Improving Design Patterns Finder Precision Using a Model Checking Approach

Mario L. Bernardi, Marta Cimitile, Giuseppe De Ruvo, Giuseppe A. Di Lucca, and Antonella Santone

Department of Engineering, University of Sannio, Benevento, Italy
e-mail: {mlbernar,dilucca,gderuvo,santone}@unisannio.it
Unitelma Sapienza University, Rome, Italy
e-mail: marta.cimitile@unitelma.it

Abstract. In this paper we propose an approach exploiting the model checking technique to automatically refine the results produced by a Design Patterns mining tool called Design Pattern Finder (DPF) to improve the precision of its results by verifying the detected DPs automatically. To assess the feasibility of the proposed approach along with its effectiveness, we have applied it to an open source Object Oriented system with good results in improving the precision of the detected DPs.

Key words: Software Engineering, Design Patterns, Model Checking, Formal Methods, Models, Mining

1 Introduction

The detection of Design Patterns (DPs) [1] istances in Object Oriented (OO) software systems is valuable to assess the quality of the source code [2] , improve program comprehension, maintenance and reuse [3]. According to this, an increasing interest is adressed to the study and experimentation of DPs detection approaches [4]. Bernardi et al. in [5] have proposed an approach called Design Pattern Finder (DPF), based on a meta-model and a Domain Specific Language (DSL) to represent both the software system and the searched DPs. The DPs models are organized as a hierarchy of declarative specifications and expressed as a wide set of high level properties that can be added, removed or relaxed obtaining new pattern variants. The DPF effectiveness, was evaluated by applying it to several systems and the obtained results are reported in [5]. Even if the obtained results are very encouraging, we observed that the precision of the DPF can be further improved. Indeed, DPF, as any other existing DPs detecting approach, can suffer in lacking of precision and completeness. Starting from these considerations, in this work we exploit formal methods to automatically refine the results produced by DPF; in particular we employ model checking using the Language of Temporal Ordering Specification (LOTOS) and selective-μ-calculus.

The model checking (MC) methodology aims to analyse the number of DPs instances, detected by the DPF, evaluating their correctness with respect to formally encoded properties checked against the entire system model represented with (basic) LOTOS. This allows to reduce the number of wrongly detected patterns (false positives) with respect to the original approach. We decided to apply the MC refinement to the DPF, mainly because DPF is based on a meta-model that can be exploited by the model checking refinement to create (basic) LOTOS processes. Therefore, we embodied a new refinement stage adopting DPF outcomes as inputs. From the DPF model we create (basic) LOTOS processes and from DPF detected patterns we generate selective-μ-calculus properties in order to verify the existence of design patterns through model checking.

The approach has been assessed by a preliminary experiment where it was applied to a system from an open benchmark proposed in [6], [7]. Of course, the proposed refining approach can be extended to any other DP mining approach. The remainder of this paper is organized as follows. Section 2 discusses related work. Section 3 gives definitions of basic LOTOS and selective-μ-calculus. Section 4 presents and discusses the proposed detection process, the implemented tools and their integration aspects. Finally, in Section 5, conclusive remarks and future works are presented.

2 Related work

Several pattern recovery techniques and tools have been introduced in the last years. Some reviews about the existing approaches are reported in [4]. Here for brevity, we limit our discussion only to the formal methods (model checking) based approaches. In [8] a formal framework to specify the DPs at different levels of abstraction is proposed. The framework uses stepwise refinement to incrementally add details to a specification after starting from the most abstract one. Moreover, a validation through model checking will verify that a specification in a given level of abstraction is indeed a refinement of a specification of a higher level. The limit of this approach is that a domain specific language to describe DPs is missing and applications in real systems has been never performed. In [9] authors propose an approach aiming to validate DPs using formal method, but the approach is not validated on real software systems. Finally, in [10], a fully automated DPs mining approach performing both static and dynamic analysis to verify the behavior of pattern instances, is proposed. The static analysis exploits model checking to analyze the interactions among objects, while the dynamic analysis of the pattern behavior is performed through a code instrumentation and monitoring phase, applied on the candidate pattern instances. This approach, differently from ours, requires the analysis of the collaboration among objects at runtime by identifying and executing test cases on the software system.

