Automated Verification of Design Patterns: A Case Study

Jon Nicholson[a], Amnon H. Eden[b,*], Epameinondas Gasparis[b], Rick Kazman[c,d]

[a] *School of Computing, Engineering and Mathematics, University of Brighton, UK*
[b] *School of Computer Science and Electronic Engineering, University of Essex, UK*
[c] *Software Engineering Institute, Carnegie-Mellon University, USA*
[d] *University of Hawaii, USA*

Abstract

Representing design decisions for complex software systems, tracing them to code, and enforcing them throughout the lifecycle are pressing concerns for software architects and developers. To be of practical use, specification and modeling languages for software design need to combine rigour with abstraction and simplicity, and be supported by automated design verification tools that require minimal human intervention. This paper examines closely the use of the visual language of Codecharts for representing design decisions and demonstrate the process of verifying the conformance of a program to the chart. We explicate the abstract semantics of segments of the Java package `java.awt` as a *finite structure*; specify the Composite design pattern as a Codechart and unpack it as a set of formulas; and prove that the structure representing the program satisfies the formulas. We also describe a set of tools for modeling design patterns with Codecharts and for verifying the conformance of native (plain) Java programs to the charts.

Keywords: object-oriented design; modelling and specification; automated verification; visual languages; design description languages

1. Introduction

Software systems are some of the most complex artefacts ever produced by humans [1, 2]. Managing complexity is one of the central challenges of software engineering. Lehman's second Law of Software Evolution [3] suggests that complexity further arises when programs are maintained in a continuous state of flux, a situation which is true for many software systems. These concerns require specification and modelling languages for software design to combine abstraction mechanisms with rigour and parsimony. In addition, practitioners generally find it easier to use a visual notation to articulate and communicate design decisions. Therefore, accurate specification of software design and the means for checking conformance of native source code thereto are

*Corresponding author
Email address: `eden@essex.ac.uk` (Amnon H. Eden)

primary concerns. Given the complexity of design verification and the frequency by which they need be carried out, practitioners also need tools that automate and report any conflicts between design and implementation at the click of a button. However, these needs have so far been difficult to reconcile in practice. Proving and preserving the conformance of a program to its design documentation is largely an unsolved problem. The result is often a growing disassociation between the design and the implementation tiers of representation [4].

The language of Codecharts [5], LePUS3, is a formal and visual design description language tailored to meet these concerns. It supports the representation of structural information about object-oriented design motifs, programs of any size, and frameworks. The Two-Tier Programming Toolkit [6] was developed to demonstrate the feasibility of specification and automated verification of Codecharts and to test it in practice.

This paper presents a case study which highlights the process by which practitioners can use Codecharts to represent design patterns and supporting tools to verify conformance thereto in Java programs fully automatically. In the remainder of this section we discuss other works related to this problem, the definition of the Composite pattern, and the java.awt package in version 1.5 of the Java Standard Development Kit which is claimed to implement the pattern. In section 2 we present an informal hypothesis (Hypothesis A) about the conformance of the java.awt package to the Composite design pattern [7]. This hypothesis is gradually rendered formal (Hypothesis E) through sections 3 and 4, which explain how design and implementation are represented formally. In section 5 we present a logic proposition that formalizes our hypothesis and prove it. In section 6 we present a tool that fully automates the verification process and reports any violations of the respective design decisions. Section 6 concludes with a brief discussion on the results of a pilot study that evaluated the tool. This study showed that the tool improved the ability of participants to detect violations of design specifications.

1.1. Related Work

There exist numerous attempts at formal specification languages for design patterns. Some preliminary work has also been published on tools which verify that such specifications were properly implemented in source code. The following set of criteria guides our analysis of these languages:

1. **Object-oriented**: the language must be suitable for modelling and specifying the building-blocks of object-oriented design patterns.
2. **Generic**: the language must have the ability to represent design motifs such as patterns in terms of generic 'participants' Gamma et al. [7] (also *placeholders* or *roles*) as distinguished from concrete implementation artefacts. A tool should support the specification of many patterns rather than being hard-coded to verify specific patterns.
3. **Implementation independent**: specifications in this language should not be bound to a specific programming language or to any specific dialects.
4. **Visual**: specifications should be represented visually and created using a visual editor for ease of use by programmers.
5. **Parsimonious**: the language must have the ability to represent complex design statements parsimoniously, using a small vocabulary.

6. **Rigorous**: the language need be mathematically sound and axiomatized such that all assumptions are articulated explicitly and precisely.
7. **Decidable**: the language is restricted to expressing properties and relations whose satisfaction can be established statically ('structural statements'), which ensures that specifications are automatically verifiable at least theoretically.
8. **Automatically verifiable**: specifications in this language must allow fully automated design verification against programs in their native, uninstrumented form (source code).

Several formal notations for specifying design patterns are described in [8]. Most of the contributions in this volume do not describe tools for automated verification, and base their notations on UML. UML is a popular visual object-oriented design description language, a powerful and expressive collection of notations [9] suitable for many common software development tasks. For example, DPML [10], which is based on UML, is capable of representing both programs and design patterns. UML, however, does not meet all of our above criteria. In particular, Fowler [11] tells us that "no formal definition exists of how UML maps to any particular programming language." In other words, UML as a specification language does not meet the criteria of being rigorous and automatically verifiable. Blewitt [12] adds that "UML cannot be used to describe an infinite set of pattern instances because the language is not designed for that purpose". Thus UML dialects that do not introduce variables for the representation of generic participants do not meet the criterion of genericity.

Many specification languages whose semantics are defined in terms of UML and tools depending on such representations face similar issues. The DEMIMA framework [13] can check the conformance of source code to design patterns specified in the Pattern and Abstract-level Description Language (PADL), which translates UML diagrams to a constraint-based language. The authors represent such constraints using a Java data structure implemented using the Ptidej tool suite. However they do not describe a tool that can create PADL models from visual specifications. The Pattern Specification Language (PSP) [14, 15] articulates design pattens in precise and generic terms. [16] define the manual process of design verification of instances of the Visitor pattern specified in PSP. However, although PSP formalizes a subset of UML, it is symbolic and not visual. LAMBDES-DP, described in [17], is a tool that detects instances of design patterns in UML models and formalized in GEBNF (Graphically Extended BNF) but not in source code.

SPINE [12] is a language outside of the UML family. It is a formal object-oriented language for representing design patterns in the logic programming language PROLOG. Specifications written in SPINE are automatically verifiable using its associated tool, HEDGEHOG [12]. However, SPINE is not a visual language, and "all of the SPINE predicates are tightly focussed on the Java implementation" [18]. Similarly, more than one tool can successfully verify the implementation of design patterns specified in the Logic Metaprogramming Model (e.g., [19]), which relies on a text-based logic language for modelling design patterns rather than UML.

