A System for Querying Program Models to Discover Design Pattern Implementation Mistakes

Tyler Harter
<tylerharter@gmail.com>

Abstract

Design patterns are an abstraction that can provide guidance in creating software. However, the adherence of design pattern implementations to design pattern descriptions is typically not enforced. While guaranteeing perfect adherence of an implementation is impossible for many non-trivial patterns given their subjective nature, many flaws can be discovered by making simple assertions about an implementation.

1. Introduction

Reusing proven solutions rather than reinventing the wheel can greatly improve productivity. For software developers, design patterns present an excellent opportunity for reuse that is well documented and that improves communication with other developers. Most object-oriented systems make use of some of the common design patterns [1]. This likely makes code maintenance easier, especially for new developers who can quickly understand how a software program is organized by comments that refer to familiar design patterns. Unfortunately, when developers are implementing patterns, poor understanding of patterns could lead to incorrect implementations, which, in turn, could cause confusion for other developers that have a different understanding of what a given pattern means. Even experienced and educated developers are likely to make occasional mistakes.

The goal of the system presented in this paper is to help developers realize their mistakes so that they can correct them. Checking for mistakes in design pattern implementations is difficult, however, since what is correct is sometimes subjective. The flyweight design pattern provides an example of this. Flyweights are objects that can be shared and used in different contexts. Any necessary information about how they're being used must be provided to them from elsewhere since they can be used in multiple ways and places at the same time, making it impossible to store all important information internally. For instance, the flyweight example in the gang of four's Design Patterns book uses flyweights to represent letters. The description of the flyweights says that any state stored by a flyweight must be intrinsic. For example, each flyweight representing a letter should know what letter it represents since that is clearly part of a letter's intrinsic state. At the same time, any additional information such as font is extrinsic state that is provided by a client. The idea of intrinsic versus extrinsic state is critical to the flyweight pattern, yet the distinction is fuzzy. Is boldness intrinsic or extrinsic? It would be make sense to have 26 flyweights for the 26 letters and let the clients inform the flyweights regarding whether or not they're bold. However, one could also argue that having 52 flyweights for both a bold and not bold version of each letter would also be reasonable. Correctly classifying data as either intrinsic or extrinsic could be tricky for people, let alone for a computer program. There might not even be a single correct answer.

Rather than give up, this paper suggests a way to at least catch some of the mistakes developers might make while implementing design patterns in Java programs. The solution presented involves a query system that allows assertions expressed in FOL (first order logic) to be evaluated against automatically generated models of Java programs. This approach makes it possible to quickly write flexible design pattern implementation checkers. The system's flexibility makes it possible to create new checkers for new patterns, but even more importantly, it allows developers to modify patterns to fit their needs. This is useful for patterns such as the singleton. Singletons can only have one instance, so it's necessary to hide a singleton's constructor behind an accessor method. For multi-

threaded programs, the accessor method should use Java's synchronized modifier to guarantee that the constructor will never be called twice. However, for single threaded programs, using the synchronized modifier is unnecessary and will just result in slower programs, so its use should not be enforced. Developers who are implementing singletons under different circumstances would find it easy to modify the singleton checker accordingly.

2. Background

This section describes the software tools that were used as building blocks for the program query system being presented and how they are utilized by the system. The two tools used by the system are the Soot framework, which is used to retrieve information about the Java programs being analyzed, and the Alloy model checking tool and its associated language.

The Soot framework is designed to allow Java programs to be optimized. To allow for optimization, Soot is capable of loading .class files or .java files into various IRs (intermediate representations). Although the described querying system does not need to modify or optimize class files, it uses Soot's Jimple IR to extract important information to put into a program model.

Most of the information provided by Soot is simply dumped into the model with little modification, but some additional work was required to obtain a useful call graph. The call graphers provided with Soot were quite slow, provided too much information, were quite cumbersome and were possibly unreliable, so a customized call grapher was implemented. The Jimple IR provides links from invocation instructions to the methods being invoked, so generating a basic call graph was relatively simple.

