A Framework to Support Behavioral Design Pattern Detection from Software Execution Data

Cong Liu[1], Boudewijn van Dongen[1], Nour Assy[1] and Wil M. P. van der Aalst[2,1]

[1]*Department of Mathematics and Computer Science, Eindhoven University of Technology, 5600MB Eindhoven, The Netherlands*

[2]*Department of Computer Science, RWTH Aachen University, 52056 Aachen, Germany*

Keywords: Pattern Instance Detection, Behavioral Design Pattern, Software Execution Data, General Framework.

Abstract: The detection of design patterns provides useful insights to help understanding not only the code but also the design and architecture of the underlying software system. Most existing design pattern detection approaches and tools rely on source code as input. However, if the source code is not available (e.g., in case of legacy software systems) these approaches are not applicable anymore. During the execution of software, tremendous amounts of data can be recorded. This provides rich information on the runtime behavior analysis of software. This paper presents a general framework to detect behavioral design patterns by analyzing sequences of the method calls and interactions of the objects that are collected in software execution data. To demonstrate the applicability, the framework is instantiated for three well-known behavioral design patterns, i.e., observer, state and strategy patterns. Using the open-source process mining toolkit ProM, we have developed a tool that supports the whole detection process. We applied and validated the framework using software execution data containing around 1000.000 method calls generated from both synthetic and open-source software systems.

1 INTRODUCTION

As a common design practice, design patterns have been widely applied in the development of many software systems. The use of design patterns leads to the construction of well-structured, maintainable and reusable software systems (Tsantalis et al., 2006). Generally speaking, design patterns are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context (Gamma, 1995). 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 system. As a result, the detection of design patterns is a challenging problem that has received a lot of attention in software engineering domain (Arcelli et al., 2009), (Arcelli et al., 2010), (Bernardi et al., 2015), (Bernardi et al., 2014), (Dabain et al., 2015), (Dong et al., 2009), (Fontana and Zanoni, 2011), (Niere et al., 2002), (Shi and Olsson, 2006).

A design pattern can be seen as a tuple of software elements, such as classes and methods, conforming to a set of constraints. Constraints can be described from both structural and behavioral aspects. The former defines classes and their inter-relationships while the latter specifies how classes and objects interact. Many techniques have been proposed to detect design pattern instances. Table 1 summarizes some typical design pattern detection approaches for object-oriented software by considering the type of analysis (i.e., static, dynamic and combination of both), artifacts that are required as inputs, the description of pattern instances (i.e., what roles are used to describe the pattern), tool support, and the extensibility.

Based on the analysis type, these techniques can be categorized as static analysis techniques, combination analysis techniques and dynamic analysis techniques. *Static analysis techniques* (e.g., (Bernardi et al., 2015), (Bernardi et al., 2014), (Dabain et al., 2015), (De Lucia et al., 2009b), (Dong et al., 2009), (Fontana and Zanoni, 2011), (Niere et al., 2002), (Shi and Olsson, 2006), (Tsantalis et al., 2006)) take the source code as input and only consider the structural connections among classes of the patterns. Therefore, the detected pattern instances based on static analysis only satisfy structural constraints. With the abundance of static analysis tools on the one hand, and the growing availability of software execution data on the other hand, combination analysis techniques come into reach. *Combination analysis techniques* (e.g., (De Lucia et al., 2009a), (Heuzeroth et al., 2003), (Ng

Liu, C., van Dongen, B., Assy, N. and M. P. van der Aalst, W.
A Framework to Support Behavioral Design Pattern Detection from Software Execution Data.
DOI: 10.5220/0006688000650076
In *Proceedings of the 13th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2018)*, pages 65-76
ISBN: 978-989-758-300-1

Table 1: Summary of Existing Design Pattern Detection Approaches.

Reference	Analysis Type	Required Artifacts	Pattern Instance	Tool	Extensibility
(Niere et al., 2002)	static analysis	source code	class roles	✓	✗
(Shi and Olsson, 2006)	static analysis	source code	class & method roles	✓	✓
(Tsantalis et al., 2006)	static analysis	source code	class roles	✓	✗
(Dong et al., 2009)	static analysis	source code	class roles	✓	✗
(De Lucia et al., 2009b)	static analysis	source code	class roles	✓	✗
(Fontana and Zanoni, 2011)	static analysis	source code	class roles	✓	✗
(Bernardi et al., 2014)	static analysis	source code	class roles	✓	✗
(Dabain et al., 2015)	static analysis	source code	class roles	✓	✗
(Bernardi et al., 2015)	static analysis	source code	class roles	✓	✗
(Heuzeroth et al., 2003)	combination analysis	source code & execution data	class & method roles	✗	✗
(Wendehals and Orso, 2006)	combination analysis	source code & execution data	class & method roles	✗	✗
(Von Detten et al., 2010)	combination analysis	source code & execution data	class roles	✓	✗
(Ng et al., 2010)	combination analysis	source code & execution data	class roles	✗	✗
(De Lucia et al., 2009a)	combination analysis	source code & execution data	class roles	✓	✗
(Arcelli et al., 2009), (Arcelli et al., 2010)	dynamic analysis	execution data	unclear	✗	✗

et al., 2010), (Von Detten et al., 2010), (Wendehals and Orso, 2006)) take as input the candidate design pattern instances that are detected by static analysis tools and software execution data, and check whether the detected candidate pattern instances conform to the behavioral constraints. Therefore, the identified pattern instances using combination techniques match both structural and behavioral constraints. Compared with static analysis techniques, the combination techniques can eliminate some of the false positives caused by static techniques because the candidate pattern instances are examined with respect to the behavioral constraints.

Both the static analysis and combination analysis techniques require source code as input. However, if the source code is not available (e.g., in case of legacy software systems) these approaches are not applicable anymore. *Dynamic analysis techniques* can detect design pattern instances directly from the software execution data by considering sequences of the method calls and interactions of the objects that are involved in the patterns. However, existing researches into dynamic analysis techniques suffer from the following problems that restrict the application:

- **Unclear Description of Detected Pattern Instances.** A *design pattern instance* is represented as a tuple of the participating classes or methods each acting a certain role. Existing dynamic techniques (Arcelli et al., 2009), (Arcelli et al., 2010) do not clearly define pattern instances. This may influence the precision of behavioral constraint checking as some (class or method) roles that need to be verified are not specified. In addition, a lot of manual work is required to understand the detected pattern instances if they are not described properly (Pettersson et al., 2010).
- **Missing Explicit Definition of Pattern Instance Invocation.** To check the behavioral constraints of a candidate pattern instance, execution data that characterize the behavior of the candidate are needed. Normally, the behavioral constraints are defined based on the notion of *pattern instance invocation* which represents one independent execution of the underlying pattern instance. One needs to check the behavioral constraints against each pattern instance invocation. Existing dynamic techniques do not define clearly a pattern instance invocation, which causes inaccurate behavioral constraint checking.
- **Inability to Support Novel Design Patterns.** An approach is extensible if it can be easily adapted to some novel design patterns rather than only supporting a sub-group of existing patterns. Existing dynamic analysis approaches do not provide effective solutions to support new emerging design patterns with novel structural and behavioral characteristics. This limits the applicability and extensibility of existing approaches in the large.
- **Lacked Tool Support.** The usability of an approach heavily relies on its tool availability. Existing dynamic analysis approaches do not provide usable tools. This unavailability prohibits other researchers to reproduce the experiment and compare their new approaches.

