A General Framework to Detect Behavioral Design Patterns

Cong Liu[1], Boudewijn van Dongen[1], Nour Assy[1], Wil M.P van der Aalst[2,1]
[1]Eindhoven University of Technology, 5600MB Eindhoven, The Netherlands.
[2]RWTH Aachen University, 52056 Aachen, Germany.
{c.liu.3,b.f.v.dongen,n.assy}@tue.nl,wvdaalst@pads.rwth-aachen.de

ABSTRACT

This paper presents a general framework to detect behavioral design patterns by combining source code and execution data. The framework has been instantiated for the observer, state and strategy patterns to demonstrate its applicability. By experimental evaluation, we show that our combined approach can guarantee a higher precision and recall than purely static approaches. In addition, our approach can discover all missing roles and return complete pattern instances that cannot be supported by existing approaches.

KEYWORDS

General Framework, Behavioral Design Pattern, Discovery and Detection, Pattern Instance Invocation

ACM Reference Format:
Cong Liu[1], Boudewijn van Dongen[1], Nour Assy[1], Wil M.P van der Aalst[2,1]. 2018. A General Framework to Detect Behavioral Design Patterns. In *ICSE '18 Companion: 40th International Conference on Software Engineering Companion, May 27-June 3, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3183440.3194947

1 INTRODUCTION

The detection of implemented design pattern instances during reverse engineering can be useful for a better understanding of the design and architecture of the underlying software. As a result, the detection of design patterns is a challenging problem that has received a lot of attention in the software engineering domain [1–3, 5]. In general, design pattern detection techniques can be categorized into (1) static techniques, (2) dynamic techniques and (3) the combination of both. *Static techniques* (e.g., [2]) take the source code as input and consider structural connections among classes. *Dynamic techniques* (e.g., [1]) take software execution data as input and consider sequences of the method calls and interactions of the objects. *Combined techniques* (e.g., [3]) take the candidate design pattern instances (detected by static analysis tools) and software execution data as input, and check whether the candidates conform to the behavioral constraints.

Compared with single static or dynamic analysis techniques, the combined techniques have the following advantages: (1) the false positives caused by static techniques can be eliminated because the candidates are examined with respect to the behavioral constraints; and (2) the search space explosion problem for large-scale software systems with big execution data can be reduced because the static techniques help to narrow down the search scope of execution data to those only related to the detected candidates. However, existing combined techniques have the following problems that may limit their accuracy and applicability: (1) incompleteness of detected pattern instances; (2) unclear definition of pattern instance invocation; and (3) inability to support novel design patterns.

This paper proposes a general framework to support the detection of behavioral design patterns by addressing the problems that existing work cannot handle. Our framework definition is generic and can be instantiated to support new patterns (or new pattern variants). To validate the proposed approach, we have developed a tool, which currently supports the observer, state and strategy patterns.

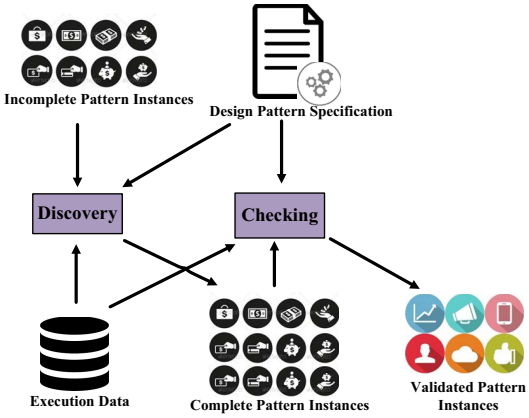

Figure 1: General Overview of the Detection Framework

2 A GENERAL FRAMEWORK TO DETECT BEHAVIORAL DESIGN PATTERNS

Let $\mathscr{U}_M$ be the method call universe, $\mathscr{U}_N$ be the method universe, $\mathscr{U}_C$ be the class universe, and $\mathscr{U}_R$ be the role universe. $\mathcal{P}(S) = \{S' | S' \subseteq S\}$ denotes the powerset of S. $f : X \rightarrow Y$ is a total function, i.e., $dom(f) = X$ is the domain and $rng(f) = \{f(x) | x \in dom(f)\} \subseteq Y$ is the range. $g : X \nrightarrow Y$ is a partial function, i.e., $dom(g) \subseteq X$ is the domain and $rng(g) = \{g(x) | x \in dom(g)\} \subseteq Y$ is the range.

