Published in the 1st NASA Formal Methods (NFM) Symposium, Moffett Field, California, USA. April 2009

Automated Verification of Design Patterns with LePUS3

Jonathan Nicholson[*] Epameinondas Gasparis[*] Amnon H. Eden[*][†] Rick Kazman[‡][§]
http://ttp.essex.ac.uk/

Abstract

Specification and [visual] modelling languages are expected to combine strong abstraction mechanisms with rigour, scalability, and parsimony. LePUS3 is a visual, object-oriented design description language axiomatized in a decidable subset of the first-order predicate logic. We demonstrate how LePUS3 is used to formally specify a structural design pattern and prove (‘verify’) whether any Java^{TM} 1.4 program satisfies that specification. We also show how LePUS3 specifications (charts) are composed and how they are verified fully automatically in the Two-Tier Programming Toolkit.

Keywords: specification, automated verification, visual languages, design description languages, object-oriented design

Related Terms: first-order predicate logic, finite model theory, Java 1.4

1 Context

Software systems are the most complex artefacts ever produced by humans [1][2], and managing complexity is one of the central challenges of software engineering. According to the second Law of Software Evolution [3], complexity also arises because most programs are in a continuous state of flux. Maintaining consistency between the program and its design documentation is largely an unsolved problem. The result is most often a growing disassociation between the design and the implementation layers of representation [4]. Formal specification of software design and tools that support automated verification are therefore of paramount importance. Of particular demand are tools which, by a click of a button, can conclusively establish whether a given implementation is consistent with (‘satisfies’) the design. Attempts towards this end include Architecture Description Languages (ADLs) [5] and formal pattern specification languages [6]. Specifically, we are concerned with the following set of desired criteria:

- **Object-orientated:** suitable for modelling and specifying the building-blocks of the design of object-oriented programs and patterns
- **Visual:** specifications serve as visual presentations of complex (possibly hypothetical) systems
- **Parsimonious:** represent complex design statements parsimoniously, using a small vocabulary
- **Scalable:** abstraction mechanisms that scale well such that charts do not clutter as the size of programs increase
- **Rigorous:** mathematically sound and axiomatized such that all assumptions are explicit
- **Decidable:** fully-automated formal verification is possible at least in principle
- **Automatically verifiable:** accompanied by a specification and (fully automated) verification tool

LePUS3 (LanguagE for Patterns Uniform Specification, version 3) is an object-oriented Design Description Language [7] tailored to meet these criteria. It is particularly suited to representing design motifs such as structural and creational design patterns. LePUS3 ‘charts’ are formal specifications, each of which stands for a set of recursive (fully Turing-decidable) sentences in the first-order predicate logic. LePUS3 logic is based on Core Specification Theory [8] which sets an axiomatized foundation in mathematical logic for many formal specification languages (including Z, B, and VDM). The axioms and

[*]School of Computer Science and Electronic Engineering, University of Essex, UK
[†]Centre for Inquiry, Amherst, NY, USA
[‡]Software Engineering Institute, Carnegie-Mellon University, USA
[§]University of Hawaii, USA

semantics of LePUS3 are defined using finite model theory. The relation between LePUS3 specifications (*charts*) and programs is well-defined [9] (formulated using the notion of an *Abstract Semantics* function) and the problem of satisfaction is Turing-decidable. Furthermore, consistency between any LePUS3 chart and a standard Java 1.4 [10] program—henceforth, the problem of *verification*—can be established by a click of a button. This quality is demonstrated with the Two-Tier Programming Toolkit (discussed in §6), a tool which fully automates the verification of LePUS3 charts against Java 1.4 programs in reasonable time.