In this paper, a general framework is proposed to support the detection of design pattern instances from execution data with full consideration of the limitations of existing work. We clearly define a pattern instance as a set of roles that each is acted by a class or a method. In addition, to ensure accurate behavioral constraint checking for execution data that involve multiple pattern instance invocations, we precisely define the notion of pattern instance invocation and propose a refactoring of the execution data (traces) in such a way that each refactored trace represents a pattern instance invocation. 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, named *DEsign PAttern Detection from execution data* (*DePaD*), which supports the whole detection process for observer, state and strategy design patterns.

The rest of this paper is organized as follows. Section 2 defines some preliminaries. Section 3 formalizes the general framework. Section 4 details each step of the instantiation of the framework by taking the observer pattern as a running example. Section 5 introduces the tool support. Section 6 conducts experimental evaluation. Section 7 discusses some threats to the validity of our approach. Finally, Section 8 concludes the paper and presents further directions.

2 PRELIMINARIES

Given a set S, $|S|$ denotes the number of elements in S. We use $\cup$, $\cap$ and $\setminus$ for the union, intersection and difference of two sets. $\emptyset$ denotes the empty set. The powerset of S is denoted by $\mathcal{P}(S) = \{S' | S' \subseteq S\}$. $f : X \rightarrow Y$ 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. A sequence over S of length n is a function $\sigma : \{1, 2, ..., n\} \rightarrow S$. If $\sigma(1) = a_1, \sigma(2) = a_2, ...\sigma(n) = a_n$, we write $\sigma = \langle a_1, a_2, ...a_n \rangle$. $|\sigma| = n$ represents the length of sequence σ is n. The set of all finite sequences over set S is denoted by S^*. Let $\sigma \in S^*$ be a sequence, $\sigma(i)$ represents the ith element of σ where $1 \leq i \leq |\sigma|$. Given a sequence σ and an element e, we have $e \in \sigma$ if $\exists i : 1 \leq i \leq |\sigma| \wedge \sigma(i) = e$.

To be able to refer to the different entities, we introduce the following universes. Let $\mathcal{U}_M$ be the method call universe, $\mathcal{U}_N$ be the method universe, $\mathcal{U}_C$ be the universe of classes and interfaces, $\mathcal{U}_O$ be the object universe where objects are instances of classes, $\mathcal{U}_R$ be the role universe and $\mathcal{U}_T$ be the time universe. To relate these universes, we use the following notations: For any $m \in \mathcal{U}_M$, $\widehat{m} \in \mathcal{U}_N$ is the method of which m is an instance. For any $o \in \mathcal{U}_O$, $\widehat{o} \in \mathcal{U}_C$ is the class of o. $pc : \mathcal{U}_N \rightarrow \mathcal{P}(\mathcal{U}_C)$ defines a mapping from a method to its parameter class set. $ms : \mathcal{U}_C \rightarrow \mathcal{P}(\mathcal{U}_N)$ defines a mapping from a class to its method set. $iv : \mathcal{U}_N \rightarrow \mathcal{P}(\mathcal{U}_N)$ defines a mapping from a method to its invoked method set.

In this paper, we mainly reason based on the software execution data. A method call is the basic unit of software execution data (Liu et al., 2016), (Leemans and Liu, 2017) and its attributes can be defined as follows:

Definition 1 (Method Call, Attribute)**.** *For any $m \in \mathcal{U}_M$, the following standard attributes are defined:*

- *$\eta : \mathcal{U}_M \rightarrow \mathcal{U}_O$ is a mapping from method calls to objects such that for each method call $m \in \mathcal{U}_M$, $\eta(m)$ is the object containing the instance of the method $\widehat{m}$.*
- *$c : \mathcal{U}_M \rightarrow \mathcal{U}_M \cup \{\perp\}$ is the calling relation among method calls. For any $m_i, m_j \in \mathcal{U}_M$, $c(m_i) = m_j$ means that m_i is called by m_j, and we name m_i as the callee and m_j as the caller. Specially, for $m \in \mathcal{U}_M$, if $c(m) = \perp$, then $\widehat{m}$ is a main method.*
- *$p : \mathcal{U}_M \rightarrow \mathcal{U}_O^*$ is a mapping from method calls to their (input) parameter object sequences such that for each method call $m \in \mathcal{U}_M$, $p(m)$ is a sequence of (input) parameter objects of the instance of the method $\widehat{m}$.*
- *$t_s : \mathcal{U}_M \rightarrow \mathcal{U}_T$ is a mapping from method calls to their timestamps such that for each method call $m \in \mathcal{U}_M$, $t_s(m)$ is start timestamp of the instance of method $\widehat{m}$.*
- *$t_e : \mathcal{U}_M \rightarrow \mathcal{U}_T$ is a mapping from method calls to their timestamps such that for each method call $m \in \mathcal{U}_M$, $t_e(m)$ is end timestamp of the instance of method $\widehat{m}$.*

Definition 2 (Software Execution Data)**.** *Software execution data is defined as a set of method calls, i.e., $SD \subseteq \mathcal{U}_M$.*

3 A GENERAL FRAMEWORK TO DETECT DESIGN PATTERNS FROM EXECUTION DATA

Fig. 1 shows an overview of the framework to detect behavioral design patterns. It involves two main phases, i.e., candidate pattern instance discovery and behavioral constraint checking. Software execution data and design pattern specification are required as the input artifacts. The design pattern specification and the two phases are detailed in the following.

3.1 Design Pattern Specification

A design pattern specification precisely defines how a design pattern should be organized, which includes the role set that is involved in the design pattern, the

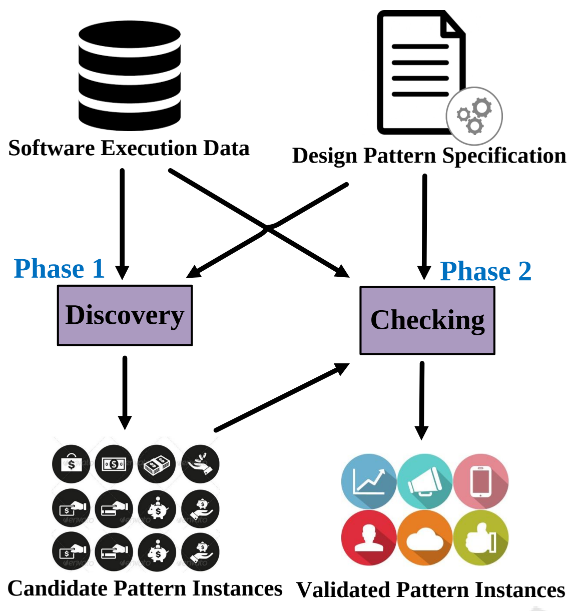

Software Execution Data
Design Pattern Specification
Phase 1
Phase 2
Discovery
Checking
Candidate Pattern Instances
Validated Pattern Instances

Figure 1: General Overview of the Framework.

definition of pattern instance, a set of structural constraints, the definition of pattern instance invocation, and a set of behavioral constraints.

