



Terms and Conditions of Use of Digitised Theses from Trinity College Library Dublin

Copyright statement

All material supplied by Trinity College Library is protected by copyright (under the Copyright and Related Rights Act, 2000 as amended) and other relevant Intellectual Property Rights. By accessing and using a Digitised Thesis from Trinity College Library you acknowledge that all Intellectual Property Rights in any Works supplied are the sole and exclusive property of the copyright and/or other IPR holder. Specific copyright holders may not be explicitly identified. Use of materials from other sources within a thesis should not be construed as a claim over them.

A non-exclusive, non-transferable licence is hereby granted to those using or reproducing, in whole or in part, the material for valid purposes, providing the copyright owners are acknowledged using the normal conventions. Where specific permission to use material is required, this is identified and such permission must be sought from the copyright holder or agency cited.

Liability statement

By using a Digitised Thesis, I accept that Trinity College Dublin bears no legal responsibility for the accuracy, legality or comprehensiveness of materials contained within the thesis, and that Trinity College Dublin accepts no liability for indirect, consequential, or incidental, damages or losses arising from use of the thesis for whatever reason. Information located in a thesis may be subject to specific use constraints, details of which may not be explicitly described. It is the responsibility of potential and actual users to be aware of such constraints and to abide by them. By making use of material from a digitised thesis, you accept these copyright and disclaimer provisions. Where it is brought to the attention of Trinity College Library that there may be a breach of copyright or other restraint, it is the policy to withdraw or take down access to a thesis while the issue is being resolved.

Access Agreement

By using a Digitised Thesis from Trinity College Library you are bound by the following Terms & Conditions. Please read them carefully.

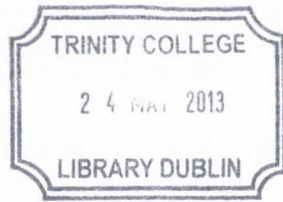
I have read and I understand the following statement: All material supplied via a Digitised Thesis from Trinity College Library is protected by copyright and other intellectual property rights, and duplication or sale of all or part of any of a thesis is not permitted, except that material may be duplicated by you for your research use or for educational purposes in electronic or print form providing the copyright owners are acknowledged using the normal conventions. You must obtain permission for any other use. Electronic or print copies may not be offered, whether for sale or otherwise to anyone. This copy has been supplied on the understanding that it is copyright material and that no quotation from the thesis may be published without proper acknowledgement.

**Specification and Verification of Design Pattern Structure,
Behaviour and Variation**

Ashley Sterritt

A Dissertation submitted to the University of Dublin, Trinity College
in fulfillment of the requirements for the degree of
Doctor of Philosophy (Computer Science)

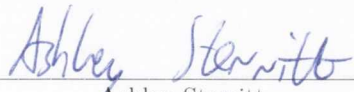
April 2013



Thesis 9959

Declaration

I, the undersigned, declare that this work has not previously been submitted to this or any other University, and that unless otherwise stated, it is entirely my own work.



Ashley Sterritt

Dated: April 30, 2013

Permission to Lend and/or Copy

I, the undersigned, agree that Trinity College Library may lend or copy this Dissertation upon request.



*Standing on the shoulders of giants...
leaves me cold*

Michael Stipe, 'King of Birds'

Acknowledgements

I would like to thank my supervisor Professor Vinny Cahill for his time and support over the years. I appreciate his rigorousness and patience. I would also like to thank Dr Siobhán Clarke for reading and providing feedback on my work. I benefited greatly from my exposure to intelligent and enthusiastic researchers. I remember especially conversations with Dr. Marcin Karpinski, Dr. Ray Cunningham and Dr. Anthony Harrington. I thank Dr. Serena Fritsch and Marco Slot for reading drafts of this thesis, and for their friendship at a stressful time. I would also like to thank Warren Kenny, my programming guru.

Mélanie has been incredibly supportive and understanding, especially since she realized that I might actually finish sometime. She has helped in many many ways. Finally, I would like to thank my family for their support. My mother has taught me to value education and learning. This thesis is for her.

