Do Coupling Metrics Help Characterize Critical Components in Component-based SPL? An Empirical Study

Anderson Uchôa[1], **Eduardo Fernandes**[1], **Ana Carla Bibiano**[1], **Alessandro Garcia**[1]

[1]Opus Research Group, Informatics Department, PUC-Rio
Rio de Janeiro, RJ, Brazil

{auchoa,emfernandes,abibiano,afgarcia}@inf.puc-rio.br

Abstract. *In component-based software product lines (SPL), each component has to encapsulate features, restrict data access, and be replaceable. For critical components, with multiple features and dependencies, these criteria are fundamental for flexible product configuration. Previous work assume that coupling metrics help characterize critical components, but we lack empirical evidence for that assumption. By characterizing critical components, we could help developers identify components that require careful maintenance and evolution. This paper relies on five well-known coupling metrics to compose a strategy for characterizing critical components in component-based SPLs. Our results suggest reasonable strategy's accuracy but the need for using additional metrics.*

1. Introduction

Since the advent of reuse techniques, component-based development is increasingly applied by organizations [Sharp 1998, Tischer et al. 2007] to build highly configurable software products, e.g., through software product lines (SPL) [Pohl et al. 2005]. Similarly, the literature introduces component-based SPLs as configurable sets of software products that share common and varying components [Atkinson et al. 2000]. To support flexible SPL product configuration, each component has to follow three criteria [Crnkovic and Larsson 2002, Krueger 2006]: encapsulate SPL functionalities, restrict data access, and be replaceable. In the case of critical components, which have several SPL functionalities and inter-component dependencies [Yacoub et al. 1999], these criteria are fundamental to assure the flexibility of SPL product configuration.

Software metrics [Lorenz and Kidd 1994] are pragmatic means to characterize code elements that are critical regard maintenance and evolution. Consequently, these metrics could also help in characterizing critical components in component-based SPLs. A previous study [Vale et al. 2016] suggests the lack of specific metrics for component-based development, which could be helpful to characterize SPL critical components. In despite of that, previous studies [Gill 2006, Her et al. 2007, Kessel and Atkinson 2015] often associate SPL critical components with conventional software metrics, mostly at the class level, such as a high coupling between components. Therefore, instead of proposing novel metrics specific to component-based SPL, *can we use conventional metrics, like coupling metrics, to help characterize critical components in component-based SPL?*

In this paper, we present an empirical study aimed at characterizing critical components in component-based SPLs. First, we collect from GitHub[1] three SPLs devel-

[1]https://github.com

oped in Java and SPL-enabling techniques. Second, we two researchers build a reference list of critical components based on the SPL developers report. Third, we rely on well-known catalogs of software metrics to select coupling metrics that somehow help capture critical components, by relying on classical definitions of critical components. Fourth, we compute both metrics and thresholds for the SPLs, based on a threshold derivation method [Vale and Figueiredo 2015]. Fifth, we define a metric-based strategy for characterizing critical components, by combining the selected coupling metrics. We also characterize the critical components per SPL using the defined strategy. Sixth, we assess the strategy's accuracy in terms of precision, recall, and F-measure [Fawcett 2006].

We summarize our contributions as follows. First, we propose a metric-based strategy for detecting critical components in component-based SPLs, by relying on five coupling metrics assumed by the literature as indicators of critical components. Second, we have observed a high rate of precision (up to 86.3%) for the strategy, but average to low rates of F-measure (up to 42.6%), and recall (up to 28.3%). Our results suggest that conventional coupling metrics may help characterize critical components in component-based SPLs. However, they do not suffice to provide a wide characterization of critical components. For instance, coupling metrics have shown unable to capture whether components properly restrict data access, which suggest the need for using additional metrics specific for component-based SPL, such as encapsulation metrics [Bouwers et al. 2014].

Section 2 provides background information. Section 3 describes the study settings. Section 4 presents our study findings. Section 5 discusses threats to validity. Finally, Section 6 concludes the paper and suggests future work.

2. Background

This section provides background information. Section 2.1 introduces component-based SPL. Section 2.2 discusses how to characterize critical components via software metrics.

2.1. Component-based Software Product Line

Component-based SPLs consist of multiple software products that share components [Atkinson et al. 2000]. They aim at providing flexible SPL product configuration. For this purpose, each component requires a careful design with focus on high encapsulation of SPL functionalities, limited data access, and high replaceability [Crnkovic and Larsson 2002, Krueger 2006]. By adopting component-based SPL, developers can spend efforts to implement novel SPL functionalities rather than re-implementing existing ones. In addition, during the life cycle of a component-based SPL, replacing an existing SPL component with another becomes easier. That is, the decomposition of SPL functionalities into components minimizes the effort required to adapt the SPL for changes, which makes easy to configure SPL products.

