Model Checking to Improve Precision of Design Pattern Instances Identification in OO Systems

Mario L. Bernardi [1] , Marta Cimitile [2] , Giuseppe De Ruvo [1] , Giuseppe A. Di Lucca [1] and Antonella Santone [1]

[1] *Department of Engineering, University of Sannio, Italy*

[2] *Unitelma Sapienza University, Italy*

mlbernar@unisannio.it, marta.cimitile@unitelma.it, gderuvo@unisannio.it, dilucca@unisannio.it, santone@unnisannio.it

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

Abstract: In the last two decades some methods and tools have been proposed to identify the Design Pattern (DP) instances implemented in an existing Object Oriented (OO) software system. This allows to know which OO components are involved in each DP instance. Such a knowledge is useful to better understand the system thus reducing the effort to modify and evolve it. The results obtained by the existing methods and tools can suffer a lack of completeness or precision due to the presence of false positive/negative. Model Checking (MC) algorithms can be used to improve the precision of DP's instances detected by a tool by automatically refining the results it produces. In this paper a MC based technique is defined and applied to the results of an existing DPs mining tool, called Design Pattern Finder (DPF), to improve the precision by verifying automatically the DPs instances it detects. To verify and assess the feasibility and the effectiveness of the proposed technique, we carried out a case study where it was applied on some open source OO systems. The results showed that the proposed technique allowed to improve the precision of the DPs instances detected by the DPF tool.

1 Introduction

In the last two decades we have seen a growth on the usage of Design Patterns (DPs) (Gamma et al., 1995) in the development of Object Oriented (OO) software systems, because their adoption contributes to greatly improve the software quality (Ampatzoglou et al., 2012), (Bergenti and Poggi, 2000). Unfortunately, the lack of adequate documentation may make difficult to understand which are the adopted Design Patterns and where they are implemented (i.e., which code components implement each instance of a DP) in a system. Thus, several approaches have been proposed to support the automatic identification of DPs instances in an existing OO software system, linking each detected instance to the OO components implementing it (Peng et al., 2008), (Dong et al., 2007), (Rasool and Streitfdert, 2011). The automatic detection of DPs provides software engineers the needed knowledge to better comprehend the system reducing the effort to modify and evolve it (Bergenti and Poggi, 2000), (Beyer, 2006), (L. Prechelt and Tichy, 2002).

However, the results obtained by the existing DP detection approaches can suffer a lack of completeness or precision due to the presence of false positive/negative. The precision of the DP's instances detected by a tool can be improved by Model Checking (MC) techniques that can automatically refine the results the tool produces. Model checking has been applied to several fields. For instance, it has been used in bioinfomatics to infer gene regulatory networks from time series data (Ceccarelli et al., 2015) or to analyse wiki quality (De Ruvo and Santone, 2015). In this paper we exploit formal methods to automatically refine the results produced by an existing DP mining tool; in particular we employ Model Checking using the Language of Temporal Ordering Specification (LOTOS) and selective-μ-calculus (we interchangeably refer to either μ or MU). The MC methodology aims to analyze the DPs' instances, detected by the mining tool, 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 apply the proposed MC technique to the Design Pattern Finder (DPF) approach presented in (Bernardi et al., 2013), (Bernardi et al., 2014). The DPF approach is 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. Moreover, the DPF approach with respect to the existing ones: i) allows to easily specify variant forms of the classic DPs; ii) takes into account a wider set of high level properties (including also the behavioural properties to better characterize DPs) to specify a pattern. The DPF effectiveness, was evaluated by applying it to several systems and the obtained results are reported in (Bernardi et al., 2014). Even if the obtained results are good, 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. We decided to apply the MC refinement to the DPF, because: (i) the authors of DPF made available both the tool and the results of previous analysis they made; (ii) DPF seems to perform better than other similar tools as shown in (Bernardi et al., 2014); (iii) DPF is based on a meta-model that can be exploited by the MC refinement to create (basic) LOTOS processes.

Therefore, we embodied a new refinement step along the DPF detection process, where the DPF outcomes are the 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 actual existence of design patterns instances through model checking.

The approach has been assessed by applying it to some systems of open benchmarks proposed in (Guéhéneuc, 2007) and in (Rasool et al., 2010).

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. Section 5 introduces and describes the case study. Finally, in Section 6, conclusive remarks and future work are presented.

