process. These histograms allow us to analyze the impact of aspects on study variables: inconsistency rate, inconsistency propagation, and inconsistency resolution effort. Each histogram focuses on the application of a particular composition algorithm. The Y-axis presents the values gathered for a particular metric. The X-axis specifies the evolution scenarios.

Note that each pair of bars is attached to a pair of values, with the first capturing the performance of the AO version and the second capturing the non-AO one. The lower the value, the better is the performance of the modeling approach used. It is important to highlight that the results shown in the histograms were gathered with respect to the entire model. Based on the inconsistencies identified by the inconsistency rate metric, Section 6.2.3.1 discusses the findings related to the first hypothesis (H1). Section 6.2.3.2 relies on the metric for quantifying model recovery effort in order to support the analysis of the second hypothesis (H2).

6.2.3.1. H1: Aspects and Inconsistency Rate

Figure 25 illustrates the results for the inconsistency rate obtained following the override algorithm. Figure 26 shows the results of the same metric for the merge algorithm. The first observation allows us to conclude that the inconsistency rate measures have favored aspect-orientation in both merge and override cases and for most of the evolution scenarios. This implies that the tally of inconsistencies to some extent is decreased whenever aspects are present in the models to-be-composed. The presence of aspects in the input models produced lower inconsistency rate than aspect-free models when the override algorithm is applied in both directions (right and left (represented by the *-columns)). For example, the inconsistency rate decreases from 1.72 (non-AO version) to 1.33 (AO version) in Scenario 2, which represents a reduction of 22.6% in favor of aspect-orientation. Similarly, the inconsistency rate decreases from 0.59 to 0.41 when the composition is performed in the left direction, which represents a reduction of 30%.

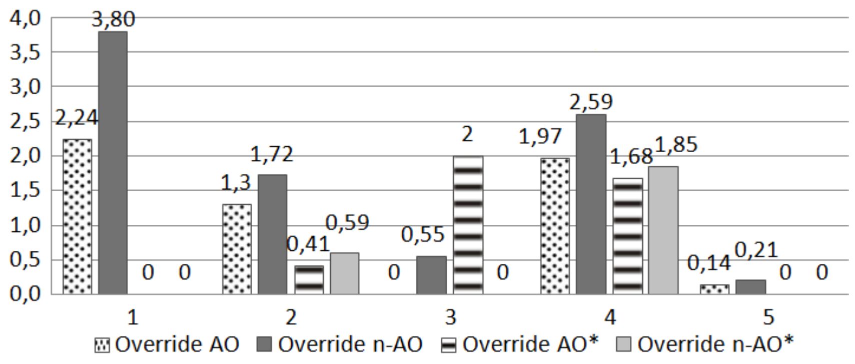

Figure 23: Inconsistency rate produced by the override algorithm

Moreover, it is well known that the higher the number of model elements that take part in compositions, the higher the likelihood of inconsistencies being generated. Nevertheless, the AO versions still had lower absolute measures of inconsistencies. For example, the absolute measure decrease from 38 (non-AO version) to 36 (AO version) in Scenario 2, which represents a reduction of 5.2% in favor of aspect-orientation. Similarly, the inconsistency rate decreases from 13 to 11 in the inverse order, which represents a reduction of 15.3%. The only case where aspect-free models led to a close inconsistency was the application of the merge algorithm in the second release; this special case is discussed in the following section.

The main reason for the superiority of the AO models is that changes, reified by the delta model, tend to be confined in fewer modules due to the superior modularization of crosscutting features in AO models. The confinement of modifications to aspects, in turn, leads to a better localization of both syntactic and semantic inconsistencies, thereby making them easier to detect and address in the output models. Therefore, we refute the null hypothesis H_{1-0} and confirm the alternative hypothesis H_{1-1}.

We have noticed that the decrease of inconsistencies observed in the AO models is potentially influenced by two factors: (i) *quantification,* the higher the quantification of aspects in input models, the higher the inconsistency rate measures, and (ii) *obliviousness*, the higher the degree of obliviousness, the lower the inconsistency rate measures in the output models. Another predominant factor in the emergence of high inconsistency rates was the nature of the change. Independently of the degree of obliviousness and quantification in AO models, the nature of the change directly affected the inconsistency rate observed in the output models. In the following, we elaborate these issues further and discuss examples that support each of these findings.