Fig. 1 shows an overview of the framework to detect behavioral design patterns by combining both static and dynamic approaches. It is explained as follows:

2.1 Design Pattern Specification. A design pattern specification includes the role set that is involved in the design pattern, the definition of pattern instance, a set of structural constraints, the definition of pattern instance invocation, and a set of behavioral constraints.

DEFINITION 1. ***(Design Pattern Specification)*** *A design pattern is defined as* $DP = (U_R^P, rs, sc, pii, bc)$ *such that: (1)* $U_R^P \subseteq \mathscr{U}_R$

ICSE '18 Companion, May 27-June 3, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5663-3/18/05.
https://doi.org/10.1145/3183440.3194947

is the role set of a design pattern; (2) $rs : U_R^P \rightarrow \mathcal{U}_N \cup \mathcal{U}_C$ *is a mapping from the roles to their values (methods or classes); (3)* $sc : (U_R^P \rightarrow \mathcal{U}_N \cup \mathcal{U}_C) \rightarrow \mathbb{B}$ *with* $\mathbb{B} = \{true, false\}$ *is used to check the structural constraints of a pattern instance; (4)* $pii : \mathcal{P}(\mathcal{U}_M) \rightarrow \mathcal{P}(\mathcal{P}(\mathcal{U}_M))$ *is a function that identifies a set of pattern instance invocations from the method call set of a pattern instance; and (5)* $bc : \mathcal{P}(\mathcal{P}(\mathcal{U}_M)) \rightarrow \mathbb{B}$ *is used to check the behavioral constraints of all invocations of a pattern instance.*

2.2 (Incomplete) Candidate Pattern Instances. Static analysis approaches take as input the source code and return a set of candidate pattern instances by considering the information extracted from source code. However, these approaches may result in a high false positive rate because of their inability to use the actual runtime information. According to Definition 1, a pattern instance is represented as a tuple of the participants (classes and methods) each representing a particular role. A statically detected candidate pattern instance may be incomplete. Formally, $crs : U_R^P \nrightarrow \mathcal{U}_N \cup \mathcal{U}_C$ is a partial function that maps a sub-set of roles to its corresponding values.

2.3 Complete Pattern Instance Discovery. As candidate pattern instances detected by existing static tools may be incomplete, some roles that are needed to do the behavioral constraint checking are missing. Therefore, we need to discover the complete description for those incomplete candidate pattern instances. Formally, $cc : (U_R^P \nrightarrow \mathrm{U}_N \cup \mathcal{U}_C) \rightarrow \mathcal{P}(U_R^P \rightarrow \mathcal{U}_N \cup \mathcal{U}_C)$ is a function that maps an incomplete pattern instance to a set of complete pattern instances. For any $crs \in U_R^P \nrightarrow \mathcal{U}_N \cup \mathcal{U}_C$, $rs \in cc(crs)$, we have $sc(rs) = true$, i.e., the structural constraints hold for all discovered complete pattern instances.

2.4 Behavioral Constraint Checking. The discovered complete pattern instances inevitably contain false positives. For each complete candidate pattern instance *rs* and its execution data SD ($SD \subseteq \mathcal{U}_M$) [4–6], we check whether the behavioral constraints given in the specification are satisfied with respect to all invocations of a pattern instance, i.e., $bc(pii(SD)) = true$.

2.5 Framework Instantiation. To execute the framework, design pattern specification, (incomplete) candidate pattern instances detected from source code, and software execution data that cover the behavior of all candidates are required. To test the applicability, we instantiate the proposed methodology by the observer, state and strategy patterns.

