On a Feature-Oriented Characterization of Exception Flows in Software Product Lines

Hugo Melo, Roberta Coelho, Uirá Kulesza
Department of Informatics and Applied Mathematics (DIMAp)
Federal University of Rio Grande do Norte (UFRN)
Natal, Brazil
hugofariamelo@gmail.com, {roberta, uira}@dimap.ufrn.br

Abstract—The Exception Handling (EH) is a widely used mechanism for building robust systems. In Software Product Line (SPL) context it is not different. As EH mechanisms are embedded in most of mainstream programming languages, we can find exception signalers and handlers spread over code assets associated to common and variable SPL features. When exception signalers and handlers are added to an SPL in an unplanned way, one of the possible consequences is the generation of faulty family instances (i.e., instances on which common or variable features signal exceptions that are mistakenly caught inside the system). This paper reports a first systematic study, based on manual inspection and static code analysis, in order to categorize the possible ways exceptions flow in SPLs, and analyze its consequences. Fault-prone exception handling flows were consistently detected during this study; such as flows on which a variable feature signaled an exception a different variable feature handled it.

Keywords—Software Product Line, Exception Handling, Static Analysis, Code Inspection.

I. INTRODUCTION

The highly dynamic and competitive market stimulated the creation of a class of applications, structured as a family of similar systems, that shares a common architecture and address a specific market segment [2]. Such system families, also called software product lines (SPLs), provide benefits such as: reduced time-to-market, as many products with small variations can be built; and improvement on systems quality, as the artifacts developed once can be reused and tested many times. The software product line (SPL) architecture addresses a set of common and variable features from which different products can be derived in a systematic way [1].

The Exception Handling (EH) is a widely used mechanism for building robust systems [4][5]. As EH mechanisms are embedded in most of mainstream programming languages, in SPL engineering it is also used as a way of structuring fault detection and recovery solutions [15]. Exception signalers and handlers can then be found spread over code assets associated to common and variable features.

Intriguing questions arise when developing an exception-aware SPL, such as: How do exceptions flow through both variable and common features of SPLs? Can we anticipate if any product/instance of the SPL will lead to a faulty exception handling behavior? A faulty exception behavior may happen when code assets implementing common or variable features signal exceptions that are mistakenly caught inside the system (a bug known as an exception handling bug very difficult to detect, known as Unintended Handler Action [6]).

Aiming at providing answers to these questions, this paper reports a systematic study, based on manual inspection and static code analysis, which quantitatively assesses the ways exceptions flow through SPL features. The study was based on two well-known benchmark software product lines, i.e., MobileMedia [7] and Berkeley DB [8]. For MobileMedia seven releases were examined, and for Berkeley DB a single release was examined. Overall, this corresponds to 55KLOC of Java source code of which around 6.3 KLOC are dedicated to exception handling. From such code base 1.175 exception flows 1 were found and characterized. Some outcomes consistently detected through out this study were the following:

- A considerable amount of exception flows originated from variable features (23% in MobileMedia 7th release and 20% in Berkeley DB).
- Several exception flows that originated from variable features were handled by core elements (63% in Berkeley DB and 33% in MobileMedia 7th release2). In the core assets such exceptions were mainly caught by generic handlers (86% in Berkeley DB) or by specialized (but many times ineffective) handlers (100% in MobileMedia 7th).
- Some of the exception flows originated by variable features were handled by the same feature that had signaled the exception (only 11% in Berkeley DB and 39% in MobileMedia 7th release). 3
- We could also detect flows on which a different variable feature caught the exception thrown by another variable feature (29% in MobileMedia 7th and 25% in Berkeley DB).

This last scenario, on which a variable feature signaled an exception and another feature handled it, alerts to an implicit interaction between features, that is usually not documented.

1 This study only considers exceptions flows generated by checked exceptions that are signaled inside the SPL assets.
2 These percentages are calculated in relation to the number of exception flows that are originated from variable features.
and that can lead to the faulty exception handling behavior mentioned previously: depending on the SPL instance, the feature that is handling the exception may not be present and the exception may remain uncaught or may be mistakenly caught inside the system. These and other fault-prone exception handling flows were consistently detected during this study and pinpoints to an exception handling confusion [22] problem that may arise from such implicit relation between features in exception handling scenarios.

The remainder of this paper is organized as follows. Section II presents the study settings, the target product lines and the assessment procedures. Section III presents and discusses the empirical data collected in this study. Section IV provides further discussions and lessons learned. Section V describes the related work. Finally, Section VI presents our conclusions. Due to space limitations, throughout this article we assume that the reader is familiar with SPL techniques and terminology [1][23].

II. THE CHARACTERIZATION PROCESS

This section describes our study settings in terms of target product line characteristics and the characterization process phases.