1.2. The Notation

The language of Codecharts, LePUS3, is an object-oriented design description language [5][20] which was created to meet the criteria set above. It is particularly suited

to representing the structural properties of design motifs, such as structural patterns, using a minimal vocabulary (Figure 1).

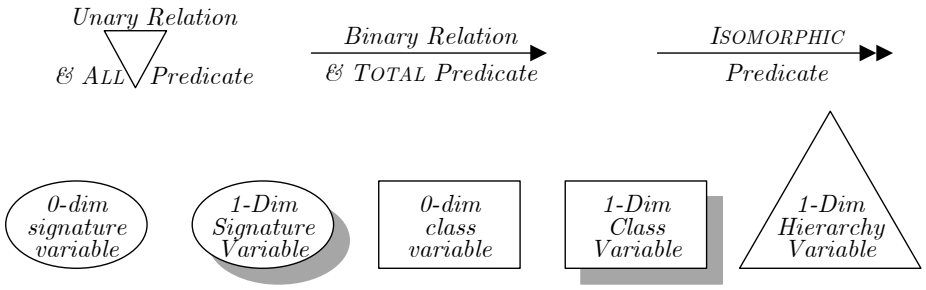

Figure 1: The vocabulary of Codecharts. Some symbols not used in this paper were omitted

A *Codechart* is a formal specification that represents a set of recursive (fully Turing-decidable) sentences in first-order predicate logic. The logic of Codecharts is based on the Core Specification Theory [21] which sets an axiomatic foundation in mathematical logic for many formal specification languages (including Z, B, and VDM). The axioms and semantics of Codecharts are defined using finite model theory. The *satisfies* relation between *Codecharts* and programs in class-based languages such as Java, C++ and C# is well-defined [5], Turing-decidable and automatically verifiable.

Figure 1 presents a subset of the vocabulary of Codecharts for generically representing the building-blocks of object-oriented design. Classes are represented by rectangles, method signatures (i.e., the method's name and argument types) by ellipses, and methods by superimposing a signature symbol on a class symbol. Sets are represented with the addition of shadow. A triangle represent inheritance class hierarchy—a set of classes that share a common superclass. Properties and relationships are represented with the respective relation symbols. Some of these symbols are explained in section 3. The detailed syntax, axioms and truth conditions which constitute the language and logic of LePUS3 are laid out in [5] and [22].

Although this paper focuses on the specification and verification of the Composite design pattern, Codecharts can also be used effectively to specify many other design patterns [5]. Additionally, the above vocabulary (and relationships expressible therein) are tailored to object-oriented concepts, such as those in [23], and not any particular implementation language. That is, Codecharts can be used to articulate the design of object-oriented programs/class libraries encoded in any class-based statically typed programming language (e.g. Java, C++, C#).

1.3. Verification vs. detection

Design verification—henceforth verification—is defined as in this context as the problem of checking whether a given implementation conforms to its specification. Design verification is distinct from the problem of detecting instances of the motif in code. This manuscript is concerned with enforcing a design decision about where and how many times a pattern is implemented. For example, software designers can decide

that the Composite pattern needs to be implemented twice in a particular program, each time by a separate set of implementation-specific classes and methods. Users should therefore need to manually indicate each and every intended implementation of the Composite pattern. Checking that each such design decision is enforced is therefore a separate problem of verification. In contrast, the challenge of detecting instances of a particular design motif arises in other circumstances. It also has a distinctly different form, e.g., instead of "class Container implements the *component* participant in the *Composite* pattern", a detection problem is posed by a claim such as "*some* class in program *p* implements the *component* participant in the *Composite* pattern". Consequently, a claim that requires detection is formalized differently (Hypothesis D) from a claim that requires verification (Hypothesis E), as illustrated in section 5. Most importantly, automating the detection process poses an interesting problem that is strictly more challenging than automating verification, since the supporting tool must first search for a suitable set of candidate classes and methods in the implementation before attempting to verify them.

2. The problem

As a leading example we focus on a claim that is commonly made informally (e.g. [24–26]) according to which the package java.awt in version 1.5 of the standard distribution ('Software Development Kit' [27]) of the Java programming language [28] 'implements' the Composite design pattern, quoted in Hypothesis A.

> **Hypothesis A.** *java.awt implements the Composite design pattern.*

In this section we examine the informal parts of Hypothesis A. The remainder of this paper is dedicated to formalizing and verifying this hypothesis.

2.1. The Composite Design Pattern

Design patterns have made a significant impact on the practice of software design, each describing an abstract design motif—a recurring theme which in principle can be implemented by an unbounded number of programs in any class-based programming language:

> *A design pattern names, abstracts, and identifies the key aspects of a common design structure that make it useful for creating a reusable object-oriented design ... Each design pattern focuses on a particular object-oriented design problem or issue.* [7]

Table 1 quotes the solution advocated by the Composite design pattern. As is the custom in most pattern catalogues, it is described informally.

Table 1: The Composite design pattern [7] (abbreviated)

Intent: Compose objects into tree structures to represent part-whole hierarchies.
Participants:
– *Component:* Declares a basic interface, implements default behaviour.
– *Leaves:* Have no children, implements/extends superclass behaviour.
– *Composite:* Has children, defines behaviour for components having children.
Collaborations: Interface of Component class is used to interact with objects in the structure. Leaves handle requests directly. Composite objects usually forward requests to each of their children, possibly performing additional operations before and/or after forwarding.

2.2. Package java.awt

Package java.awt ('Abstract Window Toolkit') is part of the standard distribution of the Java Software Development Kit 1.5 [27] which provides user interface widgets (e.g. buttons, windows, etc.) and graphic operations thereon. Class Component represents a generic widget that is extended [in]directly by all non menu-related widgets (e.g. Button, Canvas). Container represents widgets which aggregate (hold an array of instances of) widgets. Excerpts from the package's source code that corroborate Hypothesis A are provided in Table 2. All references to java.awt shall henceforth refer exclusively to those aspects listed in Table 2.

Table 2: java.awt [27] (abbreviated)

```
public abstract class Component ...{
 public void addNotify() ...
 public void removeNotify() ...
 protected String paramString() ... }

public class Button extends Component ...{
 public void addNotify() ...
 protected StringparamString() ... }

public class Canvas extends Component ...{
 public void addNotify() ...
 protected String paramString() ... }

public class Container extends Component {
 Component component[] = new Component[0];
 public Component[] getComponents() ...
 public Component getComponent(int) ...
 public void addNotify() { component[i].addNotify(); ...}
 public void removeNotify() { component[i].removeNotify(); ...}
 protected String paramString() { super.paramString(); ...} ...}
```

3. Specification

Contemporary modelling languages [9] and notations are largely designed to represent specific implementations. Design patterns however are generic design motifs: abstractions that may be implemented in any number of ways. Therefore, design patterns can only be adequately represented using generic abstractions which describe entities (e.g. '*composite*', '*component*') by their properties and relations (e.g., '*composite* is a class that has children of type *component*') and not by a particular implementation. Our specification language must therefore be capable of generically representing the category of entities and relations that constitute the building-blocks of design patterns, namely [sets of] classes, [sets of] methods, and their correlations.

