Automatic verification of design patterns in Java

Alex Blewitt
Alex.Blewitt@bandlem.com

Alan Bundy
A.Bundy@ed.ac.uk

Ian Stark
Ian.Stark@ed.ac.uk

School of Informatics
University of Edinburgh
Appleton Tower
Edinburgh
EH8 9LE
UK

ABSTRACT

Design patterns are widely used by designers and developers for building complex systems in object-oriented programming languages such as Java. However, systems evolve over time, increasing the chance that the pattern in its original form will be broken.

To verify that a design pattern has not been broken requires specifying the original intent of the design pattern. Whilst informal descriptions of design patterns exist, no formal specifications are available due to differences in implementations between programming languages.

We present a pattern specification language, SPINE, that allows patterns to be defined in terms of constraints on their implementation in Java. We also present some examples of patterns defined in SPINE and show how they are processed using a proof engine called HEDGEHOG.

The conclusion discusses the type of patterns that are amenable to defining in SPINE, and highlights some repeated mini-patterns discovered in the formalisation of these design patterns.

1. INTRODUCTION

Design patterns are a way of implementing a common solution to a common problem in object-oriented software. Many informal catalogues exist [16, 31, 4, 6] explaining how they are used and implemented. However, for all the informal descriptions of design patterns, there is still no standardised way of formally defining what a design pattern consists of, or how it can be automatically processed with verification or developer tools.

This paper presents a way in which design patterns may be represented as constraints on their implementation in the Java language [17]. This allows a proof tool HEDGEHOG to determine whether the pattern is correctly implemented. The paper follows on from an earlier short paper [3] which presented an outline of the approach and tentative results; in this paper, the SPINE language is presented along with a more detailed analysis of the results from [2].

ASE 2005 Long Beach, California, USA

2. REPRESENTING DESIGN PATTERNS

Although design patterns may be used in any object-oriented programming language, the implementation of the design pattern in each language will be different. In some cases, these differences will be minor; but the implementation of some patterns may take advantages of certain language-specific features that are not available in others. For example, implementing a *Proxy* pattern in Objective-C or Smalltalk can take advantage of dynamic faulting, which languages such as C++ and Java do not have.

A formal design pattern definition can therefore be at two levels:

1. The pattern definitions can focus solely on the object-oriented features of the pattern, and avoid language-specific issues
2. The pattern definitions can be targeted towards a single language and take advantage of language-specific features

The advantage of a higher-level definition focussing on object-oriented features alone means that it will be applicable to more target languages. Approaches such as [8, 9, 22, 18] provide pattern languages that focus on the object-oriented features of design patterns.

The advantage of a lower-level definition is that as well as object structure playing a part in the definition of the design pattern, it is possible to use knowledge of the language in order to perform extra checking on the implementation of the design pattern.

As well as deciding what level to focus on, there are three different ways of encoding patterns, depending on the use of the tool dealing with them. For example, a tool that deals with manipulation of patterns may have a different set of requirements than one that is used for verification.

2.1 Behavioural definition

One way of specifying a design pattern is to define its behaviour. It would then be possible to use this definition for automated verification purposes, provided that a suitable proof tool is available.

Design by contract [24] uses this approach to ensure the correct implementation of code. Each method has a pre- and post-condition that can be used to constrain the correct behaviour of the methods, as well as a class invariant that ensures that the instance's key properties are not violated. Tools such as iContract [20] provide this behaviour checking in Java, and Eiffel [23] provides this as part of the base language. Other tools, such as [7, 21, 28] allow arbitrary constraints to be checked against implementations of code.

However, this approach does not work well for pattern definition. Partially this is because patterns are often above the method layer, dealing with how classes are related, but mainly because behaviour alone is not enough to capture the essence of a design pattern. Indeed, refactoring existing code to introduce design patterns [25, 30] is an active area, and since refactoring, by definition, [26] does not change the existing behaviour, a pattern definition based solely on semantics would not be able to tell the difference between the code prior to the refactoring or the code afterwards.

2.2 Metaprogramming definition

Another approach to defining design patterns is to represent them as metaprograms [25, 10, 30]. In these cases, a design pattern is represented as a metaprogram that, when executed, will instantiate or transform existing code into one that exhibits a pattern.

Clearly, this approach is well suited for a tool that needs to instantiate new patterns in code, or alternatively refactor the code so that a pattern evolves out of the existing structure. However, care must be taken when using a metaprogramming approach if the pattern application is re-applied; a simple metaprogram for instantiating the *Singleton* pattern may repeatedly add instances to its target class, for example, thus accidentally violating one of the principles of the *Singleton* pattern that only a single instance exists.

By definition, the pattern is encoded within the metaprogram, which makes it potentially difficult to relate the definition to the resultant pattern instance. As well as the pattern itself, a certain amount of error-handling code may be present in the metaprogram as well, which can hide further the specification of the pattern. This method tends to be coupled tightly with the implementation of the metalanguage being used to implement the transformations; so examples in Smalltalk are common, because the Smalltalk environment allows classes to be changed dynamically at run-time.