A. Target Software Product Lines

One major decision that had to be made for our investigation was the selection of the target SPLs. We have selected two medium-sized well-known benchmark SPLs implemented in Java: MobileMedia [7], and Berkeley DB [8]. Such SPLs implement variable behavior and associated features using #ifdef constructs and CIDE (Colored IDE) [8], respectively. CIDE is a tool that enables SPL developers to annotate code with feature information using background colors (instead of #ifdefs). Both SPLs were used in several empirical studies [24][14][8][18], and each of them is a representative of different application domains, and heterogeneous realistic ways of incorporating exception handling.

The MobileMedia (MM) is a software product line (SPL) of applications that manipulate media (e.g., photo, music and video) on mobile devices. It has been implemented in 9 subsequent Java releases, each incorporates new optional or alternative features. In our study, we have analyzed releases 1 to 7. All of them adopt the same architecture style (i.e., model-view-controller), varying in terms of the number of features available and design decisions taken in each version. Figure 1 illustrates the feature model of the 7th release of MobileMedia.

![Figure 1: Feature model for the 7th release of MobileMedia.](image)

Berkeley DB (BkDB) [8] is a product line for embeddable databases of moderate size (42 features). For the sake of space limitation, details of concerning Berkeley DB architecture and feature model are omitted in this paper. Table I summarizes code characteristics of both target SPLs: the number of lines of code; the number of lines of code dedicated to exception handling; the number of throw-clauses and catch-blocks; number of classes; number of user-defined checked and unchecked exceptions; and the number of features.

![Table I. Code characteristics per system.](image)

<table>
<thead>
<tr>
<th>Software Product Lines</th>
<th>MM01</th>
<th>MM02</th>
<th>MM03</th>
<th>MM04</th>
<th>MM05</th>
<th>MM06</th>
<th>MM07</th>
<th>BkDB</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOC</td>
<td>1158</td>
<td>1315</td>
<td>1363</td>
<td>1557</td>
<td>2056</td>
<td>2512</td>
<td>3016</td>
<td>39233</td>
</tr>
<tr>
<td>EHLOC</td>
<td>277</td>
<td>306</td>
<td>315</td>
<td>356</td>
<td>443</td>
<td>471</td>
<td>575</td>
<td>3028</td>
</tr>
<tr>
<td>#throw statements</td>
<td>19</td>
<td>19</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>21</td>
<td>24</td>
<td>402</td>
</tr>
<tr>
<td>#catch blocks</td>
<td>35</td>
<td>42</td>
<td>44</td>
<td>51</td>
<td>69</td>
<td>68</td>
<td>91</td>
<td>252</td>
</tr>
<tr>
<td>#exceptional flows</td>
<td>33</td>
<td>35</td>
<td>36</td>
<td>44</td>
<td>55</td>
<td>77</td>
<td>91</td>
<td>796</td>
</tr>
<tr>
<td>#classes</td>
<td>24</td>
<td>25</td>
<td>25</td>
<td>30</td>
<td>37</td>
<td>46</td>
<td>50</td>
<td>238</td>
</tr>
<tr>
<td>#user-defined checked exceptions</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>22</td>
</tr>
<tr>
<td>#user-defined unchecked exceptions</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>#features</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>6</td>
<td>9</td>
<td>42</td>
</tr>
</tbody>
</table>

3 Releases 1 to 7 were downloaded from the repository http://sourceforge.net/scm/?type=repo&group_id=226960 and are equivalent to the releases 2-8 out of 8 releases described at [24].
TABLE II. QUANTIFICATION OF EACH FLOW TYPE PER SPL RELEASE, AND THE PERCENTAGE OF SUCH FLOWS IN RELATION TO ALL FLOWS PER SPL RELEASE.