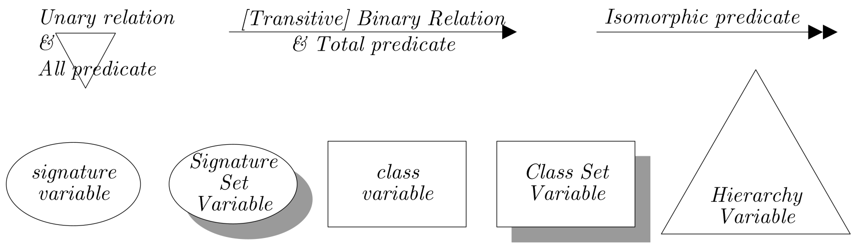


Figure 1: LePUS3 vocabulary, some visual tokens not used in this paper were omitted

The detailed syntax, axioms and truth conditions which constitute the logic of LePUS3 are laid out in [11]. Due to space limitations, our presentation focuses on a single example of the specification and verification of an informal hypothesis. LePUS3 also effectively specifies many other design patterns [12] and the design of object-oriented class libraries encoded in any class-based programming language (e.g. C++, Smalltalk). However, to maintain decidability the language is largely limited to the structural an creational aspects of such designs. In §2 we define informally our initial informal hypothesis, which we refine in §3 and §4 by specifying the design in LePUS3 and offer an abstract representation ('abstract semantics') of the implementation, respectively. In §5 we present a logic proposition that formalizes our original hypothesis and prove it. In §6 we present a tool which fully automates the verification process and discuss an experiment we are currently undertaking, which will test our predicted benefits in program comprehension, conformance and maintenance.

2 The problem

As a leading example we focus on a common claim (e.g. [13][14][15]) that the package `java.awt` in version 1.4 of the standard distribution ('Software Development Kit' [16]) of the Java programming language [10] 'implements' the Composite design pattern, quoted in Hypothesis 1.

Hypothesis 1. `java.awt` implements the Composite design pattern

In this section we examine the informal parts of Hypothesis 1: the Composite design pattern and package `java.awt`. 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 contribution to 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.*[17]

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

Intent: Compose objects into tree structures to represent part-whole hierarchies.
Participants:
– *Component:* Declares a basic interface, implementing default behaviour as appropriate.
– *Leaves:* Have no children, implements/extends superclass behaviour as appropriate.
– *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 its children, possibly performing additional operations before and/or after forwarding.

Table 1: The Composite design pattern [17] (abbreviated)

2.2 Package `java.awt`

Package `java.awt` (‘Abstract Window Toolkit’) is part of the standard distribution of Java Software Development Kit 1.4 [16] 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 of the package’s source code that corroborate Hypothesis 1 are provided in Program 1. All references to `java.awt` shall henceforth refer exclusively to those aspects listed in Program 1.

```
public abstract class Component ... {
  public void addNotify() ...
  public void removeNotify() ...
  protected String paramString() ... }
public class Button extends Component ... {
  public void addNotify() ...
  protected String paramString() ... }
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() { String str = super.paramString(); ... } ... }
```

Program 1: `java.awt` [16] (abbreviated)

3 Specification

Most contemporary modelling languages [18] and notations offer a means of representing implementation minutiae. Design patterns however describe design motifs: abstractions that are not tied in to specific programs. Therefore, the representation of design patterns requires accurately capturing generic abstractions involving collections of entities (e.g. ‘*composite*’, ‘*component*’) that are characterized not by a particular implementation but by their properties and relations (e.g., ‘*composite* is a class that has children of type *component*'). Our Design Description Language must therefore be useful in representing generically, among others, the category of entities and relations which constitute the building-blocks of design patterns, namely [sets of] classes, [sets of] methods, and their correlations.