Lastly, the metaprogram normally has some kind of default method names or fields to instantiate, which works fine for a green-field implementation of the pattern; but for existing patterns that are being verified or modified can present a problem.

2.3 Declarative constraints

The last way of representing a design pattern is to define a set of declarative constraints on the implementation of the pattern itself. These constraints can be used to define both static (inheritance) and dynamic (method invocation) behaviours of the class (or classes) involved in the design pattern.

The ability to represent design patterns as constraints has been considered previously [1, 3] and is used in some tools [22] to define design patterns. The benefit of using an external declarative language to define design patterns allows the specification and the tool that processes the specification to be separate. As a result, different tools can be created to process the same specification, but have different results – such as an automated verification tool or a repairing/refactoring tool.

Since this approach works for the verification of design patterns, and the other two are less well suited for verification, HEDGEHOG uses this form of pattern representation.

3. THE SPINE LANGUAGE

Patterns can be defined in a Prolog-like language called SPINE that allows patterns to be defined in terms of built-in functions and predicates. It supports simple existential qualifiers and allows iteration over both class structure (classes, interfaces, methods) and method implementations (determining if a method instantiates a class, or writes a field definition etc.). This allows a pattern to be defined both in terms of its structure and also its behaviour in the same specification.

Terms in SPINE are either variables (which begin with capital letters such as '`F`'), lists (enclosed with '`[`' and '`]`') or compound terms (which begin with a lower case letter such as '`and(A,B)`'). Rules define compound terms as a comma separated list of terms, which is interpreted by HEDGEHOG as a logical conjunction. In addition, built-in terms such as '`and`' and '`or`' allow arbitrary lists of terms to be combined, and the terms '`true`' and '`false`' can be used.

The special-case quantifiers '`forAll`' and '`exists`' operate on a list of terms, and are equivalent to a conjunction and disjunction respectively of generated terms. As an example, '`forAll([1,2],X.F(X))`' is logically expanded to '`and([F(1),F(2)])`'. These are only used over finite lists, and so are decidable.

Figure 1: SPINE **definition of** ***Immutable***

```
realises('Immutable',[C]) :-
  forAll(fieldsOf(C),
    F.or([
      isStatic(F),
      isFinal(F),
      and([
        hasModifier(F,private),
        forAll(methodsOf(C),
          M.not(modifies(M,F)))
      ])
    ])
  )
```

Figure 1 shows a SPINE definition of the *Immutable* pattern. This pattern requires that there are no modifiable instance fields in the given class. This can be easily verified provided that every field is either `static` (in which case, it is not an instance field) or `final` (which by definition cannot change). Additionally, any non-`private` fields would be open to changes by external objects, so non-`static` and non-`final` fields must be `private`. But as well as requiring that the fields are `private`, it is necessary to ensure that no methods defined in the class can modify the field. The meanings of the terms '`fieldsOf`' and '`modifies`' are covered in Section 4.

Figure 2 shows a SPINE definition and Figure 3 shows a Java example for the *Singleton* pattern:

Figure 2: SPINE definition of public *Singleton*

```
realises('PublicSingleton',[C]) :-
  exists(constructorsOf(C),true),
  forAll(constructorsOf(C),
    Cn.isPrivate(Cn)),
  exists(fieldsOf(C),F.and([
    isStatic(F),
    isPublic(F),
    isFinal(F),
    typeOf(F,C),
    nonNull(F)
  ]))
```

Figure 3: Java example of public *Singleton*

```
public class PublicSingeton {
  public static final PublicSingleton
    instance = new PublicSingleton();
}
```

The two-argument predicate '`realises`' specifies a pattern definition (in this case, '`PublicSingleton`') that can be matched against any class '`C`'. The function '`constructorsOf`' can be evaluated to produce a result of constructors, which can then be used in the remainder of the definition. Similarly, '`fieldsOf`' produces a list of fields defined in the given class. The '`forAll`' and '`exists`' predicates operate over these lists, binding each value to the variable in the expression.

In this case, the SPINE definition says that a class is a true *Singleton* iff:

- There is at least one constructor
- All constructors are `private`
- There is a field that is `static`, `public`, `final`, is the type of the enclosing class, and is non-`null`

Of course, this is a valid definition for only a single variant of *Singleton*; it is not the only way that it can be implemented. For example, the singleton instance could be instantiated through lazy instantiation, or there could be a method that accesses a `private static` instance instead.

3.1 Variants

Although the intent of patterns remains the same, there are several different ways to implement them. These may differ in small ways (such as whether a variable is initialised in the constructor or as a default value) or large ways (by using a completely different variety of the same pattern). Catalogues such as [16] often have a number of implementation notes regarding each pattern that allow the same pattern to evolve in slightly different ways.