2 Related work

In the last years, many design pattern recovery techniques and tools have been proposed. In (Dong et al., 2007) and (Rasool and Streitfdert, 2011) reviews regarding some of the main existing approaches can be found.

Several approaches, as the ones in (Tsantalis et al., 2006), (Dong et al., 2009), (Paakki et al., 2000), use UML structures, represented as matrices, to model structural and behavioral information of software systems. These techniques are applied to match a DP template matrix with the matrix generated for the system. In particular, a DP detection methodology based on similarity scoring between graph vertexes is proposed in (Tsantalis et al., 2006). The approach is able to also recognize patterns that are slightly modified from their standard representation. It exploits the fact that patterns reside in one or more inheritance hierarchies (in order to reduce the size of the graphs to which the algorithm is applied). These approaches are computationally efficient and have good precision and recall rates. Their limit is that they miss to detect patterns variants of similar design patterns. Furthermore, they are only limited to the patterns coded as matrices and hence it is not suitable to be easily extended.

Some DP mining approaches are based on metric techniques: program related metrics (i.e. generalizations, aggregations, associations, interface hierarchies) are computed from different source code representations and their values compared with source code DP metrics (von Detten and Becker, 2011), (Paakki et al., 2000), (Antoniol et al., 1998). These techniques are computationally efficient because metric computation is less expensive than structural pattern recognition and do not require heuristic approach to reduce search space through filtration (Guéhéneuc et al., 2010). Their precision and recall are usually low; moreover they were experimented on few design patterns in literature.

Other DP detection approaches exploit other techniques (such as, fuzzy reasoning, bit vector compression, minimum key structure method, dynamic analysis using run-time execution traces, machine learning based approaches and concept analysis) that are good as a complement to improve the DP detection based on structural methods. For example, in (De Lucia et al., 2009), De Lucia et. al. use a recovery technique based on the parsing of visual languages, and supported by a visual environment automatically produced by a grammar based visual environment generator. A tool, using a mixed structural and metric approach, for design pattern detection and software architecture reconstruction is proposed in (Arcelli and Zanoni, 2011).

Other studies (Dong et al., 2009), (Tonella et al., 2007) have been focused on the formalization of empirical evaluation criteria (Dong et al., 2009), (Tonella et al., 2007).

There is few work, at best of our knowledge, that exploits formal methods (model checking) based ap-

proaches to detect DP instances in existing OO systems. In (Taibi et al., 2009) 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 (Aranda and Moore, 2002), authors propose an approach aiming to validate DPs using formal method. Similarly, in (Flores et al., 2001) formal methods are used to demonstrate that a particular design conforms to a given DPs. Both these approaches, are not validated on real software systems. Finally, in (De Lucia et al., 2010), 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

Historically, process algebras have been developed as formal descriptions of complex computer systems, and in particular of those involving communicating, concurrently executing components. The crucial idea in the definition of Process Algebras is the algebraic structure of the concurrent processes. This uses a state-based approach with labeled transitions, where states and transitions correspond to processes and actions, respectively. There are many examples of process algebras, like for example Milner's Calculus of Communicating Systems (CCS) (Milner, 1989) and Language of Temporal Order Specification (LOTOS) (Bolognesi and Brinksma, 1987), which we will use in this paper.

3.1 Basic LOTOS

Let us now recall the main concepts of Basic LOTOS. 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 | a;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 `a` ranges over $\mathcal{A}$. The operational semantics of a behaviour expression `B` is a labelled transition system, i.e., an automaton whose states correspond to behaviour expressions (the initial state corresponds to `B`) and whose transitions (arcs) are labeled by actions in $\mathcal{A}$. The meaning of the operators composing behavior expressions is the 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`.

$a \in \mathcal{A},\ l \in \mathcal{A} - \{\mathtt{i}\}$

$$\textbf{pre}\ \frac{}{a;B \xrightarrow{a}_{\mathcal{S}} B} \qquad \textbf{choice}\ \frac{B_1 \xrightarrow{a}_{\mathcal{S}} B_1'}{B_1\ [\,]\ B_2 \xrightarrow{a}_{\mathcal{S}} B_1'}$$

$$\textbf{inst}\ \frac{B \xrightarrow{a}_{\mathcal{S}} B'}{P \xrightarrow{a}_{\mathcal{S}} B'}\ P := B \in E \qquad \textbf{rel}\ \frac{B \xrightarrow{a}_{\mathcal{S}} B'}{B[f] \xrightarrow{f(a)}_{\mathcal{S}} B'[f]}$$