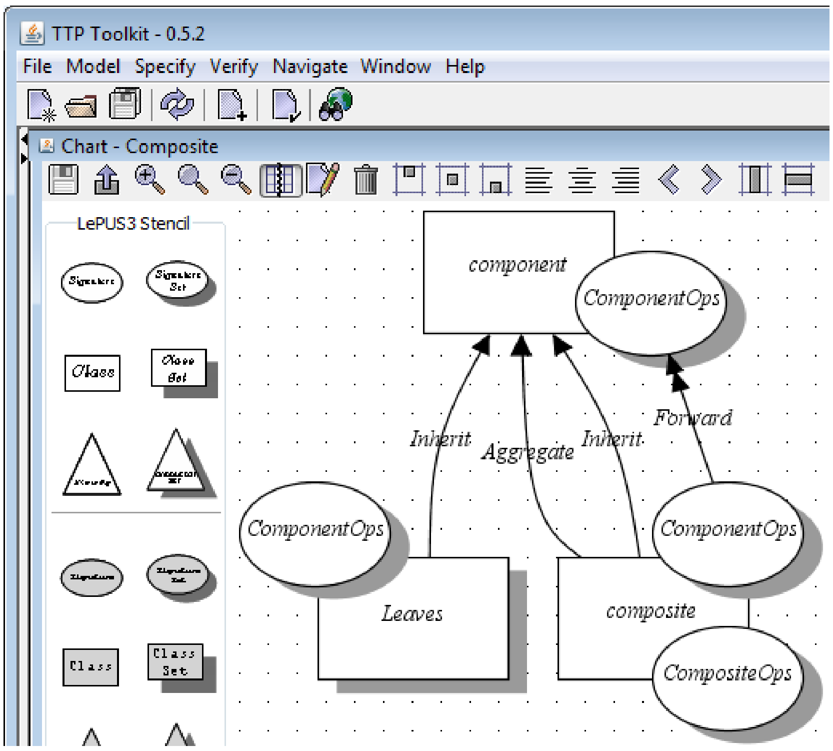


Chart 1: The Composite design pattern specified in LePUS3 (chart Composite) using the TTP Toolkit

LePUS3 was tailored specifically for this purpose, while keeping focus on automated verification. For example chart Composite (Chart 1) captures much of the informal description of the Composite design pattern (Table 1). A LePUS3 chart consists of a set of *well-formed formulas*, each of which is composed of a well-formed 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*, which represent correlations between the entities being modelled. The meaning of chart Composite is captured by the truth conditions spelled out in Table 2.

Given the formal specification of the pattern and the truth conditions of satisfying this specification, we may now rephrase our informal hypothesis in slightly more rigorous fashion as demonstrated in Hypothesis 2:

Hypothesis 2. `java.awt` 'implements' chart Composite

4 Abstract Semantics

When speaking of a 'program' or an 'implementation' we usually refer to the source code, normally represented by a complex collection of text files describing myriad implementation minutiae distributed across a directory structure. Source code is a difficult medium to reason about, not the least so because in practical circumstances it normally contains thousands (or millions) of lines of code. For example, the unabbreviated source code of only four classes from `java.awt` spans over ten thousand lines. Reasoning

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 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: `extend` or `implement`) (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

Table 2: Truth conditions for chart Composite

therefore requires a simplified picture of the program, hence the motivation for introducing the notion of *abstract semantics*.

In this context, an abstract semantics is a finite structure in model theory, which is simply a set of atomic entities and relations between them (implemented as a set of tables in a relational database). Specifically, a *finite structure* $\mathfrak{F}$ [11][19] 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, and $\mathbb{R}$ is the (finite) set of relations between them, e.g.:

$$\mathbb{U} = \{ \underline{\texttt{Container}}, \ldots, \underline{\texttt{addNotify()}}, \ldots \} \tag{1}$$

$$\mathbb{R} = \{ \underline{\mathit{Class}}, \underline{\mathit{Method}}, \underline{\mathit{Signature}}, \underline{\mathit{Inherit}}, \underline{\mathit{Aggregate}}, \ldots \} \tag{2}$$