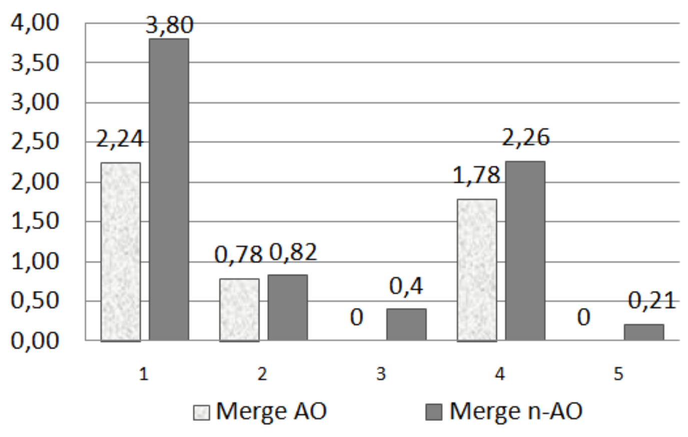

Figure 24: Inconsistency rate produced by the merge algorithm

a. *Obliviousness and Quantification*

We have observed that quantification (Filman & Friedman, 2000) influenced the inconsistency rate measures. The presence of aspects with lower quantification (in the input models) led to fewer syntactic and semantic inconsistencies in the output models. When aspects were being used, for example, to encapsulate domain–specific features, a lower inconsistency rate manifested in the output models. On the other hand, we also observed that when a conflict arises in aspects with higher quantification (in the input models), higher rates of syntactic and semantic inconsistencies occurred in the output models. Therefore, the quantification mechanism may (or may not) improve inconsistency rate results.

This category of aspects is the case where the aspects work as glue between a few elements in the base model and the changes realized by the delta model. Aspects with a higher degree of quantification, such as exception handling (Figure 22, Figure 23, and Figure 24), affect the input base model in many places (join points). This was exactly the case in Scenario 2, where the non-AO version (Rate = 0.82) has a measure close to the AO version (Rate = 0.78) (Figure 22). Higher quantification increases the aspect scope and, therefore, the likelihood of aspects interfering with each other. When the merge algorithm was applied, the exception handling aspect (Figure 23) led to undesired superimpositions with other aspectual behaviors advising the same join points.

The overall inconsistency rate (Rate measure) has been usually lower in the AO version because most of the aspects were not affecting more than three elements. By overall rate, we mean the average of inconsistencies considering all the model elements. However, a careful analysis of the number of inconsistencies in individual model elements (e.g., a particular component) reveals some interesting information. The composition output of AO models consistently caused an increase on the number of inconsistencies for some specific model elements. For example, this can be observed in Scenario 4, when the highest number of inconsistencies emerged in both non-AO and AO versions. Despite the significant Rate difference favoring the AO version, the component *BaseController* presented an increase (Rate = 38) in relation to *BaseController* of the non-AO version (Rate = 24). We noted that this problem occurred in situations where the components were affected by two aspects or more in the delta model. In other words, when a base component had a high density of join points shared by multiple aspects; it generated a higher number of inconsistencies.

An additional interesting finding was that the composition of AO models tended to manifest fewer inconsistencies when the obliviousness degree of the base elements was higher. We have noted that the creation of new aspects (via the delta model) for encapsulating new features implies that the modules in the input base model are more oblivious to the modification being implemented in the release. This observation holds for both mandatory and varying(optional or alternative) features. Consequently, the combination of the AO modules tended to ripple fewer inconsistencies in the output models.

This finding implies that the presence of obliviousness is a good indicator that the model composition at hand will better adhere to the Open-Closed principle (Meyer, 1988). This principle states “software should be open for extensions, but closed for modification.” AO modeling conformed more closely to this principle in scenarios where the behavior in the new aspect (part of the Delta model) is more independent of the affected elements in the base model. Release 3 illustrates this finding. For instance, the *AlbumData* component demanded modifications in the non-AO version of Release 3 in order to include the feature of sorting photos by highest viewing frequency. On the other hand, the AO counterpart required no modification in this component. The reason was that new components and the *PhotoSorting* aspect in the delta model modularly implemented the feature.