$$\textbf{par}\ \frac{B_1 \xrightarrow{a}_{\mathcal{S}} B_1'}{B_1\ |[S]|\ B_2 \xrightarrow{a}_{\mathcal{S}} B_1'\ |[S]|\ B_2}\ a \notin S$$

$$\textbf{com}\ \frac{B_1 \xrightarrow{a}_{\mathcal{S}} B_1',\ B_2 \xrightarrow{a}_{\mathcal{S}} B_2'}{B_1\ |[S]|\ B_2 \xrightarrow{a}_{\mathcal{S}} B_1'\ |[S]|\ B_2'}\ a \in S$$

$$\textbf{hide}_1\ \frac{B \xrightarrow{a}_{\mathcal{S}} B'}{\mathtt{hide\ S\ in\ B} \xrightarrow{a}_{\mathcal{S}} \mathtt{hide\ S\ in\ B'}}\ a \notin S$$

$$\textbf{hide}_2\ \frac{B \xrightarrow{l}_{\mathcal{S}} B'}{\mathtt{hide\ S\ in\ B} \xrightarrow{\mathtt{i}}_{\mathcal{S}} \mathtt{hide\ S\ in\ B'}}\ l \in S$$

Table 1: *Standard operational semantics of Basic LOTOS*

- 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.

Assume the precedence of the operators as specified by the following list, ordered in decreasing order:

```
;   [f]   hide   |[S]|   []
```

The semantics of a process `B` is precisely defined by means of the structural operational semantics (in Table 1). The semantic definition is given by a set of conditional 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 $\mathcal{S}(\mathtt{B})$. In Table 1 the symmetrical rules for choice and parallel composition are not shown.

We consider only finite Basic LOTOS programs, i.e., programs with finite standard transition systems. A sufficient condition for finiteness is that the parallel operator does not occur inside recursive process declarations. From now on, we write LOTOS instead of Basic LOTOS.

3.2 Selective-μ-calculus

The selective-μ-calculus, introduced in (Barbuti et al., 1999), is a branching temporal logic to express behavioral properties of systems. It is equi-expressive to μ-calculus (Stirling, 1989), 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. $[K]_R \varphi$ and $\langle K \rangle_R \varphi$ are the selective modal operators:

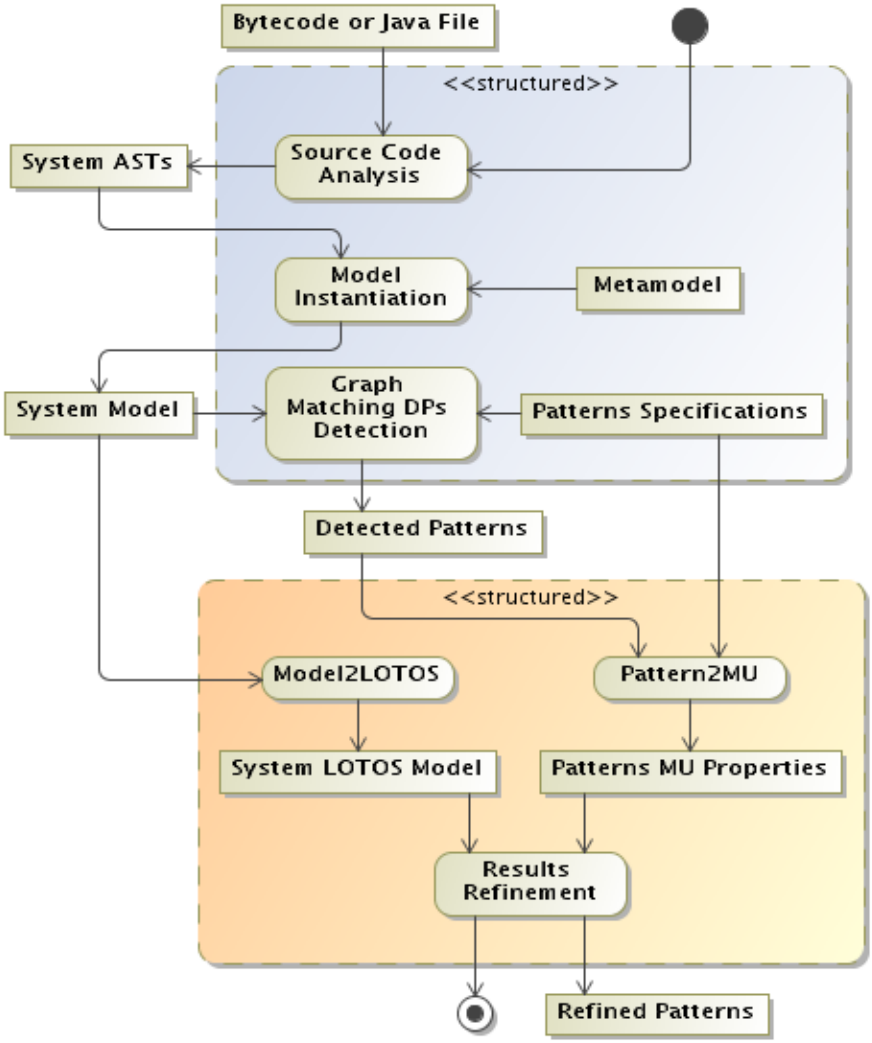

Figure 1: The DPs mining process represented as a UML activity diagram

$[K]_R\varphi$ 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 selective modal operators $\langle K \rangle_R \varphi$ and $[K]_R \varphi$ substitute the standard modal operators $\langle K \rangle \varphi$ and $[K] \varphi$. The basic characteristic of the selective-μ-calculus is that each formula allows us to immediately point out the parts of the transition system that do not alter the truth value of the formula itself. More precisely, *the only actions relevant for checking a formula are the ones explicitly mentioned by the selective modal operators used in the formula itself*. Thus, the result of checking the formula is independent from all other actions. This information can be exploited to obtain reduced transition systems on which the formula can be equivalently checked (see, for example, (Barbuti et al., 2005)). The precise definition of the satisfaction of a closed formula φ by a state of a transition system can be found in (Barbuti et al., 1999).

4 The approach

In this section we introduce the overall Design Pattern mining approach.

The process is structured in two main subprocesses. The first one, shown in the upper part of

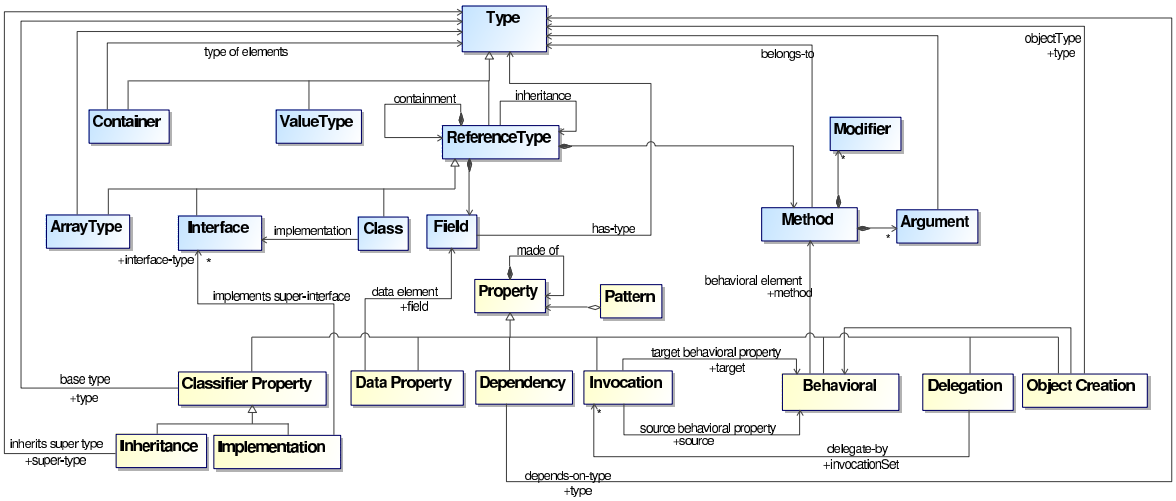

Figure 2: The meta-model represented as a UML class diagram

the Figure 1, performs the design pattern detection applying the Graph-Matching approach implemented by DPF, discussed in (Bernardi et al., 2014). The second sub-process, depicted in the lower part of the figure, performs the refinement of DPF results using the model checking approach proposed in this paper.