Note that to maintain their distinction, items in the model are underlined. Each atomic entity in the universe $\mathbb{U}$ is 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). In other words, $\mathbb{U}$ is the (disjoint) union of the unary relations $\underline{\mathit{Class}}$, $\underline{\mathit{Method}}$ and $\underline{\mathit{Signature}}$. Each unary (binary) relation in $\mathbb{R}$ is a finite set of 1-tuples (2-tuples) of atomic entities. For example, the unary relation $\underline{\mathit{Class}}$ is a set of 1-tuples, one for each of the four classes in `java.awt`. The binary relation $\underline{\mathit{Inherit}}$ is a set which contains all the pairs of types $\langle cls, supercls \rangle$ in `java.awt` such that *cls* extends/implements/is-subtype-of *supercls*. Likewise, the binary relation $\underline{\mathit{Aggregate}}$ contains pairs of classes $\langle cls, elementType \rangle$ 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, $\langle invoker, invoked \rangle$, 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 (expressions in the programming language) into finite structures. For example, $\mathcal{A}_{Java1.4}$ [9] is an abstract semantics function which represents the mapping from each Java 1.4 program to a finite structure.

$$\mathcal{A}_{Java1.4} : \mathbb{JAVA}1.4 \longmapsto \mathfrak{F}^* \tag{3}$$

where $\mathbb{JAVA}1.4$ stands for the set of all well-formed Java 1.4 programs and $\mathfrak{F}^*$ stands for the (enumerable) set of all possible finite structures.

Abstract semantics functions allow us to determine exactly how the source code of programs can be abstracted. For example, we may use $\mathcal{A}_{Java1.4}$ to define the finite structure for `java.awt` as follows:

$$\mathcal{A}_{Java1.4}(\texttt{java.awt}) \tag{4}$$

We require that abstract semantics functions are fully Turing-decidable such that they always terminate; in practical terms this means that $\mathcal{A}_{Java1.4}$ can, in principle, be implemented as a static analyzer. Such an analyzer is implemented in the Two-Tier Programming Toolkit (§6), a tool which parses any Java 1.4 program and generates a representation of a finite structure therefrom (a relational database) [9]. However, static analysis is not without its limitations, and as such we do not currently capture much of the behavioural aspects of programs, for example temporal information and program state.

Alternatively, other abstract semantics functions can be equally used to represent programs in any class-based object-oriented programming language (e.g. C++, C#, Smalltalk). For example, if an abstract semantics function is defined for the C++ programming language: $\mathcal{A}_{C++} : \mathbb{C}++ \longmapsto \mathfrak{F}^*$, the very same specification and verification mechanisms described in this paper can be applied to programs written in C++.

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 LePUS3 chart which specifies that pattern. Hypothesis 2 can thus be redefined in these terms as follows:

Hypothesis 3. $\mathcal{A}_{Java1.4}$(`java.awt`) *satisfies* Composite

In the following section we define this 'stisfies' relation, and recast Hypothesis 3 as a mathematical proposition.

5 Verification

By verification we refer to the rigorous, conclusive, and decidable process of establishing or refuting whether a particular program is consistent with a given LePUS3 specification (chart). An automated process of verification therefore consists of executing an algorithm which determines if a program p (modelled using the notion of abstract semantics) satisfies a LePUS3 chart Ψ.

The conditions for 'satisfying' Ψ are modelled after the standard Tarski's truth conditions for the classical logic, as demonstrated in Table 2. A satisfies proposition is represented using the standard semantic entailment symbol $\models$, allowing us to recast Hypothesis 3 as the following (decidable) proposition:

Hypothesis 4. $\mathcal{A}_{Java1.4}$(`java.awt`) $\models$ Composite

Charts modelling design motifs such as Composite contain variable terms. To show that such a chart 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. Formally the semantic entailment in Hypothesis 4 holds if and only if there exists an assignment that maps each variable in Composite to specific elements of a given program, in this case `java.awt`. Such an assignment is presented in Table 3, where the search for assignments is a matter of *pattern detection* and therefore beyond the scope of this paper.