Ashley Sterritt

University of Dublin, Trinity College

April 2013

Abstract

Design patterns are generic solutions to commonly-occurring object-oriented software design problems that display good design properties such as extensibility or loose coupling. During software maintenance, earlier design decisions, such as the application of design patterns, can be violated, gradually reducing software quality in a phenomenon known as ‘architectural drift’. Specifications serve to formalize design decisions and can be compared directly to implementations, as well as being useful in communication. Precise specification of patterns and automated verification of the conformance of implementations to the specifications can help to avoid architectural drift, preserving software quality. Specification languages and verification of design patterns can also be used for legacy code understanding and the generation of quality metrics.

Design patterns place constraints on multiple entities (objects, classes and inheritance hierarchies). They also describe generic interactions between entities, whose number and type are unknown. This second characteristic is a distinguishing feature of patterns and means they present a subtly different specification challenge to concrete software architectures. Pattern catalogues typically describe a number of trade-offs and optional features to consider when implementing a particular design pattern. Therefore, it is difficult to produce one specification that covers all the potential pattern implementation variants. For this reason, many existing approaches to design-pattern specification and verification have focused on only the structure and behaviour common to all variants, producing specifications that are vague and lead to many false positives during verification. Some recent research has focused on addressing design pattern variants directly, but this has focused on structure only and lacked an accompanying verification tool.

In our analysis of the widely-used Gang of Four (GoF) pattern catalogue, we identified five categories of invariants that a design pattern specification language should be capable of specifying. Of these, three were found to be insufficiently addressed by the state of the art in

design pattern specification and verification: invariants relating to inter-class dependency, an object’s runtime state and the runtime properties of data structures. Existing design pattern specification languages were found to suffer from numerous deficiencies, such as, a lack of expressiveness, imprecise semantics, no verification support and/or verification based on only simple or sporadically-applied program analyses. In this thesis, we focus on the specification and verification of pattern variants and of the insufficiently-addressed invariant categories identified.

This thesis presents Alas, a precise specification language that is capable of expressing constraints in each of the five invariant categories that we identified. It provides syntax and semantics for the description of design pattern variants as well as for the generic specification necessary to describe design patterns. The Alas Verification Tool (AVT) can read Alas specifications and verify that Java source code conforms to them. It uses data-flow analysis and deals with object-oriented issues such as aliasing and modular verification.

To evaluate Alas and AVT, we created our own benchmark based on Alas specifications, identifying GoF pattern instances in a number of code bodies that make extensive use of patterns and are commonly analyzed by related work. We aggregated some existing benchmarks by including pattern instances included in them that also conform to our specification. Small extensions to the benchmark were made to increase the generality of the analysis results. To our knowledge, this is the first sizeable benchmark to include design pattern variants.

Verification of novel invariant categories by AVT is demonstrated on the benchmark and is shown to be accurate and scalable to medium-sized examples. The novel invariant categories provided by Alas and verified by AVT allow us to address patterns typically overlooked by the literature, as well as novel aspects of more well-supported patterns. Specification and verification of design pattern variants allows us to identify pattern instances overlooked by existing tools, and to distinguish between instances of different variants indistinguishable by existing tools.

Contents

Acknowledgements	ix
Abstract	ix
List of Tables	xvii
List of Figures	xix
Chapter 1 Introduction	1
1.1 Motivation	2
1.2 Background	3
1.3 Scope of the Thesis	5
1.4 Key Contributions of the Thesis	6
1.5 Roadmap of the Thesis	9
Chapter 2 Design Pattern Specification and Verification Classification	10
2.1 Introduction	10
2.2 Scope and related work	12
2.2.1 Other DPSL and DPVT reviews in the literature	15
2.3 Abstraction Levels in Design Pattern Specification	15
2.3.1 Generic pattern specification	16
2.3.2 Variant-specific pattern specification	17
2.4 Specification of Design Patterns	18
2.4.1 Syntactic elements	20
2.4.2 Invariant elements	22
2.4.3 Invariant dynamism	36
2.4.4 Conceptual elements	36