3 Preliminaries

Let us now recall the main concepts of Basic LOTOS [11]. A Basic LOTOS program is defined as:

```
process ProcName := B
where E
endproc
```

where `B` is a *behaviour expression*, `process ProcName := B` is a *process declaration* and `E` is a *process environment*, i.e., a set of process declarations. A behaviour expression is the composition, by means of a set of operators, of a finite set $\mathcal{A}$={`i,a,b, ...`} of atomic *actions*. Each occurrence of an action in $\mathcal{A}$ represents an event of the system. An occurrence of an action `a` $\in \mathcal{A}-$ {`i`} represents a communication on the gate `a`. The action `i` does not correspond to a communication and it is called the *unobservable action*. The syntax of behaviour expressions (also called *processes*) is the following:

B ::= stop | α;B | B[]B| P | B|[S]|B | B[f] | hide S in B | exit
| B>>B | B[>B

where `P` ranges over a set of process names and α ranges over $\mathcal{A}$. he following:

- The *action prefix* `a;B` means that the corresponding process executes the action `a` and then behaves as `B`.
- The *choice* `B1 [] B2` composes the two alternative behavior descriptions `B1` and `B2`.
- The expression `stop` cannot perform any move.
- The *parallel composition* `B1|[S]|B2`, where `S` is a subset of $\mathcal{A}-$ {`i`}, composes in parallel the two behaviors `B1` and `B2`. `B1` and `B2` interleave the actions not belonging to `S`, while they must synchronize at each gate in `S`. A synchronization at gate `a` is the simultaneous execution of an action `a` by both partners and produces the single event `a`. If S=$\emptyset$ or S=$\mathcal{A}$, the parallel composition means pure interleaving or complete synchronization.
- Cyclic behaviors are expressed by recursive process declarations.
- The *relabeling* `B[f]`, where `f`: $\mathcal{A} \rightarrow \mathcal{A}$ is an action relabeling function, renames the actions occurring in the transition system of `B` as specified by the function `f`. This function is syntactically defined as `a0 -> b0,...,an->bn`, meaning `f(a0)=b0,...,f(an)=bn`, and `f(a)=a` for each `a` not belonging to {`a0,...,an`}. Note that each relabelling function has the property that `f(i)` = `i`.
- The *hiding* `hide S in B` renames the actions in `S`, occurring in the transition system of `B`, with the unobservable action `i`.
- The expression `exit` represents successful termination; it can be used by the enabling (`B >> B`) and disabling (`B[> B`) operators: `B >> B` represents sequentialization between `B1` and `B2` and `B[> B` models interruptions. For the sake of simplicity, we do not discuss these operators in the paper.

The semantics of a process `B` is rules describing the transition relation of the automaton corresponding to the behavior expression defining `B`. This automaton is called *standard transition system* for `B` and is denoted by $\mathcal{S}$(`B`). The reader can refer to [11] for details. From now on, we write LOTOS instead of Basic LOTOS.

In the following we recall the selective-μ-calculus, introduced in [12], which is a branching temporal logic to express behavioral properties of systems. It is equi-expressive to μ-calculus [13], but it differs from it in the definition of the modal operators. Given a set $\mathcal{A}$ of actions and a set Var of variables, the selective-μ-calculus logic is the set of formulae given by the following inductive definition:

- `tt` and `ff` are selective-μ-calculus formulae;
- Y, for all $Y \in Var$, is a selective-μ-calculus formula;
- if φ_1 and φ_2 are selective-μ-calculus formulae then $\varphi_1 \wedge \varphi_2$ or $\varphi_1 \vee \varphi_2$ are selective-μ-calculus formulae;
- if φ is a selective-μ-calculus formula then $\langle K \rangle_R \, \varphi$ and $[K]_R \, \varphi$ are selective-μ-calculus formulae, where $K, R \subseteq \mathcal{A}$;
- if φ is a selective-μ-calculus formula then $\mu X.\varphi$ and $\nu X.\varphi$ are selective-μ-calculus formulae, where $X \in Var$.