<table>
<thead>
<tr>
<th>Flow Type</th>
<th>Signaler</th>
<th>Handler</th>
<th>Handler Action</th>
<th>MM01</th>
<th>MM02</th>
<th>MM03</th>
<th>MM04</th>
<th>MM05</th>
<th>MM06</th>
<th>MM07</th>
<th>BKDB</th>
<th>All of F Flows</th>
</tr>
</thead>
<tbody>
<tr>
<td>CC</td>
<td>Core</td>
<td>Core</td>
<td>any</td>
<td>32 (100%)</td>
<td>32 (100%)</td>
<td>34 (94.9%)</td>
<td>36 (81.8%)</td>
<td>35 (64.8%)</td>
<td>42 (54.5%)</td>
<td>43 (47.3%)</td>
<td>434 (53.6%)</td>
<td>688</td>
</tr>
<tr>
<td>CCS</td>
<td>Core</td>
<td>Core</td>
<td>specialized</td>
<td>22 (68.8%)</td>
<td>22 (66.7%)</td>
<td>24 (66.7%)</td>
<td>26 (59.1%)</td>
<td>25 (46.3%)</td>
<td>29 (39.7%)</td>
<td>30 (33%)</td>
<td>64 (7.9%)</td>
<td>242</td>
</tr>
<tr>
<td>CCG</td>
<td>Core</td>
<td>Core</td>
<td>generic</td>
<td>10 (31.3%)</td>
<td>10 (30.3%)</td>
<td>10 (27.8%)</td>
<td>10 (22.7%)</td>
<td>10 (18.5%)</td>
<td>13 (16.9%)</td>
<td>13 (14.3%)</td>
<td>370 (45.7%)</td>
<td>446</td>
</tr>
<tr>
<td>CV</td>
<td>Variable</td>
<td>any</td>
<td>any</td>
<td>0 (0%)</td>
<td>1 (3%)</td>
<td>2 (5.6%)</td>
<td>7 (15.9%)</td>
<td>17 (31.5%)</td>
<td>22 (28.6%)</td>
<td>27 (29.7%)</td>
<td>416 (50.5%)</td>
<td>789</td>
</tr>
<tr>
<td>CVS</td>
<td>Variable</td>
<td>specialized</td>
<td>any</td>
<td>0 (0%)</td>
<td>0 (0%)</td>
<td>0 (0%)</td>
<td>0 (0%)</td>
<td>0 (0%)</td>
<td>0 (0%)</td>
<td>0 (0%)</td>
<td>0 (0%)</td>
<td>0 (0%)</td>
</tr>
<tr>
<td>CVG</td>
<td>Core</td>
<td>Generic</td>
<td>any</td>
<td>0 (0%)</td>
<td>0 (0%)</td>
<td>0 (0%)</td>
<td>0 (0%)</td>
<td>0 (0%)</td>
<td>0 (0%)</td>
<td>0 (0%)</td>
<td>0 (0%)</td>
<td>0 (0%)</td>
</tr>
<tr>
<td>Sub-Total (CV + CC)</td>
<td>32 (100%)</td>
<td>33 (100%)</td>
<td>36 (100%)</td>
<td>43 (97.7%)</td>
<td>52 (96.3%)</td>
<td>64 (83.1%)</td>
<td>70 (76.9%)</td>
<td>648 (80%)</td>
<td>978</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VC</td>
<td>Variable</td>
<td>Core</td>
<td>any</td>
<td>0 (0%)</td>
<td>0 (0%)</td>
<td>0 (0%)</td>
<td>0 (0%)</td>
<td>0 (0%)</td>
<td>0 (0%)</td>
<td>0 (0%)</td>
<td>0 (0%)</td>
<td>0 (0%)</td>
</tr>
<tr>
<td>VCS</td>
<td>Variable</td>
<td>Core</td>
<td>specialized</td>
<td>0 (0%)</td>
<td>0 (0%)</td>
<td>0 (0%)</td>
<td>0 (0%)</td>
<td>0 (0%)</td>
<td>0 (0%)</td>
<td>0 (0%)</td>
<td>0 (0%)</td>
<td>0 (0%)</td>
</tr>
<tr>
<td>VVG</td>
<td>Variable</td>
<td>Variable</td>
<td>generic</td>
<td>0 (0%)</td>
<td>0 (0%)</td>
<td>0 (0%)</td>
<td>2 (3.2%)</td>
<td>4 (6.2%)</td>
<td>8 (8.8%)</td>
<td>18 (22.2%)</td>
<td>33</td>
<td></td>
</tr>
<tr>
<td>VV</td>
<td>Variable</td>
<td>Variable</td>
<td>any</td>
<td>0 (0%)</td>
<td>0 (0%)</td>
<td>0 (0%)</td>
<td>1 (2.3%)</td>
<td>2 (3.7%)</td>
<td>4 (5.2%)</td>
<td>8 (8.8%)</td>
<td>18 (22.2%)</td>
<td>33</td>
</tr>
<tr>
<td>VaVb</td>
<td>Variable (a)</td>
<td>Variable (b)</td>
<td>any</td>
<td>0 (0%)</td>
<td>0 (0%)</td>
<td>0 (0%)</td>
<td>0 (0%)</td>
<td>0 (0%)</td>
<td>0 (0%)</td>
<td>0 (0%)</td>
<td>0 (0%)</td>
<td>0 (0%)</td>
</tr>
<tr>
<td>VaVbS</td>
<td>Variable (a)</td>
<td>Variable (b)</td>
<td>specialized</td>
<td>0 (0%)</td>
<td>0 (0%)</td>
<td>0 (0%)</td>
<td>0 (0%)</td>
<td>0 (0%)</td>
<td>0 (0%)</td>
<td>0 (0%)</td>
<td>0 (0%)</td>
<td>0 (0%)</td>
</tr>
<tr>
<td>VaVbG</td>
<td>Variable (a)</td>
<td>Variable (b)</td>
<td>generic</td>
<td>0 (0%)</td>
<td>0 (0%)</td>
<td>0 (0%)</td>
<td>0 (0%)</td>
<td>0 (0%)</td>
<td>0 (0%)</td>
<td>0 (0%)</td>
<td>0 (0%)</td>
<td>0 (0%)</td>
</tr>
<tr>
<td>Sub-total (VC + VV + VaVb)</td>
<td>80(0%)</td>
<td>80(0%)</td>
<td>80(0%)</td>
<td>80(0%)</td>
<td>80(0%)</td>
<td>80(0%)</td>
<td>80(0%)</td>
<td>80(0%)</td>
<td>80(0%)</td>
<td>35(4.3%)</td>
<td>35</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td></td>
<td></td>
<td>32</td>
<td>33</td>
<td>36</td>
<td>44</td>
<td>56</td>
<td>77</td>
<td>91</td>
<td>809</td>
<td>1175</td>
</tr>
</tbody>
</table>

variable and common features of SPLs? Can we anticipate if any product/instance of the SPL will lead to a faulty exception handling behavior? Details concerning the manual inspection and the feature-oriented exception flow analysis tool are presented below.