2.4.5	Summary	37
2.5	Verification of Design Pattern Implementations	38
2.5.1	Invariant elements	39
2.5.2	Languages compared during verification	45
2.5.3	Conformance relation	46
2.5.4	Pattern instance classification	48
2.5.5	DPVT use cases	49
2.5.6	Program analysis	50
2.5.7	Assigning pattern roles to implementation actors	55
2.5.8	Summary	56
2.6	Conclusions	57
Chapter 3 Alas		59
3.1	Introduction	60
3.1.1	Language basis rationale	60
3.1.2	Alas design decisions	63
3.1.3	UML as a basis for design pattern specification	64
3.1.4	Sample Alas specification	65
3.2	Structural specification in Alas	68
3.2.1	Dependency invariants	69
3.2.2	Structural variant specification	72
3.2.3	Structural cardinality invariants	77
3.3	Behavioural specification in Alas	80
3.3.1	Control-flow invariants	81
3.3.2	Object-state invariants	87
3.3.3	Data structure invariants	98
3.3.4	Interaction invariants	102
3.3.5	Behavioural pattern variant specification	104
3.3.6	Behavioural cardinality invariant specification	107
3.4	Summary	108
Chapter 4 AVT Implementation		111
4.1	Verification requirements imposed by Alas	112
4.2	Shape analysis algorithm	113

4.2.1	Intra-procedural analysis	116
4.2.2	Inter-procedural analysis	120
4.2.3	Termination	123
4.2.4	Efficiency	124
4.3	Alas clause verification	125
4.3.1	Conservative verification	126
4.3.2	Dependency	127
4.3.3	Object state	127
4.3.4	Data-structure	128
4.4	Miscellaneous implementation issues	129
4.4.1	Synthetic method CFGs	129
4.4.2	Unimplemented features	130
4.5	Summary	131
Chapter 5 Pattern Specification and Benchmark		133
5.1	Benchmark Construction Methodology	133
5.1.1	Code Body Selection	134
5.1.2	Inspection method	136
5.2	Benchmark	138
5.2.1	Dependency	140
5.2.2	Object State	145
5.2.3	Data structure	148
5.3	Summary	153
Chapter 6 AVT Verification Evaluation		155
6.1	Methodology	156
6.1.1	Scope	156
6.1.2	Metrics	156
6.2	Results	157
6.2.1	Dependency	157
6.2.2	Object State	159
6.2.3	Data Structure	164
6.3	Summary	168

Chapter 7 Conclusions	170
7.1 Specific conclusions	170
7.2 General conclusions	173
7.3 Future work	175
Appendix A Control-flow invariant semantics	177
A.1 Sequencing	177
A.2 Selection	178
A.3 Iteration	181
A.4 Lifeline semantics	182
A.5 Generic behaviour	183
Appendix B Benchmark Observations and Ancillary Specifications	185
B.1 Dependency	185
B.1.1 Abstract Factory	185
B.1.2 Command	186
B.1.3 Builder, Strategy, State	186
B.2 Object-state	187
B.2.1 Prototype	187
B.2.2 Observer and Memento	188
B.3 Data-structure	189
B.3.1 Decorator	189
B.3.2 CoR	190
B.3.3 Composite	190
Appendix C Verification Examples	193
C.1 Dependency	193
C.2 Object state	195
C.3 Data structure	196
C.3.1 Decorator	196
C.3.2 Composite	197
Appendix D Aggregated benchmarks	200
D.1 Dependency	200
D.1.1 Abstract Factory	200

D.1.2 Command	201
D.2 Object state	203
D.2.1 Prototype	203
D.3 Data structure	204
D.4 Decorator	204
D.4.1 Composite	204
Appendix E Generic behaviour specification	208
Appendix F List of Acronyms	229