The satisfaction of a formula φ by a state s of a transition system, written $s \models \varphi$, is defined as follows: each state satisfies `tt` and no state satisfies `ff`; a state satisfies $\varphi_1 \vee \varphi_2$ $(\varphi_1 \wedge \varphi_2)$ if it satisfies φ_1 or (and) φ_2. $[K]_R \, \varphi$ is satisfied by a state which, for every performance of a sequence of actions not belonging to $R \cup K$, followed by an action in K, evolves to a state obeying φ. $\langle K \rangle_R \, \varphi$ is satisfied by a state which can evolve to a state obeying φ by performing a sequence of actions not belonging to $R \cup K$, followed by an action in K. The precise definition of the satisfaction of a closed formula φ by a state of a transition system can be found in [12].

4 Approach

The overall Design Pattern mining approach follows a process structured in two main sub-processes. The first performs the design pattern detection applying the Graph-Matching approach implemented by DPF [5]. The second performs the refinement of DPF results using the model checking approach proposed in this paper.

In the following is a short description of each process activity, while next sub-sections will provide more details about them:

- **Source Code Analysis** — The source and bytecodes of the system under study are parsed and the complete ASTs of the system are produced.
- **Model Instantiation** — A traversal of the system AST is performed to generate an instance of the system model (i.e. the system graph S), conforming to the meta-model defined for DPF. Rapid type analysis (RTA), class flattening and inlining of not public methods are exploited in order to build a system's representation suitable for the matching algorithm.
- **Graph-Matching DPs Detection** — The DPF graph matching algorithm, described in [5], is performed to match the system graph, built in the previous step, with the pattern specifications graphs of the DPs to be detected.

- **Pattern2MU** — Each pattern specification to be detected is written as a set of templates μ-properties (also MU-properties used in the following). These properties involve the patterns roles and their relationships. The template parameters are bound to the concrete system elements using information extracted from the pattern instances found in the detection step (i.e. roles and the system elements related to them).
- **Model2LOTOS** — In order to check if a given set of parametrized MU-properties holds, the system graph should be expressed in a suitable model (in our approach LOTOS was exploited). Hence this step takes the system graph as input and translates it to a LOTOS model instance. This translation has to be performed only one time for each system to be mined.
- **Results refinement** — This step checks the parametrized sets of MU-properties obtained from the pattern specifications catalogue against the LOTOS model of the system in order to reduce the number of false positives.

4.1 Graph-Matching DPs Detection

The detection of the DPs instances is performed according to the DPF approach [5], based on a meta-model and a Domain Specific Language (DSL) to model the structure of both OO systems and DPs. Each pattern, in order to be detected, is modeled by a DSL pattern specification that can be translated into DP Graph (DPG) which is part of the input for the graph-matching detection algorithm.

Along the execution of the DPF Graph Matching algorithm, the system graph (i.e., the instance of the system model) is traversed and each pattern instance sub-graph is mapped to the corresponding matching DPG (to identify the actually implemented patterns). More insights and details about the DPF approach can be found in [5].

4.2 DPF Refinement

The proposed approach is based on the use of formal methods (to the authors' knowledge, never used before). From the DPF outcomes we derive LOTOS processes, which are successively used to perform model checking. The goal of the approach is to increase the precision of DPs mining results produced by DPF. This part of the approach is addressed by the second sub-process which comprises the following steps:

1. LOTOS System model creation (Model2LOTOS activity)
2. Pattern Property generation (Pattern2MU activity)
3. Pattern Matching through Model Checking (Results Refinement activity)

In the following subsections the three steps are discussed in detail.