1) The Analysis of Exception Handling Behaviour

The main goal of this step was to firstly explore the possible ways exceptions could flow inside a SPL. In this task we took into account the information concerning which features were associated to the pieces of code responsible for signaling and handling exceptions. The exception flows, identified through manual inspection, were characterized according to three attributes:

(i) The feature associated with the exception signaler, the piece of code responsible for signaling the exception can be associated to the core (C) or to a variable\(^4\) feature (V);

(ii) The feature associated with the exception handler, the piece of code responsible for handling the exception can be associated to the core (C) or to a variable feature (V); and finally

(iii) The exception handling action, the exception can be caught by an specialized handler (S), a handler whose type is of the same of the exception being caught, or by a generic handler (G), a handler whose type is a supertype of the exception being caught.

Table II presents the number of exception flows found in this study, characterized according to these attributes. This table also shows the percentage of each flow type in relation to all flows found per SPL release. The flow type, first column of

\(^4\) In this study we refer to optional and alternative features indistinctively as variable features.
Analyzer performs an inter-procedural analysis on the SPL bytecode. It implements one of the most used algorithms for call graph construction called Class Hierarchy Analysis (CHA) [28], in order to build the program dependency graph (PDG) [16]. Then, it traverses the PDG, firstly looking for the checked exceptions, explicitly thrown by the SPL assets, and then looking for handlers that may handle them. As a result the plug-in reports a set of exception flows.

The Flow Identifier is responsible for accessing the Configuration Knowledge (CK) [23] of the LPS to obtain the information concerning the features associated to each signaler and handler of each exception flow. The CK format depends on the tool used for the SPL implementation and variability management. The current version of PLEA accepts the CK defined by CIDE [8], the tool used for the variability management of Berkeley DB.

C. Study Operation

This study was undertaken between August 2011 and March 2012. During this period the target product lines were selected, the manual inspection was conduced over seven releases on MobileMedia, and the static analysis tool was implemented and executed for over BerkeleyDB. Moreover, after the exception flows characterization, a deeper inspection of specific flows was conducted.

III. EMPIRICAL DATA

This section summarizes the study results collected during the manual inspection of releases 1 to 7 of MobileMedia and the execution of PLEA tool over Berkeley DB. Next subsections discuss over the information presented in Tables II and III. Section A analyzes the evolution of exception flows across 7 releases of MobileMedia. Then, Section B contrasts the occurrence of exception flows in the 7th release of MobileMedia with Berkeley DB, focusing on the analysis of the fault-proneness of specific exception flow types.

### Table III. Quantification of each flow type per SPL release, and the percentage of flows in relation to the flows originated by variable features.