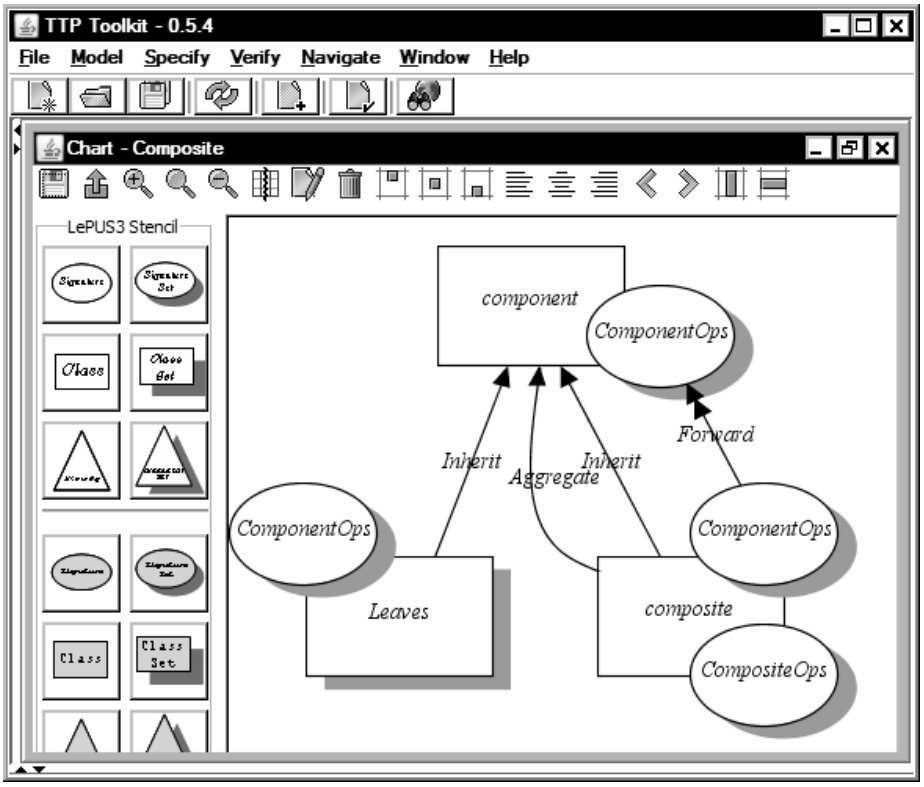

Figure 2: The Composite design pattern specified as a Codechart (designated Composite) using the Toolkit

Codecharts were specifically designed for this purpose. To ensure that every Codechart is automatically verifiable, the scope of the language is restricted to representing recursive properties of programs. The significance of this is that Codecharts are not designed to model other aspects of programs, such as their behaviour, events, or state. For example, Codechart Composite (Figure 2) captures the structural properties in the informal description of the Composite design pattern (Table 1). Promoting abstraction, it does not specify exactly how many classes must be in the set *Leaves*, only that it must

not be empty. The precise meaning of Composite is spelled out by the truth conditions listed in Table 3.

Table 3: Truth conditions for Composite

Terms
(a) *composite* and *component* are variables ranging over individual types (in Java: class, interface, or primitive type)
(b) *Leaves* is a variable that ranges over non-empty sets of types
(c) *CompositeOps* and *ComponentOps* are variables ranging over sets of method signatures
Formulas
(d) *composite* must have an 'aggregate' (an array or a Java collection) of instances of type *component* (or of subtypes thereof)
(e) *composite* must 'inherit' (in Java: **extends** or **implements**) (possibly indirectly) from class *component*
(f) Every class in *Leaves* must 'inherit' (possibly indirectly) from class *component*
(g) *composite* must define (or inherit) a method for each of the signatures in the set *CompositeOps*
(h) Every class in *Leaves* must define (or inherit) a method for each of the signatures in the set *ComponentOps*
(i) Each method defined in (or inherited by) *composite*, with a signature in *ComponentOps*, must at some point forward the method call (invocation) to that (unique) method with same signature that is a member of (or inherited by) *component*, and vice versa

Formally, a well-formed Codechart represents a set of *well-formed formulas* in terms of a combination of visual tokens (Figure 1). Each formula consists of *terms*, which stand for [sets of] classes or [sets of] methods, a *relation* and possibly a *predicate symbol*. Terms representing methods consist of the combination of a signature s and a class c using the binary operator called *superimposition*, written $s \otimes c$. Relations describe properties (such as being abstract) of and relations (e.g., inheritance) between entities. Predicate symbols articulate properties of sets of entities and their correlations. There are three predicates, All, Total, and Isomorphic which can be roughly understood as:

- All(R, t) requires that all elements of set t are in relation R;
- Total(R, s, t) requires that for all elements of set s there is an element in t for which R holds;
- Isomorphic(R, s, t) requires that the elements of set s be uniquely paired with the elements in set t under relation R.

We define these predicates and the superimposition function in section 5. Using this notation we may unpack Composite as the set of well-formed formulas identified in Table 4.

Table 4: Composite as a set of well-formed formulas

$composite \in Class$	(1)
$component \in Class$	(2)
$Leaves \in \mathcal{P}(Class)$	(3)
$CompositeOps \in \mathcal{P}(Signature)$	(4)
$ComponentOps \in \mathcal{P}(Signature)$	(5)
$Aggregate(composite, component)$	(6)
$Inherit(composite, component)$	(7)
$\textsc{Total}(Inherit, Leaves, component)$	(8)
$\textsc{All}(Method, CompositeOps \otimes composite)$	(9)
$\textsc{All}(Method, ComponentOps \otimes Leaves)$	(10)
$\textsc{Isomorphic}(Forward, ComponentOps \otimes composite, ComponentOps \otimes component)$	(11)

These formulas (Table 4) and truth conditions (Table 3) tell us how to understand Composite as a mathematical artefact. And, given this formal specification of the Composite pattern, we may now rephrase our informal hypothesis in a slightly more rigorous fashion as demonstrated in Hypothesis B:

> **Hypothesis B.** *java.awt 'implements'* Composite

4. Abstract Semantics

The terms 'program' and 'implementation' usually refer to a set of source code (text) files distributed across a file system which normally contains myriad implementation minutiae. Source code can be a difficult medium to reason about in part because of its scale, it is not uncommon for a moderately complex system to contain thousands (or even millions) of lines of code. For example, the unabbreviated source code of only four classes from java.awt spans over ten thousand lines. Furthermore, each programming language rightly adopts a different set of syntactic and semantic rules. As such, reasoning over the source code directly would restrict the modelling language to the idiosyncrasies of a specific programming language. Reasoning therefore requires some intermediate representation of the program in a simplified form. This motivates the notion of *abstract semantics*. A program's abstract semantics contains all relevant decidable properties in a standard format, in-line with the notion of program equivalence classes [29], obtained via static analysis.