The call grapher has two major simplifications: use of just CHA (class hierarchy analysis) for determining what methods can possibly be invoked by a given call, and exclusion of classes that are not part of a pattern implementation. The first simplification, the use of CHA, means that when a method is called, an edge is added to the call graph from the caller to the callee and to all the methods that override the callee in subclasses. The one exception is when the callee is a constructor, as calls to a constructor of a class are obviously always for that exact class, not one of its subclasses. “Smarter” call graphers can sometimes do more complicated analysis to actually determine the exact class of an object, thus eliminating many of the edges that indicate potential calls to overriding methods in subclasses. Using such a call grapher to replace the one currently in use would be a straightforward matter and would provide greater accuracy; however, a “smarter” call graph would also probably be slower.

The other simplification, the exclusion of classes that are not part of a pattern, keeps the call graph from becoming unreasonably large by discarding lots of information that probably is not relevant to most design pattern implementations. This means that if a method calls another method that is not part of a pattern implementation, edges will not be added to the callee or even to methods that override the callee in subclasses. This approach drastically reduces the size of the call graph. Suppose a method casts an object to the “Object” type and then invokes toString on it. Without the simplification, such a call would create edges from the caller to the toString methods of every other class in the program. This would be unnecessary and undesirable. Furthermore, if complete data were loaded for all the classes that are reachable from even an extremely simple Java program, class information for dozens of classes in the Java runtime library would be loaded.

Alloy is the other software tool utilized by the software query system. Alloy is typically used as a model checker. Normally, it is provided with a description of a model that has basic constraints expressed in FOL. Alloy checks models in two ways. First, given FOL “predicates” that describe some interesting property that the model should be able to have, Alloy uses a SAT solver to see if it can find any solutions that satisfy the constraints of both the model and the interesting property. In Alloy, any such solutions that are discovered are called “core instances”. The second way Alloy checks models is with assertions. For assertions, which are also expressed in FOL, Alloy attempts to find core instances where the assertion does not hold. Any core instances that are found are presented to the user as counterexamples that prove that the provided assertion is not always true. Once core instances are found, it's possible to use Alloy's Evaluator to evaluate FOL Alloy expressions about

the instances.

Having a basic understanding of Alloy's syntax for FOL, set-builder notation, and relations is necessary for understanding the examples in this paper and for creating design pattern checkers that use the program query system. Table 1 shows some important aspects of Alloy's syntax and the corresponding mathematical symbols where appropriate.

Table 1: Traditional FOL and equivalent Alloy syntax

Symbol description	Symbol	Alloy syntax
for all	∀	all
there exists	∃	some
there does not exist	¬∃	no
such that	|	|
member of	ϵ	item:set
Subset	⊆	set1 in set2
union	⋃	+
set size	|setname|	#setname
not	¬	not
and	∧	and
implication	→	implies
relational product		->
relation composition		.
transitive closure		.^
reflexive transitive closure		.*

Much of Alloy's syntax should be clear from table 1, but the last four rows deserve an explanation. The relational product simply maps one element to another. This can be thought of as a way of expressing fields by mapping an element to its fields. Sets of relational products are also used to implement functions by creating mappings from every element in a function's domain to an element in its range.

The mappings created by sets of relational products can be followed from the domain to the range with the relation composition operator. The composition operator's left operand is simply a set and its right operand is a set of relational products. For instance, if you had an object with a field, the following expression would return a set containing the field:

```
{object}.{object->field}
```

Alternatively, if the operator is considered in terms of functions, the following expression would return the set of all the elements in the function's range:

```
{dom1, dom2}.{dom1->range1, dom2->range2}
```

The transitive closure and reflexive transitive closure are powerful variants of the relation composition operator and are explained with an example in section 4.1.

3. Creating and Using a Program Model

Once all the program information is retrieved from Soot, it is given to Alloy so queries can be made. Alloy typically searches for solutions (called “core instances”) for a model. However, all the important information is taken directly from the program being analyzed, so there is no need to search for anything. Thus, instead of providing Alloy with a model and then having it search for core instances, Alloy is directly provided with both a model that represents all Java programs as well as a core instance that represents the specific Java program being analyzed.