2.2. Critical Components and Software Metrics

Certain SPL components are inherently more essential for operating the SPL than others [Her et al. 2007]. For instance, a functionality often used by others may be critical for maintaining and evolving. Previous work [Atkinson et al. 2000, Her et al. 2007] mention these components as *critical components*, but they do not properly characterize such components [Gill 2006, Vale et al. 2016]. They associate SPL critical component with

metrics like as the number of concentrated SPL functionalities and the number of dependencies with other components. However, this assumption lacks empirical evidence on whether coupling metrics actually help characterize critical components in component-based SPLs. However, to the best of our knowledge, we did not find empirical studies that characterize critical components in component-based SPLs with coupling metrics.

3. Study Settings

This section presents the settings of our empirical study. Section 3.1 introduces the study goal and research questions. Section 3.2 describe the study steps.

3.1. Goal and Research Questions

This paper presents an empirical study aimed at characterizing critical components in component-based SPLs with coupling metrics. According to the guidelines of Wohlin el al. (2012), we describe the study goal as follows: *analyze* a software metric-based strategy; *for the purpose of* characterizing SPL critical components; *with respect to* precision, recall, and F-measure; *from the viewpoint of* SPL developers with experience in software metrics and component-based SPL development; *in the context of* conventional coupling software metrics, and component-based SPLs available at GitHub and developed in Java with support of SPL-enabling techniques. In order to guide our empirical study, we designed the following research question.

RQ1. *Do coupling metrics help characterize critical components in component-based software product lines?*

With RQ1, we assess coupling metrics as possible indicators of critical components. We did not find similar study for component-based SPLs. We expect to validate means for developers characterize SPL components that require careful maintenance and evolution. To assess RQ1, we compute precision (P) and recall (R) [Fawcett 2006] in terms of true positives (TP), false positives (FP), true negatives (TN), and false negatives (FN). TP is the number of critical components correctly identified by the strategy. FP is the number of critical components incorrectly identified by the strategy. TN is the number of non-critical components correctly not identified as critical by the strategy. Finally, FN is the number of non-critical component incorrectly not identified as critical by the strategy. We also compute F-measure (F) that balances precision and recall. The formula are $P = \frac{TP}{TP+FP}$, $R = \frac{TP}{TP+FN}$, and $F = 2 \times \left(\frac{P \times R}{P+R}\right)$.

3.2. Study Steps

Figure 1 presents the six steps of our empirical study. We describe each step as follows.

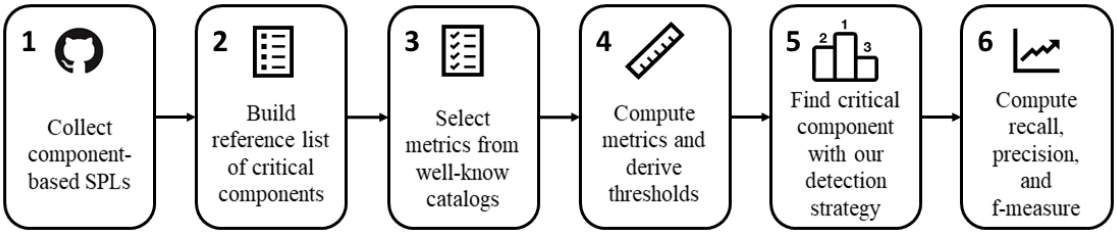

Figure 1. Study Steps

Step 1 aims at collecting SPLs from GitHub with the following search string: *"component-based" AND ("software product line" OR "product line").* Table 3.2 overviews each SPL: BET-SPL [Donegan and Masiero 2007], MobileMedia v7 [Tizzei et al. 2011], and NotePad [Soares et al. 2015], developed in Java. The columns present the SPL purpose, the Number of Lines of Code (LOC), the Number of Methods (# of Methods), and the Number of Classes (# of Classes), respectively. **Step 2** aimed at building reference lists of critical components per SPL. Two paper authors contributed for this task, by relying on definitions of critical component provided by the developers of each collected SPL, via online report, and their experience with component-based SPL.