Definition 3 (Design Pattern Specification)**.** *A design pattern is defined as* $DP = (U_R^P, rs, sc, pii, bc)$ *such that:*

- ***Role Set.*** $U_R^P \subseteq \mathcal{U}_R$ *is the role set of a design pattern.*
- ***Pattern Instance.*** $rs : U_R^P \to \mathcal{U}_N \cup \mathcal{U}_C$ *is a mapping from the roles to their values (methods or classes). Essentially, it is an implementation of the pattern.*
- ***Structural Constraints.*** $sc : (U_R^P \to \mathcal{U}_N \cup \mathcal{U}_C) \to \mathbb{B}$ *with* $\mathbb{B} = \{true, false\}$ *is used to check the structural constraints of a pattern instance.*
- ***Pattern Instance Invocation Identification.*** $pii : \mathcal{P}(\mathcal{U}_M) \to \mathcal{P}(\mathcal{P}(\mathcal{U}_M))$ *is a function that identifies a set of pattern instance invocations from the method call set of a pattern instance.*
- ***Behavioral Constraints.*** $bc : \mathcal{P}(\mathcal{P}(\mathcal{U}_M)) \to \mathbb{B}$ *is used to check the behavioral constraints of all invocations of a pattern instance.*

Note that the way invocations are identified is specific to each type of design pattern and independent of the number of runs included in the software execution data. In addition, for some structural patterns (e.g., adapter pattern, composite pattern), they do not necessary contain the definition of pattern instance invocation and behavioral constraints.

3.2 Phase 1: Candidate Pattern Instance Discovery

By taking the execution data as input, we need to discover a set of candidate pattern instances, i.e., finding the values (classes or methods) that play certain roles in the pattern instances. 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. In our case, we have $dom(crs) = \emptyset$, i.e., all roles do not have values and we need to brute force all possibilities according to the structural constraints. $dis : (U_R^P \nrightarrow \mathcal{U}_N \cup \mathcal{U}_C) \to \mathcal{P}(U_R^P \to \mathcal{U}_N \cup \mathcal{U}_C)$ is the discovery function that maps an empty pattern instance to a set of complete pattern instances.

For any $crs \in U_R^P \nrightarrow \mathcal{U}_N \cup \mathcal{U}_C, rs \in dis(crs)$, we have $sc(rs) = true$, i.e., the structural constraints hold for each discovered candidate pattern instance.

Existing static pattern instances discovery approaches focus on structural analysis of classes, operations and their inter-relationships (e.g., inheritance, realization, dependency) by exploring the source code. As for the discovery from execution data, class operations and some of the relationships, e.g., dependency and inheritance relationships, can be recovered. Realization relationship cannot be recovered as the interfaces cannot be instantiated and recorded during execution. Hence, some of the roles that are played by interfaces as detected from source code will be replaced by the implemented classes from the execution data.

3.3 Phase 2: Behavioral Constraint Checking

The pattern instance discovery takes as input the execution data and return a set of candidate pattern instances by considering only the structural constraints of the pattern, i.e., relationship among classes and methods. So the discovered candidate pattern instances inevitably contain false positives, especially for behavioral design patterns. For each candidate pattern instance *rs* and its execution data *SD*, we check whether the behavioral constraints given in the specification are satisfied with respect to all invocations of a pattern instance, i.e., $bc(pii(SD)) = true$.

3.4 Instantiation of the Framework

Given a novel design pattern X, the framework is required to support its detection. To execute the framework, design pattern specification of pattern X and software execution data that cover the behavior of such pattern are required. To test the applicability,

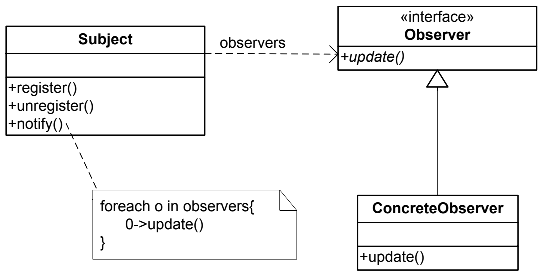


Figure 2: Structure of the Observer Design Pattern.

we instantiate the proposed methodology to discover the observer, state and strategy patterns. Detailed explanations of the applicability to the observer pattern is given in the next section with a running example. For the state and strategy patterns, their instantiation details are not repeated for page limits.

4 DETECTION OF OBSERVER DESIGN PATTERN

We first use the observer design pattern as a running example to illustrate the proposed framework. This is one particular instance of our general framework. Later we provide other examples.

4.1 Observer Design Pattern and A Running Example

The *observer design pattern* (Gamma, 1995) defines one-to-many dependency between objects so that when one object changes state, all its dependents are automatically notified. The structure of the observer design pattern is shown in Fig. 2. The key participants of this pattern are *Subject* and *Observer*. In this pattern, there are many Observer objects which are observing a particular Subject object. Observers are interested and want to be notified when the Subject undergoes a change. Therefore, they register themselves to that Subject. When an observer loses interest in the subject they simply unregister from it.

More specifically, a *Subject* is a class that (1) keeps track of a list of *Observer* references; (2) sends notifications to its registered observers on state changes; and (3) provides interfaces for registering and deregistering *Observer* objects. The *Observer* interface defines an update interface for objects that should be notified when the subject changes. When the state of *Subject* is changed, it invokes the *notify* method that implements a sequential call of the *update* method on all registered *Observer* objects.

The following naming conventions refer to roles that are involved in the observer pattern. Note that this naming convention is only used for explanations and our approach does not rely on these names.

- *Observer* is an interface that defines an updating interface for objects that should be notified of changes;
- *Subject* is a class that stores state of interest to *Observer* objects and sends notifications to its interested objects when its state changes;
- *notify* is a method that is responsible for notifying the observers of a state change in the *Subject*;
- *update* is a method implemented by the *Observer* objects and can be called by the *notify* method;
- *register* is a method responsible for adding *Observer* objects to a *Subject* object; and
- *unregister* is a method responsible for removing *Observer* objects from a *Subject* object.

A concrete implementation of the observer pattern is *AISWorld* which is an academic community software for researchers and practitioners. All its members subscribe to the news updates by registering to a public mailing server. When new events of the community occur, the mailing server will push these news items to all its subscribed members. Community members can also unsubscribe if they do not want to follow any more.