Formally, the abstract semantics of a program is captured in a finite model theoretic structure called a *finite structure*[1] [5, 30] defined as follows:

[1] Finite structures are implementable as a set of tables in a relational database.

Definition 1. *A **finite structure** $\mathfrak{F}$ is a pair $\mathfrak{F} = \langle \mathbb{U}, \mathbb{R} \rangle$ where $\mathbb{U}$ (the 'universe' of $\mathfrak{F}$) is the finite set of all atomic entities (each of which stands either for a specific class, method, or signature in the program), and $\mathbb{R}$ is the finite set of relations over $\mathbb{U}$.*

For example, the abstract semantics of java.awt (Table 2) is that finite structure which is represented by the pair $\langle \mathbb{U}, \mathbb{R} \rangle$ where:

$$\begin{aligned}\mathbb{U} =& \{\underline{\texttt{Component}}, \underline{\texttt{Component[]}}, \underline{\texttt{Container}}, \ldots, \\ & \underline{\texttt{Component.addNotify()}}, \underline{\texttt{Container.addNotify()}}, \ldots, \\ & \underline{\texttt{addNotify()}}, \ldots\} \\ \mathbb{R} =& \{\underline{\mathit{Class}}, \underline{\mathit{Method}}, \underline{\mathit{Signature}}, \underline{\mathit{Inherit}}, \underline{\mathit{Aggregate}}, \ldots\}\end{aligned}$$

Entities and relations in the model are underlined. Every atomic entity in the universe $\mathbb{U}$ is either a class (an element of the unary relation $\underline{\mathit{Class}}$), a method (an element of $\underline{\mathit{Method}}$), or a method signature (an element of $\underline{\mathit{Signature}}$, which identifies method name and argument types). For example, the entity $\underline{\texttt{Container}}$ that models the class Container is a member of $\underline{\mathit{Class}}$ but not $\underline{\mathit{Method}}$ or $\underline{\mathit{Signature}}$. In other words, $\mathbb{U}$ is the disjoint union of the unary relations $\underline{\mathit{Class}}$, $\underline{\mathit{Method}}$ and $\underline{\mathit{Signature}}$. Each relation in $\mathbb{R}$ is a finite set of tuples of atomic entities. For example, the unary relation $\underline{\mathit{Class}}$ is a set of 1-tuples, one for each class in java.awt. The binary relation $\underline{\mathit{Inherit}}$ is a set that contains all pairs of classes (i.e. a subset of $\underline{\mathit{Class}} \times \underline{\mathit{Class}}$) $\langle \mathit{cls}, \mathit{supercls} \rangle$ in java.awt such that *cls* extends/implements/is-subtype-of *supercls*. Likewise, the binary relation $\underline{\mathit{Aggregate}}$ contains pairs of classes $\langle \mathit{cls}, \mathit{elementType} \rangle$ such that *cls* contains a collection (or array) of instances of the class *elementType* (or subtypes thereof). The binary relation $\underline{\mathit{Forward}}$ represents a special kind of method call between two methods, $\langle \mathit{invoker}, \mathit{invoked} \rangle$, that share the same signature.

The precise relation between a program and its abstract semantics is formally captured using the *abstract semantics function*: a mapping from programs in a programming language into finite structures defined as:

Definition 2. *An **abstract semantics function** $\mathcal{A}$ maps program source code written in some class-based object-oriented language $\mathbb{L}$ to the enumerable set of possible finite structures $\mathfrak{F}^*$, written $\mathcal{A} : \mathbb{L} \to \mathfrak{F}^*$.*

For example, $\mathcal{A}_{Java}$ [31] is an abstract semantics function [5] which represents the mapping from each Java program to a finite structure. Given the package java.awt as input (Table 2) the function $\mathcal{A}_{Java}$ yields the finite structure described above. How this abstract semantics is obtained is summarised in Table 5.

Table 5: Summary of the abstract semantics for java.awt (Table 2)

Code Description	**Tuple**	**Relation**
Class Component defined	Component	*Class*
Class Container defined	Container	*Class*
Method Container.addNotify() with signature addNotify() defined in class Container	Container.addNotify() addNotify() ⟨addNotify(), Container.addNotify()⟩ ⟨Container, Container.addNotify()⟩	*Method* *Signature* *SignatureOf* *Member*
Class Component is abstract	Component	*Abstract*
Class Container extends class Component	⟨Container, Component⟩	*Inherit*
Class Container has a field of class Component[]	⟨Container, Component[]⟩	*Member*
Class Container has (or is) an aggregation of class Component	⟨Container, Component⟩	*Aggregate*
Method Container.addNotify() forwards its method call to method Component.addNotify()	⟨ Container.addNotify(), Component.addNotify() ⟩	*Forward*
...		

Abstract semantics functions allow us to determine exactly how the source code of programs can be abstracted. For example, we may use $\mathcal{A}_{Java}$ to define the finite structure for java.awt:

$$\mathcal{A}_{Java}(\text{java.awt})$$

Alternatively, other abstract semantics functions can be equally used to represent programs in any class-based object-oriented programming language, such as C#, C++, Object Pascal, PHP and Eiffel. For example, if we describe an abstract semantics function for the C++ programming language: $\mathcal{A}_{CPP} : \mathbb{CPP} \rightarrow \mathfrak{F}^*$, we can use the same specification and verification mechanisms described in this paper to analyse programs written in C++.

Abstract semantics functions must be recursive (fully Turing-decidable) such that their computation always terminates within a bounded and predetermined number of steps. In practical terms this means that $\mathcal{A}_{Java}$ can, in principle, be implemented as a static analyser. Such an analyser is implemented in the Toolkit (see section 6). However, static analysis has its limitations. Codecharts therefore do not capture many behavioural aspects of programs, for example temporal information and program state.

The notion of abstract semantics allows us to articulate informal claims concerning the relationship between a design pattern and a program precisely as a mathematical proposition. Specifically, we stipulate that a program p implements a design pattern if and only if the abstract semantics of p (a finite structure) *satisfies* that Codechart which specifies that pattern. Hypothesis B can thus be redefined in these terms as follows:

Hypothesis C. $\mathcal{A}_{Java}$(java.awt) *satisfies* Composite

In the following section we define the *satisfies* relation and recast Hypothesis C as a mathematical proposition.