Table 1. General Data about each Collected Component-based SPL

SPL	Purpose	LOC	# of Methods	# of Classes
BET-SPL	Managing electronic transport cards	13,318	1,576	177
MobileMedia	Managing media in mobile devices	14,250	1,333	332
NotePad	Deriving text editors	2,626	128	15

Step 3 aims at selecting coupling metrics from well-know metric catalogs [Chidamber and Kemerer 1994, Lorenz and Kidd 1994, McCabe 1976]. We intend to propose a simple strategy, which combine conventional metrics, since complex strategies could be difficult for developers to adopt in different contexts. Table 2 describes the five selected metrics: *Coupling between Objects* (CBO) for general dependencies, *Depth in Inheritance Tree* (DIT) and *Number of Children* (NOC) for hierarchical dependencies, and *Fan-in* (FANIN) and *Fan-out* (FANOUT) for incoming and outgoing dependencies.

Table 2. Software Metrics to Characterize Critical Components

Software Metric	Description
CBO	Number of components coupled to one specific component
DIT	Maximum depth of a component in the inheritance tree
FANIN	Number of dependencies from other components to a specific component
FANOUT	Number of dependencies from one components to others
NOC	Number of children components of a component

Step 4 aimed at computing each metric per SPL using a well-known tool called Understand[2], which considers inner classes and methods as different code elements with respect to the main classes and methods. We analyzed the metric distribution to select a threshold derivation method that best fits our metric distributions. Figure 2 suggest that at least CBO, FANIN, and FANOUT follow a heavy-tailed distribution. However, a previous work observes that software metrics mostly follow such distribution [Filó et al. 2015]. In the context of our study, the higher the metric value, the more critical is the measure.

Figure 3 presents our metric-based strategy for characterizing critical components. A detection strategy combines multiple metrics to characterize components that satisfy a property [Marinescu 2004], the component criticality in our case. We assume that both FANIN and CBO explicitly capture coupling. However, FANOUT has to be combined with DIT and NOC, since the two last metrics are potentially critical when the component also depends on other components (FANOUT). Details in our research website[3].

[2]https://scitools.com/features/

[3]https://anderson-uchoa.github.io/coupling-metrics-spl-critical-components/

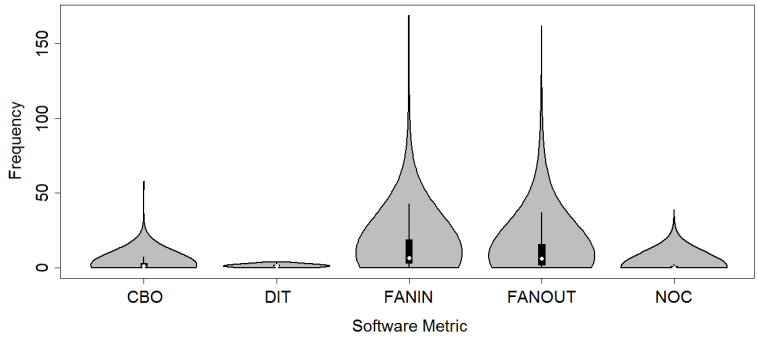

Figure 2. Distribution of Frequency per Software Metric

To derive thresholds that characterize critical metric values (see discussion of Figure 2), we used a quantile-based method from the literature [Vale and Figueiredo 2015]. It defines five value intervals, including high (values $>$ 90%) and very high (values $>$ 95%). We call *Critical* the interval [90%, 95%], which has the critical SPL component, and *Strongly Critical* the interval [95%, 100%], which has the most critical components.

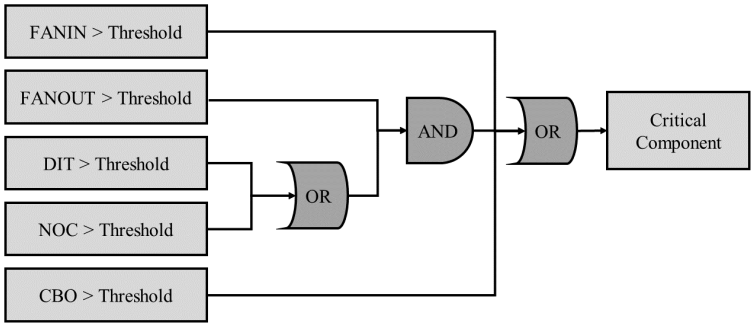

Figure 3. Metric-based Strategy for Characterizing Critical Components

Table 3 presents the quantiles computed with support of the R tool[4]. We highlight that all three component-based SPLs of (Table 3.2) were used to compute these quantiles.