At the risk of causing confusion, the representation of the Java program being analyzed is referred to as a “program model” throughout this paper. This seems like the most natural term. Unfortunately, though, the program model is stored as an Alloy “core instance”, which is not the same as an Alloy “model”. Thus, the word “model” is not used in the sense that Alloy users are familiar with.

Alloy has the ability to load previously saved core instances. The core instances are represented using the XML format, so the query system simply creates temporary core instance XML files for a Java program and then lets Alloy parse them. The partial representation of a core instance is given in Figure 1.

Figure 1: Alloy Core Instance XML Encoding

```
<instance bitwidth="8" maxseq="0" filename="~temp.als">
    <sig label="class" ID="7" builtin="no" parentID="5">
        <atom label="atom41"/>
        <atom label="atom47"/>
        ...
    </sig>

    <field label="superClass" ID="26" parentID="7">
        <tuple>
            <atom label="atom41"/>
            <atom label="atom47"/>
        </tuple>
        <tuple>
            <atom label="atom47”/>
            <atom label="atom215"/>
        </tuple>
        ...
    </field>

    <field label="id" ID="25" parentID="4">
        <tuple>
            <atom label="atom41"/>
            <atom label="7"/>
        </tuple>
        <tuple>
            <atom label="atom47"/>
            <atom label="18"/>
        </tuple>
        ...
    </field>
    ...
</instance>
```

The program model is represented in the XML file by sets of atoms and the relations between them. Sets are called sigs and relations are called fields. In the XML file, the sig with the “class” label contains all the classes which are represented by atoms. The “superClass” field is essentially a set of tuples representing relational products that connect a class's atom to its superclass's atom. Each tuple can be interpreted as meaning that the first element in the tuple has the specified relationship to the second element in the tuple. Thus, according to the first tuple in the “superClass” field, the class associated with atom47 is the superclass of the the class associated with atom41. The “id” field works just like the “superClass” field, except that it maps each class to a unique integer. This is useful, as Alloy core instances cannot encode strings. Thus, in order to print informative error messages and warnings, pattern checkers need a way to lookup the names of the offending classes that are returned by queries. The “id” field accomplishes this. When IDs are initially being assigned, the query system keeps track of which IDs are associated with which classes, methods, and fields so that the original names provided by Soot can be looked up by checkers later.

Checkers gain access to the program model by extending an abstract Patterns.Checker class. A settings file is provided to the query system that specifies what classes have what roles in a design pattern. Sets of classes for each participant in a design pattern are created and then added to a program model. The settings file also provides the name of the checker class that extends Patterns.Checker. This class is loaded at runtime using reflection and provided with an evaluator that provides access to the program model via queries and a lookup tables that allows IDs to be converted to names. The query system then calls a “run” method that is implemented in the checker class that allows the checker to begin checking for mistakes in the pattern implementation that is represented in the program model.

The checker can then query the program model by evaluating Alloy expressions that generate a set using set-builder notation. Typically, sets are built of classes that have some property that indicates that they do not correctly fulfill their role in a pattern. The resulting sets can be used to provide informative warnings to the developer. This is a rough example of how the set-builder queries are generally formed:

$$\{c \in classes \mid c \text{ is not implemented correctly}\}$$

4. Examples

This section demonstrates how design pattern implementation checkers can be created for the abstract factory and visitor patterns. Also, the flexibility of the system is demonstrated by a “checker” that does not actually check singleton pattern implementations but rather detects them.

4.1 Abstract Factory Pattern Checker

This example illustrates how Alloy expressions can be constructed that can be used to query a program model to obtain useful information about a design pattern implementation. Figure 2 shows the complete code for an abstract factory implementation checker. The custom checker

extends an abstract checker, Patterns.Checker, which implements functions that allow Alloy expressions to be evaluated and element IDs to be translated to names.