LOTOS model creation We use, as internal representation, the LOTOS language. Thus, LOTOS specifications are generated starting from the internal representation of DPF. This is obtained by defining a DPF-to-LOTOS transform operator $\mathcal{T}$. The function $\mathcal{T}$ directly applies to Java system outcomes of DPF and translates them into LOTOS process specifications. The function $\mathcal{T}$ is defined for each part of a Java system such as classes, interfaces, methods, fields. Each one has been translated into LOTOS processes. First of all, a System is composed of a set of Types. A Type may be a ClassType or an InterfaceType. A ClassType is made up of Methods. Types may be tied by inheritance relations and a ClassType may implement an InterfaceType, as usually occurs in OO software systems.

System

The generic Java *System* containing k types is translated into the following LOTOS process:

$$\mathcal{T}(C) = \ process\ SYSTEM := Type_1[] \cdots []Type_k\ endproc$$

where $Type_i$ is written using the fully qualified Java name. The LOTOS process $SYSTEM$ represents the parent process of all the types. Each translated LOTOS model has a $System$ process.

Type

As stated, a Type may be a ClassType or an InterfaceType. For example, if FQN is the fully qualified name of a Type, an InterfaceType is translated into the following LOTOS process:

$$\begin{array}{l} \mathcal{T}(I) = process \\ FQN\_InterfaceType := \\ name\_InterfaceType; (FQN\_\ Method_i; FQN\_\ Method_i\_\ Method[] \cdots [] \\ FQN\_\ Method_k; FQN\_\ Method_k\_\ Method[] \\ inherits; (FQN\_\ InterfaceType_l[] \cdots [] \\ FQN\_\ InterfaceType_y)) \\ endproc \end{array}$$

where *implements* and *inherits* are actions which respectively indicate implementation of interfaces and inheritance relation between types.

Method

A method is represented with its own arguments and with a modifier, thus it is translated into the following LOTOS process:

$$\begin{array}{l} \mathcal{T}(M) = process \\ \ FQN\_\ Method := name\_Method; (arg_i[] \cdots []arg_k[]modifier\_mod) \\ endproc \end{array}$$

where arg_i is the name of the argument and mod is the type of modifier such as public, private, protected.

Pattern Property generation In our approach, we use model checking to verify the existence of specific patterns. Once we have the LOTOS processes of the Java software system, we can use selective-μ-calculus logic to specify desired properties. A pattern is translated into a selective-μ-calculus property. Each design pattern leads to a different property, although a set of common properties are used as building blocks:

1. Existence of Interface Implementation:
   $\langle implements \rangle_\emptyset \, \langle name\_ \, InterfaceType \rangle_\emptyset \, \mathtt{tt}$
2. Existence of Inheritance:
   $\langle inherits \rangle_\emptyset \, \langle name\_ \, ClassType \rangle_\emptyset \, \mathtt{tt} \wedge \langle inherits \rangle_\emptyset \, \langle name\_ \, InterfaceType \rangle_\emptyset \, \mathtt{tt}$
3. Existence of a Method:
   $\langle name\_ \, Method \rangle_\emptyset \, \mathtt{tt}$
4. Existence of a Field:
   $\langle field \rangle_\emptyset \, \langle name\_InterfaceType \rangle_\emptyset \, \mathtt{tt} \wedge \langle field \rangle_\emptyset \, \langle name\_ClassType \rangle_\emptyset \, \mathtt{tt}$
5. Existence of an Argument:
   $\langle arg \rangle_\emptyset \, \langle name\_ \, InterfaceType \rangle_\emptyset \, \mathtt{tt} \wedge \langle arg \rangle_\emptyset \, \langle name\_ \, ClassType \rangle_\emptyset \, \mathtt{tt}$

Pattern Matching through Model Checking Once we have created the LOTOS model of a Java software system and we also have built all the properties which represent the design patterns, we can proceed with model checking. As aforementioned, in this paper both model and properties (patterns) come out translating the ones of DPF. We have used CADP [14] as formal verification environment. The CADP model checker is applied verifying each pattern against the System model. When the result is TRUE, it means that the pattern has been found. FALSE otherwise. Thanks to a very detailed LOTOS model we are able to detect false positives of DPF.