Table 3. Derived Thresholds per Software Metric

Metric Level	Software Metric	3%	15%	90%	95%
Method	FANOUT	0	1	34	46.85
	FANIN	0	2	30	40
Class	CBO	0	0	6	8
	NOC	0	0	1	1
	DIT	0	0	3	3

Step 5 aims at identifying the critical components per SPL. We run the metric-based strategy of Figure 3 in spreadsheets with the metrics computed for each component-based SPL. As a result, we obtained the list of critical components for each SPL, by considering both *Critical* and *Strongly Critical* intervals. Finally, **Step 6** aims at computing TP, FN, precision, recall, and F-measure for both intervals. The goal is to assess the accuracy of our strategy in identifying critical components.

[4]https://cran.r-project.org/

4. Results and Discussion

RQ1. *Do coupling metrics help characterize critical components in component-based software product lines?*

Table 4 presents the accuracy of our strategy in characterizing critical components, in terms of five metrics: TP, FN, precision, recall, and F-measure. The first column lists each SPL. The second to sixth columns present the five metrics, considering the *Critical* interval (from 90% to 95%). The last five columns present the same metrics, considering the *Strongly Critical* interval (from 95% to 100%). From Table 4, we can draw some interesting conclusions. First, we observe a high precision of the strategy in characterizing critical components for both *Critical* and *Strongly Critical* intervals. Thus, our results suggest that coupling metrics can effectively characterize at least a few actual critical components with high accuracy (i.e., RQ1 is positively answered).

Table 4. TP, FN, Precision, Recall and F-Measure per SPL

SPL	Critical (90%)					Strongly Critical (95%)				
	TP	**FN**	**P**	**R**	**F**	**TP**	**FN**	**P**	**R**	**F**
BET-SPL	26	76	81.2%	25.5%	38.8%	13	89	72.2%	12.7%	21.6%
MobileMedia	37	96	90.2%	27.8%	46.7%	20	113	95.2%	15%	25.9%
NotePad	6	2	85.7%	75%	80%	6	4	80%	50%	61.5%
All	**69**	**175**	**86.3%**	**28.3%**	**42.6%**	**37**	**216**	**84.1%**	**14.6%**	**24.9%**

> ***Finding 1.*** Our strategy has provided a high precision (up to 86.3%) when characterizing critical components in component-based SPLs. In other words, developers may apply our strategy for characterizing *Critical* components with a high precision rate. Similar observation is valid for *Strongly Critical* components.

In turn, we observe low recall (up to 28.3%) of identified SPL critical components. In fact, none of the five selected software metrics properly capture SPL component properties like data access, which may have affected recall. The high number of FN (up to 216 for all SPLs) reinforces this assumption, which suggests the need for using additional metrics that help capturing other SPL critical components and exploring properties of the code other than coupling. Moreover, the researchers that build the reference lists of SPL critical components reported that a critical component implements several SPL functionalities that are largely used in the SPL, e.g., components that realize features and manage data. Therefore, it suggests the need for proposing novel SPL-specific metrics that capture critical components in component-based SPLs, such as concern metrics [Sant'Anna et al. 2003]. Finally, we could explore possible SPL product configuration to characterize critical components, which may have caused high FN.

> ***Finding 2.*** Our strategy has provided low recall (up to 28.3%), which suggests the need for using additional metrics, proposing novel metrics for capturing properties like restricted data access, or exploring properties other than coupling to characterize critical components.

5. Threats to Validity

We discuss threats to the study validity [Wohlin et al. 2012] as follows.

Construct and Internal Validity. We carefully designed our study for replication. However, we used a limited set of metrics to characterize critical components. We minimize this threat by selecting metrics from well-known catalogs [McCabe 1976, Lorenz and Kidd 1994] in pairs via Understand. We discussed solutions when diverging to avoid biases. Regarding the limited set of SPLs, we collected them from GitHub with a carefully designed search, in pairs to assure the discard of SPLs that are not aligned to our study. With respect to our definition of critical component, there is an inherent threat to which we minimize by relying on the viewpoint of component-based SPL developers. Finally, we overlooked possible SPL product configuration to characterize critical components, which may have affected the strategy's accuracy with several false positives. We mitigate this threat by considering metrics that characterize SPL components with high dependency and tend to be mandatory in the product configuration.

Conclusion and External Validity. We carefully designed the data analysis protocol. We computed largely used accuracy metrics [Fawcett 2006], such as precision, recall, F-measure for the accuracy analysis of the critical components. We performed the data analysis in pairs to avoid missing data. We also re-conducted the analysis to prevent biases. Regarding the generalization of findings, we analyzed only three component-based SPLs. To mitigate this threat, we collected SPLs from different domains. Overall, we expect that our findings apply to other SPL development contexts than component-based SPL. However, further investigation is required.