The open-closed principle was more closely adhered by the composition of AO models than non-AO models. However, this observation did not occur in all the cases. In general, this principle was fully achieved only when the delta model was adding new elements to the base models. The other types of changes realized by the delta model exerted more specific implications in the rate of inconsistencies detected in the output models. This issue is discussed in the following section.

b. The Effect of the Change Category

A careful analysis of the results has pointed out that the inconsistency rate is strictly affected by the category of changes to be applied to the base model. We identified four types of changes throughout our target SPL study:

- *Addition:* new model elements are inserted into base model; for instance, the new method *getFormType()* is inserted into the provided interface, named *ManageLabel*, of the component *NewLabelScreen* (Figure 23).
- *Removal*: a model element in the base model is removed; for example, the required interface *ControlPhoto* of the component *AlbumListScreen* is removed in the fourth Mobile Media release;

- *Modification*: a model element has some properties modified; for instance, the component *NewAlbumScreen* (Release 1) has its name modified to *NewLabelScreen* in Release 2.
- *Derivation*: model elements are refined and/or move to accommodate the changes; for example, the provided interface *ControlPhoto* (with 14 methods) of the component *BaseController* (Release 3) has some methods moved to the provided interface *ControlPhoto* of the component *PhotoController* (Release 4).

Additions. As previously discussed in the previous section, the use of aspects has contributed to produce an output model with much lower inconsistency rate when the evolution scenarios were dominated by *additions.* This finding is supported by the low inconsistency rate in Scenarios 3 and 5. The main reason is that the created aspects (in the delta model) modularize the changes and insert them into the target model elements, without requiring their modifications. In these cases, we also observed that lower Rate measures were observed in the AO models when the override algorithm is used and performed in the left direction. For all the other compositions, the inconsistency rate of the AO releases was equal or lower than the non-AO releases.

A concrete example of the superiority of the AO version was the decrease of the inconsistency rate from 3.8 to 2.24 in Scenario 1. This was due to the aspectual component, included in this release (via the delta model), which advises 9 methods: (i) three of them in the interface *ManagePhotoInfo* of the component *AlbumData*; and (ii) 6 of them in the interface *PersistPhoto* of the *ImageAcessor*. This led to a Rate decrease in the interface *PersistPhoto* from 11 (non-AO version) to 4 (AO version). In the same way, the *ManagePhotoInfo* had its inconsistency rate decreased from 9 to 6.

Modifications, Removals and Derivations. We could not find a recurring Rate pattern (in favor of AO or non-AO versions) when modification was being realized. The AO version performed better in certain cases, while the non-AO version was better in others. On the other hand, the inconsistency rate was slightly higher in non-AO models when removals and derivations were applied. We also observed that a very high inconsistency rate occurred simultaneously in both AO and non-AO models when the change scenario was complex. This was the case when the change scenario involved a blend of modifications, removals, and derivations. More specifically, this occurred in Scenario 4, when there is a significant architectural change: a single controller was restructured as a set of specialized controllers, for example.

Therefore, the heuristic composition algorithms were inefficient in widely scoped architecture evolution, such as the refinement of the MVC (Model-View-Controller) architecture style of Mobile Media. This is also due in part to the name-based model comparison, which is not able to recognize more intricate equivalence relationships between the model elements. This comparison strategy is very restrictive whenever there is a 1:N correspondence relationship between elements in the two input models. An example of the 1:N relationship category encompassed the required interface *ControlPhoto* (Release 3) of the *AlbumListScreen* component. This interface was decomposed into two new required interfaces *ControlAlbum* and *ControlPhotoList* (Release 4), thereby characterizing a 1:2 relationship. 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 4), the *AlbumListScreen* component provides duplicated services to the environment giving rise to an inconsistency. However, even in these cases the aspect orientation presented a lower inconsistency rate (e.g., see Scenario 4 in Figure 27 and Figure 28).

It is known that a higher number of model elements may lead to a higher inconsistency rate when the composition is put in practice. However, this was not the case with aspect-orientation. For instance, let us consider the fourth scenario. Although fewer composed elements (25) were observed in the non-AO version, the latter presents a higher Rate measure (2.59). On the other hand, the AO version has a higher number of compositions (27), but the inconsistency rate is lower (Rate = 1.97). A real example would be the *PhotoViewScreen* component, which decreased the number of inconsistencies from 3 (non-AO version) to 1 (AO version).