Figure 2: Complete Abstract Factory Checker

```
public class AbstractFactoryChecker extends Patterns.Checker {

    void error(String msg) {
        System.out.println("Error: " + msg + "\n");
    }

    public void run() {
        String result[];

        //make sure concrete products descend from abstract products
        result = lookup(eval(
            "{cp:ConcreteProduct|no ap:AbstractProduct|ap in cp.^(superClass + interfaces)}.id"
        ));

        if (result.length > 0) {
            error("These ConcreteProduct(s) do not descend " +
            "from AbstractProduct(s):\n" + join(result));
        }

        //make sure concrete factories descend from abstract factories
        result = lookup(eval(
            "{conFact:ConcreteFactory|no absFact:AbstractFactory" +
            "|absFact in conFact.^(superClass + interfaces)}.id"
        ));

        if (result.length > 0) {
            error("These ConcreteFactory(s) do not descend " +
            "from AbstractFactory(s):\n" + join(result));
        }

        //make sure at least 1 AbstractFactory method returns each AbstractProduct
        result = lookup(eval(
            "{ap:AbstractProduct|some af:AbstractFactory|no m:af.methods|m.returnType = ap}.id"
        ));

        if (result.length > 0) {
            error("These AbstractProduct(s) are not returned by any " +
            "methods in some AbstractFactory(s):\n" + join(result));
        }

        //make sure at least 1 ConcreteFactory can create a given ConcreteProduct
        result = lookup(eval(
            "{cp:ConcreteProduct|#{cf:ConcreteFactory|some m:cp.methods" +
            "|m.isConstructor = true and m in cf.methods.^callees} = 0}.id"
        ));

        if (result.length > 0) {
            error("These ConcreteProducts(s) are not created " +
            "by any ConcreteFactory(s):\n" + join(result));
        }

        //make sure only 1 ConcreteFactory can create a given ConcreteProduct
        result = lookup(eval(
            "{cp:ConcreteProduct|#{cf:ConcreteFactory|some m:cp.methods" +
            "|m.isConstructor = true and m in cf.methods.^callees} > 1}.id"
        ));

        if (result.length > 0) {
            error("These ConcreteProducts(s) are created by " +
            "multiple ConcreteFactory(s):\n" + join(result));
        }
    }
}
```

The abstract factory checker performs five checks in the “run” method. Each check is essentially a query that returns classes that participate in a pattern but that do not follow the pattern correctly. The checks are performed by calling the “eval” method to obtain a list of IDs for offending classes. The IDs are then passed to the “lookup” function, which returns an array of the names corresponding to the IDs. If the array of names contains at least one element, then the pattern implementation being checked must be incorrect, and an appropriate error message is printed that indicates which classes are wrong.

The first query obtains a list of classes that the user indicates are concrete products but that do not descend from abstract products. The query begins with “cp:ConcreteProduct” followed by a vertical bar and a condition. This is Alloy's way of building a set of all ConcreteProduct(s) that meet the requirements after the bar. The vertical bar can be read in English as “such that”.

The requirement after the bar is that there are no Abstract factories that meet a second requirement which is also preceded by a vertical bar. Numerous sub-requirements can be strung together in this way. The requirement following the second vertical states that for the “ap” is that the “cp” being considered for inclusion in the set either descends from or implements the “ap”. This relationship is expressed as the combination of two other relationships: “superClass” and “interfaces”. The “+” creates a union of these two relationships, so “someclass.(superClass + interfaces)” would return the set of all the classes and interfaces that “someclass” either directly extends or implements. The “.^”, which is Alloy's transitive closure operator, is similar to the basic “.” operator, but it includes relationships that involve multiple links instead of just a single link. For example, “someclass.superClass” would get the class that “someclass” extends, but “someclass.^superClass” would retrieve the set of all classes that “someclass” descends from. Thus, the full expression, “cp.^(superClass + interfaces) ”, returns all the classes and interfaces that are ancestors of “cp”.

Now that most of the details of the Alloy query has been described, it should be clear that the part of the Alloy expression that includes the curly braces and everything in between will generate the set of all concrete products that do not descend from abstract products. Furthermore, the “.id” at the end of the expression will apply the ID relation to the set, effectively converting the set of classes to the set of those classes' IDs. The IDs can then be processed by the lookup function to obtain class names.