6. Conclusion and Future Work

We present an empirical study aimed at characterizing critical components in component-based SPLs with conventional coupling metrics. We collected three Java SPLs. We also proposed a metric-based strategy for detecting critical components in component-based SPLs with five well-known coupling metrics of the literature. We then built a reference list of critical components per SPL based on the SPL developers report. Finally, we computed the strategy's accuracy. Our data suggest a reasonable strategy's accuracy and the need for using additional metrics to characterize critical components. As future work, we aim to: analyze industry-scale SPLs; assess other techniques for SPL product configuration; and assess whether mandatory components are more critical than others.

Acknowledgments. This is work is funded by CAPES/Procad (175956, 117875), CNPq (309884/2012-8, 483425/2013-3, 477943/2013-6, 465614/2014-0, 308380/2016-9), and FAPERJ (225207/2016, 102166/2013).

References

Atkinson, C., Bayer, J., and Muthig, D. (2000). Component-based product line development. In *1st SPLC*, pages 289–309.

Bouwers, E., van Deursen, A., and Visser, J. (2014). Quantifying the encapsulation of implemented software architectures. In *30th ICSME*, pages 211–220.

Chidamber, S. and Kemerer, C. (1994). A metrics suite for object oriented design. *IEEE Trans. Softw. Eng. (TSE)*, 20(6):476–493.

Crnkovic, I. and Larsson, M. (2002). *Building reliable component-based software systems*. Artech House.

Donegan, P. and Masiero, P. (2007). Design issues in a component-based software product line. In *1st SBCARS*, pages 3–16.

Fawcett, T. (2006). An introduction to roc analysis. *Pattern Recogn. Lett.*, 27(8):861–874.

Filó, T., Bigonha, M., and Ferreira, K. (2015). A catalogue of thresholds for object-oriented software metrics. *1st SOFTENG*, pages 48–55.

Gill, N. (2006). Importance of software component characterization for better software reusability. *ACM SIGSOFT Softw. Eng. Notes*, 31(1):1–3.

Her, J., Kim, J., Oh, S., Rhew, S., and Kim, S. (2007). A framework for evaluating reusability of core asset in product line engineering. *Inform. Softw. Tech. (IST)*, 49(7):740–760.

Kessel, M. and Atkinson, C. (2015). Ranking software components for pragmatic reuse. In *6th WETSoM*, pages 63–66.

Krueger, C. (2006). New methods in software product line practice. *Comm. of the ACM*, 49(12):37–40.

Lorenz, M. and Kidd, J. (1994). *Object-oriented software metrics*. Prentice-Hall.

Marinescu, R. (2004). Detection strategies. In *20th ICSM*, pages 350–359.

McCabe, T. (1976). A complexity measure. *IEEE Trans. Softw. Eng. (TSE)*, (4):308–320.

Pohl, K., Böckle, G., and van Der Linden, F. (2005). *Software product line engineering*. Springer.

Sant'Anna, C., Garcia, A., Chavez, C., Lucena, C., and Von Staa, A. (2003). On the reuse and maintenance of aspect-oriented software. In *17th SBES*, pages 19–34.

Sharp, D. (1998). Reducing avionics software cost through component based product line development. In *17th DASC*, pages G32:1–G32:8.

Soares, L., Machado, I., and Almeida, E. (2015). Non-functional properties in software product lines. In *9th VaMoS*, page 67.

Tischer, C., Muller, A., Ketterer, M., and Geyer, L. (2007). Why does it take that long? In *11th SPLC*, pages 269–274.

Tizzei, L., Dias, M., Rubira, C., Garcia, A., and Lee, J. (2011). Components meet aspects. *Inform. Softw. Tech. (IST)*, 53(2):121–136.

Vale, G. and Figueiredo, E. (2015). A method to derive metric thresholds for software product lines. In *29th SBES*, pages 110–119.

Vale, T., Crnkovic, I., Almeida, E., Silveira, P., Cavalcanti, Y., and Meira, S. (2016). Twenty-eight years of component-based software engineering. *J. Syst. Softw. (JSS)*, 111:128–148.

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M., Regnell, B., and Wesslén, A. (2012). *Experimentation in software engineering*. Springer.

Yacoub, S., Cukic, B., and Ammar, H. (1999). Scenario-based reliability analysis of component-based software. In *10th ISSRE*, pages 22–31.