In the following there 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 (Bernardi et al., 2014), is performed to match the system model, 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 templated μ-properties. 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 μ-properties holds, the system model graph should be expressed in a suitable model (in our approach LOTOS was exploited). Hence this step takes the system model 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 μ-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 (Bernardi et al., 2014). The DPF approach is based on a meta-model and a Domain Specific Language (DSL) to model the structure of both OO systems and DPs. The meta-model uniformly describes the DPs and systems in terms of relationships among code elements, and allows to trace down to the DPs Properties and Types components both the structural and behavioral relationships among the types. The meta-model is reported in Figure 2 as a UML class diagram. The upper part of the figure shows the structure of an OO system as a set of Types (i.e., Container, Value, Reference, and Compound Types) along with their relationships. Reference Types, composed by Fields and Methods can inherit from another ReferenceType as well as can contain another ReferenceType. Similarly, in the bottom part of the figure a DP is modeled as the aggregation of several Properties (Classifier, Behavioral, Dependency, Invocation, Delegation, Object Creation). The

```
pattern Singleton {
   final type X {
   X has private constructor c;
   X has field f of type X;
   X has public static methods-set creationHooks
    each {depends on f; }
   }
}
```

Figure 3: The DSL of Singleton DP

meta-model is exploited to define the DSL to represent structural and behavioral relevant properties of OO software systems, as well as to express the specifications of the DPs to be detected. Each pattern, in order to be detected, has been modeled writing a DSL pattern specification stored into a repository. The current repository stores a catalog composed of 18 patterns with 56 variants. Each DSL specification can be translated into a graph, called DP Graph (DPG), in which elements are nodes and properties are labeled edges. The DPG is part of the input for the graph-matching detection algorithm. As an example of how a pattern is modeled by the DSL and represented by the corresponding DP Graph , let us consider a classic Singleton DP as defined in (Gamma et al., 1995). The DSL, reported in Figure 3, provides a Singleton definition implemented with a final class, a private constructor and a public static getter method. To mine multiple instance getters, the variant defines a method set called "creationHooks" (the box labelled by cH in Figure 4). Each method in this set requires a dependency on the static Singleton field "f".

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 (Bernardi et al., 2014).

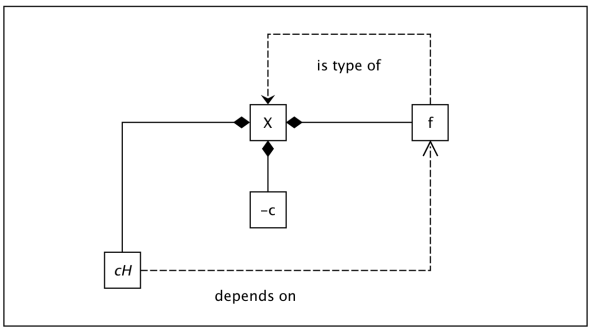

Figure 4: GoF Singleton DP graph

4.2 DPF Refinement

The proposed approach is based on the use of formal methods. 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 shown in the bottom of Figure 1 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 by more details.

4.2.1 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. If FQN is the fully qualified name of a Type, a ClassType is translated into the following LOTOS process:

$\mathcal{T}(T) = process$
$FQN\_ClassType := name\_ClassType;$
$(FQN\_Method_i; FQN\_Method_i\_Method[] \cdots []$
$FQN\_Method_k; FQN\_Method_k\_Method[]$
$implements; (FQN\_InterfaceType_j[] \cdots []$
$FQN\_InterfaceType_w)[]$
$inherits; (FQN\_ClassType_l[] \cdots []$
$FQN\_ClassType_y)[].field.(FQN\_InterfaceType_h[]$
$\cdots [] FQN\_ClassType_z))$
$endproc$

Instead, an InterfaceType is translated into the following LOTOS process:

$\mathcal{T}(I) = process$
$FQN\_InterfaceType :=$
$name\_InterfaceType; (FQN\_Method_i;$
$FQN\_Method_i\_Method[] \cdots []$
$FQN\_Method_k; FQN\_Method_k\_Method[]$
$inherits; (FQN\_InterfaceType_l[] \cdots []$
$FQN\_InterfaceType_y))$
$endproc$

where *implements* and *inherits* are actions which respectevely 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:

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

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

4.2.2 Pattern Property generation

After the LOTOS processes of the Java software system are generated, 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:
   $\langle implements \rangle_{\emptyset} \langle name\_InterfaceType \rangle_{\emptyset}$ tt