The second check is very similar to the first check, except that it is looking for concrete factories that do not descend from abstract factories. The third check merely looks for abstract products that are not returned by any of the methods in the abstract factory. The last two checks are interesting though, as they make use of the call graph. Transitive closure is used here again to discover what methods are reachable from a given method with this expression: “somemethod.^callees”. Together, the last two checks make sure that each concrete product can be constructed by one and only one concrete factory. This is necessary if the implementation organizes the concrete factories and concrete products into distinct families as proscribed by the abstract factory pattern.

4.2 Visitor Pattern Checker

Visitor patterns are used to group similar functionality that would otherwise be scattered among many classes. This approach is commonly used for ASTs (Abstract Syntax Trees) to group functions for pretty printing, type checking, and other functionality that would otherwise be distributed across the AST nodes in the tree.

Figure 3: Visitor Checker Snippet

```
//each concrete element should have an accept
//method that calls a visit method in the
//visitor
result = lookup(eval(
    "{ce:ConcreteElement|some v:Visitor” +
    “|no m:v.methods|m in ce.methods.^callees” +
    “ and ce in m.parameters.localType}.id"
));

if (result.length > 0) {
    error("These ConcreteElement(s) do not” +
    “have an \"accept\" method that calls “ +
    “a \"visit\" method in some Visitor(s):\n" +
    join(result));
}
```

The most important participant groups in a visitor pattern are Visitor, ConcreteVisitor, Element, and ConcreteElement. In the common AST case, a ConcreteElement is simply an AST node, and Element is just a class that all the AST nodes descend from. The ConcreteVisitor contains the functions that perform some specific functionality on each of the AST nodes. Each ConcreteVisitor descends from a Visitor, which defines an interface for invoking some function for each of the AST nodes. Each AST node that participates in the pattern must be able to "accept" a visitor. To do this, each AST node must implement an "accept" method that receives a visitor as an argument, and passes itself to an appropriate "visit" function in the visitor.

An incorrect implementation of a visitor pattern might not have an accept method in one of the AST nodes that calls the visit method for its visitor as it should. For pretty printing, such a mistake might cause the textual representation of some of the AST nodes to not be displayed. If the bad AST node is for something like type qualifiers, such a mistake might go unnoticed because the printed output would still look like it represented a valid program that simply lacked modifiers.

The code snippet in Figure 3 looks for AST nodes that do not have a correctly implemented accept method. Basically, the Alloy expression that is evaluated says, "give me a list of all the AST nodes such that there is a visitor that does not have a method that is reachable from the AST node and that takes the AST node as an argument."

The checker containing the check in figure 3 was run on a visitor pattern implementation for an AST in the Kodkod constraint solver. Kodkod is used by the Alloy tool for model checking as well as by other applications [5]. The Kodkod visitor pattern implementation for an AST was initially correct; however, modifying the code to introduce a mistake resulted in a bad implementation and an incorrect program that could be compiled and run without any obvious misbehavior. The mistake was introduced by simply commenting out a single line in the Decls AST node's accept method that makes a call to a visitor's visit method. Running the checker on Kodkod's visitor pattern took about 21 seconds and resulted in the checker correctly identifying the offending AST node without raising any false positives. There were 28 participating classes in the pattern implementation. The 28 classes had a total of 246 methods. The test was run on a laptop running Ubuntu Linux with 1GB of RAM and a 1.73GHz processor.

4.3 Singleton Pattern Detector

One of the primary benefits of doing pattern checking by allowing individual checkers to query a program model is flexibility. In fact, this approach is so flexible that it can also be used to detect some patterns in addition to just checking for mistakes in them.

Figure 4 provides an example of this, giving the complete code for a singleton detector. The detector works by using set-builder notation to create a list of all classes that meet certain requirements. The "req1" requirement simply states that any classes considered must be complete. This means that the class was actually specified as being part of the program and was not a library that was automatically included in the model. Incomplete classes just have class hierarchy information and leave out any details about methods or fields. The "req2" states that for all methods in a singleton, if a method is a constructor, it must be declared to be either private or protected. This is because making a constructor available to code outside of the class would allow the creation of multiple instances of the singleton in violation of the pattern. "req3" just says that singletons must have a static access method. Finally, "req4" requires that singletons keep a static reference to the instance of the class.