$g(composite)$	=	<u>`Container`</u>
$g(component)$	=	<u>`Component`</u>
$g(Leaves)$	=	{ <u>`Button`</u> , <u>`Canvas`</u> }
$g(ComponentOps)$	=	{ <u>`addNotify()`</u> , <u>`removeNofity()`</u> , <u>`paramString()`</u> }
$g(CompositeOps)$	=	{ <u>`getComponents()`</u> , <u>`getComponent(int)`</u> }

Table 3: Assignment g mapping variables in Composite to entities in `java.awt`

Hypothesis 4 can now be recast as a proposition such that the abstract semantics of `java.awt` satisfy the Composite chart under assignment g, a claim represented in Hypothesis 5 using the standard notation.

Hypothesis 5. $\mathcal{A}_{Java1.4}(\texttt{java.awt}) \models_g \mathsf{Composite}$

The proposition in Hypothesis 5 imposes specific conditions on the existence of specific entities and sets of entities in `java.awt` and on specific correlations amongst them. To prove it we refer back to Table 2, which constitutes two kinds of conditions:

1. **Truth conditions for terms.** For example, class *composite* is satisfied by virtue of assignment g, and the 1-tuple $\langle\, \underline{\texttt{Container}} \,\rangle$ in relation $\underline{\mathit{Class}}$.
2. **Truth conditions for formulas.** For example, the Inherit relation between class *composite* and class *component* is satisfied by virtue of g, and the pair $\langle\, \underline{\texttt{Container}}, \underline{\texttt{Component}} \,\rangle$ in the relation $\underline{\mathit{Inherit}}$.

Table 4 is demonstrates the proof for Hypothesis 5, the precise elements of $\mathcal{A}_{Java1.4}(\texttt{java.awt})$ which satisfy the truth conditions of Chart Composite (Table 2).

$\ldots, \langle\, \underline{\texttt{Container}} \,\rangle, \langle\, \underline{\texttt{Component}} \,\rangle \in \underline{\mathit{Class}}$	$\models$	(a)
$\ldots, \langle\, \underline{\texttt{Container}}, \underline{\texttt{Component}} \,\rangle \in \underline{\mathit{Aggregate}}$	$\models$	(d)
$\ldots, \langle\, \underline{\texttt{Container}}, \underline{\texttt{Component}} \,\rangle \in \underline{\mathit{Inherit}}$	$\models$	(e)
$\ldots, \langle\, \underline{\texttt{Container.getComponents()}} \,\rangle \in \underline{\mathit{Method}}$		
$\ldots, \langle\, \underline{\texttt{getComponents()}}, \underline{\texttt{Container.getComponents()}} \,\rangle \in \underline{\mathit{SignatureOf}}$		
$\ldots, \langle\, \underline{\texttt{Container}}, \underline{\texttt{Container.getComponents()}} \,\rangle \in \underline{\mathit{Member}}$	$\models$	(g)
$\ldots, \langle\, \underline{\texttt{Container.addNotify()}}, \underline{\texttt{Component.addNotify()}} \,\rangle \in \underline{\mathit{Forward}}$	$\models$	(i)
		$\ldots$