2. Existence of Inheritance:
   $\langle inherits \rangle_{\emptyset} \langle name\_ClassType \rangle_{\emptyset}$ tt $\wedge$
   $\langle inherits \rangle_{\emptyset} \langle name\_InterfaceType \rangle_{\emptyset}$ tt

3. Existence of a Method:
   $\langle name\_Method \rangle_{\emptyset}$ tt

4. Existence of a Field:
   $\langle field \rangle_{\emptyset} \langle name\_InterfaceType \rangle_{\emptyset}$ tt $\wedge$
   $\langle field \rangle_{\emptyset} \langle name\_ClassType \rangle_{\emptyset}$ tt

5. Existence of an Argument:
   $\langle arg \rangle_{\emptyset} \langle name\_InterfaceType \rangle_{\emptyset}$ tt $\wedge$
   $\langle arg \rangle_{\emptyset} \langle name\_ClassType \rangle_{\emptyset}$ tt

4.2.3 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. Such translations have been completely automated.

One of the most popular toolbox for the design of asynchronous concurrent systems is CADP (Garavel et al., 2013). It supports high-level descriptions written in various languages, mainly LOTOS. In the CADP the verification of temporal logic formulae is based on model checking (Clarke et al., 2001), a formal technique for proving the correctness of a system with respect to a desired behavior. This is accomplished by checking whether a structure representing the system (typically a labelled transition system) satisfies a temporal logic formula describing the expected behaviour.

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 identify false positives among the DPs detected by DPF. Eventually, we have all the necessary information to improve the precision of the overall results, as explained in the following section.

5 Case study

The effectiveness and efficiency of the proposed approach has been validated applying it to some middle-sized OO systems. These systems were available from the publicly available benchmarks proposed in (Guéhéneuc, 2007) and in (Rasool et al., 2010).

Due to space constraints we only present the results for the two systems reported in Table 2 and the 4

System Name	Version	Size (KLOC)	#Types	#Methods
JHotDraw	5.1	8,9K	174	1316
QuickUML	2.1	9,2K	230	1082

Table 2: Analyzed systems characteristics

Step →	Detection							Refinement					
↓Design Pattern	GS	D	T_P	F_P	F_N	P	R	D	T_P	F_P	F_N	P	R
System→	JHotDraw												
Composite/spec{GoF}	16	19	14	5	2	0,74	0,88	17	14	3	2	0,82	0,88
Factory Method/spec{Parametrized}	15	14	12	2	3	0,86	0,8	14	12	2	3	0,86	0,8
System→	QuickUML												
Command/spec{GoF}	10	8	7	1	3	0,88	0,7	8	8	0	1	1	0,89
Strategy/spec{GoF}	15	18	12	6	3	0,67	0,8	12	12	0	3	1	0,8

Table 3: Results obtained on JHotDraw and QuickUML

GoF patterns (Command, Composite, Factory methods and Strategy) for which the DPF method provides the lowest precision.

According to (Pettersson et al., 2010), in order to assess effectiveness and correctness of the proposed approach, we evaluated precision and recall. To compute recall and precision we assume that a pattern instance can be classified into one of four categories (T_P: true positive, F_P: false positive, T_N: true-negative, and F_N false-negative).

Precision is defined as the ratio of correctly found occurrences to occurrences provided by the tool whereas recall is defined as the ratio of correctly found occurrences to all correct occurrences:

$$Precision = T_P/(T_P + F_P) \quad (1)$$

$$Recall = T_P/(T_P + F_N) \quad (2)$$

To verify the correctness of the results we considered as Gold Standard (GS) the union of both the benchmarks cited in (Guéhéneuc, 2007) and in (Rasool et al., 2010) (assumed to be correct) with the correct results produced by DPF approach (i.e., also the instances not present in the benchmarcks, mainly due to DP variants, but correctly detected by DPF as verified by manual code inspection). [1]

Since in this context we are interested to assess the improvement in precision obtained after the model-checking driven refinement, we evaluate and compare precision and recall at the end of both DPs Matching and DPF Refinement steps.

[1]Of course, the different formats of the benchmarks were translated into a unique common format to store the considered GS.