The singleton detector was run on all of the classes that are part of the program query system itself which consists of 21 classes and 103 methods. The only singleton implemented in the system was detected correctly, and no false positives were suggested. Running the detector took about 20 seconds to run on a laptop running Ubuntu Linux with 1GB of RAM and a 1.73GHz processor.

Figure 4: Complete Code for Singleton Detector

```
public class SingletonFinder extends Patterns.Checker {
    void error(String msg) {
        System.out.println("Error: " + msg + "\n");
    }

    public void run() {
        String result[];

        String req1 = "c.isComplete = true";
        String req2 = "all m:c.methods|m.isConstructor = true " +
                "implies (privateModifier in m.methodModifiers " +
                "or protectedModifier in m.methodModifiers)";
        String req3 = "some m:c.methods|m.returnType = c and staticModifier in m.methodModifiers";
        String req4 = "some f:c.fields|f.fieldType = c " +
                "and (staticModifier + privateModifier) in f.fieldModifiers";

        result = lookup(eval(
            "{c:class|("+req1+") and ("+req2+") and ("+req3+") and ("+req4+")}.id"
        ));

        if (result.length > 0) {
            System.out.println("Found potential singletons:\n" + join(result));
        } else {
            System.out.println("No singletons found!");
        }
    }
}
```

5. Related Work

There are many tools available that are designed to automatically generate code that implements some pattern. Examples include ModelMaker [7], PatternBox [8], one of the tools in IBM's Rational Software Architect suite [9], and the Design Pattern Automation Toolkit [6]. Many of these tools are plugins for IDEs such as Visual Studio (ModelMaker) or Eclipse (PatternBox and IBM's tool are examples). Initially generating code automatically probably significantly reduces the mistakes in pattern implementations. However, code changes over time, so being able to check for mistakes in implementations is likely also useful.

The DisCo (Distributed Co-operation) language was originally designed to be a means of formally describing reactive systems. In [4], it has been shown that DisCo is expressive enough to describe design patterns. Using DisCo to describe patterns was motivated by a desire to formally reason about various patterns and to make it possible to use theorem provers to prove temporal properties about patterns. However, DisCo is a very expressive language, and as such it would be difficult for pattern specification writers to learn. Also, there are not tools available that allow pattern implementations to be checked against Disco pattern specifications.

Using an extension to the Prolog language called Prolog-SandD to describe patterns is discussed in [2]. Also, SanD, a less expressive higher level language that can be compiled to Prolog-SanD, is also briefly described. The SanD tool is used to detect patterns in Java programs by using a combination of static and dynamic analysis. First, the tool statically looks for possible patterns in the Java source code. This naturally comes up with a lot of potential matches. The tool weeds many of these potential matches out by running the program and dynamically observing how various objects interact. Checking for pattern implementation mistakes could conceivably be done by running a SanD pattern detector and seeing if it detects a pattern that is being implemented, but this approach would have several disadvantages. If it did not detect the pattern that was being implemented, there would be no explanatory error messages. Also dynamic analysis might have trouble recognizing patterns in some cases. For instance, will it recognize that a class is a concrete factory in an abstract factory pattern if that concrete factory is never used during the runtime analysis? Using SanD for pattern detection seems useful, but adapting it for pattern checking would probably not be effective.

The PQL (Program Query Language) discussed in [3] lets users write queries to obtain facts about a program. The tool works statically by doing flow analysis and letting the user make queries about sequences of events. Similarly, dynamic analysis is done by instrumenting the code and looking for scenarios that match queries. PQL basically provides a simple way for software engineers to write custom specifications about their programs. According to the PQL paper, "the focus of PQL is to track method invocations and accesses of fields and array elements in related objects." The query system described in this paper similarly allows users to write queries about a program using Alloy's set-builder notation. The query system described in this papers differs, however, in that it has a focus on the relationships between classes, hopefully making it more useful for finding mistakes in design pattern implementations.