Sometimes these differences just deal with the way that object instances are created: for example, whether they are instantiated at load-time or at request-time. Or there might be different sorts of data structure (lists, hashtables), that might give different performance, based on how the pattern is intended to be used.

We can capture these variants by defining additional rules for matching these patterns, as in Figure 4:

Figure 4: SPINE definition of lazy *Singleton*

```
realises('LazySingleton',[C]) :-
  exists(constructorsOf(C),true),
  forAll(constructorsOf(C),
    Cn.isPrivate(Cn)),
  exists(fieldsOf(C),F.and([
    isStatic(F),
    isPrivate(F),
    typeOf(F,C),
    exists(methodsOf(C),M.and([
      isStatic(M),
      isPublic(M),
      lazyInstantiates(M,F)
    ]))
  ]))
```

Figure 5: Java example of lazy *Singleton*

```
public class LazySingeton {
  private static LazySingleton
instance;
  public static LazySingleton
getInstance() {
    if (instance == null) {
      instance = new LazySingleton();
    }
    return instance;
  }
}
```

Figure 5 shows an alternative *Singleton* realisation, this time using lazy instantiation to set up a static instance variable. Much of the pattern is the same as before, but instead of requiring a `public` instance field, we have a `public` method that instantiates the field on demand and then returns it. The predicate '`lazyInstantiates`' is covered below.

In order to tie these two variants into the same pattern, we can define rules that express the fact that a *Singleton* is either a *PublicSingleton* or a *LazySingleton*, as shown in Figure 6:

Figure 6: Combining the *Singleton* variants

```
realises('Singleton',[C]) :-
  or([
    realises('PublicSingleton',[C]),
    realises('LazySingleton',[C])
  ])
```

This allows code to be checked against a group of variants at the same time. Instead of having to use a specific variant (e.g. '`realises('PublicSingleton',['C'])`' a group name

may be used (e.g. '`realises('Singleton',['C'])`'. It is even possible to group separate patterns together that have a similar intent; for example, both *Singleton* and *Utility* have a similar intent, and so could be grouped into a *SingleAccess* group.

4. THE HEDGEHOG PROOF ENGINE

The HEDGEHOG proof engine reads the SPINE definitions, along with Java source code, and attempts to automatically prove whether or not the class correctly realises the design pattern. The proof is entirely automated; the user simply asks whether or not a class meets a specific design pattern, and the answer is automatically returned. The functionality has been built into Eclipse giving a menu of design patterns that allows the user to determine if the currently selected Java class meets the design pattern selected.

The automated proof system reads in the goal to prove, such as '`realises('Singleton',['ExampleSingleton'])`', and constructs a goal for it. It then repeatedly applies rules from the SPINE patterns library, to build up a proof tree for the class. Built-in predicates, such as '`constructorsOf()`', derive the information from an internal Java parser, which caches the class for efficiency of later use.

By the end of the proof, the system has either been successful (in which case, it prints out that the class does realise the given pattern), or unsuccessful (that the class does not meet the pattern). A post-processing mechanism then translates the failed proof tree into an understandable error message.

4.1 Static semantics

Since one of the key parts in a design pattern is its relationship with other classes, it is necessary to be able to define predicates and functions that will allow HEDGEHOG to interrogate Java classes. These static semantics can be derived directly from the object-oriented structure of the Java code:

exists($List$,$X.P(X)$) at least one X in $List$, then $P(X)$ holds; if $List$ is empty, then fail

forAll($List$,$X.P(X)$) for every X in $List$, then $P(X)$ holds; if $List$ is empty, then succeed

hasModifier(X,M) X has modifier M

implements(C,I) class C `implements` interface I

implies(A,B) the logical equivalent of or(not(A),B)

isAbstract(X) X is a field, method, or type that is `abstract` – uses **hasModifer**

isClass(T) T is a class

isConstant(E) E does not change value

isFinal(X) X is `final` – uses **hasModifer**

isFriendly(X) X is `friendly` – uses **hasModifer**

isInterface(T) T is an interface

isLiteral(E) E is a literal Java expression

isPrivate(X) X is `private` – uses **hasModifer**

isProtected(X) X is `protected` – uses **hasModifer**

isPublic(X) X is `public` – uses **hasModifer**

isStatic(X) X is `static` – uses **hasModifer**

isSideEffectFree(E) E does not change the object's state

prefix(M,$name$) method M begins with the prefix $name$

sameSignature(M_1,M_2) method M_1 has the same signature as method M_2

subtypeOf(X,T) X is a sub-type of T (or equal to T)

typeOf(F,T) F is a field of type T

typeOf(M,T) M is a method that has a declared return type of T

Since these these predicates over a finite lists (e.g. a Java class has only a finite number of constructors) the logic is essentially propositional and thus decidable.