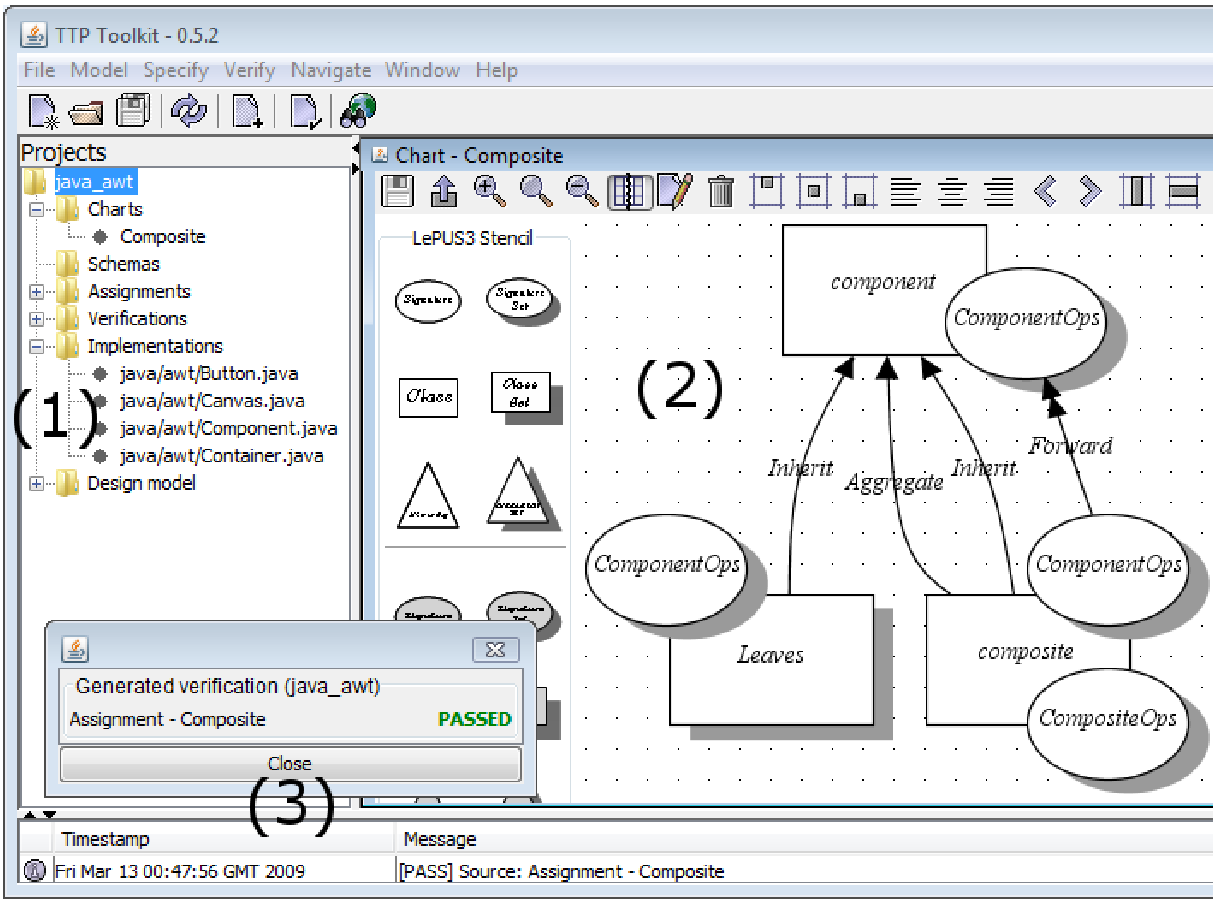
Table 4: Proof of Hypothesis 5 (abbreviated)

From this proof we conclude that `java.awt` indeed satisfies the Composite pattern. While the notion of verification as demonstrated is straightforward, manually producing the proof is a tedious, error-prone process. Such proofs require the software designer to check the validity of hundreds and thousands of clauses. It also requires intimate knowledge of both the abstract semantics of the implementation and of the specification language. Worse, the proof process would have to be repeated each time the implementation, or the design, change. However, verifying LePUS3 charts need not be a Herculean manual task as it can be fully automated, as described in the next section.

6 Tool Support

The Two-Tier Programming project [20] has recently completed implementing version 0.5.2 of the Two-Tier Programming (TTP) Toolkit. The TTP Toolkit is a prototype that integrates the representation of programs in two layers of abstraction: the design—a set of LePUS3 charts, and the implementation—a set of standard Java 1.4 source code files.