Table 3 reports, for each of the analyzed systems: the name of the DPs searched in the code (first column), the number of true positive instances as provided by the benchmark (GS), and two groups of columns for the DPs detection performed by DPF and the DPF refinement steps performed using the model checker. Each group contains the number of detected patterns (column D), the number of true positive (column Tp), the number of false positive and negative ones (columns F_p and F_n). The last two columns report respectively precision (P) and recall (R).

In the Composite-GoF pattern, for the JHotDraw system, the model-checking step reduced the number of false positive from five to three raising the precision from 0.74 to 0.82. Looking at the three remaining false positive we can see that these are cases in which the assignment of the element to a role in the pattern can be only done looking at the semantics of the element. This is confirmed by the presence in the results of two methods of 3 concrete composite classes (*read()* and *readObject()* in subclasses of CompositeFigure) that were mistakenly bound to Operation role by both the steps but are not part of the interface. This is also the case for the parametrized Factory Method for which the two false positives have the same structure and behaviour of the defined property but cannot be considered as factory methods (*decompose(...)* and *flip(...)* method of Figure class).

In QuickUML system in both cases properties were able to consider structural or behavioral relationships that the original approach was unable to take into account. For instance, for the Strategy pattern several false positive (e.g. the ToolPalette, Clipboard Tool, PropertyChangeHandler and SelectioModel contexts) were detected since the properties

were able to better identify indirect relationships and type nesting relationships.

The overall average improvement for all the considered patterns and systems was above 19%.

6 Conclusions and future works

In this work we exploit formal methods to automatically refine the results produced by an existing DP mining approach, in particular we selected the DPF approach. DPF approach introduces a metamodel 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. The performed experiments confirmed the feasibility, correctness, and effectiveness of the approach showing, on the analyzed systems, an improvement of the precision (19% on average) with a very reduced impact on the original recall. As future work, a more complete translation of pattern specifications to selective-μ-calculus properties will be defined. Moreover, we will perform the translation of the entire DP catalog defined in (Bernardi et al., 2014) as selective-μ-calculus properties allowing the experimentation on the complete benchmark comprised of 12 OO systems. Finally, we want to assist software engineers providing WYSIWYG tools that support our approach as done in (De Ruvo and Santone, 2014).

REFERENCES

Ampatzoglou, A., Frantzeskou, G., and Stamelos, I. (2012). A methodology to assess the impact of design patterns on software quality. *Inf. Softw. Technol.*, 54(4):331–346.

Antoniol, G., Fiutem, R., and Cristoforetti, L. (1998). Design pattern recovery in object-oriented software. In *Proceedings of the 6th International Workshop on Program Comprehension*, IWPC '98, pages 153–, Washington, DC, USA. IEEE Computer Society.

Aranda, G. and Moore, R. (2002). A formal model for verifying compound design patterns. In *Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering*, SEKE '02, pages 213–214, New York, NY, USA. ACM.

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

Barbuti, R., De Francesco, N., Santone, A., and Vaglini, G. (1999). Selective mu-calculus and formula-based equivalence of transition systems. *J. Comput. Syst. Sci.*, 59(3):537–556.

Barbuti, R., De Francesco, N., Santone, A., and Vaglini, G. (2005). Reduced models for efficient ccs verification. *Formal Methods in System Design*, 26(3):319–350.

Bergenti, F. and Poggi, A. (2000). Improving uml designs using automatic design pattern detection. In *Proc. 12th. International Conference on Software Engineering and Knowledge Engineering (SEKE 2000*, pages 336–343.

Bernardi, M., Cimitile, M., and Di Lucca, G. (2013). A model-driven graph-matching approach for design pattern detection. In *20th Working Conference on Reverse Engineering (WCRE)*, pages 172–181.

Bernardi, M., Cimitile, M., and Di Lucca, G. (2014). Design patterns detection using a dsl-driven graph matching approach. *Journal of Software: Evolution and Process*, Published online in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/smr.1674.

Beyer, D. (2006). Relational programming with crocopat. In *Proceedings of the 28th international conference on Software engineering*, ICSE '06, pages 807–810, New York, NY, USA. ACM.

Bolognesi, T. and Brinksma, E. (1987). Introduction to the iso specification language lotos. *Computer Networks*, 14:25–59.

Ceccarelli, M., Cerulo, L., De Ruvo, G., Nardone, V., and Santone, A. (2015). Infer gene regulatory networks from time series data with probabilistic model checking. *FormaliSE 2015*.

Clarke, E. M., Grumberg, O., and Peled, D. (2001). *Model checking*. MIT Press.