4.2 Weak semantics

As well as requiring a pattern to have the correct collaborators (supertypes, additional classes and so forth), it is also necessary that the methods have the correct behaviour: they must be defined with knowledge of the semantics of the Java class.

Although a full Java semantics would be able to prove certain properties about the class in question, it turns out that it is not necessary to have a full semantics. In much the same way that ESC/Java[21] can perform static analysis of code, so too can the implementation of HEDGEHOG. Although neither can prove everything, they can produce useful results. Two key points from [13] are just as applicable to HEDGEHOG:

1. Although program proofs are undecidable in general, "the kinds of programs that occur in undecidability proofs rarely occur in practice." Whilst recursion (and determine whether a program will terminate) are hard mathematical problems, these kind of issues do not appear in design patterns.

2. Although it is desirable for an ideal automated system to be both sound and complete, it is not an absolute requirement. "The competing technologies (manual code reviews and testing) are neither sound nor complete ... if the checker finds enough errors to repay the cost of running and studying its output, then the checker will be cost-effective and a success."

The following weak semantic predicates are built-in to HEDGEHOG:

adds(M,$Type$,$Collection$) the method M adds an instance of $Type$ to $Collection$

instantiates(M,T) method M creates an instance of T and returns it, although the method may have a declared return type of T or one of its super-types

invokes($Method$,$Delegate$) code in $Method$ invokes the method called $Delegate$

invokes($Method$,$Delegate$,$Field$) code in $Method$ invokes $Delegate$ on $Field$

lazyInstantiates(M,F) M lazily instantiates F and returns it

modifies(M,F) method M modifies the value of field F

navigable(C_1,C_2) it is possible to navigate between C_1 and C_2

nonNull(F) the field F is non-null (i.e. it has been assigned an instance, either in the default field initialisation or in the constructor)

removes(M,$Type$,$Collection$) method M removes an instance of $Type$ from $Collection$

returns(M,F) method M returns the value of field F

So how are these weak semantic predicates proved by HEDGEHOG? The answer lies in the contents of the Java methods. It is worth noting that HEDGEHOG is neither sound nor complete; it does not perform loop unbundling or determine which branch of a conditional to process, but rather assumes all possible call paths throughout a method. In other words, in a loop statement it assumes that the loop executes once, and that in an if statement, both the true and false branches are executed. This allows HEDGEHOG to determine potential reachability of expressions and methods, and uses these to determine whether a predicate can be satisfied or not. A more advanced static analysis tool may yield more accurate results at the code level; but this would be unnecessary for most SPINE pattern definitions.

As an example, the '`nonNull`' predicate is defined to return `true` if the field F has been assigned a non-`null` value. It does this by ensuring one of the following:

1. The field F is initialised with an in-line initialiser with a non-`null` expression
2. There is at least one constructor, and all constructors ensure that field F is assigned a non-`null` expression
3. The field F is assigned the value of a parameter in the constructor, and the value is guarded with an `if` test that throws an exception if the parameter is `null`

A non-`null` expression is one of the following:

1. A call to `new` Type()
2. A call to a method that returns a `new` Type()
3. A reference to a field or local variable that has been initialised with a non-`null` value

Although this is very conservative, it is possible to use these definitions to match most of the instances where the '`nonNull`' predicate is used. The important factor is that the whole process is decidable, and only uses the static Java code as its basis.

A similar argument holds for the '`modifies`' predicate. This checks to see whether a method can potentially change the value of a field in the current class. In the case of a conditional, the proof system assumes that both parts are called; in other words, it traces out the potential call paths through a method and condenses them into one possibility. This means the process is not sound; for example, we can imagine a code statement such as '`if (true == false) {f=a;}`' which would clearly never happen, but the proof system assumes that the value of '`f`' may have been modified. This may result in some false negatives being raised. However, such code does not tend to exist in programs.

In the case of the '`modifies`' predicate, the proof process iterates through the abstract syntax tree of the method in question determining whether there are any assignments in the tree. If there are assignments, it resolves them to either a local variable or a field; and in the case of a field, compares it to see if it is the same field as the '`modifies`' predicate is looking for.

This process recurses through called methods; if m_1 calls m_2, and the proof system is trying to prove $modifies(m_1, f)$ then it will also try to prove $modifies(m_2, f)$.

Any code wrapped in a conditional block (`if` or `switch`) is assumed to potentially happen; similarly, looping operators (`for` and `while`) are assumed to happen at least once.

5. RESULTS

The HEDGEHOG proof engine has been tested with samples obtained from the Java language source code, as well as using [16] to provide a list of candidate patterns for representation in SPINE.

Of the 24 patterns defined in [16], a total of 7 of them could not be represented suitably in SPINE definitions. These were: *Builder*, *Façade*, *Chain of responsibility*, *Command*, *Interpreter*, *Mediator* and *Memento*.