5 Conclusions and future works

In this work we exploit formal methods to automatically refine the results produced by a previous approach called DPF. DPF approach introduces a meta-model to represent both the patterns and the system under study as graphs in order to apply a graph matching algorithm. In this paper the detection process is enriched with a model-checking refinement step in which the system model is represented using LOTOS and patterns as selective-μ-calculus properties checked against it. The defined LOTOS model allows to check a wider set of properties that lead to a reduction of the number of false positives. Preliminary experiments performed on a middle sized system (QuickUML 2.1) confirmed the feasibility, correctness, and effectiveness of the approach showing, improvement of the precision (30% on average) with a very reduced impact on the original recall. The model-checking step indeed reduced to zero the number of false positives for Command and Strategy patterns, raising the precision, respectively, from 0.88 and 0.67 to 1. In QuickUML system in both cases the MC properties were able to consider structural or behavioral relationships that the original DPF approach was unable to take into account.

As future works, a more complete translation of pattern specifications to selective-μ-calculus properties will be defined. Moreover, we want to develop new user friendly tools to assist software engineers during the model checking step, as done in [15]. Finally, we plan to perform the translation of the entire DP catalogue defined in [5] as selective-μ-calculus properties.

References

1. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design patterns: elements of reusable object-oriented software. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1995)
2. Bergenti, F., Poggi, A.: Improving uml designs using automatic design pattern detection. In: SEKE 2000. (2000) 336–343
3. L. Prechelt, B. Unger-Lamprecht, M.P., Tichy, W.: Two controlled experiments assessing the usefulness of design pattern documentation in program maintenance. IEEE Trans. Softw. Eng. **28**(6) (2002) 595–606
4. Rasool, G., Streitfdert, D.: A survey on design pattern recovery techniques. IJCSI International Journal of Computer Science Issues **8**(2) (2011) 251 – 260
5. Bernardi, M., Cimitile, M., Di Lucca, G.: Design patterns detection using a dsl-driven graph matching approach. Journal of Software: Evolution and Process **Wiley Online Library** (2014)
6. Guéhéneuc, Y.G.: P-mart: Pattern-like micro architecture repository,. In: Proceedings of the 1st EuroPLoP Focus Group on Pattern Repositories, Michael , Aliaksandr Birukou, and Paolo Giorgini (2007, `http://www.ptidej.net/tool/designpatterns/`)
7. : Comsats institute of information technology. `http://research.ciitlahore.edu.pk/Groups/SERC/DesignPatterns.aspx`
8. Taibi, T., Herranz-Nieva, Á., Moreno-Navarro, J.J.: Stepwise refinement validation of design patterns formalized in TLA+ using the TLC model checker. Journal of Object Technology **8**(2) (2009) 137–161
9. Aranda, G., Moore, R.: A formal model for verifying compound design patterns. In: Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering. SEKE '02, New York, NY, USA, ACM (2002) 213–214
10. De Lucia, A., Deufemia, V., Gravino, C., Risi, M.: Improving behavioral design pattern detection through model checking. In: CSMR, 2010. (2010) 176–185
11. Bolognesi, T., Brinksma, E.: Introduction to the iso specification language lotos. Computer Networks **14** (1987) 25–59
12. Barbuti, R., De Francesco, N., Santone, A., Vaglini, G.: Selective mu-calculus and formula-based equivalence of transition systems. J. Comput. Syst. Sci. **59**(3) (1999) 537–556
13. Stirling, C.: An introduction to modal and temporal logics for ccs. In: Concurrency: Theory, Language, And Architecture. (1989) 2–20
14. Garavel, H., Lang, F., Mateescu, R., Serwe, W.: CADP 2011: a toolbox for the construction and analysis of distributed processes. STTT **15**(2) (2013) 89–107
15. De Ruvo, G., Santone, A.: An eclipse-based editor to support lotos newcomers. In: WETICE, 2014 IEEE 23rd. (June 2014)