List of Tables

2.1	Classification of the support of each DPSL for each of the syntactic elements	23
2.2	Classification of the support of each DPSL for each of the invariant types .	29
2.3	Classification of the support of each DPVT for each of the invariant types .	41
2.4	Relation between specification and implementation language for each DPSL	48
2.5	Relation between specification and implementation language for each DPVT	49
2.6	Analysis use case, program analysis and implementation language classification of each DPSL's associated DPVT	51
2.7	Analysis use case, program analysis and implementation language classification of each DPVT	52
3.1	Ordering and element uniqueness of each of the Alas (and OCL) collection kinds	97
4.1	Distinguishing features of the shape analysis algorithms implemented by Rinetzky et al. and AVT	123
4.2	A summary of the requirements imposed by the novel invariant categories of Alas and features of object-oriented programming languages, and the AVT design approaches addressing each requirement	132
5.1	Code bodies and how often they occur in the evaluation of existing DPVTs	135
5.2	Novel invariant categories in Alas required for each of the patterns in GoF design pattern catalogue specifications	139
5.3	Intended instances of variants of the Prototype pattern in the Alas benchmark for each code body	146
6.1	AVT analysis results for the Abstract Factory pattern	158
6.2	AVT analysis results for the Command pattern	159

6.3	AVT analysis results for the Prototype pattern	162
6.4	AVT analysis results for the Decorator pattern	165
6.5	AVT analysis results for the Composite pattern	167
D.1	Instances of variants of the Abstract Factory pattern in each benchmark from the JHotDraw code body	201
D.2	Instances of variants of the Abstract Factory pattern in each benchmark from the JUnit code body	201
D.3	Instances of variants of the Abstract Factory pattern in each benchmark from the Swing code body	202
D.4	Actor names in candidate Command instances	203
D.5	Command instance classification	203
D.6	Instances of variants of the Decorator pattern in each benchmark from the JHotDraw code body	205
D.7	Instances of variants of the Decorator pattern in each benchmark from the JUnit code body	205
D.8	Instances of variants of the Decorator pattern in each benchmark from the Swing code body	205
D.9	Instances of the Composite pattern identified in both the Alas benchmark and aggregated benchmarks	207

List of Figures

2.1	Bayley and Zhu's [2010] specification of the Abstract Factory pattern, including two variants: Single factory method and Multiple factory methods	18
2.2	DPSL classification framework dimensions	19
2.3	Syntactic elements of GoF design patterns	21
2.4	Invariant categories and types: Part 1: Dependency and cardinality invariants	25
2.5	Invariant categories and types: Part 2: Control-flow invariants	26
2.6	Invariant categories and types: Part 3: Object-state and data-structure invariants	27
2.7	The specification of the Visitor pattern given in Mak et al. [2004]	31
2.8	Flyweight pattern conditional branching expressed in BPSL	32
2.9	RBML [France et al., 2004] behavioural specification of the Visitor pattern	33
2.10	Conceptual elements of GoF patterns	38
2.11	Classification framework dimensions specific to DPVTs	46
2.12	Control-flow graph for an implementation of the Singleton's <code>getInstance()</code> method (PINOT)	54
2.13	FUJABA [Wendehals and Orso, 2006] DFA for the State pattern	55
3.1	A textual specification of the FalseFaçade pattern, equivalent to the graphical specification in Figure 3.2	67
3.2	The structural specification of the FalseFaçade pattern, including a structure diagram with three class roles, three method roles and three reference variable roles. The constraint boxes define a dependency invariant, a pattern variant differing in structure from the structure diagram and a data-structure definition.	68