<table>
<thead>
<tr>
<th>Flow Type</th>
<th>Signaler</th>
<th>Handler</th>
<th>Handler Action</th>
<th>MM01</th>
<th>MM02</th>
<th>MM03</th>
<th>MM04</th>
<th>MM05</th>
<th>MM06</th>
<th>MM07</th>
<th>BKDB</th>
<th>All</th>
</tr>
</thead>
<tbody>
<tr>
<td>VC</td>
<td>Variable</td>
<td>Core</td>
<td>any</td>
<td>0(0%)</td>
<td>0(0%)</td>
<td>0(0%)</td>
<td>0(0%)</td>
<td>0(0%)</td>
<td>0(0%)</td>
<td>0(0%)</td>
<td>0(0%)</td>
<td>38(38%)</td>
</tr>
<tr>
<td>VCS</td>
<td>Variable</td>
<td>Core</td>
<td>specialized</td>
<td>0(0%)</td>
<td>0(0%)</td>
<td>0(0%)</td>
<td>0(0%)</td>
<td>0(0%)</td>
<td>0(0%)</td>
<td>0(0%)</td>
<td>0(0%)</td>
<td>8(8%)</td>
</tr>
<tr>
<td>VCG</td>
<td>Variable</td>
<td>Core</td>
<td>generic</td>
<td>0(0%)</td>
<td>0(0%)</td>
<td>0(0%)</td>
<td>0(0%)</td>
<td>0(0%)</td>
<td>0(0%)</td>
<td>0(0%)</td>
<td>0(0%)</td>
<td>88(55%)</td>
</tr>
<tr>
<td>VV</td>
<td>Variable</td>
<td>Variable</td>
<td>any</td>
<td>0(0%)</td>
<td>0(0%)</td>
<td>0(0%)</td>
<td>0(0%)</td>
<td>0(0%)</td>
<td>0(0%)</td>
<td>0(0%)</td>
<td>0(0%)</td>
<td>0(0%)</td>
</tr>
<tr>
<td>VVS</td>
<td>Variable</td>
<td>Variable</td>
<td>specialized</td>
<td>0(0%)</td>
<td>0(0%)</td>
<td>0(0%)</td>
<td>0(0%)</td>
<td>0(0%)</td>
<td>0(0%)</td>
<td>0(0%)</td>
<td>0(0%)</td>
<td>0(0%)</td>
</tr>
<tr>
<td>VVG</td>
<td>Variable</td>
<td>Variable</td>
<td>generic</td>
<td>0(0%)</td>
<td>0(0%)</td>
<td>0(0%)</td>
<td>0(0%)</td>
<td>0(0%)</td>
<td>0(0%)</td>
<td>0(0%)</td>
<td>0(0%)</td>
<td>0(0%)</td>
</tr>
<tr>
<td>VaVb</td>
<td>Variable (a)</td>
<td>Variable (b)</td>
<td>any</td>
<td>0(0%)</td>
<td>0(0%)</td>
<td>0(0%)</td>
<td>0(0%)</td>
<td>0(0%)</td>
<td>0(0%)</td>
<td>0(0%)</td>
<td>0(0%)</td>
<td>0(0%)</td>
</tr>
<tr>
<td>VaVbS</td>
<td>Variable (a)</td>
<td>Variable (b)</td>
<td>specialized</td>
<td>0(0%)</td>
<td>0(0%)</td>
<td>0(0%)</td>
<td>0(0%)</td>
<td>0(0%)</td>
<td>0(0%)</td>
<td>0(0%)</td>
<td>0(0%)</td>
<td>0(0%)</td>
</tr>
<tr>
<td>VaVbg</td>
<td>Variable (a)</td>
<td>Variable (b)</td>
<td>generic</td>
<td>0(0%)</td>
<td>0(0%)</td>
<td>0(0%)</td>
<td>0(0%)</td>
<td>0(0%)</td>
<td>0(0%)</td>
<td>0(0%)</td>
<td>0(0%)</td>
<td>0(0%)</td>
</tr>
<tr>
<td>Sub-total (VC + VV + VaVb)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>13</td>
<td>21</td>
<td>161</td>
<td>98</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A. Exception Flow Types across different Releases

Figure 2 illustrates the number of exception flows of each type found in releases 1 to 7 of MobileMedia. It shows how exception flows evolved across the different releases, as SPLs features were changed and new features were added.

To better interpret the exception flow evolution across MobileMedia releases Table IV briefly describes how features evolved through releases 1 to 7.

### Table IV. Summary of changes between MobileMedia releases.

<table>
<thead>
<tr>
<th>Change</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>core assets were created</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>an optional and a common feature were added</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>an optional feature was added</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>an optional feature was added</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>a common feature was converted into two alternative features</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>an alternative and two optional features were added</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>

We can observe from Figure 2 that the first type of exception flow, involving variable features, found in this study, was the CV flow, which represents exceptions signaled by core
elements and caught by variable feature elements. Figure 2 shows that the number of CV flows increased across the releases more significantly than the other flow types. In the 7th version it represents almost 30% of all flows (see Table II).

Figure 3 (a) shows how the number of flows originated from core elements rises through the evolution of MobileMedia. Inspecting the code of MM releases we could observe that the main reason for the increase in CV flows was the fact the new features added (as illustrated in Table IV) reused core elements; and the exceptions thrown by the core elements were then handled by the new variable features.

All together, the exception flows originated from variable features (VC, VV and VaVb) correspond to 23% of all flows. In all of such flows, specialized handlers caught the exceptions. Figure 3 (b) shows how the number of these three types of flows (originated from variable features) rises through the evolution of MobileMedia. The main reason for such increase in the number such flows was the major refactoring performed in the 6th release of MobileMedia, on which a common feature was converted to two variable features (as illustrated in Table III).

However, to know whether the exceptions signaled or handled in such flows had been adequately caught, a deeper analysis of each flow was needed. Next subsection will illustrate examples of such flow types while comparing and contrasting with exception flows found in Berkeley DB.

B. Exception Flow Types across different SPLs

This section compares the exception flow information obtained while analyzing the 7th release of MobileMedia and the single release of Berkeley DB.

Figure 4 illustrates the proportion of exception flows found in each SPL. We could observe that although distinct in domain and number of features (Berkeley DB contains 42 variable features, while the 7th release of MobileMedia contains 9) such SPLs were similar in the proportion of the flow types found.

However, considering the handling action associated to the exception flows found, we could observe that the exception handling policy strongly differed between the analyzed SPLs. As illustrated in Figure 5, specialized handlers caught most of the flows signaled by the core elements in MobileMedia, but in Berkeley DB, generic handlers caught most of the exceptions signaled by core elements.