6. Future Work

Generating a call graph by assuming a method invocation could be invoking any methods that override the callee is often an unnecessary assumption. For instance, if an object has just been constructed, it should be clear what methods are being referred to in the immediately following code. A better call grapher that is aware of this could generate a sparser call graph, which could help identify more mistakes in programs since it would be possible to identify more cases where one method is not reachable by another method even though it should be.

Section 4.2 described how the visitor pattern checker was used on Kodkod. Originally, the intention was to find a pattern implementation in Alloy and then test a pattern checker on it. The grep program was run with the words “design pattern” on the source code for Alloy, which included Kodkod's source code. The visitor pattern in Kodkod was discovered and was then used for testing. While doing testing with Kodkod it was learned that Kodkod is a constraint solver that can evaluate Alloy expressions and that Alloy actually uses Kodkod to evaluate its own expressions. Kodkod is intended to be used in projects such as the one described in this paper that need to evaluate Alloy expressions. Currently, the query system uses code which was ripped out of Alloy's GUI code and drastically modified to allow the ability to automatically evaluate expressions without going through a user interface. The solution is cumbersome, and using Kodkod directly would be much cleaner. It would also probably be possible to avoid the generation of the temporary XML core instance, a step that significantly increase the time necessary to run a pattern checker.

Another way to improve performance would be to allow multiple pattern checkers to run without reloading the classes with Soot each time. For Java programs with overlapping patterns, a significant amount of time could be saved by only parsing the .class files once.

One feature Alloy has that is not utilized by the query system is the ability to write functions. It would be convenient for users if they were allowed to define functions in their checkers. Also, for some checks that are very common, such as the one that makes sure one class descends from another, simple functions could allow the checks to be expressed more compactly and intuitively.

One relationship that should certainly be a part of the program models but is not is the overrides relationship. It is possible to tell which classes extend which other classes, but it is not possible to tell which methods in a subclass override methods in a superclass. The abstract factory checker in section 4.1 cannot perform as many checks for this reason. The checker merely checks that each abstract product is returned by an abstract factory and that each concrete product is returned by a concrete factory. It would also be nice to check that the methods in the concrete factories that return concrete products override the methods in the abstract factory that return abstract products. Such a check is not possible without the overrides relationship.

7. Conclusion

The examples in this paper demonstrate the ease with which concise checks can be written for pattern implementation checkers. The approach is flexible, and many developers are already familiar with FOL, so users of the system should be able to readily create new patterns or adjust existing patterns according to their needs. Many of the checkers for the patterns in the gang of four book have been created, and creating more is straightforward.

While it is not possible to write checkers that catch all possible mistakes for any of the patterns, it is possible to catch many mistakes and to perform at least some checks for all of the patterns. For patterns where the checks are primarily limited to class hierarchy information, the query system might not be extremely useful, but most patterns have more interesting checks that can be done on individual methods and their relations to other methods. If some of the improvements in the future work section are implemented, it is possible that the query system could become a useful tool during software development.

References

[1] Erich Gamma , Richard Helm , Ralph Johnson , John Vlissides, Design patterns: elements of reusable object-oriented software, pp xv, 195-206, 87-95, 127-134, 331-344, 1995

[2] D. Heuzeroth, S. Mandel, and W. Lowe. Generating Design Pattern Detectors from Pattern Specifications. Automated Software Engineering, 2003.

[3] M. Martin, B. Livshits, and M. Lam. Finding Application Errors and Security Flaws Using PQL: a Program Query Language. OOPSLA, 2005

[4] T. Mikkonen. Formalizing Design Patterns. 1998.

[5] E. Torlak and G. Dennis. Kodkod for Alloy Users. First ACM Alloy Workshop. Portland, Oregon, 2006

[6] Design Pattern Automation Toolkit website, http://dpatoolkit.sourceforge.net/, 2008

[7] ModelMaker website. http://www.modelmakertools.com/modelmaker/design-patterns.html, 2008

[8] PatternBox website, http://www.patternbox.com/, 2008

[9] Rational Software Architect, http://publib.boulder.ibm.com/infocenter/rtnlhelp/v6r0m0/index.jsp?topic=/com.ibm.xtools.pttrn.author.doc/topics/c_design_ptrns.html, 2008