5. Verification

The form of *design verification*—henceforth *verification*—which we consider in this paper is the rigorous, conclusive, and decidable process of establishing or refuting whether a particular program conforms to a given Codechart. An automated process of verification in this context, therefore, consists of executing an algorithm that determines whether the abstract semantics of a program p satisfies Codechart Ψ.

The conditions for satisfying a Codechart are modelled after the standard Tarski's truth conditions for classical logic, as demonstrated in Table 3. A *satisfies* proposition is represented using the standard semantic entailment symbol $\models$ defined as follows:

Definition 3. *Let $\mathfrak{F}$ be a finite structure and Ψ a Codechart. $\mathfrak{F}$ satisfies Ψ, written $\mathfrak{F} \models \Psi$, if and only if all the following hold:*[2]

1. *each atomic term t in Ψ interprets to an entity $\underline{\mathtt{t}}$ in $\mathfrak{F}$*
2. *each term of the form $s \otimes c$ in Ψ interprets to an entity in $\mathfrak{F}$ such that:*
   - *if $\underline{\mathtt{s}} \in \underline{Signature}$ and $\underline{\mathtt{c}} \in \underline{Class}$ then:*
     - *there exists an $\underline{\mathtt{m}} \in \underline{Method}$ such that $\langle \underline{\mathtt{s}}, \underline{\mathtt{m}} \rangle \in \underline{SignatureOf}$ and $\langle \underline{\mathtt{c}}, \underline{\mathtt{m}} \rangle \in \underline{Member}$ then $s \otimes c = m$, or*
     - *there exists some class $\underline{\mathtt{super}}$ such that $\langle \underline{\mathtt{c}}, \underline{\mathtt{super}} \rangle \in \underline{Inherit}$ and $s \otimes super$ is defined then $s \otimes c = s \otimes super$*
   - *if $\underline{\mathtt{s}} = \{\underline{\mathtt{s}_1}, \ldots, \underline{\mathtt{s}_n}\}$ then $s \otimes c = \{s_1 \otimes c, \ldots, s_n \otimes c\}$,*
   - *if $\underline{\mathtt{s}}$ is atomic and $\underline{\mathtt{c}} = \{\underline{\mathtt{c}_1}, \ldots, \underline{\mathtt{c}_n}\}$, then $s \otimes c = \{s \otimes c_1, \ldots, s \otimes c_n\}$.*
3. *for every formula f in Ψ the following hold:*
   - *if f is of the form $t \in R$ then $\underline{\mathtt{t}}$ is a member of $\underline{R}$*
   - *if f is of the form $t \in \mathcal{P}(R)$ then $\underline{\mathtt{t}}$ is a member of the power set of $\underline{R}$*
   - *if f is of the form $R(t_1, t_2)$ then $\langle \underline{\mathtt{t}_1}, \underline{\mathtt{t}_2} \rangle \in \underline{R}$*
   - *if f is of the form $\textsc{All}(R, t)$ then either:*
     - *$\underline{\mathtt{t}}$ is in $\underline{R}$, or*
     - *$\underline{\mathtt{t}}$ is a set and for every $\underline{\mathtt{x}} \in \underline{\mathtt{t}}$ $\textsc{All}(R, x)$ holds*
   - *if f is of the form $\textsc{Total}(R, t_1, t_2)$ then either:*

[2] This definition has been condensed for presentation in this paper. A more detailed definition can be found in [5].

  - $\langle \underline{\texttt{t}_1}, \underline{\texttt{t}_2} \rangle$ *is in* $\underline{R}$*, or*
  - $\underline{\texttt{t}_1}$ *is a set and for every* $\underline{\texttt{x}} \in \underline{\texttt{t}_1}$ $\textsc{Total}(R, x, t_2)$ *holds, or*
  - $\underline{\texttt{t}_2}$ *is a set and there exists a* $\underline{\texttt{y}} \in \underline{\texttt{t}_2}$ $\textsc{Total}(R, t_1, y)$ *holds.*
- *if* f *is of the form* $\textsc{Isomorphic}(R, t_1, t_2)$ *then either:*
  - $\langle \underline{\texttt{t}_1}, \underline{\texttt{t}_2} \rangle$ *is in* $\underline{R}$*, or*
  - *there exists an* $\underline{\texttt{x}} \in \underline{\texttt{t}_1}$ *and* $\underline{\texttt{y}} \in \underline{\texttt{t}_2}$ *such that* $\textsc{Isomorphic}(R, x, y)$ *and* $\textsc{Isomorphic}(R, t_1 - \{x\}, t_2 - \{y\})$ *holds.*

The above definition demonstrates that the question whether a program satisfies a Codechart is reduced to a series of queries about set membership.

Using the semantic entailment notation we can recast Hypothesis C as the following proposition:

> **Hypothesis D.** $\mathcal{A}_{Java}(\text{java.awt}) \models \text{Composite}$

However, Codecharts modelling design motifs, such as Composite (Figure 2), contain variable terms. Hence, the problem described in Hypothesis C is that of detection (see section 1), not of verification. To verify that such a Codechart is satisfied in the context of a specific program its variables must first be mapped to entities in the appropriate finite structure. Such a mapping is commonly referred to as an *assignment*, defined as follows:

Definition 4. *An* ***assignment*** *is a function mapping each variable in a Codechart to [a set of] entities in a finite structure. Let* Ψ *be a Codechart and* g *an assignment. We write* $\Psi[g(x_1)/x_1, \ldots, g(x_n)/x_n]$ *for the Codechart resulting from the consistent replacement of each variable* x_i *with* $g(x_i)$ *in* Ψ*.*

Given this we define the satisfaction of Codecharts that contain variables, such as those that represent design patterns and application frameworks, as follows:

Definition 5. *Let* $\mathfrak{F}$ *be a finite structure and* Ψ *be a Codechart that contains variable terms.* $\mathfrak{F}$ *satisfies* Ψ *if and only if there exists an assignment* g *from* Ψ *to* $\mathfrak{F}$ *such that* $\mathfrak{F} \models \Psi[g(x_1)/x_1, \ldots, g(x_n)/x_n]$ *holds, written* $\mathfrak{F} \models_g \Psi$*.*

Therefore, the key difference between the problems of pattern detection and verification is whether this assignment must be discovered or is given. That is, the semantic entailment in Hypothesis D holds if there exists an assignment (either discovered or given) that maps each variable in Composite to specific entities in java.awt. In this case, we define an assignment g (Table 6) based on claims made elsewhere (e.g. [24–26]).