An excerpt of the execution data that are generated by an independent run of the *AISWorld* software are given in Table 2.[1] Its behavior can be described as: (1) three Member objects (Observer objects) are first created and registered to a MailingServer object (Subject object); (2) the notifyMember method of the MailingServer object is called and the update methods of the three registered Member objects are invoked during its execution; (3) these three Member objects are unregistered from the MailingServer object; (4) another two Member objects are created and registered to the second MailingServer object; (5) the notifyMember method of the second MailingServer object is called and the update methods of the two registered Member objects are invoked during its execution; and (6) these two Member objects are unregistered from the second MailingServer object.

[1]Note: methods are fully quantified including package, class and method names. *init* represents the constructor of a class.

Table 2: An Example of Software Execution Data

ID	(Callee) Method	(Callee) P O	(Callee) O	Caller Method	Caller O	Start Time	End Time
m_1	Member.init	–	1807970113	mainclass.main	–	709020268	709120368
m_2	Member.init	–	1807567788	mainclass.main	–	709244786	709267786
m_3	Member.init	–	1488142454	mainclass.main	–	709378641	709378641
m_4	MailingServer.init	–	1333401746	mainclass.main	–	717761066	717961966
m_5	MailingServer.register	1807970113	1333401746	mainclass.main	–	718086509	718086619
m_6	MailingServer.register	1807567788	1333401746	mainclass.main	–	718261847	718361447
m_7	MailingServer.register	1488142454	1333401746	mainclass.main	–	718420506	718588315
m_8	MailingServer.notifyMembers	–	1333401746	mainclass.main	–	718686715	719110738
m_9	Member.update	–	1807970113	MailingServer.notifyMembers	1333401746	718788715	718880715
m_{10}	Member.update	–	1807567788	MailingServer.notifyMembers	1333401746	718929841	718929941
m_{11}	Member.update	–	1488142454	MailingServer.notifyMembers	1333401746	719050867	719100467
m_{12}	MailingServer.unregister	1807970113	1333401746	mainclass.main	–	719270253	719370253
m_{13}	MailingServer.unregister	1807567788	1333401746	mainclass.main	–	719408812	719507712
m_{14}	MailingServer.unregister	1488142454	1333401746	mainclass.main	–	719570465	719880462
m_{15}	Member.init	–	1491288577	mainclass.main	–	719712873	719722873
m_{16}	Member.init	–	805469502	mainclass.main	–	719843307	719943908
m_{17}	MailingServer.init	–	1936493073	mainclass.main	–	720398401	720698461
m_{18}	MailingServer.register	1491288577	1936493073	mainclass.main	–	720719568	720919968
m_{19}	MailingServer.register	805469502	1936493073	mainclass.main	–	721077514	721276516
m_{20}	MailingServer.notifyMembers	–	1936493073	mainclass.main	–	721526122	721976868
m_{21}	Member.update	–	1491288577	MailingServer.notifyMembers	1936493073	721526122	721626122
m_{22}	Member.update	–	805469502	MailingServer.notifyMembers	1936493073	721825906	721875908
m_{23}	MailingServer.unregister	1491288577	1936493073	mainclass.main	–	722279646	722379646
m_{24}	MailingServer.unregister	805469502	1936493073	mainclass.main	–	722579858	722779768
m_{25}	TestMail.mainclass.main	5336152135	–	–	–	703720242	723873758

Note: O is short for object, P is short for parameter and – means the value is unavailable.

4.2 Candidate Observer Pattern Instances Discovery

In this subsection, we introduce how to discover candidate observer pattern instances from software execution data. Note that different levels of granularity can be used for the roles of a design pattern. For example, most existing works (e.g., (Bernardi et al., 2015), (Bernardi et al., 2014), (Dabain et al., 2015), (De Lucia et al., 2009a), (De Lucia et al., 2009b), (Dong et al., 2009), (Fontana and Zanoni, 2011), (Ng et al., 2010), (Niere et al., 2002) and (Tsantalis et al., 2006)) use a tuple of *Subject* and *Observer* as roles of observer pattern. The coarse-grained descriptions of pattern instances are not enough to preform precise behavioral constraint checking. This paper aims to give a complete description of design patterns (in terms of class and method roles) . The following definition specifies the observer pattern in detail.

Definition 4 (Role Set and Pattern Instance of Observer Pattern)**.** *Role set of the observer pattern is defined as* $U_R^O = U_{RC}^O \cup U_{RM}^O$ *where* $U_{RC}^O = \{Sub, Obs\}$ *and* $U_{RM}^O = \{not, upd, reg, unr\}$. $rs^o : U_R^O \to \mathscr{U}_C \cup \mathscr{U}_N$ *is a mapping from the role set of observer pattern to their values such that* $\forall r \in U_{RC}^O : rs^o(r) \in \mathscr{U}_C$ *and* $\forall r \in U_{RM}^O : rs^o(r) \in \mathscr{U}_N$.

An observer pattern instance is an implementation of the observer pattern and it defines a binding from the role set to its values. The discovery process aims to find the missing values for all roles involved in the role set of observer pattern with respect to the structural constraints. The following definition formalizes the structural constraints of the observer pattern.

Definition 5 (Structural Constraints of Observer Pattern)**.** *For each observer pattern instance* rs^o, *we have* $sc^o(rs^o) = true$ *iff:*

- $rs^o(not) \in ms(rs^o(Sub))$, *i.e., notify is a method of Subject; and*
- $rs^o(reg) \in ms(rs^o(Sub))$, *i.e., register is a method of Subject; and*
- $rs^o(unr) \in ms(rs^o(Sub))$, *i.e., unregister is a method of Subject; and*
- $rs^o(upd) \in ms(rs^o(Obs))$, *i.e., update is a method of Observer; and*
- $rs^o(Obs) \in pc(rs^o(reg))$, *i.e., register should contain a parameter of Observer type; and*
- $rs^o(Obs) \in pc(rs^o(unr))$, *i.e., unregister should contain a parameter of Observer type; and*
- $rs^o(Obs) \notin pc(rs^o(not))$, *i.e., notify should not contain a parameter of Observer type; and*
- $rs^o(upd) \in iv(rs^o(not))$, *i.e., update should be invoked by notify; and*

- *$rs^o(reg) \neq rs^o(unr)$, i.e., register and unregister can not be played by the same method.*

Based on the structural constraints, we propose an algorithm to discover candidate observer pattern instances from software execution data. Before outlining the algorithm, some important concepts and notations that are used in the reminder are introduced.

Given software execution data *SD*, we define:

- $cs(SD) = \{\widehat{\eta(m)} | m \in SD\}$ is the class set involved in the software execution data *SD*;
- For any $c \in \mathscr{U}_C$, $ms(c,SD) = \{\widehat{m} | \widehat{\eta(m)} = c\}$ is the method set of class *c* in execution data *SD*;
- For any $n \in \mathscr{U}_N$, $pc(n,SD) = \{\widehat{o} | \exists m \in SD : \widehat{m} = n \wedge o \in p(m)\}$ is the parameter class set of method *n* in the software execution data *SD*; and
- For any $n \in \mathscr{U}_N$, $iv(n,SD) = \{\widehat{m} | m \in SD \wedge \widehat{c(m)} = n\}$ is the invoked method set of method *n* in the software execution data *SD*.