3.3	The behavioural specification of the <code>FalseFaçade</code> pattern. The behaviour within the <code>opt</code> operator is conditional. A data structure invariant is attached to the end of <code>receivingMethod</code> , indicating a post-condition.	69
3.4	Composite Structure diagram with two structural variant definitions, both of which add static roles to the core specification. This allows for four valid variants of the Composite pattern.	73
3.5	A variant specified in a separate SD, illustrating the use of the scoping (<code>::</code>) and substitution (<code>-></code>) operators	75
3.6	Behavioural specification of the CoR pattern involving a two-operand <code>alt</code> with an operand guarded by a basic state invariant on an object role. A single path in a valid implementation may not contain the behaviour of both operands. The call event involving <code>successor</code> 's <code>handleRequest</code> only occurs if the successor has been initialized.	84
3.7	Specifying iteration over and interaction with an unbounded collection by matching the name of the quantified variable in the loop guard (<code>obs</code>) and the selector string in the lifeline.	86
3.8	A potential (non-Alas compliant) Memento pattern specification where the objects pointed to by reference variable roles are strictly equal when the Memento's constructor returns.	88
3.9	Illustration of the usage of the <code>CopyState</code> reference variable role in a structural specification. <code>CopyState</code> may be bound to a different set of primitive and user-defined reference variables in each implementation of the Memento	90
3.10	Illustration of the <code>isCopy</code> operator, relating the copystate of two objects at a particular point in the execution. Note also the matching parameter and lifeline object role name. The diagram states that the Originator object calls the Memento passed to it as a parameter.	90
3.11	Class A satisfies the strict ownership relation with respect to class B, but class B does not satisfy the strict ownership relation with respect to class C	92
3.12	Graph of all reachable state and the included copy state for the classes defined in Figure 3.11 using the <code>comp</code> copy state definition	93
3.13	Factory method specification using the <code>isAlias</code> operator and the qualified and unqualified version of the <code>returnval</code> keyword. The <code>factoryMethod</code> is required to return the object returned by the constructor of <code>Product</code>	96

3.14	A graphical definition of a collection. Subject contains an ordered collection of potentially non-unique Observer objects	97
3.15	Subject's <code>detachObserver</code> specification illustrating collection operators and matching parameter and constraint role names. The method should remove the parameter from the list and have no other side effects.	98
3.16	CoR Behaviour diagram along with an interaction invariant using Alas data-structure operators. The specification states that a <code>DefaultHandler</code> should be reachable from every <code>handler</code> in any valid <code>chainOfResponsibility</code> . This constraint must be satisfied at the end of every method that refers to and/or mutates the <code>chainOfResponsibility</code>	104
3.17	Composite Behaviour diagram illustrating the use of the <code>var</code> operator and variant-labelled constraint boxes to specify variant-specific behaviour. The same variant is named in both the <code>var</code> operator and constraint box, meaning that both of these constraints must apply in a valid <code>parentLinks</code> variant of the Composite pattern.	106
3.18	BD illustrating the use of the scoping operator for the definition of a sub-variant and the alternative conformance relations provided by Alas. The do nothing variant may only be satisfied by implementations that satisfy the unsafe variant	106
3.19	Specification of two behavioural variants of the CoR pattern using a multiple-operand <code>var</code> . Each valid CoR implementation must perform the behaviour specified in <code>viaSuperDelegation</code> or <code>directDelegation</code> , but not both or neither.	107
3.20	Structural cardinality invariant specifying a surjective relation between Factory Methods and AbstractProducts in each <code>ConcreteFactory</code> : every <code>FactoryMethodSet</code> contains some method that initializes some subclass of each of the <code>AbstractProduct</code> classes. Both the method role representing a Factory Method and the class role representing a <code>ConcreteProduct</code> have been substituted with variables bound in the quantified specification.	109
3.21	Structural cardinality invariant specifying an injective relation between Factory Methods and <code>ConcreteProducts</code> in each <code>ConcreteFactory</code> : no two Factory Methods create the same <code>ConcreteProduct</code> . Two quantified variables are bound to a single variable substituting for a method role.	109