5.1 Unrepresentable patterns

The main reason for not being able to represent these patterns in SPINE is that a suitably abstract definition could not be found. A definition for one of these patterns would either be too narrow (and thus produce a number of false negatives) or too vague (and thus produce a number of false positives). For example, the *Command* pattern is one of the more common patterns, but does not have a suitable SPINE definition. The main problem is that the *Command* pattern is very simple; there is an `abstract` class, with an `abstract` method, and a number of subclasses. This results in either a very vague pattern definition (matching almost any `abstract` class) or a very specific pattern definition (where the class name is constrained to be '`Command`'). Neither case works well for verification purposes; although the more specific may work for a pattern instantiation tool.

Others are difficult to define because they do not have a clear realisation; for example, the *Memento* pattern is very difficult to determine whether it is present or not. The difficulty in representing a pattern is inversely proportional to the number of identifying artefacts the pattern has. In the case of the *Memento*, a class with a simple data structure such as a `Map` may be sufficient; but that does not mean that all classes that hold a `Map` are realisations of the *Memento* pattern.

What all SPINE pattern specifications fail to do is capture the *intent* of the pattern; how (and where) it is used. The *Proxy*, *Adapter* and *Decorator* patterns are examples of how important the intent of a pattern is; all of them are superficially similar to one another, and often have a single collaborating instance to which they forward one (or more) messages. However, the intent of the *Proxy* pattern is to

forward the requests through some protocol (such as over a network); the *Adapter* and *Decorator* intercept methods between the caller and the delegation class and "work together" [16, page 139]. However, "work together" is neither well defined, nor definable. This point is explicitly raised in [16, pages 219–220] when comparing *Adapter* and *Bridge*: "the key difference between the patterns lies in their intents. *Adapter* focuses on resolving incompatibilities between two interfaces; *Bridge* bridges an abstraction and its (potentially numerous) implementations."

5.2 Results table

Pattern examples were taken from four sources: Applied Java Patterns [27]; the Eclipse Pattern Box[11]; and the Java language source code, versions 1.1 and 1.2. The sources were manually scanned to find instances of the design patterns, and then HEDGEHOG was asked to declare whether the pattern was present or not. The results are shown in Table 1.

Table 1: Results

Pattern	AJP [27]	PB Eclipse	Java 1.1	Java 1.2
Creational				
Abstract Factory	√+	√+	√+	√+
Factory Method	√+	√+	√+	√+
Prototype	×-	×-	∅	∅
Singleton	√+	√+	√+	√+
Structural				
Adapter	√+	√+	√+	√+
Bridge	√+	∅	√+	√-
Composite	√+	√+	√+	√+
Decorator	√+	∅	×-	×-
Flyweight	√+	√+	√+	√+
Proxy	√+	∅	∅	×-
Behavioural				
Immutable	√+	√+	√+	√+
Iterator	√+	×-	√+	√+
Observer	√+	√+	√+	√+
State	×-	√+	∅	∅
Strategy	√+	×-	∅	∅
Template Method	√+	√+	√+	√+
Visitor	√+	∅	∅	∅

√+ True positive; the pattern was found and correctly identified

×+ False positive; the pattern was found where no pattern exists

√- True negative; the pattern was not found where no pattern exists

×- False negative; the pattern was not correctly identified where the pattern existed

∅ Unfound; no pattern definition could be found from the sample set to test against

The results are summarised in Table 2. Of the 24 pattern types, 7 could not be defined in SPINE. That leaves 17 patterns, of which there were 4 examples each (less 13 examples which could not be found in the samples) which gives a total of 55 pattern instances to be verified. Of these, 47 were correctly classified (46 true positives and 1 true negative), whilst 8 were incorrectly classified as not realising the design pattern (false negative). There were no false positives.

Table 2: Summary of results

	True (√)	False (×)	Others	
Positive (+)	46	0	**Unfound**	13
Negative (-)	1	8	**Unrepresentable**	7
Total	47	8		20

5.3 Analysis

A brief analysis of the results follows, with a commentary on each type of failure and the kinds of problems or successes that are associated with each. A more detailed account is presented in [2].

5.3.1 True positives

The true positives (√+) are the success of HEDGEHOG giving the correct answer. Most of the design pattern examples that could be found from the 16 that had pattern definitions could be matched with real uses of the patterns in the source examples. Out of the 55 pattern examples, 46 of them resulted in true positives – a success rate of 83%.

5.3.2 False positives

There were no false positives (×+) found in the results. This can be attributed to the fact that only suspected patterns were tried, rather than randomly trying to match any pattern against any implementation. However, note that not finding a false positive does not mean that they do not exist; there may be patterns that HEDGEHOG reports as being a realisation of a design pattern, but in fact are not.

Partially, the reluctance to implement vague pattern specifications (such as *Command*, as discussed earlier) has limited the number of false positives that could be recorded. Had vaguely defined patterns such as *Command* been provided, then the false positive rate might have been much higher.