Table 6: Assignment g mapping variables in Composite to entities in java.awt

$g(composite)$	=	Container
$g(component)$	=	Component
$g(Leaves)$	=	{Button, Canvas}
$g(ComponentOps)$	=	{addNotify(), removeNotify()}
$g(CompositeOps)$	=	{getComponents(), getComponent(int)}

Hypothesis D can now be recast as a proposition where, under assignment g, the abstract semantics of java.awt satisfy Codechart Composite, a claim represented in Hypothesis E using the standard notation for assignments:

> **Hypothesis E.** $\mathcal{A}_{Java}(\text{java.awt}) \models_g \text{Composite}$

The proposition in Hypothesis E imposes conditions on the existence of entities and sets of entities in java.awt and on correlations amongst them. To prove it we refer back to Table 4 to see what terms and formulas appear in Composite. We then employ the assignment g (Table 6) to fix each variable to their respective entities allowing us to use Definition 3 to decide if Hypothesis E holds. Table 7 demonstrates the proof for Hypothesis E, which depicts the precise elements of $\mathcal{A}_{Java}$(java.awt) (Table 5) which satisfy the truth conditions of Codechart Composite (Table 3)[3].

Table 7: Proof of Hypothesis E (abbreviated)

$\langle$Container$\rangle \in$ *Class*	$\models$	(1)
$\langle$Component$\rangle \in$ *Class*	$\models$	(2)
{Button, Canvas} $\in \mathcal{P}$(*Class*)	$\models$	(3)
{getComponents(), getComponent(int)} $\in \mathcal{P}$(*Signature*)	$\models$	(4)
{addNotify(), removeNotify()} $\in \mathcal{P}$(*Signature*)	$\models$	(5)
$\langle$Container, Component$\rangle \in$ *Aggregate*	$\models$	(6)
$\langle$Container, Component$\rangle \in$ *Inherit*	$\models$	(7)
$\langle$Button, Component$\rangle \in$ *Inherit*		
$\wedge\langle$Canvas, Component$\rangle \in$ *Inherit*	$\models$	(8)
$\ldots \wedge \langle$Container.getComponents()$\rangle \in$ *Method*		
$\wedge\langle$getComponents(), Container.getComponents()$\rangle \in$ *SignatureOf*		
$\wedge\langle$Container, Container.getComponents()$\rangle \in$ *Member*	$\models$	(9)
$\ldots \wedge \langle$Button.addNotify()$\rangle \in$ *Method*		
$\wedge\langle$addNotify(), Button.addNotify()$\rangle \in$ *SignatureOf*		
$\wedge\langle$Button, Button.addNotify()$\rangle \in$ *Member*		
$\wedge\langle$Component.removeNotify()$\rangle \in$ *Method*		
$\wedge\langle$removeNotify(), Component.removeNotify()$\rangle \in$ *SignatureOf*		
$\wedge\langle$Component, Component.removeNotify()$\rangle \in$ *Member*		
$\wedge\langle$Button, Component$\rangle \in$ *Inherit*	$\models$	(10)
$\langle$Container.addNotify(), Component.addNotify()$\rangle \in$ *Forward*		
$\wedge\langle$Container.removeNotify(), Component.removeNotify()$\rangle \in$ *Forward*	$\models$	(11)

This proves that Hypothesis E holds, thereby confirming that package java.awt indeed conforms to the Composite pattern (Hypothesis A).

[3] The omitted statements for formulas 9 and 10 mirror those presented.

5.1. Analysis

To summarize, *verification* consists of the process of checking the truth value of a set of propositions, each of which is unpacked as a *sentence* (also *closed formula*) about finite sets and relations. Verification of a Codechart Ψ given assignment g can therefore be implemented as an algorithm which checks whether each formula in $\Psi[g(x_1)/x_1, \ldots, g(x_n)/x_n]$ satisfies the relevant condition in Definition 3. It is straightforward to show that the computational complexity of such an algorithm is bounded by the number of steps that is required to check the predicates $\textsc{All}(R, t)$, $\textsc{Total}(R, t_1, t_2)$ or $\textsc{Isomorphic}(R, t_1, t_2)$, which is $O(|\mathbb{U}|)$, $O\left((|\mathbb{U}|)^2\right)$, and $O\left((|\mathbb{U}|)^{|t_1|}\right)$, respectively, where $|\mathbb{U}|$ stands for the size of the universe and $|t_1|$ is the number of entities in t_1. In other words, the complexity of a verification algorithm for any Codechart with predicates in the form $\textsc{Isomorphic}(R, t_1, t_2)$ with terms t_1 that are not too large ($|t_1| < c$ for some small constant c) is at most polynomial in the size of the implementation (i.e. number of classes, methods and signatures).

6. Tool Support

While the notion of verification demonstrated above is relatively straightforward, verifying conformance of non-trivial programs is a tedious and error-prone process. It involves making sure that a disproportionally large number of conditions are met. It also requires intimate knowledge of formal techniques such as computing the abstract semantics of Java programs and the truth conditions of the specification language. The manual task is even less feasible for software systems that evolve in iterations, since the proof would have to be repeated each time the implementation or the design change. Fortunately, the discussion in the previous section demonstrates that verifying a Codechart can be formulated and fully automated, and as long as the terms in the Isomorphic predicates are under a fixed size, the verification algorithm need not exceed a number of steps squared in the size of the size of the implementation. Automating the verification process so defined by a tool may therefore be feasible. Below we describe a set of tools which, among others, were built to test the feasibility of automated verification of Codecharts in practical settings.

The current prototype (0.5.4) of the Toolkit is a tool that parses any Java program[4] and generates a representation of its abstract semantics in the form of a simple relational database [5]. The Toolkit we describe below is freely available[5] [6].

The Toolkit consists of a collection of tools that were designed to support visualization (reverse engineering Codecharts), specification (composition of Codecharts), and verification of object-oriented programs. Figure 3 demonstrates the Toolkit's facilities for specifying the Composite design pattern and for verifying the conformance of `java.awt` package thereto. Window (1) shows the source code of the relevant Java files in this package, which the Toolkit analyses to create the abstract semantics. Window

[4] Version 0.5.4 of the Toolkit incorporates a static analyser for Java 1.5 without support for generics. All other parts of the Toolkit are capable of working with other versions of Java, as well as other programming languages.