Next subsections discuss how the exceptions signaled by core elements and variable features are caught in MobileMedia and Berkeley DB.
1) How the Exceptions Signaled by Core Elements are Handled

We could observe that in both SPLs most of the exception flows were signaled by core elements. From this set, most of the exception flows were also caught by core elements (see CC flows in Table II, 47.3% in MM7 and 53.6% BkDB). Such flows were not inspected in detail since they did not involved any variable feature, which was the focus of this study. The second flow type in terms of number of occurrences, in both MM7 and MkDB SPLs, were the ones signaled by core elements and handled by a variable feature (see CV flows in Table II, 29.7% in MM7 and 26.4% in BkDB) - which were generated by core elements and caught by elements associated with variable features.

One instance of the CV flows is depicted in Figure 8(a) It presents a method call chain on which: a method defined on a Sorting variable feature (i.e., incrementCountViews()) calls a method defined on a core element (i.e., updateMedia()), which transitivity calls a method that signals an exception (i.e., updateMediaInfo()). We could observe that this kind of flow was common in the analyzed SPLs. A reason for this is that most of SPL variable features use supporting services provided by the core, such as persistence, security, and so on. Once the developers responsible for variable features may know the exceptions that can flow from core elements (defined during domain engineering), they can define specific handlers to such exceptions.

a) Specialized x Generic Handling

Considering the way the exceptions thrown by core elements were handled (i.e., by generic or specialized handlers), we could observe that in MM7 most of the exceptions thrown by core elements were handled by specialized handlers (see Figure 6(a)); only few of them were caught by generic handlers (14.3%). In contrast, in BerkeleyDB, generic handlers caught almost all exceptions signaled in the core.

2) How the Exceptions Signaled by Variable Features are Handled

We could also observe that a considerable amount of exception flows originated from variable features (23% in MM7 and 20% in BkDB, see Table II). Table III focus on such exception flows, originated from variable features. From this set, most of them were handled by core elements. Only part of such flows was indeed handled by the same variable feature that had signaled it (see the VV flows in Table II, 38% in MM7 and 11% in BkDB). We could also detect flows on which a different variable feature caught the exception (29% in MM7 and 25% in BkDB, see the VaVb flows in Table III). Next subsections present examples of these flow types and discuss about their fault-proneness.

a) Exceptions signaled by a variable feature and handled in the core (VC).

In Berkeley DB product line, most of the exceptions signaled by variable features were captured by generic handlers defined in a core element (see VCG flows of BkDB in Table II).
Figure 8(b) illustrates a scenario on which a method from the core of Berkeley DB calls methods from different variable features. Figure 7 presents the partial code of such method. This method from the core assets contains about 100 lines of code, and refers to 8 distinct variable features (through method calls), which can signal specific exceptions. The code snippet only presents three of such variable features that can signal instances of DatabaseException (see A, B and C tags on the code). Moreover, such method also accesses other core methods that can also signal specific exceptions. Although distinct exceptions may flow in such method a single and generic treatment is given for all of them (in lines 30-34).

b) Exceptions signaled and handled by the same variable feature (VV).

Manually inspecting all such flows in detail for MobileMedia SPL we could observe that, in all of them, the intermediate elements (the methods called between the signaler and the handler) were also associated to a single feature, which means that, in such flows, no core supporting functions were reused. Figure 8(c) illustrates one of such flows. In the analyzed SPLs, all flows signaled and caught inside the feature context (the VV flows) were handled using specialized handlers – which can lead to an adequately handling as a single feature can have enough information to handle the exception. An interesting finding was that even in Berkeley DB SPL, where most of the exceptions were handled by generic handlers (see Figure 5), the exceptions signaled and handled in the context of a single feature were caught by specific handlers.

c) Exceptions signaled and handled by distinct variable features (VaVb).

Flows VaVb are the ones whose signaler was defined on a variable feature and the handler on a different variable feature. Such flows bring out an implicit relation between features that may lead to EH faulty behavior as presented next.

Figure 9 illustrates the code snippet of a method from a variable feature of MM07 called Sorting, which is responsible for sorting lists of different kinds of medias. The method called in line 5 transitively calls other methods as illustrated in Figure 8(d); one of such methods in the call chain belongs to the Music feature (i.e., getBytesFromMediaInfo())\(^5\) which can signal InvalidImageDataException (represented as IIDE in Figure 8(d)). When such exception is signaled by

---

\(5\) This method overrides getBytesFromMediaInfo() method defined in a core element. This is the reason why this method appears twice in the call chain. Part of this is also present in the VC flow example in Figure 7(a) but the exceptions signaled and their signalers are different in both cases.
Music feature it is caught by the handler defined in line 11-12 (see Figure 9), and a message related to another feature (i.e., Photo feature that may also interact with Sorting feature) is presented to the user. Such exception handling confusion problem is a consequence of the implicit feature interaction in exception handling scenarios.