6.2.3.2. H2: Aspects and Inconsistency Propagation

We focus our discussion about inconsistency propagation on the analysis of model recovery effort, the resolution effort ($g(M_{CM})$) measure (Section 6.2.2.4). This $g(M_{CM})$ measure is a useful indicator to support the analysis of the presence (or absence) of inconsistency propagation (H_2) in both AO and non-AO models. The higher the effort of recovering the output model (towards the intended composed model), the higher the chance of inconsistency propagation being observed in the output model. Figure 27 depicts the recovery effort measures to transform the output model produced by the override algorithm in the intended model. Similarly, Figure 28 shows the results of the same metric for the merge algorithm. The structure of the histograms follows those in the previous section.

We have concluded that aspects indeed affect the manner of the inconsistencies spread over the output models. We identified a number of recurring inconsistencies in the AO models, which did not occur in the non-AO models. In general, some inconsistencies specific to aspect orientation were caused by a conflict (or several) arising at a single aspect and spreading through all the affected elements in the base model. Therefore, we have found that there is a sensible difference on the way composition inconsistencies are propagated in non-AO and AO models. Therefore, we refute the null hypothesis $H_{2\text{-}0}$ and confirm the alternative hypothesis $H_{2\text{-}1}$.

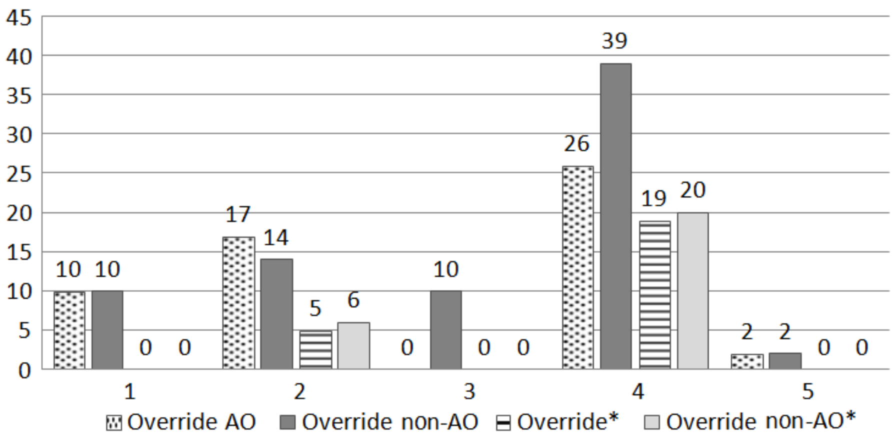

Figure 25: Inconsistency resolution effort to recover the output model produced by override algorithm

a. *Quantification and Model Recovery Effort*

According to previous discussion, aspects with higher quantification contribute to higher inconsistency rates in AO models. An inspection of the output models, however, pointed out that this problem occurred because these aspects led to higher inconsistency propagation manifesting during the model composition process. Surprisingly, increase the inconsistency rates in AO models does not imply in more effort to transform the output composed into the intended composed model. In other words, the finding is that a high degree of quantification does not lead to more effort to recover the output model. The $g(M_{CM})$ measure often tends to be similar in AO and non-AO models.

This phenomenon can be illustrated, for example, in Scenario 2 (Figure 28), where the AO version presents an inconsistency rate closer to (Rate = 0.78) than the non-AO version (Rate = 0.82). However, the model resolution effort is equal to 9 for both AO and non-AO versions (Figure 28). This was the case of inconsistencies arising in a reusable exception handling aspect (modified by the delta model). When inconsistencies arose in such an aspect, they spread over all the model elements directly advised by the aspect. During the model recovery process, there was a need to fix only the inconsistency in the specification of the exception handling aspect.