For the discovery of observer pattern instances from execution data, we first identify a set of possible values for each role as intermediate results. This is defined as $rs^o_* : U^O_R \to \mathcal{P}(\mathscr{U}_C \cup \mathscr{U}_N)$. Then we define a function ω that generates a set of pattern instances by exploiting all possible combinations of different role values. Formally, we have $\omega : (U^O_R \to \mathcal{P}(\mathscr{U}_N \cup \mathscr{U}_C)) \to \mathcal{P}(U^O_R \to \mathscr{U}_N \cup \mathscr{U}_C)$. For any $rs^o_* \in U^O_R \to \mathcal{P}(\mathscr{U}_C \cup \mathscr{U}_N)$, (1) for any $rs^o \in \omega(rs^o_*)$: $dom(rs^o) = U^O_R$; (2) for any $r \in U^O_R : rs^o_*(r) = \bigcup_{rs^o \in \omega(rs^o_*)} rs^o(r)$; (3) $\nexists rs^o, rs^{o\prime} \in \omega(rs^o_*)$: $\forall r \in U^O_R$, $rs^o(r) = rs^{o\prime}$; and (4) $|\omega(rs^o_*)| = \prod_{r \in U^O_R} |rs^o_*(r)|$.

Assume that design pattern W involves with two type of roles (X and Y), we have $rs^w_*(X) = \{a,b\}$ and $rs^w_*(Y) = \{c,d\}$. Then we generate four pattern instances $\omega(rs^w_*) = \{\{rs^w_1(X) = \{a\}, rs^w_1(Y) = \{c\}\}, \{rs^w_2(X) = \{a\}, rs^w_2(Y) = \{d\}\}, \{rs^w_3(X) = \{b\}, rs^w_3(Y) = \{c\}\}, \{rs^w_4(X) = \{b\}, rs^w_4(Y) = \{d\}\}$. The pseudocode description of the observer pattern instance discovery is given in the following algorithm.

As the algorithm discovers a set of candidate observer pattern instances using the structural constraints and the software execution data, it inevitably produces some false positives. For example, by taking the software execution data in Table 2 as input, we obtain two candidate observer pattern instances using Algorithm 1, denoted as rs^o_1 and rs^o_2, as shown in Table 3. Note that the methods that play the roles of register and unregister are undistinguishable by only considering the structural constraints.

4.3 Behavioral Constraint Checking

This section introduces how to check whether a discovered candidate observer pattern instance conforms to the behavioral constraints. Because one or more invocations may be involved in the execution data, we need to identify independent observer pattern instance invocations from the execution data. Because not all method calls in the software execution data are relevant with the observer pattern instance that we are going to check, we first define execution data for an observer pattern instance.

Definition 6 (Execution Data of Observer Pattern Instance)**.** *Let rs^o be an observer pattern instance and SD be the execution data. $SD^O = \{m \in SD \mid \exists r \in U^O_R : rs^o(r) = \widehat{m}\}$ are the execution data of observer pattern instance rs^o.*

According to specification of observe design pattern, an observer pattern instance invocation starts with the creation of one *Subject* object and involves all method calls such that: (1) the method plays a role in the observer pattern instance; and (2) its object is the Subject object or its caller object is the *Subject* object and the object is an *Observer* object. In addition, by taking the same observer pattern instance execution data as input, the set of identified observer pattern instance invocations should be unique.

Definition 7 (Observer Pattern Instance Invocation)**.** *Let rs^o be an observer pattern instance and SD^O be its execution data. We define invocation set of rs^o as $pii^o(SD^O) = \{I_1, I_2, \ldots, I_n\} \subseteq \mathcal{P}(SD^O)$, such that:*

- *for any $I_i \in pii^o(SD^O)$, $m \in I_i$ where $1 \leq i \leq n$, there exists $o \in \mathscr{U}_O$ and $rs^o(Sub) = \widehat{o}$ such that:*
  - *$(\widehat{m} = rs^o(reg) \vee \widehat{m} = rs^o(unr) \vee \widehat{m} = rs^o(not)) \wedge (\eta(m) = o)$, i.e., the object of each method call is the Subject object and the method should play the role of register, unregister or notify; or*
  - *$\widehat{m} = rs^o(upd) \wedge \eta(c(m)) = o \wedge rs^o(Obs) = \widehat{\eta(m)}$, i.e., the caller object of each method call is the Subject object and the method should play the role of update.*
- *for any $I_i, I_j \in pii^o(SD^O)$, $m \in I_i$, $m' \in I_j$ where $1 \leq i < j \leq n$, there does not exist $o \in \mathscr{U}_O$ and $rs^o(Sub) = \widehat{o}$ such that $(\eta(m) = o \vee \eta(c(m)) = o) \wedge (\eta(m') = o \vee \eta(c(m')) = o)$.*

Considering the software execution data in Table 2, the execution data of candidate observer pattern instance rs^o_1 is $SD^O = \{m_i | 5 \leq i \leq 14 \wedge 18 \leq i \leq 24\}$. We have $pii^o(SD^O) = \{I^1_O, I^2_O\}$ with $I^O_1 = \{m_i | 5 \leq i \leq 14\}$ and $I^O_2 = \{m_i | \ 18 \leq i \leq 24\}$ be two invocations.

Algorithm 1: Candidate observer pattern instances discovery.

```
   Input: Software execution data $SD$.
   Output: Observer pattern instance set $RS$.
1  $RS \leftarrow \emptyset$, $ClassSet \leftarrow cs(SD)$. /**initialization.**/
2  for $c_i \in ClassSet$ do
3      /**subject class should at least contain three methods**/
4      if $|ms(c_i,SD)| \geq 3$ then
5          for $c_j \in ClassSet$ do
6              /**observer class should at least contain one method.**/
7              if $c_i \neq c_j$ & $|ms(c_j,SD)| \geq 1$ then
8                  /**create intermediate result $op$.**/
9                  $op(Sub) \leftarrow \{c_i\}$, $op(Obs) \leftarrow \{c_j\}$, $op(not) \leftarrow \emptyset$,
10                 $op(upd) \leftarrow \emptyset$, $op(reg) \leftarrow \emptyset$, $op(unr) \leftarrow \emptyset$;
11                 for $m_i \in ms(c_i,SD)$ do
12                     /**register and unregister are played by methods with a parameter of observer class.**/
13                     if $c_j \in pc(m_i,SD)$ then
14                         /**set values for register and unregister roles of $op$.**/
15                         $op(reg) \leftarrow op(reg) \cup \{m_i\}$;
16                         $op(unr) \leftarrow op(unr) \cup \{m_i\}$;
17                 if $|op(reg)| \geq 2$ then
18                     for $m_j \in ms(c_i,SD)$ do
19                         /**notify should not contain a parameter of observer class.**/
20                         if $m_j \notin op(reg)$ & $c_j \notin pc(m_j,SD)$ then
21                             for $m_k \in iv(m_j,SD)$ do
22                                 /** update is invoked by notify.**/
23                                 if $m_k \in ms(c_j,SD)$ then
24                                     $op(not) \leftarrow op(not) \cup \{m_j\}$;
25                                     $op(upd) \leftarrow op(upd) \cup \{m_k\}$;
26                 /** for each $op$, we generate all possible role to value combinations as candidate instances.**/
27                 for $rs \in \omega(op)$ do
28                     if $rs(reg) \neq rs(unr)$ then
29                         $RS \leftarrow RS \cup \{rs\}$;
30 return All detected observer pattern instances RS.
```

Table 3: Two Observer Candidate Pattern Instances.