4.1	The running example used to illustrate the features of the shape analysis algorithm implemented in AVT	115
4.2	Control-flow graph for the example of Figure 4.1	115
4.3	Simple set method example with two formal parameters and one reference variable that may be all aliased when the method call dispatches	118
4.4	Shape graph resulting from the analysis of the first path through the example of Figure 4.1, until the end of the IF block	118
4.5	Shape graph representing the second path through the method of Figure 4.1, through the ELSE block	119
4.6	The call graph edges resulting from the analysis of the call to <code>setRealSubject</code> in Figure 4.1	122
4.7	The relationship between conservative Alas predicate verification and the choice of meet operator	126
4.8	Pseudocode for <code>isCopy</code> verification on a single shape graph	128
5.1	Alas specification of the GoF (core) and No AF variants of the Abstract Factory pattern	141
5.2	Alas specification of the structure of all variants of the Command pattern	143
5.3	Alas specification of the behaviour of the Command pattern	144
5.4	Alas specification of the Prototype’s clone method. The clone method is required to make and return a copy of ‘this’	146
5.5	OptionListModel’s clone method in the Swing code body. The <code>ListenerList</code> variable is not made an alias or assigned a deep-copied object	147
5.6	Structural specification of the Decorator pattern, along with a single structural variant	149
5.7	Alas specification of the operation method of the Composite subclass	151
6.1	DefaultListSelectionModel’s clone method in the Swing code body	160
6.2	Shape graphs output by AVT after analyzing DefaultListSelectionModel’s clone method	161
6.3	Result of applying the <code>isCopy</code> predicate to the shape graphs of Figure 6.2	162
6.4	SelectionTool’s <code>createDragTracker</code> method from the JHotDraw benchmark	164
6.5	Result of applying the data-structure invariant to SelectionTool’s <code>createDragTracker</code> method	165

B.1	Specification of the operation method for the Forward if not null variant of the Decorator pattern.	190
C.1	StandardDrawingView's selectionZOrdered in the JHotDraw code body is a 'bad' client of the Factory Method UndoableAdapter.getAffectedFigures as it creates an instance of a FigureEnumeration subclass	193
C.2	AVT output when applying the implementation dependency invariant of the Abstract Factory Client role to the method of Figure C.1	194
C.3	ElementIterator's clone method in the Swing code body	195
C.4	Result of applying the isCopy predicate to the source code of Figure C.3	196
C.5	BouncingDrawing's replace method from the JHotDraw benchmark	196
C.6	Result of applying the data-structure invariant to BouncingDrawing's replace method	197
C.7	Segment of the AVT analysis output when analyzing SuiteTest's suite method. The Composite tree is correctly classified as being free from sharing	198
C.8	Modified source code of JTree's getDefaultModel method from the Swing code body, with sharing introduced. The original simply omits the addition of the 'blue' node to the parent 'sports'	199
D.1	Composite class actor of each identified instance, along with their variant classification	207
E.1	A generic specification of the Looping Director variant of the Builder pattern. No selector is provided for the method set: any arbitrary member of the BuildPartSet method set may be called on each iteration of the loop.	209
E.2	A generic specification of a Template Method. Each of the members of the PrimitiveOperationSet is called in sequence. The set may vary in size between valid implementations.	210
E.3	Generic specification of the ConcreteFactory's factoryMethod. The specification states that for each class in the ConcreteProduct class set role, there exists an alternative path that creates an object of that class.	211

Chapter 1

Introduction

This thesis presents Alas (Another Language for pAttern Specification): a specification language capable of expressing the constraints imposed by object-oriented design patterns from each of the invariant categories identified in a novel classification framework of design pattern specification languages (DPSLs). Alas is also capable of specifying variants of design patterns that differ in terms of structure and/or behaviour. Verification involves comparing a design pattern specification in a given language to an implementation and categorizing the implementation as conforming or not conforming to the specification. We use the term Design Pattern Verification Tool (DPVT) for tools that perform this function. The Alas Verification Tool (AVT) is a DPVT capable of checking that Java source code conforms to Alas specifications. The evaluation contained in this thesis demonstrates how the combination of Alas and AVT allows properties of design pattern implementations that were previously not addressed to be specified and verified. The specification and verification of design pattern variants allows for more pattern instances to be identified in the analyzed bodies of code. Also, instances of different variants can be distinguished that are indistinguishable by other languages and tools. Finally, the thesis illustrates the wider applicability of Alas and AVT due to characteristics of the code bodies analyzed and instances uncovered.