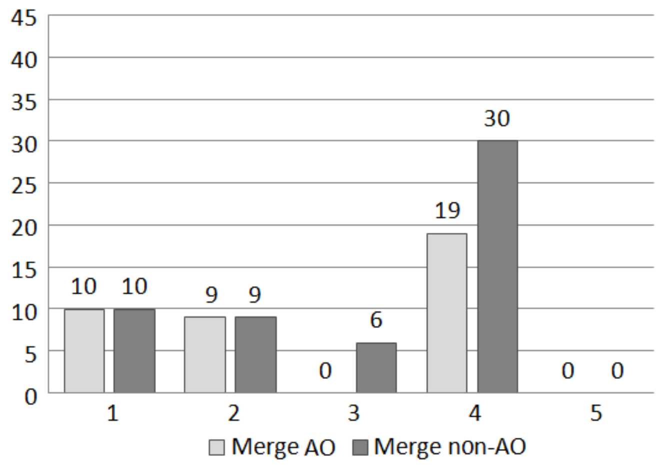

Figure 26: Effort to recover the output model produced by merge algorithm

Therefore, although AO and non-AO versions present different inconsistency rates in certain evolution scenarios (e.g., Scenario 1 in Figure 28), the effort to recover the output model from the inconsistencies in both versions is similar. The effort directly depends on how instances of inconsistencies are interrelated. Propagation channels of inconsistencies were more common in AO models as discussed above. For example, despite aspect orientation exhibiting an inconsistency rate close to the non-AO inconsistency rate in Scenario 2 (Figure 27 and Figure 28), the inconsistency resolution effort is similar to non-AO models. Thus, when the inconsistency that is responsible for propagation is identified and resolved, all inconsistencies are indirectly resolved as well.

b. Shared Join Points and Cyclic Propagation

We have noticed that when an inconsistency emerged in a highly coupled base module (e.g., a controller in Mobile Media), it led to a higher degree of inconsistency propagation in the AO versions than the non-AO versions. This problem was particularly observed when the highly coupled base module was the source of join point shadows shared by multiple aspects. For instance, we have analyzed the inconsistency channels triggered by an inconsistency arising in the *BaseController*, a central model element in the Mobile Media architecture. We observed that the inconsistency propagation affected four components in the non-AO version, namely *AlbumListScreen, PhotoListScreen, PhotoView Screen*, and *AddPhotoToAlbumScreen*. However, the propagation affected three additional modules (aspects) in the AO version.

The *HandleExceptions* interface had a method signature modified from *String[] getImages(String record-Name)* to *ImageData[] getImages(String record-Name)*. However, the *R1.HandleExceptions* incorrectly overrides *Delta(R1,R2).HandleExceptions*. As a result, this method was incorrectly present into the output model, which gives rise to some functionality inconsistencies. This propagation was spread through the component *AlbumData*, because the aspect is no longer able to introduce the expected method *ImageData[] getImages(String record Name)* into the provided interface *ManagePhotoInfo* of *AlbumData*. Consequently, *AlbumData* does not provide any expected service to the environment. Hence, inconsistencies are propagated through the component *BaseController* and *ImageAcessor*.

It is interesting to note that *ImageAcessor* is also affected by an inconsistency that emerged from *AlbumData*. As *ImageAcessor* requires the service (*ImageData[] getImages(…)*) provided by the interface *ManagePhotoInterface*, it is not able to correctly provide the all services defined in the provided interface *PersistPhoto*. Hence, the *AlbumData* is also re-affected by an inconsistency that previously arose from it. This phenomenon represents the cyclic conflict propagation. On the other hand, this propagation is solved in the composition $R_{2,\text{overide,left}}$ due to the Delta*(R1,R2).HandleExceptions* override the `R1`*.HandleExceptions*, decreasing the inconsistency rate from 1.3 in R2,overide,right to 0.41 in R2,overide,left.

6.2.4. Limitations of Related Work

Model composition is a very active research field in many domains, including database integration (Bernstein & Melnik, 2007), composition of web services (Milanovic & Malek, 2004), merging of statecharts (Nejati et al., 2007) , model composition in product lines (Jayaraman et al., 2007), composition of UML models (Dingel et al., 2008; Clarke & Walker, 2005; Farias et al., 2010), aspect-oriented modeling (Whittle et al., 2009; Klein et al., 2006), and AO composition of models (Reddy et al., 2006; Cottenier et al., 2007). However, there is little related work focusing on the quantitative and qualitative assessment of AOM. In general, most of the research on the interplay of AOM and model composition rest on subjective assessment criteria. Even worse, they lead to dependence on experts who have built up an arsenal of mentally held indicators to evaluate the growing complexity of models in general (France & Rumpe, 2007; Lange et al., 2006a, Lange et al., 2006b). Consequently, the truth is that modelers ultimately rely on feedback from experts to determine "how well" the input models and their compositions can be. According to (Figueiredo et al., 2008), the state of the practice in assessing model quality provides evidence that modeling is still in the craftsmanship era and when we assess model composition this problem is accentuated.

More specifically, to the best of our knowledge, researchers have neglected the assessment of how aspects affect model composition effort. The need for assessing models during a model composition process has neither been pointed out nor proposed by current model composition techniques (Cottenier et al., 2008;

Nejati et al., 2007; Reddy et al., 2006; Apel et al., 2011; IBM RSA, 2011). For example, the UML built-in composition mechanism, namely package merge (OMG, 2011; Dingel et al., 2008), does not define metrics or criteria to assess the merged UML models. Moreover, it has been found to be incomplete, ambiguous, and inconsistent (OMG, 2011).