3 EXPERIMENTAL RESULTS

In this section, we report the results obtained by our approach on six software cases that each implements one or more typical design patterns. To evaluate the accuracy of our approach, we computed the precision and recall metrics. The metrics are computed at two different levels, i.e., the *instance* and *role* levels. The *instance-level precision* measures the percentage of the detected pattern instances that are correct pattern instances while *instance-level recall* measures the percentage of correct pattern instances that have been correctly detected. On the other hand, the *role-level precision* measures the percentage of the retrieved pattern instance roles that are correct roles, while the *role-level recall* measures the percentage of the correct pattern instance roles that have been correctly retrieved.

The instance-level and the role-level measures are reported in Tables 1 and 2. Based on Table 1, we conclude that the proposed approach can reduce all false positive instances caused by static tools without losing any true positive instances. Moreover, in Table 2, we can clearly see that our approach can discover all the missing roles that cannot be detected with static analysis.

Table 1: Comparison by Instance-level Precision and Recall

	Observer Pattern		State Pattern		Strategy Pattern	
	Prec.	Rec.	Prec.	Rec.	Prec.	Rec.
DPD	60%	100%	66.6%	100%	66.6%	100%
DPDC	100%	100%	100%	100%	100%	100%

Table 2: Comparison by Role-level Precision and Recall

	Observer Pattern		State Pattern		Strategy Pattern	
	Prec.	Rec.	Prec.	Rec.	Prec.	Rec.
DPD	60%	50%	66.6%	60%	66.6%	60%
DPDC	100%	100%	100%	100%	100%	100%

4 DISCUSSION

The precision and recall of the detection results heavily rely on the accuracy and completeness of the design pattern specification. On the one hand, if the pattern specification is over-defined (e.g., some unnecessary constraints are included), this will cause low recall as some true positives may be missing. On the other hand, if the pattern specification is under-defined (e.g., some essential constraints are not included), this will cause low precision as some false positives may be incorrectly detected.

5 CONCLUSION

This paper proposes a general framework to detect behavioral design patterns by combining static and dynamic techniques. To test the applicability, the framework was instantiated for three typical behavioral design patterns. The proposed approaches have been implemented as part of the open source process mining toolkit ProM.

In the future, the detection of other behavioral patterns (e.g., command pattern) will be included in our framework. In addition, we are working on analyzing large-scale software systems to evaluate the scalability and the extensibility of the approach.

REFERENCES

[1] Francesca Arcelli, Fabrizio Perin, Claudia Raibulet, and Stefano Ravani. 2010. Design Pattern Detection in Java Systems: A Dynamic Analysis Based Approach. *Evaluation of Novel Approaches to Software Engineering* (2010), 163–179.

[2] Mario Luca Bernardi, Marta Cimitile, and Giuseppe Di Lucca. 2014. Design pattern detection using a DSL-driven graph matching approach. *Journal of Software: Evolution and Process* 26, 12 (2014), 1233–1266.

[3] Andrea De Lucia, Vincenzo Deufemia, Carmine Gravino, and Michele Risi. 2009. Behavioral pattern identification through visual language parsing and code instrumentation. In *13th European Conference on Software Maintenance and Reengineering, CSMR'09*. IEEE, 99–108.

[4] Cong Liu, Boudewijn van Dongen, Nour Assy, and Wil van der Aalst. 2016. Component Behavior Discovery from Software Execution Data. In *International Conference on Computational Intelligence and Data Mining*. IEEE, 1–8.

[5] Cong Liu, Boudewijn van Dongen, Nour Assy, and Wil van der Aalst. 2018. A Framework to Support Behavioral Design Pattern Detection from Software Execution Data. In *13th International Conference on Evaluation of Novel Approaches to Software Engineering*. 1–12.

[6] Cong Liu, Boudewijn van Dongen, Nour Assy, and Wil van der Aalst. 2018. Software Architectural Model Discovery from Execution Data. In *13th International Conference on Evaluation of Novel Approaches to Software Engineering*. 1–8.