The remainder of this chapter is structured as follows: we motivate and provide background on the specification and verification of design patterns in Sections 1.1 and 1.2, respectively. We describe the scope and key contributions of this work in Sections 1.3 and 1.4, respectively, before providing a brief roadmap of the rest of the thesis in Section 1.5.

1.1 Motivation

Design patterns are ‘simple and elegant solutions to specific problems in object-oriented software design... that have been developed and evolved over time’ [Gamma et al., 1995]. The concept of design patterns as generic solutions to be applied in multiple contexts originated with Christopher Alexander, who developed a pattern language for designing buildings and cities [Alexander, 1979]. Alexander’s work differs from subsequent work on software design patterns in that he dictates an order in which the patterns should be applied, and that his aim is to generate a complete design. Similarities, however, include the use of templates to explain patterns in natural language and a discussion in terms of a concrete example. Beck and Cunningham [1987] introduced the concept of patterns to software design, with a small group of graphical user interface (GUI) patterns for Smalltalk. Since then, a multitude of new patterns have been proposed in other catalogues and academic papers [Coplien, 1998][Gamma et al., 1995][Schmidt et al., 2000][Erl, 2008] with many focusing on object-oriented programming languages. A *pattern catalogue* is a collection of design patterns, where for each pattern a description of its intent, applicability and possibly some illustrative sample code is provided. The *intent* of a design pattern describes its intended function and (sometimes implicit) non-functional properties that a correct implementation of the pattern should display. The value of object-oriented design patterns (subsequently referred to simply as design patterns) lies in the fact that they are not the most obvious solution to novices in object-oriented software design: instead they represent years of collective experience in how to “find pertinent objects, factor them into classes at the right granularity, define class interfaces and inheritance hierarchies, and establish key relationships between them”[Gamma et al., 1995], so that the design displays some positive non-functional property, such as extensibility or loose coupling.

A software designer or developer applies a design pattern when they intend some piece of code to exhibit one of these non-functional properties. The intent is sometimes captured in comments, documentation or in variable naming conventions, but in other instances, they remain implicit. In the latter case, the intention that a piece of code should conform to some design pattern may be lost if, for example, the original developer leaves the organization to which the code belongs. A pattern can be incorrectly applied by the original developer or original design decisions can be violated during maintenance, so that the code no longer conforms to the pattern. Bieman et al. [2003] have shown that code utilizing

design patterns can be more prone to change than other code. Because the application of patterns is not always fully documented and pattern implementations are prone to change during maintenance, the invariants imposed by a design pattern may be broken and the original design may degrade in a phenomenon known as architectural erosion or drift [Perry and Wolf, 1992][van Gorp and Bosch, 2002]. Precise specification of the invariants a design pattern imposes on an implementation and automated verification based on these specifications is useful to protect the developer's initial intent when applying the design pattern and can protect against this phenomenon.

1.2 Background

Pattern specifications define a number of *roles* (classes, objects or methods), most of which are mutually exclusive, and also place constraints on how these roles interact. Thus, patterns define object-oriented protocols to be satisfied by *actors* in the implementation. Design patterns place constraints on multiple entities (objects, classes and inheritance hierarchies) and are also more generic than concrete software architectures as they describe interactions between entities, whose number, type and precise behaviour are unknown. For example, when specifying conditional behaviour in a concrete software architecture, the condition to be evaluated is known. However, some design patterns involve conditional behaviour where the actual condition differs between implementations and is not known at specification time. Also, while specifying concrete software architectures, it is not necessary to place a constraint on the number of classes in a particular role, as the number of classes is known, while such constraints are a central concern for design pattern specification [Lauder and Kent, 1998][Eden, 2001][Mak et al., 2004]. Due to the specific requirements on specification posed by the generic nature of design patterns, existing concrete software architecture specification languages are unsuitable for their specification. This requirement has motivated the development of numerous DPSLs, most of which focus on the description of the Gang of Four (GoF) pattern catalogue [Gamma et al., 1995].