	Sub	Obs	not	upd	reg	unr
rs_1^o	MS.	M.	notifyM	update	register	unregister
rs_2^o	MS.	M.	notifyM	update	unregister	register

MS. and M. are short for MailingServer and the Member.

After obtaining candidate pattern instances and refactoring execution data by invocation identification, we can check whether a candidate conforms to the behavioral constraints. To this end, we formally define the behavioral constraints of the observer pattern. The following notations and operators are defined on the basis of each invocation.

Given an observer pattern instance invocation $I \subseteq SD$, we define:

- For any $n \in \mathscr{U}_N$, $N(I,n) = \{m \in I | \widehat{m} = n\}$ is a set of method calls with n being its method in I;
- For any $M \subseteq I$, $CO(M) = \{o \in \mathscr{U}_O | \exists m \in M : \eta(m) = o\}$is the object set of M;
- For any $n \in \mathscr{U}_N$, $c \in \mathscr{U}_C$, $PS(I,n,c) = \{o \in \mathscr{U}_O | \exists m \in I : \widehat{m} = n \wedge o \in p(m) \wedge \widehat{o} = c\}$ is a set of (input) parameter objects of method calls with n being their method and these objects are of class type c in I;
- For any $m \in \mathscr{U}_M$, $I_v(I,m) = \{m' \in I | c(m') = m\}$ is the invoked method call set of method call m in I; and
- For any $m \in \mathscr{U}_M$, $Pre(I,m) = \{m' \in I | t_e(m') < t_s(m)\}$ is the set of method calls that are invoked before method call m in I.

Definition 8 (Behavioral Constraints of Observer Pattern)**.** *For each observer pattern instance rs^o, the behavioral constraints $bc^o(pii^o(SD^O)) = true$ iff there exists an invocation $I \in pii^o(SD^O)$ such that:*

- $|N(I,rs^o(not))| \geq 1 \wedge |N(I,rs^o(upd))| \geq 1 \wedge$ $|N(I,rs^o(reg))| \geq 1 \wedge |N(I,rs^o(unr))| \geq 1$, *i.e., for each observer pattern invocation, notify, update, register and unregister methods should be invoked at least once; and*
- $\forall_{o \in PS(I,rs^o(reg),rs^o(Obs)) \cup PS(I,rs^o(unr),rs^o(Obs))} (\exists_{m \in I} \widehat{m} =$

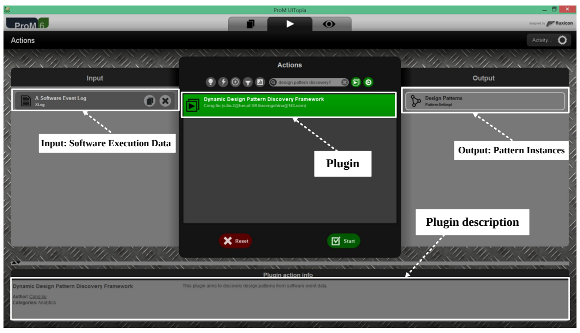
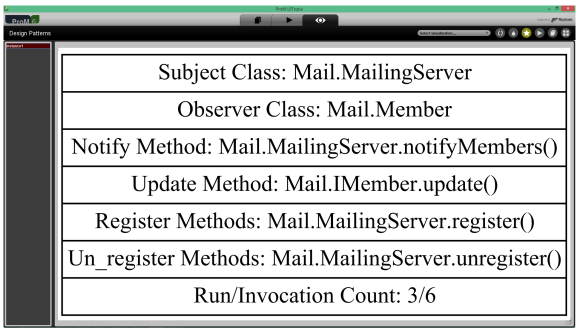


Figure 3: Screenshot of the *DePaD*.



Subject Class: Mail.MailingServer
Observer Class: Mail.Member
Notify Method: Mail.MailingServer.notifyMembers()
Update Method: Mail.IMember.update()
Register Methods: Mail.MailingServer.register()
Un_register Methods: Mail.MailingServer.unregister()
Run/Invocation Count: 3/6

Figure 4: A Detected Observer Pattern Instance.

$rs^o(reg) \wedge o \in p(m)) \wedge (\forall_{m\in I} \exists_{m' \in I} \widehat{m} = rs^o(reg) \wedge \widehat{m'} = rs^o(unr) \wedge o \in p(m) \wedge o \in p(m') \wedge t_e(m) < t_s(m'))$, *i.e., for each observer pattern invocation an observer object should be first registered to the Subject object and then unregistered from it; and*

- $\forall_{m\in I \wedge \widehat{m}=rs^o(not)} CO(N(I_v(I,m), rs^o(upd))) = (PS(Pre(I,m), rs^o(reg), rs^o(Obs)) \setminus PS(Pre(I,m), rs^o(unr), rs^o(Obs)))$, *i.e., a notify method should invoke the update methods of all Observer objects that are currently registered to the Subject object.*

A candidate observer pattern instance is valid if there exists at least one invocation that satisfies all behavioral constraints, otherwise, it is not valid. Considering the candidates in Table 3 and its invocations I_1^O and I_2^O, rs_1^o is a valid observer pattern instance but rs_2^o is not. This is because the second behavioral constraint is violated, i.e., some observer objects are only registered to the subject object but not unregistered, for both invocations when checking rs_2^o.

5 TOOL IMPLEMENTATION

The proposed approach to detect behavioral design pattern has been implemented as a plug-in, called *DEsign PAttern Discovery from execution data* (*DePaD*), in our *ProM 6* package.[2] It takes the software execution data as input, and returns a set of design pattern instances. Currently, this tool supports observer pattern, state pattern and strategy pattern. A snapshot of the tool is shown in Fig. 3.

Fig. 4 shows the observer pattern instance rs^{o1} by taking the execution data that are generated by three runs of the *AISWorld* software as input. Note that the *Run/Invocation Count* indicates the number of runs in the input software execution data and the number of invocations that support the current pattern instance.

[2]We will link to the code in the camera-ready version

All experimental results in the following section are based on this tool.