5.3.3 True negatives

HEDGEHOG only found one true negative (√-); that of the *Bridge* pattern in Java's AWT. In Java 1.1, the Java AWT `Component` types had a one-to-one mapping with a `ComponentPeer` class. This was broken in the migration to Java 1.2, when Swing 'lightweight' components were created that had no corresponding peer class. In this instance, HEDGEHOG correctly reports that the pattern is broken in 1.2. Had this gone unnoticed, it might have caused problems. As it happens, it was a design decision to break the pattern, so not something to be worried about, although it does highlight HEDGEHOG's benefits as a pattern verification tool. Indeed, this is a success not only because HEDGEHOG correctly identified the result, but also that this validates the purpose of HEDGEHOG in identifying broken patterns over time.

5.3.4 False negatives

A false negative (×-) is one where HEDGEHOG could not correctly verify a design pattern that was actually present in a piece of code. Of the 55 examples, 8 were reported as false negatives.

Given that HEDGEHOG's use as a pattern verification tool that helps highlight pattern violations, a false negative is much less of a problem than a false positive. If HEDGEHOG is unable to prove that a pattern exists, it does not mean that it is not present; only that HEDGEHOG is not capable of deciding this.

One likely source of false negatives comes from patterns that are implemented by unknown variants. The approach taken in representing design patterns as constraints on their implementation is likely to capture some, but not all, variants of a particular design pattern. As such, HEDGEHOG does not aim to be complete, but provides extensible mechanisms to allow user-defined variants to be added at a later time. As HEDGEHOG's pattern library grows, this source of false negatives should diminish over time.

The other likely source of false negatives is to do with HEDGEHOG's proof system not being able to determine aspects, particularly to do with the weak semantics. For example, the *LazySingleton* pattern must ensure that when a method is called a non-`null` instance is returned. If that instance were to be obtained from deserialising an object, then the built-in predicate would not be able to deduce that the returned value is a non-`null` value. It is highly unlikely, but not impossible, that a *Singleton* may be implemented in this fashion, which means that HEDGEHOG would not be able to identify this as a correct implementation of a *Singleton*.

The examples of false negatives found in these results are due to the patterns being implemented with a different variant to the one defined in SPINE. For example, the *Prototype* pattern definition specified that the Java interface `Cloneable` is implemented by classes that are *Prototype* instances; however, the examples used a similar but distinct interface for representing the `cloneable` method. It could be argued that it is an incorrect implementation of the pattern; but more realistically, it is just a different variant of the *Prototype* pattern that is not encoded in SPINE.

6. RELATED WORK

A number of other projects focus on the formalisation of design patterns. A number of these are specifically focussed on the editing or instantiation of design patterns in existing code, either by creating them from an empty class, or by refactoring existing code into a design pattern.

Marco Meijers' Fragment Tool [22] provides a mechanism for representing design patterns such that they could be modified by a development tool. Its purpose was to facilitate the use of design patterns as elements that could be used for design and implementation, rather than focussing on the lower-level classes, methods and fields. Patterns are represented as fragments, which are associated with classes in the fragment browser.

When a new pattern is instantiated, a template set of fragments are associated with the given class. It is then up to the user to select which fragment roles are filled by which other classes/methods/fields; and gradually, the collaborators in the pattern are assembled into an interconnected set of fragments. Should the pattern require changing at any time, it is possible to edit the fragments directly, which then changes the underlying code. This approach used a combination of the metaprogramming and declarative specification for design patterns; and because the fragments have to exist prior to their use in the tool, existing code has to be appropriately marked up prior to its use.

The fragment tool was continued with a slightly different slant in [18] to provide a framework for working with design patterns. It also used metaprogramming to represent the design patterns, so that they could be instantiated by executing the metaprogramming code. It also attached handlers to the frameworks, so that if another user modified the code (say, by adding a subclass to a visitor pattern) then appropriate changes could be made to the code (say, by adding a method to the visitors). It also investigated the possibility of having an intermediate language to deal with language-agnostic design patterns; but concluded that it would be necessary to have at least 3 different target languages for such an intermediary language to make sense.

LePUS [9] proposes a different mechanism for specifying design patterns; by using a graphical representation called "LanguagE for Patterns' Uniform Specification". The graphical language allows relationships between patterns to be represented graphically, and includes similar basic primitives to that encoded in SPINE. For example, classes can be related with inheritance, and methods (or sets of methods) can be defined with call sequences between them. It is somewhat similar to UML [15] except that the latter is normally used for expressing only relationships between classes and method call sequences; and not specifically pattern specifications.

While LePUS is a useful tool for describing design patterns, it still has two features that limit its usefulness to verification of design patterns:

- LePUS is intended to be used without a specific target language in mind. As such, it cannot take advantage of any implementation tricks or techniques, nor verify the correct implementation of individual methods.
- The graphical language that defines LePUS does not lend itself well to representation in an automated proof system.