Our demonstration focuses on Figure 2, and begins with choosing the implementation (point 1); the TTP Toolkit supports the selection and static analysis (generating finite structures) of Java 1.4 source code, which in this case is the four .java files from `java.awt` (`Button.java`, `Canvas.java`, `Component.java` and `Container.java`). The TTP Toolkit also supports the composition and editing of specifications (LePUS3 charts), such as Chart Composite (point 2). Finally, the TTP Toolkit fully automates verification of programs against charts at the click of a button, such as the proof discussed in



Generated verification (java_awt)
Assignment - Composite PASSED

Fri Mar 13 00:47:56 GMT 2009 [PASS] Source: Assignment - Composite

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

this paper. This is depicted by a window stating that verification was successful (point 3), as indicated by the text ‘PASSED’, as well as by a status message in the console.

Additionally, it is extremely important that when verification fails it generates a simple explanation so that the inconsistency between specification and implementation can be rectified. For example, consider reversing the forwarding relation in chart Composite, a change which will fail verification against `java.awt`. The TTP Toolkit symbolically reports this failure to the user (Table 5), clearly indicating where the problem originates in chart Composite.

`java.awt.Component.removeNotify()` does not forward to any of the entities in: { `java.awt.Container.removeNotify()` , `java.awt.Container.addNotify()` }

Table 5: Summary of the explanation provided by the TTP Toolkit on verification failure

To summarize, the TTP Toolkit supports the following tasks:

- **Specifying.** The TTP Toolkit can be used to create, edit and view LePUS3 charts.
- **Analysing.** The TTP Toolkit statically analyses any (arbitrarily-large) well-formed Java 1.4 program and generates a relational database representing its abstract semantics, defined by $\mathcal{A}_{Java1.4}$.

- **Verifying.** The TTP Toolkit can conclusively and efficiently determine whether a given implementation satisfies a LePUS3 chart by a click of a button, and within reasonable time.

The TTP Toolkit has also been used to model, specify and verify other cases than that presented here[20]. For example it has been used to prove that `java.io` package is not consistent with the Decorator design pattern [12][17], while adding evidence to the claim that said package is consistent with a variation of the pattern [15].

Indeed, specification need not necessarily precede verification; verification being just one of many tools that aid in the development and understanding of programs. Specifically the TTP Toolkit also supports reverse engineering and program visualization. The Design Navigator [7][21] is a design recovery tool that allows a user to navigate through the design of Java 1.4 programs by reverse-engineering LePUS3 charts therefrom. To do so, the Design Navigator creates the abstract semantic representation of programs, uses the verification engine to detect correlations between [sets of] classes and methods, and represents them at the appropriate level of abstraction.

6.1 Empirical validation

We believe that TTP Toolkit can dramatically increase the productivity of software engineers. To test this claim we have designed and started conducting an experiment that compares the TTP Toolkit against a standard commercial integrated development environment. The experiment measures the performance of (mostly postgraduate) students in carrying out a variety of tasks under controlled conditions, designed to test the following specific claims:

- **Comprehension:** Effort required to understand the design and structure of arbitrarily-large programs, measured in time, is significantly reduced;
- **Conformance:** Overall dependability of programs, measured in terms of conformance of the implementation to the design specifications, can be significantly improved;
- **Evolution:** The cost of software maintenance and/or re-engineering, measured in terms of time, can be significantly reduced.

Preliminary results suggest that the TTP Toolkit radically decreases the length of time required to carry out software engineering tasks.

7 Summary

We presented LePUS3, an object-oriented Design Description Language, and demonstrated how LePUS3 can be used to specify (model) design patterns. We re-formulated an informal hypothesis, which claimed that the Composite design pattern is implemented by the `java.awt` package, as a mathematical proposition and sketched its proof. We also described the Two-Tier Programming Toolkit, a tool which can be used to compose object-oriented design specifications in LePUS3, statically analyse Java 1.4 programs, and verify them to establish whether they are consistent with the design specifications. Finally, we discussed an experiment designed to test our claims concerning the Two-Tier Programming Toolkit.