[5] Under the Creative Commons Attribution-No Derivative Works 2.0 UK: England & Wales License

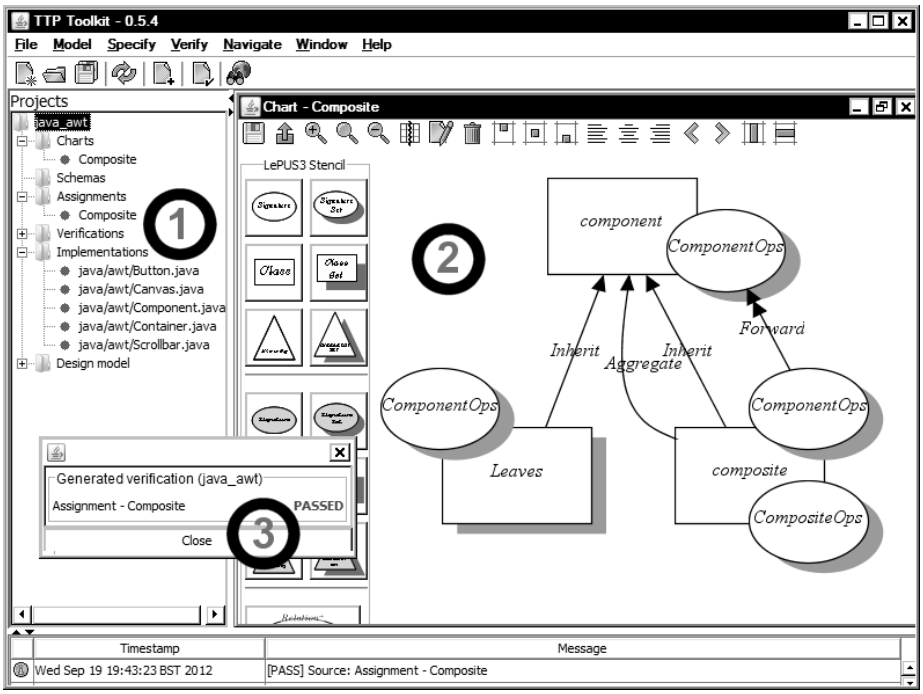

Figure 3: Source files (1), Codechart Composite (2), and the verification result (3) in the Toolkit

(2) depicts the vocabulary of Codecharts (left pane) as a set of icons which the user can drag-and-drop to create the Codechart modelling the Composite pattern (right pane). Dialogue box (3) shows the result of executing the verification process, indicating that the implementation conforms to the specification.

As demonstrated above, the Toolkit supports creating and editing Codecharts for encoding design decisions, automatic generation of a program's abstract semantics (e.g., Table 5) by static analysis of Java source code, and conformance checking at the click of a button. The graphical user interface can be used to define assignments which map variables to source code artefacts, such as the one presented above (Table 6). Verification as described in section 5 is fully automated and efficient, concluding in this example under a fraction of a second. It compares the conditions imposed by the truth conditions expressed in the Codechart with the abstract semantics it generated and reports whether all have been met. In this manner the Toolkit closes the round-trip engineering cycle. This ensures that the documentation of the program—Codecharts and assignments—is always current and correct, reflecting the program's true structure.

If conformance fails, the user is likely to seek ways to resolve the conflict by changing either the design or the implementation. To support this, the Toolkit reports exactly which truth condition has not been met. Let us demonstrate such a scenario by changing the Codechart and reversing the *Forward* relation in 3 such that it specifies that the methods in *component* (Component) forward the call to the respective methods in *composite* (Composite), as demonstrated in Figure 4. When the user clicks Verify,

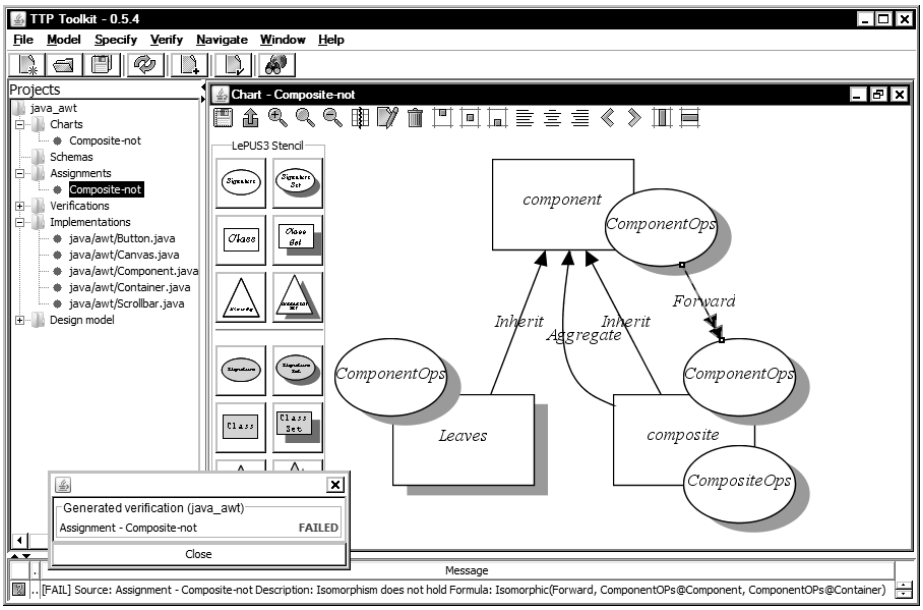

Figure 4: The attempt to verify that java.awt conforms to a different Codechart fails for the reasons detailed in the message displayed at the bottom pane

the Toolkit checks the revised specification and detects that java.awt does not conform to the revised Codechart, reporting the formula whose truth conditions have not been met. The error message it displays is depicted at the bottom pane of 3.

The Toolkit has also been used to model (specify) and verify several implementations of other design patterns [5]. For example it has been used to prove that the package java.io does not conform to the Decorator design pattern [7] as commonly accepted [26] but to a variation of the pattern.

A pilot study conducted at the University of Essex tested the contribution of the Toolkit to practitioners [20]. The results of the Conformance experiment in this study suggest gains in accurately deciding whether an implementation conforms to design specifications when using the Toolkit over a market-standard commercial tools, namely NetBeans 6.1 and relevant Javadoc files. Participants in this experiment were mostly graduate computer science students at the University of Essex who had no prior experience with the Toolkit. All participants were paid a fixed amount regardless of the time it took them to complete the tasks. The participants were given one hour of training in using the Toolkit for design verification, balanced with one hour training in using NetBeans 6.1 and relevant Javadoc files for the same task. We prepared two equivalent tasks and randomly split the participants into two groups: the experiment group who used the Toolkit and the control group who used NetBeans. In the first task, participants in both groups were given a set of source code files taken from a Java package in Java's SDK and a copy of the chapter about the Composite design pattern from [7]. They were asked to judge whether or not the implementation conforms to the pattern, where

the correct answer was Yes. To minimize bias between the groups, participants who used the Toolkit (the experiment group) in carrying out the first task switched to using NetBeans (therefore becoming the control group) to carry out the second task, and vice versa. The second task required all participants to determine whether a different selection of source code conforms to the Decorator design pattern [7]. Again, participants were asked to give a Yes/No answer, where this time the correct answer was No. Of the eight participants in this experiment, one participant's data was excluded as s/he did not complete both sessions and therefore failed to complete both tasks. The results were that all seven participants in the experiment group completed both tasks correctly, whereas three participants in the control group delivered an incorrect answer.