Other work on introducing design patterns to existing code took the metaprogramming approach for representing patterns [25, 29, 30]. These approaches aimed to convert existing code that had a specific behaviour into code that kept the same behaviour, but by using a design pattern.

The concept of refactoring [26, 14] is not specific to design patterns, but rather introduces a set of techniques for translating code from one implementation to another whilst preserving behaviour. Often, refactorings can be as simple as renaming a variable, but they can be as complicated as changing an algorithm (for example, changing a binary sort into a quick sort). In neither case is the behaviour of the code changed; but the newly refactored code may have other benefits (it runs faster; it is easier to maintain; it is easier to extend). This highlights why it is not possible to represent patterns based on their externally observed behaviour; both ugly and beautiful implementations may have exactly the same behaviour, but the pattern may only be considered to be present in the beautiful one.

Refactoring existing code into a design pattern is more difficult than instantiating a pattern from a template. Apart from anything else, the pattern application tool needs to know whether it is valid to try to convert a set of classes into a design pattern. The approach taken by [25] is to determine a precursor to the pattern; something which closely, but not completely, represents the design pattern. It can then be

incrementally improved until it meets the specification of the design pattern. It is expected that the user will drive the tool; and if there are names that need to be decided for method or fields that are being created, the user can be prompted to make a decision.

Other works concentrate on specifying the semantics of the Java language; tools such as ESC/Java [21, 13, 12] and iContract [20] allow statements to be made about existing Java code, and then checked for correctness. These are used in modelling languages such as JML [19, 5] which attempts to model the constraints of code implemented in the Java language. This is similar to the purpose of SPINE and its weak semantic constraints; a more extensible mechanism for defining additional weak semantic constraints may be able to benefit from such modelling tools in the future.

7. FURTHER WORK AND CONCLUSIONS

The results show that it is possible to specify patterns in terms of declarative constraints on their implementation, and that they can be used to verify the implementation of code examples from real systems. However, the results could be improved by focussing on the following areas:

- *Provide new pattern specifications.* Some of the false negatives are due to the fact that a different variant was used than the one specified in SPINE. Although it would be possible to fix these problems by defining a pattern variant for each false negative, this was not done in order to obtain an impartial set of results for this test. However, the purpose of any extensible design patterns library is to increase the size of the library, so it is possible to build up the SPINE library with other variants to capture these false negatives.

- *Extend the capabilities of the weak semantics.* Although there are relatively few patterns that specifically require a weak semantic predicate, it is necessary to be able to extend HEDGEHOG's capability over time. One way this might be achieved is to provide an extension point that would allow weak semantic predicates to be defined externally to the HEDGEHOG proof engine. Alternatively, other proof systems such as ESC/Java [21] could be used to provide invariant and assertion checking that is currently done internally in HEDGEHOG.

- *Provide template definitions for specific patterns.* Patterns such as *Command* and *Adapter* could be defined to expect use specific implementation names such as `Command` and `Adapter`. Although this may not be much use for pattern verification, it might provide a way in which these patterns could be recognised from existing systems.

There are also other uses for the SPINE pattern definitions that are not currently used by HEDGEHOG. For example, a pattern verification tool can be used as a pattern detection tool by brute-force searching of existing code bases. Although this would be an inefficient use of HEDGEHOG at present, the use of the SPINE patterns may help to provide hints as to where to start looking.

HEDGEHOG does not currently offer any automated resolutions when a pattern is not correctly realised, other than an error message to help guide the user. Given the detail of the pattern that is currently present, it should be possible to use the SPINE definitions to try to automatically repair (or suggest repairs to) the problems whilst preserving semantics. Taken to its extreme, a pattern system that could automatically repair a failed pattern could also be used to instantiate a pattern from scratch; although this would probably be an inefficient use of a verification tool. Furthermore, when the pattern requires the existence of a specific method (such as the accessor method in the *LazySingleton*) it would need to either use a pre-defined method name or be able to prompt the user for such a name.

Another possibility for extension is to migrate HEDGEHOG to use different target languages, instead of Java. It is expected that the SPINE pattern definitions will be mostly similar if a new target language is chosen, though it will require the re-implementation of HEDGEHOG's language parsing and implementation of the built-in predicates. However, there is no reason why this approach would not work on other target languages.

These results show that it is possible to define patterns as a set of constraints on their implementation, and that with a sufficient library of design patterns and their variations, be used to automatically verify the existence of patterns in code.

8. ACKNOWLEDGMENTS

The research reported in this paper is supported by an EPSRC CASE studentship in conjunction with Edinburgh University and International Object Solutions Limited.

9. REFERENCES

[1] A. Blewitt. A formal catalogue of design patterns. Technical Report Blue book note 1373, Division of Informatics, Edinburgh University, 7 2000.