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

De Lucia, A., Deufemia, V., Gravino, C., and Risi, M. (2010). Improving behavioral design pattern detection through model checking. In *Software Maintenance and Reengineering (CSMR), 2010 14th European Conference on*, pages 176–185.

De Ruvo, G. and Santone, A. (2014). An eclipse-based editor to support lotos newcomers. In *Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), 2014 IEEE 23rd International Conference on*.

De Ruvo, G. and Santone, A. (2015). Analysing wiki quality using probabilistic model checking. In *2015 IEEE 24th International WETICE Conference, WETICE 2015, Larnaca, Cyprus, 15-17 June, 2015*.

Dong, J., Zhao, Y., and Peng, T. (2007). Architecture and design pattern discovery techniques - a review. In Arabnia, H. R. and Reza, H., editors, *Software Engineering Research and Practice*, pages 621–627. CSREA Press.

Dong, J., Zhao, Y., and Sun, Y. (2009). A matrix-based approach to recovering design patterns. *Trans. Sys. Man Cyber. Part A*, 39(6):1271–1282.

Flores, A., Moore, R., and Reynoso, L. (2001). A formal model of object-oriented design and gof design patterns. In *Proceedings of the International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity*, FME '01, pages 223–241, London, UK, UK. Springer-Verlag.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Garavel, H., Lang, F., Mateescu, R., and Serwe, W. (2013). CADP 2011: a toolbox for the construction and analysis of distributed processes. *STTT*, 15(2):89–107.

Guéhéneuc, Y. G. (`http://www.ptidej.net/tool/designpatterns/`, 2007). P-mart: Pattern-like micro architecture repository,. In *Proceedings of the 1st EuroPLoP Focus Group on Pattern Repositories*. Michael , Aliaksandr Birukou, and Paolo Giorgini.

Guéhéneuc, Y. G., Guyomarc'H, J. Y., and Sahraoui, H. (2010). Improving design-pattern identification: a new approach and an exploratory study. *Software Quality Control*, 18(1):145–174.

L. Prechelt, B. Unger-Lamprecht, M. P. and Tichy, W. (2002). Two controlled experiments assessing the usefulness of design pattern documentation in program maintenance. *IEEE Trans. Softw. Eng.*, 28(6):595–606.

Milner, R. (1989). *Communication and concurrency*. PHI Series in computer science. Prentice Hall.

Paakki, J., Karhinen, A., Gustafsson, J., Nenonen, L., and Verkamo, A. I. (2000). Software metrics by architectural pattern mining. In *Proceedings of the International Conference on Software: Theory and Practice (16th IFIP World Computer Congress*, pages 325–332.

Peng, T., Dong, J., and Zhao, Y. (2008). Verifying behavioral correctness of design pattern implementation. In *Proceedings of the Twentieth International Conference on Software Engineering & Knowledge Engineering (SEKE'2008)*, pages 454–459.

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

Rasool, G., Philippow, I., and Mäder, P. (2010). Design pattern recovery based on annotations. *Advances in Engineering Software*, 41(4):519 – 526.

Rasool, G. and Streitfdert, D. (2011). A survey on design pattern recovery techniques. *IJCSI International Journal of Computer Science Issues*, 8(2):251 – 260.

Stirling, C. (1989). An introduction to modal and temporal logics for ccs. In *Concurrency: Theory, Language, And Architecture*, pages 2–20.

Taibi, T., Herranz-Nieva, Á., and Moreno-Navarro, J. J. (2009). Stepwise refinement validation of design patterns formalized in TLA+ using the TLC model checker. *Journal of Object Technology*, 8(2):137–161.

Tonella, P., Torchiano, M., Du Bois, B., and Systä, T. (2007). Empirical studies in reverse engineering: state of the art and future trends. *Empirical Softw. Engg.*, 12(5):551–571.

Tsantalis, N., Chatzigeorgiou, A., Stephanides, G., and Halkidis, S. T. (2006). Design pattern detection using similarity scoring. *IEEE Trans. Softw. Eng.*, 32(11):896–909.

von Detten, M. and Becker, S. (2011). Combining clustering and pattern detection for the reengineering of component-based software systems. In *Proceedings of the joint ACM SIGSOFT conference QoSA-ISARCS*, QoSA-ISARCS '11, pages 23–32, New York, NY, USA. ACM.