We identify three primary factors that impact the strength of this result: sample group size, tool coverage and the measurement of accuracy. First, the small number of participants means that the sample is not sufficiently representative of the population of software engineers. Second, due to limited resources we compared the Toolkit to a single tool, the NetBeans integrated development environment, out of numerous possible candidates. Further experimentation is therefore required to see if similar results can be obtained over a wider range of tools. Third, the method of measuring participant accuracy was reliant on a correct boolean (Yes/No) response which left little room for analysis. A task of the form "identify all entities in program p that participate in design pattern d" would provide greater insight into the participant's precision and accuracy. However, as a pilot study the results are encouraging and suggest what might be seen in a larger study.

7. Summary

We have presented the language of Codecharts and demonstrated how it can be used to specify (model) design patterns. To illustrate the process we quoted a widely held claim that the Composite design pattern is implemented by the java.awt package. We recast this informal hypothesis as a mathematical proposition and sketched its proof. We also described the Toolkit, a set of tools which can be used to compose object-oriented design specifications as Codecharts, statically analyse Java programs, and verify them to establish whether they conform to the design specifications. Finally, we discussed a pilot study demonstrating potential gains of using Codecharts and the Toolkit.

Acknowledgements

This work was partially funded by the UK's Engineering and Physical Sciences Research Council (EPSRC) and the University of Essex's Research Promotion Fund. The authors wish to thank Raymond Turner for his numerous contributions to this project. We also wish to thank Olumide Iyaniwura, Gu Bo, Maple Tao Liang, Dimitrious Fragkos, Omololu Ayodeji, Xu Yi, and Christina Maniati for their contributions to this research.

References

[1] F. P. Brooks, No Silver Bullet: Essence and Accidents of Software Engineering, IEEE Computer 20 (4) (1987) 10–19.

[2] W. Gibbs, Software's chronic crisis, Scientific American 271 (3) (1994) 72–81.

[3] M. M. Lehman, Laws of Software Evolution Revisited, in: Proceedings of the 5th European Workshop on Software Process Technology, Springer-Verlag, 108–124, 1996.

[4] D. E. Perry, A. L. Wolf, Foundations for the study of software architecture, SIGSOFT Software Engineering Notes 17 (4) (1992) 40–52.

[5] A. H. Eden, J. Nicholson, Codecharts: Roadmaps and Blueprints for Object-Oriented Programs, Wiley-Blackwell, ISBN 0470626941, 2011.

[6] J. Nicholson, E. Gasparis, A. H. Eden, The Two-Tier Programming Project Website, http://ttp.essex.ac.uk/, (Accessed 07 May 2013) .

[7] E. Gamma, R. Helm, R. Johnson, J. M. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley Professional, 1994.

[8] T. Taibi, Design Patterns Formalization Techniques, IGI Global, Hershey, USA, 2007.

[9] Object Management Group, UML 2.4.1 Superstructure Specification, Tech. Rep., 2011.

[10] D. Maplesden, J. Hosking, J. Grundy, A Visual Language for Design Pattern Modeling and Instantiation, in: Design Patterns Formalization Techniques, IGI Global, Hershey, USA, ISBN 1599042193, 2007.

[11] M. Fowler, UML Distilled: A Brief Guide to the Standard Object Modeling Language, Addison Wesley, third edn., ISBN 0321193687, 2003.

[12] A. Blewitt, Spine: Language for Pattern Verification, in: Design Patterns Formalization Techniques, IGI Global, Hershey, USA, ISBN 1599042193, 2007.

[13] Y.-G. Guéhéneuc, G. Antoniol, DeMIMA: A Multilayered Approach for Design Pattern Identification, IEEE Transactions on Software Engineering 34 (5) (2008) 667–684.

[14] D.-K. Kim, A meta-modeling approach to specifying patterns, PhD, Colorado State University, Fort Collins, CO, USA, 2004.

[15] R. B. France, D. Kim, S. Ghosh, E. Song, A UML-Based Pattern Specification Technique, IEEE Transactions on Software Engineering 30 (3) (2004) 193–206.

[16] L. Lu, D.-K. Kim, Y. Zhu, S. Kim, Verification of Structural Pattern Conformance Using Logic Programming, Journal of Universal Computer Science 16 (17) (2010) 2455–2474.

[17] H. Zhu, I. Bayley, L. Shan, R. Amphlett, Tool Support for Design Pattern Recognition at Model Level, in: Computer Software and Applications Conference, 2009. COMPSAC '09. 33rd Annual IEEE International, vol. 1, 228 –233, 2009.

[18] A. Blewitt, Hedgehog: Automatic verification of Design patterns in Java, PhD, University of Edinburgh, UK, 2006.

[19] J. Fabry, T. Mens, Language-independent detection of object-oriented design patterns, Computer Languages, Systems & Structures 30 (1) (2004) 21–33, ISSN 1477-8424.

[20] A. H. Eden, E. Gasparis, J. Nicholson, R. Kazman, Modeling and Visualizing Object-Oriented Programs with Codecharts, Formal Methods in System Design (2013) .

[21] R. Turner, The Foundations of Specification, J Logic Computation 15 (5) (2005) 623–662.

[22] A. H. Eden, E. Gasparis, J. Nicholson, LePUS3 and Class-Z Reference Manual, Technical Report CSM-474, ISSN 1744-8050, School of Computer Science and Electronic Engineering, University of Essex, 2007.

[23] I. Craig, The interpretation of object-oriented programming languages, Springer, London, ISBN 9781852335472, 1999.

[24] J. Dong, Y. Zhao, Experiments on Design Pattern Discovery, in: Proceedings of the 3rd International Workshop on Predictor Models in Software Engineering, IEEE Computer Society, 12, 2007.

[25] J. Seemann, J. W. von Gudenberg, Pattern-based design recovery of Java software, in: Proceedings of the 6th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ACM, Lake Buena Vista, Florida, USA, 10–16, 1998.

[26] S. A. Stelting, O. Maassen, Applied Java Patterns, Prentice Hall, 2002.

[27] Sun Microsystems, Java 2 Platform Standard Edition Documentation, version 1.5.0, Sun Microsystems, 2004.

[28] J. Gosling, B. Joy, G. Steele, G. Bracha, Java Language Specification, Addison-Wesley Professional, 3rd edn., 2005.

[29] J. M. Wing, A Specifier's Introduction to Formal Methods, IEEE Computer 23 (9) (1990) 8–23.

[30] M. R. A. Huth, M. D. Ryan, Logic in Computer Science: Modelling and Reasoning about Systems, Cambridge University Press, Cambridge, England, 2000.

[31] J. Nicholson, A. H. Eden, E. Gasparis, Verification of LePUS3/Class-Z Specifications: Sample models and Abstract Semantics for Java 1.4, Technical Report CSM-471, ISSN 1744-8050, School of Computer Science and Electronic Engineering, University of Essex, 2007.