[2] A. Blewitt. HEDGEHOG*: Automatic Verification of Design Patterns in Java*. PhD thesis, School of Informatics, University of Edinburgh, 2005. Unpublished PhD Thesis. `http://www.bandlem.com/Alex/Papers/PhDThesis.pdf`.

[3] A. Blewitt, A. Bundy, and I. Stark. Automatic verification of Java design patterns. In *ASE 2001: Proceedings of the 16th IEEE International Conference on Automated Software Engineering*, pages 324–327. IEEE Computer Society Press, November 2001. `http://www.ed.ac.uk/~stark/autvjd.html`.

[4] F. Buschmann. *Pattern-oriented Software Architecture: A System of Patterns*. Wiley, April 1996.

[5] Y. Cheon. *A Runtime Assertion Checker for the Java Modeling Language*. PhD thesis, Department of Computer Science, Iowa State University, April 2003.

[6] J. Coplien. *Pattern Languages of Program Design 1*. Pattern Languages of Program Design. Addison Wesley, June 1995.

[7] D. Detlefs, K. R. Leino, G. Nelson, and J. Saxe. Extended static checking. Technical Report 159, Systems Research Center, Digital Equipment Corporation, December 1998.

[8] A. Eden. LePUS - a declarative pattern specification language. Technical report, Tel Aviv University, 1998. `http://www.cs.concordia.ca/~faculty/eden/lepus/`.

[9] A. Eden. *Precise Specification of Design Patterns and Tool Support in Their Application*. PhD thesis, Department of Computer Science, Tel Aviv University, 2000. `http://www.eden-study.org`.

[10] A. Eden, J. Gil, and A. Yehudai. Precise specification and automatic application of design patterns. In *12th Annual Conference of Automated Software Engineering*, 1997.

[11] D. Ehms. Patternbox eclipse tool. `http://www.patternbox.com`.

[12] Esc/Java 2. web. `http://www.sos.cs.ru.nl/research/escjava/index.html`.

[13] C. Flanagan, K. R. M. Leino, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI)*, June 2002. `http://www.hpl.hp.com/personal/Mark_Lillibridge/ESCOverview/revised-p25-leino.pdf`.

[14] M. Fowler. *Refactoring: Improving the design of existing code*. Object Technology Series. Addison-Wesley, 2000.

[15] M. Fowler and K. Scott. *UML Distilled*. Addison-Wesley Professional, September 2003.

[16] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of reusable object-oriented software*. Professional Computing Series. Addison Wesley, 1995. ISBN 0-201-63361-2.

[17] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison Wesley, 1996. `http://java.sun.com/docs/books/jls/html/index.html`.

[18] D. Gruijs. *A Framework of Concepts for Representing Object-Oriented Design and Design Patterns*. PhD thesis, Department of Computer Science, Utrecht University, August 1998.

[19] B. P. F. Jacobs and E. Poll. A logic for the Java Modeling Language JML. Technical Report CSI-R0018, Computing Science Institute, University of Nijmegen, December 2000. `http://www.cs.kun.nl/csi/reports/info/CSI-R0018.html`.

[20] R. Kramer. iContract - The Java Design by Contract Tool. In *Proceedings of the Technology of Object Oriented Languages and Systems*. IEEE Computer Society, 1998.

[21] K. R. M. Leino, G. Nelson, and J. B. Saxe. ESC/Java user's manual. Technical Report 2000-002, Compaq Systems Research Center, October 2000. `http://gatekeeper.dec.com/pub/DEC/SRC/technical-notes/abstracts/src-tn-2000-002.html`.

[22] M. Meijers. Tool support for object-oriented design patterns. Master's thesis, Utrecht University, 1996. INF-SCR-96-28.

[23] B. Meyer. *Eiffel: The Language*. Prentice-Hall, 1999.

[24] B. Meyer. *Design by Contract*. Prentice Hall, 2002.

[25] M. Ó Cinnéide. *Automated Application of Design Patterns: A Refactoring Approach*. PhD thesis, University of Dublin, Trinity College, October 2000.

[26] W. F. Opdyke. *Refactoring Object-Oriented frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992. `ftp://st.cs.uiuc.edu/pub/papers/refactoring/opdyke-thesis.ps.Z`.

[27] S. Stelting and O. Maassen. *Applied Java patterns*. Java series. Prentice Hall, December 2001.

[28] D. Syme. *Declarative Theorem Proving for Operational Semantics*. PhD thesis, University of Cambridge, Computer Laboratory, 1998.

[29] L. Tokuda. *Evolving Object-Oriented Designs with Refactorings*. PhD thesis, The University of Texas at Austin, December 1999.

[30] L. Tokuda and D. Batory. Evolving object-oriented designs with refactorings. In *Automated Software Engineering*, pages 89–120, 2001.

[31] J. Vlissides. *Pattern Hatching: Design Patterns Applied*. Software Pattern Series. Addison Wesley, July 1998.