6 EMPIRICAL EVALUATION

In this section, we evaluate our approach using both synthetic and open-source software systems that overall produce around 1000.000 method calls in the execution data. For these experiments we used a laptop with a 2.40 GHz CPU, Windows 8.1 and Java SE 1.7.0 67 (64 bit) with 4 GB of allocated RAM.

6.1 Subject Software Systems and Execution Data

For our experiments, we use six synthetic software systems. Each system implements one or more design patterns. For each synthetic software system, we create its execution data by instrumenting different execution scenarios. The advantages of using synthetic software systems are that we have enough up-to-date knowledge to (1) collect execution data that cover all scenarios; and (2) evaluate the accuracy (in terms of precision and recall) of our proposed approach.

In addition, to show the applicability and scalability of our approach for real-life software systems, we use the execution data that were collected from three open-source software. Different from synthetic software that we have enough knowledge to guarantee that the execution data cover all software usage scenarios, we collected execution data from typical usage scenarios of these open-source software systems.

Table 4 shows the detailed statistics of execution data collected from these software systems, including the number of packages/classes/methods that are loaded during execution and the number of method calls analyzed. More specifically,

- Synthetic 1 is a calender software system implementing a state design pattern instance;

Table 4: Statistics of Subject Software Execution Data.

Software		#Pac.	#Cla.	#Meth.	#Meth. Call
Synthetic	Synthetic 1	1	5	9	155
	Synthetic 2	1	5	11	135
	Synthetic 3	1	5	15	160
	Synthetic 4	1	6	12	104
	Synthetic 5	1	8	14	258
	Synthetic 6	8	12	28	9728
Real-life	Lexi 0.1.1	5	68	263	20344
	JUnit 3.7	3	47	213	363948
	JHotDraw 5.1	7	1.8	549	583423

Note: The number of packages (#Pac.), the number of classes (#Cla.), the number of methods (#Meth.), and the number of method calls (#Meth. Call).

- Synthetic 2 is a testing software system implementing a state and a strategy pattern instances;
- Synthetic 3 is a short message software system implementing an observer design pattern instance;
- Synthetic 4 is a lights control software system implementing a strategy design pattern instance;
- Synthetic 5 is a sensing alarm software system implementing an observer design pattern instance;
- Synthetic 6 is a product management software system implementing an observer pattern instance;
- *Lexi 0.1.1*[3] is a Java-based open-source word processor. Its main function is to create document, edit text, save file, etc. The format of exported files are compatible with the Microsoft word.
- *JUnit 3.7*[4] is a simple framework to write repeatable tests for java programs. It is an instance of the xUnit architecture for unit testing frameworks.
- *JHotDraw 5.1*[5] is a GUI framework for technical and structured 2D Graphics. Its design relies heavily on some well-known GoF design patterns.

Note that the execution data of *Lexi 0.1.1* and *JHotDraw 5.1* are collected by monitoring typical execution scenarios of the software system. For example, a typical scenario of the *JHotDraw 5.1* is: launch JHotDraw, draw two rectangles, select and align the two rectangles, color them as blue, and close JHotDraw. For the *JUnit 3.7*, we monitor the execution of the project test suite with 259 independent tests provided in the *MapperXML*[6] release.

6.2 Quality Metrics

To evaluate the accuracy of the proposed design pattern detection approach, we use precision and recall measures that are widely adopted in the information retrieval area. *Precision* measures the percentage of the detected pattern instances that are correct pattern instances while *recall* measures the percentage of correct pattern instances that have been correctly detected by our approach.

Formally, *Correct* refers to the set of actual design pattern instances implemented in the software, *Detected* refers to the set of detected pattern instances based on our approach. Precision and recall can be computed as follows:

$$precision = \frac{|Correct \cap Detected|}{|Detected|} \quad (1)$$

$$recall = \frac{|Correct \cap Detected|}{|Correct|} \quad (2)$$

For the synthetic software systems, we have enough knowledge of the implementation, i.e., *Correct* is known. In addition, the collected execution data cover all possible execution scenarios, i.e., the execution data can fully represent the behavior of all potential pattern instances. Therefore, we can measure precision and recall of our approach for the synthetic software systems. As *Correct* is not available for the open-source software and the execution data that are collected by monitoring typical scenario execution only cover a fraction of the software behavior, we only evaluate the precision. In the experiment, we manually validate each detected pattern instances.

Table 5: Observer Pattern Instances Detected from the Execution Data of Synthetic Software Systems.

	#1	#2	#3
Sub	GlobalClock	ActiveSensorSystem	CommentaryObject
Obs	GlobalClockObs	ActiveAlarmLis	Observer
not	run()	soundTheAlarm()	notifyObservers()
upd	periodPasses()	alarm()	update()
reg	attach()	addAlarm()	subscribe()
unr	detach()	removeAlarm()	unsubscribe()

6.3 Evaluation based on Synthetic Software Systems

In this section, we report the detection results obtained by our tool for the six synthetic software systems.

We executed the *DePaD* tool by taking the execution data collected from the six synthetic software systems as input. Three observer pattern instances were returned. Detailed information on the discovered observer pattern instances are shown in Table 5. By manually inspecting these observer pattern instances with respect to our domain knowledge, we found that (1) all these detected observer pattern instances are valid; and (2) all observer pattern instances implemented in Synthetic 3, Synthetic 5, and Synthetic 6 were fully detected.

[3]http://essere.disco.unimib.it/svn/DPB/Lexi%20v0.1.1%20alpha/

[4]http://essere.disco.unimib.it/svn/DPB/JUnit%20v3.7/

[5]http://www.inf.fu-berlin.de/lehre/WS99/java/swing/JHotDraw5.1/

[6]http://essere.disco.unimib.it/svn/DPB/MapperXML%20v1.9.7/

Table 6: State Pattern Instances Detected from the Execution Data of Synthetic Software Systems.

	#4	#5
Context	TContext	Context
State	TS	State
setState	setS()	setState()
request	request()	writeName()
handle	handle()	write()

Table 7: Strategy Pattern Instances Detected from the Execution Data of Synthetic Software Systems.

	#6	#7
Context	RemoteControl	TContext
State	Command	TS
setStrategy	setCommand()	setS()
contextInterface	pressButton()	contextInterface()
algorithmInterface	execute()	algorithmI()

Similarly, we executed *DePaD* tool by taking the execution data collected from the six synthetic software systems as input for state and strategy patterns, two state pattern instances and two strategy pattern instances were returned. Detailed information of the detected state and strategy pattern instances are shown in Tables 6-7. By manually inspecting these detected pattern instances with respect to the domain knowledge, we found that (1) all detected state/strategy pattern instances are valid; and (2) all state/strategy pattern instances implemented in Synthetic 1, Synthetic 2, and Synthetic 4 were fully detected.