Acknowledgments. This work was partially funded by the UK's Engineering and Physical Research Council, and the University of Essex's Research Promotional Fund. The authors wish to thank Raymond Turner for his numerous contributions to this project.

References

[1] Frederick P. Brooks. No silver bullet: Essence and accidents of software engineering. *IEEE Computer magazine*, 20(4):10–19, April 1987.

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

[3] M. M. Lehman. Laws of software evolution revisited. In *Proceedings of the 5th European Workshop on Software Process Technology*, pages 108–124, Nancy, France, October 1996. Springer-Verlag.

[4] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *SIGSOFT Softw. Eng. Notes*, 17(4):40–52, 1992.

[5] Nenad Medvidovic and Richard N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Trans. Softw. Eng.*, 26(1):70–93, 2000.

[6] Toufik Taibi. *Design Patterns Formalization Techniques*. IGI Global, Hershey, USA, March 2007.

[7] Epameinondas Gasparis, Jonathan Nicholson, and Amnon H. Eden. LePUS3: an Object-Oriented design description language. In *5th Intl. Conf. on the Theory and Application of Diagrams*, Herrsching, Germany, September 2008.

[8] Raymond Turner. The foundations of specification. *J Logic Computation*, 15(5):623–662, October 2005.

[9] Jonathan Nicholson, Amnon H Eden, and Epameinondas Gasparis. Verification of LePUS3/Class-Z specifications: Sample models and abstract semantics for java 1.4. Tech. Rep. CSM-471, ISSN 1744-8050, School of Computer Science and Electronic Engineering, University of Essex, December 2007. `http://lepus.org.uk/ref/verif/verif.pdf`.

[10] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java Language Specification*. Addison-Wesley Professional, 3rd edition edition, 2005.

[11] Amnon H Eden, Epameinondas Gasparis, and Jonathan Nicholson. LePUS3 and Class-Z reference manual. Tech. Rep. CSM-474, ISSN 1744-8050, School of Computer Science and Electronic Engineering, University of Essex, December 2007. `http://lepus.org.uk/ref/refman/refman.pdf`.

[12] Amnon H Eden, Epameinondas Gasparis, and Jonathan Nicholson. The 'Gang of four' companion: Formal specification of design patterns in LePUS3 and Class-Z. Tech. Rep. CSM-472, ISSN 1744-8050, School of Computer Science and Electronic Engineering, University of Essex, December 2007. `http://lepus.org.uk/ref/companion/companion.pdf`.

[13] Jing Dong and Yajing Zhao. Experiments on design pattern discovery. In *Proceedings of the Third International Workshop on Predictor Models in Software Engineering*, page 12. IEEE Computer Society, 2007.

[14] Jochen Seemann and Jürgen Wolff von Gudenberg. Pattern-based design recovery of java software. In *Proceedings of the 6th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 10–16, Lake Buena Vista, Florida, United States, 1998. ACM.

[15] Stephen A. Stelting and Olav Maassen. *Applied Java Patterns*. Prentice Hall, 2002.

[16] Sun Microsystems. *Java 2 SDK, Standard Edition Documentation, version 1.4.2*. Sun Microsystems, 2004.

[17] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, November 1994.

[18] Object Management Group. UML 2.0 superstructure specification. Technical report, August 2005. `http://www.omg.org/cgi-bin/apps/doc?formal/05-07-04.pdf`.

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

[20] Jonathan Nicholson, Epameinondas Gasparis, and Amnon H Eden. The Two-Tier programming project, 2008. `http://ttp.essex.ac.uk/`.

[21] Epameinondas Gasparis, Amnon H. Eden, Jonathan Nicholson, and Rick Kazman. The design navigator: Charting java programs. In *Companion of the 30th international conference on Software engineering*, pages 945–946, Leipzig, Germany, May 2008. ACM.