When a feature, different from the one that signaled the exception, handles an exception, such feature does not have enough contextual information to adequately handle it. Such scenarios usually happen by mistake, in another words, the exception signaled by the feature would remain uncaught but was mistakenly caught by a handler defined by another feature (an instance of the Unintended Handler Action problem).

Even though the exception signaled by the variable feature Music was of a different type, the Sorting feature could handle it if it defined a catch all clause (such as catch Exception or catch Throwable clauses). Hence, currently there is no way of preventing the exceptions signaled by a feature to be caught by any other element defined on the core or another feature as such elements share the same exception hierarchy.

d) The Implicit Feature Interaction in EH Context

In this study, when an exception signaled by a variable feature was handled by a different variable feature (as illustrated by the VaVb exception flows) an implicit interaction between features was brought out. Such kind of interaction, which only happens in exception handling scenarios, was not documented in any of the SPLs analyzed in this study.

Moreover, in the scenarios were VaVb flows were detected there was no constraint associating the feature which signaled the exception and the feature that handled it. One of the consequences of such implicit interaction was in fact the EH faulty behavior presented previously. Since, there was no explicit relation between the exception signaler and handler features, in such SPLs we could a product (instance of the SPL) on which one of such features was added. We could for instance generate a product on which the feature responsible for handling the exception was not present. In such cases the exceptions could remain uncaught or was mistakenly caught by any other element (in a feature or in the core).

e) Specialized Handling x Adequate Handling

Considering the way the exceptions thrown by variable features were handled (i.e., by generic or specialized handlers), we could observe that in MM7 specialized handlers caught all the exceptions signaled by variable features (see Figure 6(b)). It may give the impression that the exceptions signaled by features are being adequately handled inside the MM7 SPL. However, only a deeper analysis could show the real scenario.

After manually inspecting these exception flows we observed that 7 out of 9 variable features defined in MM7 reused exceptions defined by core elements and added a new semantic to them. Hence, even though such exceptions were caught by specialized handlers (handler of the same type of the exception being caught), the handling action was not adequate, as illustrated in Figure 9. The problems derived by reusing exception types and adding different semantics were already mentioned by [6] and are not restricted to SPL context but can happen in any Java system. But what we could noticed was that such problem was strongly observed in MM7.

IV. DISCUSSIONS

This section provides further discussion of issues and lessons learned while performing this study.

A. Collateral Effects of Specific Flow Types

In our study we classified all exception flows found on two SPLs according to their signalers and handlers. Such characterization enabled us to identify the most common flow types and analyze how the characteristics of such flows could lead to faulty exception handling behaviors.

It is known that the exception handling policy of a system or product lines depends on others factors than the intrinsic ones (i.e., software architecture). Design decisions, coding patterns or company-specific policies, and developer’s experience [26] may also affect the way exceptions are signaled and handled inside the system. Hence, what is an inadequate handling for a system may be a design decision for another. However, it is also known that specific exception handling patterns may lead to faulty exception handling behavior affecting the system robustness (e.g., Unintended Handler Action, and Generic handling [6]). In this study, by performing a deeper analysis of specific flow types, we could consistently detect such faulty scenarios in the analyzed SPLs, such as the ones associated to VC and VaVb flows presented in Section III.

B. The Implicit Feature Interaction and The Exception Handling Confusion Problem.

The implicit interactions between features were already pointed out as a source of failures by other works [11]. However such works focused on detecting implicit relationships in the normal control flow when features share data values. In our work, we found an implicit relation that arises between features in the exceptional flow (when a feature handles the exception signaled by other feature), and could observe that it could lead to the exception handling confusion problem already mentioned in the context of aspect oriented development [22].

C. Exception Handling Guidelines for Software Product Lines

The outcomes of our study also emphasizes the need for the definition of EH guidelines for SPLs. Such guidelines could motivate, for instance, application engineers to avoid throwing exceptions from their variable feature implementations to the core assets. In practice, however, there are several technical and organizational factors that impairs the full adoption of such kind of guidelines, such as: (i) the runtime exceptions from by third-party libraries used by the variable code assets; (ii) the difficulty in coordinating the work of product line and application engineers; and (iii) the natural complexity of the dependencies between SPL common and variable code assets. Given such restrictions to the full adoption of an EH guideline,
feature-oriented exception flow analysis tools, such as PLEA, would be strongly useful to detect violations of the practices defined on EH guidelines.

D. Study Limitations

One may argue that performing the characterization in a sample of 8 releases for two different SPLs is a limiting factor. The study phase based on manual code inspections (a very time-consuming task) restricted the number of releases evaluated in the study. Even under such restriction, the study analyzed 55KLOC of Java source code of which around 6,3 KLOC are dedicated to EH handling. From these base code 1175 exception flows were found, categorized and analyzed (i.e., all flows of MobileMedia and the flows of Berkeley DB that were signaled or caught by variable features were manually inspected). Another limitation of this study is the fact that it only considered the exceptions explicitly thrown by SPL code assets - excluding exceptions signaled from libraries and Java runtime environment. There are also the limitations inherent to the use of a static analysis tool (i.e., inheritance, polymorphism and virtual calls [19]), however such limitations were overcame by the manual inspection phase. Hence, the limitations of this study are similar to the ones imposed on the other empirical studies with similar goals [24][17][18][25].