The lack of quantitative and qualitative indicators for model compositions hinder the understanding of side effects peculiar to certain model composition strategies (in the presence of aspects or not). Many different types of metrics have been developed during the past few decades for different UML models. These metrics have certainly helped designers analyze their UML models to an extent. However, as researchers' focus has shifted to the activities related to model management (such as model composition, evolution, and transformation), the shortcomings, and limitation of UML model metrics have become more apparent. Some authors (Fenton & Pfleeger, 1996; Lorenz & Kidd, 1994; Chidamber & Kemerer, 1994) have proposed a set of metrics that can be applied to measure UML models' properties. These works have shown that their measures satisfy some properties expected for good measures of design models. However, these metrics cannot be employed to assess problems that may arise in a model composition process such as semantic inconsistencies.

There are some specific metrics available in the literature for supporting the evaluation of model composition specifications. For instance, Chitchyan and colleagues (Chitchyan et al., 2009) have defined some metrics to quantify the effort to specific compositions between two or more requirements models, such as scaffolding and mobility. However, their metrics are targeted at evaluating the reusability and stability of explicit model composition specifications. Boucké and colleagues (Bouke et al., 2006) propose a number of metrics for evaluating the complexity and reuse of architectural model compositions. However, in this study, we have focused on the evaluation of heuristic composition algorithms, such as merge and override, where explicit model compositions are not provided up front. In addition, we have focused on analyzing the impact of aspects on the effort to resolve emerging inconsistencies in output models. Therefore, existing metrics (such as those described in (Chitchyan et al., 2009; Bouke et al., 2006)) cannot be directly applied to our context.

6.2.5. Threats to Validity

The exploratory study obviously has a number of threats to validity that range from (Wohlin et al., 2000): (i) the use of single target application and a single AOM language, to (ii) the use of specific metrics to compute the conflict resolution effort. Obviously, more investigations involving other case studies with compositions of larger UML models are required. We observed that the number of properties and details (i.e., granularity) of the model elements taken into consideration throughout the compositions affect directly the composition results. Consequently, it is necessary to observe that, to generalize our findings, other types of model with different levels of abstraction are needed to make further investigation.

Further empirical evaluations are indeed fundamental to confirm or refute our findings in other real-world design settings involving UML model compositions. However, it was never our goal to conduct a controlled study. Our investigation represents a first stepping-stone, where a number of initial findings can be used to drive the experimental designs of more controlled studies in the future.

6.2.6. Conclusions and Future Work

Model composition is one of the pillars of AOM, and it is an operation intended to be used in many software development activities. Hence, software designers naturally become concerned about the quality of the composed models. This study represents a first exploratory study to assess the potential advantage of aspect-orientation in reducing conflict resolution effort. In our study, model composition was used to express the evolution of architectural models along six releases of a software product line. Three canonical algorithms for heuristic model composition have been applied, and two of them were discussed in detail in this study. As expected, we found that the presence of aspects in input models improved modularization and, therefore, tended to better localize inconsistencies.

We have also observed: (i) a higher degree of obliviousness between base models and aspects led to a significant decrease of inconsistencies when compared to the non-AO model counterparts, and (ii) aspects with higher quantification were the cause of higher inconsistency rates in AO models. Another interesting finding was that, even in scenarios where the inconsistency rate of AO models was close to (or higher than) the inconsistency rate of non-AO models, conflict resolution effort was similar in AO and non-AO models. This means that the time spent in recovering output AO models from emerging inconsistencies is, at least, similar to non-AO models. All these findings were independent of the specific composition algorithms being assessed. These results provide some initial indication that aspect-orientation may alleviate conflict resolution effort.

We should point out that assessing the benefit of AOM in model composition is in its initial stage and there is little experience that can be used to determine the feasibility of current approaches. This study represents a first exploratory study that investigates the impact of aspects on conflict resolution effort. However, further empirical studies are still required to evaluate the impact of AO modeling on model composition in real-world settings. We also need to better understand if aspect orientation provides some gain or not: (i) when applied to other composition algorithms, and (ii) with respect to the time spent to identify the inconsistencies rather than the effort to resolving them. We hope that the issues outlined throughout the study encourage researchers to replicate our study in the future under different circumstances.