To validate the accuracy of the detection results, we also manually analyzed the detected pattern instances in terms of precision and recall. Table 8 reports the precision and recall for observer, state and strategy pattern instances detected from the synthetic software execution data. According to the comparison, we conclude that the proposed approach does not include false positives and can find all true positives in case they are included in the execution data.

6.4 Evaluation based on Open-source Software Systems

In this section, we report the evaluation of our approach using three open-source software systems. We executed the *DePaD* tool by taking the software execution data as input, the number of detected observer, state and strategy pattern instances are shown in Table 9. By manually inspecting the detected pattern

Table 8: Precision and Recall of the Detected Pattern Instances from Synthetic Software Systems.

	Observer Pattern	State Pattern	Strategy Pattern
Precision	100%	100%	100%
Recall	100%	100%	100%

instances, we found that all of them are valid, i.e., the precision of our approach is 100%. This can be explained by the fact that our approach guarantees all detected pattern instances satisfy both structural and behavioral constraints.

Table 9: Number of Detected Pattern Instances from the Execution Data of 3 Open-source Software Systems.

	Observer Pattern	State Pattern	Strategy Pattern
Lexi 0.1.1	–	–	8
JUnit 3.7	2	2	–
JHotDraw5.1	4	22	24

Note that – means no pattern instance is detected from the data.

7 THREATS TO VALIDITY

In the following, we discuss the main threats that may affect the validity of our approach.

- Similar to other dynamic analysis techniques, the quality of the proposed approach heavily depends on the completeness of the execution data. If the execution data do not cover fractions of the software's behavior including all pattern candidates, the results would be unreliable.
- The precision and recall of the detection approach heavily rely on the design pattern specification. On 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.

8 CONCLUSION

Existing dynamic analysis-based design pattern detection approaches generally have problems in performing accurate behavioral constraint checking and providing extensible mechanism to support novel design patterns. This paper proposes a general framework to support the detection of behavioral design patterns from software execution data. To test the applicability, the framework was instantiated for three typical behavioral design patterns, i.e., observer pattern, state pattern and strategy pattern. In addition, the proposed approaches have been implemented as a tool in the open source process mining toolkit ProM. Currently, this tool supports observer pattern, state pattern and strategy pattern. The applicability of this tool was demonstrated by a set of software execution data containing around 1 million method calls.

This work opens the door for several further research directions. The detection of other typical creational and behavioral design patterns (e.g., singleton pattern, factory method pattern, command pattern, visitor pattern) should be included in our framework. In addition, we are working on analyzing large-scale software projects and try to detect more design pattern instances to evaluate the scability and empirically validate extensibility of the approach.

ACKNOWLEDGEMENTS

This work is supported by the *NIRICT 3TU.BSR (Big Software on the Run)* research project[7].

REFERENCES

Arcelli, F., Perin, F., Raibulet, C., and Ravani, S. (2009). Jadept: Dynamic analysis for behavioral design pattern detection. In *4th International Conference on Evaluation of Novel Approaches to Software Engineering, ENASE*, pages 95–106.

Arcelli, F., Perin, F., Raibulet, C., and Ravani, S. (2010). Design pattern detection in java systems: A dynamic analysis based approach. *Evaluation of Novel Approaches to Software Engineering*, pages 163–179.

Bernardi, M. L., Cimitile, M., De Ruvo, G., Di Lucca, G. A., and Santone, A. (2015). Model checking to improve precision of design pattern instances identification in oo systems. In *10th International Joint Conference onSoftware Technologies (ICSOFT)*, volume 2, pages 1–11. IEEE.

Bernardi, M. L., Cimitile, M., and Di Lucca, G. (2014). Design pattern detection using a dsl-driven graph matching approach. *Journal of Software: Evolution and Process*, 26(12):1233–1266.

Cong Liu, Boudewijn van Dongen, N. A. W. v. d. A. (2018). A general framework to detect behavioral design patterns. In *International Conference on Software Engineering (ICSE2018)*, pages 1–2. ACM.

Dabain, H., Manzer, A., and Tzerpos, V. (2015). Design pattern detection using finder. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, pages 1586–1593. ACM.

De Lucia, A., Deufemia, V., Gravino, C., and Risi, M. (2009a). Behavioral pattern identification through visual language parsing and code instrumentation. In *Software Maintenance and Reengineering, 2009. CSMR'09. 13th European Conference on*, pages 99–108. IEEE.

De Lucia, A., Deufemia, V., Gravino, C., and Risi, M. (2009b). Design pattern recovery through visual language parsing and source code analysis. *Journal of Systems and Software*, 82(7):1177–1193.

Dong, J., Zhao, Y., and Sun, Y. (2009). A matrix-based approach to recovering design patterns. *IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans*, 39(6):1271–1282.

Fontana, F. A. and Zanoni, M. (2011). A tool for design pattern detection and software architecture reconstruction. *Information sciences*, 181(7):1306–1324.

Gamma, E. (1995). *Design patterns: elements of reusable object-oriented software*. Pearson Education India.

Heuzeroth, D., Holl, T., Hogstrom, G., and Lowe, W. (2003). Automatic design pattern detection. In *Program Comprehension, 2003. 11th IEEE International Workshop on*, pages 94–103. IEEE.

Leemans, M. and Liu, C. (2017). Xes software event extension. *XES Working Group*, pages 1–11.

Liu, C., van Dongen, B., Assy, N., and van der Aalst, W. (2016). Component behavior discovery from software execution data. In *International Conference on Computational Intelligence and Data Mining*, pages 1–8. IEEE.

Ng, J. K.-Y., Guéhéneuc, Y.-G., and Antoniol, G. (2010). Identification of behavioural and creational design motifs through dynamic analysis. *Journal of Software Maintenance and Evolution: Research and Practice*, 22(8):597–627.

Niere, J., Schäfer, W., Wadsack, J. P., Wendehals, L., and Welsh, J. (2002). Towards pattern-based design recovery. In *Proceedings of the 24th international conference on Software engineering*, pages 338–348. ACM.

Pettersson, N., Lowe, W., and Nivre, J. (2010). Evaluation of accuracy in design pattern occurrence detection. *IEEE Transactions on Software Engineering*, 36(4):575–590.

Shi, N. and Olsson, R. A. (2006). Reverse engineering of design patterns from java source code. In *21st IEEE/ACM International Conference on Automated Software Engineering, 2006.*, pages 123–134. IEEE.

Tsantalis, N., Chatzigeorgiou, A., Stephanides, G., and Halkidis, S. T. (2006). Design pattern detection using similarity scoring. *IEEE transactions on software engineering*, 32(11).

Von Detten, M., Meyer, M., and Travkin, D. (2010). Reverse engineering with the reclipse tool suite. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, volume 2, pages 299–300. IEEE.

Wendehals, L. and Orso, A. (2006). Recognizing behavioral patterns atruntime using finite automata. In *Proceedings of the 2006 international workshop on Dynamic systems analysis*, pages 33–40. ACM.

[7]http://www.3tu-bsr.nl/doku.php?id=start