V. RELATED WORK

This section presents related work organized in three categories: (i) empirical studies investigating the exception handling code of SPLs [24][18][15]; (ii) studies on implicit feature interactions [11][27][14]; and (iii) exception flow analysis tools and methods [20][19][17][21].

Empirical studies investigating the exception handling code of SPLs. Figueiredo et al. [24] presents an empirical study that aims to compare AO and OO Java implementations of the MobileMedia SPL. In their study, they have analyzed the stability of the EH feature across the SPL evolution in terms of modularization metrics (on the EH source code). In our study we discovered the exception flows originated from the EH code (manually and automatically through PLEA) and evaluated how such flows differ across different MM releases.

Coelho et al [18] performed an empirical study considering the fault-proneness of aspect-oriented implementations for handling exceptions. Two releases of both Java and AspectJ implementation of MobileMedia product line were assessed as part of that study. Although the study has analyzed the EH code of MM product line, it neither performed a feature-oriented analysis of the EH exception flows, nor discussed the fault-proneness of specific flow types related to variable features, as we have investigated in our work. Bertoncello et al [15] proposes a method for refactoring OO product line architecture in order to separate their normal and exceptional behavior into different software components. The proposed method motivates the introduction of variations points in the SPL core architecture to address different choices of exception handlers during product derivation. Our approach can be seen as complementary to the refactoring method proposed. First, the static analysis tool proposed in our work can be used to detect violations in the EH strategies established when evolving an SPL implementation with the introduction of new features or modification of existing ones. Second, our exploratory study also emphasizes the need to establish effective EH strategies to address the design and implementation of the core and variable features.

Studies on implicit feature interactions. Recent research work shows the importance and difficulty to analyze features dependencies in the context of SPL implementations using conditional compilation techniques, or similar approaches like CIDE [8]. Ribeiro et al. [11] propose the concept of emergent interface in order to address the analysis of feature dependencies when evolving a software product line. An emergent interface is used to capture the dependencies between code assets previously annotated and associated to specific features. Emergo [27] tool is used to automatically compute the emergent interfaces on demand based intra-procedural or inter-procedural dataflow analysis. Brabrand et al [14] propose and compare three different intra-procedural data analysis to detect the feature dependencies, in terms of undeclared variables, unused variables and null pointer. One of the great benefits of their approach is the capacity to analyze the feature dependencies for the complete SPL implementation instead of analyzing the code assets for each individual product separately. Similar to these works, our work improves the code assets dependency analysis using information about the kind of features (common or variable) they implement. However, such works address neither the analysis of exception flows in the context of SPL implementations, nor the implicit feature relation that comes about in the exceptional control flow.

Exception flow analysis tools and methods. Some research works propose solutions based on static analysis to calculate the exception flows of a system [20][19][17][21]. None of these tools however perform a feature-sensitive analysis as the one proposed in this study. The tool presented in this study performs a feature-oriented analysis of the exception flows, which allows a more accurate and detailed analysis of how exceptions flow through the code assets implementing the mandatory and variable features of a SPL.

VI. CONCLUDING REMARKS

This paper reported a systematic study which characterized and quantitatively assessed the ways exceptions flow through SPL features. Moreover, it also investigated fault-proneness of specific exception flow types in SPL implementations. The analysis was conducted in two existing benchmark software product lines – Mobile Media (releases 1 to 7) and Berkeley DB – using manual inspection and static code analysis respectively. As part of our study, we also developed a tool that performs an automated feature-oriented exception flow analysis. The tool can be useful to support the developer when implementing the exception handling behavior of variable and common features.

Overall 1175 exception flows were analyzed, and some outcomes were consistently detected through study, such as: most of the exceptions signaled by variable features were not adequately handled in the analyzed product lines. Moreover, we could also detect that some of the flows originated from
variable features were caught by a different variable feature (29% in MobileMedia/7th release and 25% in Berkeley DB). We believe that these and the other study outcomes presented in this paper are helpful in several ways, such as: (i) enhancing the general understanding of how exceptions flow through mandatory and variable features; (ii) providing information about the potential problems related to specific kinds of flows detected in this study (for instance the FaFb flows); (iii) briefly presenting how a static analysis tool can be used to support the identification of potentially faulty exception handling flows.

ACKNOWLEDGMENT
This work was partially supported by the National Institute of Science and Technology for Software Engineering (INES) - CNPq under grants 573964/2008-4 and CNPQ 560256/2010-8. Roberta Coelho is also supported by CNPq under grant 484037/2010-2.

REFERENCES