Pattern catalogues typically describe a number of trade-offs and optional features to consider when implementing a particular design pattern. Therefore, it is difficult to produce one specification that covers all the potential pattern variants. Because of the existence of design pattern variants, many approaches to specification and verification of design patterns focus on only the structure and behaviour common to all variants, producing specifications

that are vague and lead to many false positives during verification. Some promising work on design pattern variants has emerged in recent years, but has tended to focus on one or a small number of patterns, analyze small or self-coded benchmarks and/or lack a DPSL or corresponding DPVT [Stencel and Wegrzynowicz, 2008][Bayley and Zhu, 2010].

Design patterns impose different types of constraints that must be satisfied by conforming implementations. The structural class patterns in the GoF catalogue require particular inheritance relations between classes, while the structural object patterns describe ways to compose objects into structures that display particular properties. The Composite pattern, for example, describes the creation of trees of composed objects, where a single object and a composite can be treated uniformly by clients, as they expose the same interface. The creational patterns aim to improve extensibility by ‘abstract[ing] the instantiation process’, and impose invariants on the creation of objects. The Singleton pattern, for example, ‘ensure[s] that a class has only one instance, and provide[s] a global point of access to it’. Finally, the behavioural patterns are concerned with the assignment of responsibilities between objects. They involve patterns of communication between objects involving sequences of method calls, but also relationships between the state of objects. The Memento pattern, for example, stores a copy of an object’s internal state ‘so that the object can be restored to this state later’. A DPSL that is capable of expressing each of the different types of invariants imposed by design patterns enables better understanding and documentation, as well as more accurate verification of design pattern implementations. Existing DPSLs suffer from a lack of expressiveness, imprecise semantics, the lack of an accompanying verification tool, or the verification tool based on them performs only simple or sporadically-applied program analyses.

Design pattern specification and verification can be used as part of either a forward or reverse engineering use case. In a forward engineering use case, the developer of a piece of code can manually identify which actors are intended to play which roles in the specification. A DPVT can then confirm that the pattern has been implemented correctly. In a reverse engineering use case, a DPVT compares every class or group of associated classes to the input specifications. Unlike in a forward engineering use case, in reverse engineering a large number of spurious instances can be identified that were never intended to be instances of the pattern, especially if the specification is vague. A large proportion of design pattern verification tools (DPVTs) fall into the category of design pattern mining tools: tools that do not have a corresponding DPSL and target reverse engineering focused on legacy code

understanding. These have hard-coded specifications of design patterns that suffer from a number of drawbacks. Firstly, they are limited to the design pattern variants that the tool developer has considered. Secondly, the tool developer's specification of the pattern is difficult to infer as it is hard-coded by the DPVT and is not always clearly documented. The provision of a DPSL along with a DPVT that is capable of verifying code against specifications written in the DPSL provides a means to specify patterns and variants not considered by the DPSL/DPVT developers, document important design decisions with precise specifications, and guard against architectural drift by enabling verification and re-verification implementations as they are extended or maintained.

1.3 Scope of the Thesis

A *design pattern* is a generic solution to a software design problem with a stated intent. A design pattern *implementation* or *instance* is the application of a pattern to a particular design problem embodied in programming language code. Some structural and behavioural features of design patterns may be implemented in different ways while still satisfying the design patterns intent. These features are *trade-off points* and their existence creates a number of valid design pattern variants. In this thesis, we define a design pattern *variant* as a solution, with an associated name, that chooses particular alternatives at some trade-off points, but may leave other choices open. The constraints imposed by all the valid variants of a design pattern can be seen as defining a space of pattern implementations that conform to these constraints. The space of all possible implementations in a given language that possess a structure and behaviour satisfying a design pattern's intent is termed the pattern's *code signature*. Each DPSL and DPVT also provide their own code signature that approximates the true signature of each pattern it specifies or supports. A code signature that is too permissive has the potential for false positives during verification, while a code signature that is too strict has the potential for false negatives during verification.

This thesis focuses on specifying and verifying design patterns that are outlined in the GoF catalogue [Gamma et al., 1995]. This catalogue has proved to be very popular and numerous documented instances of the patterns from this catalogue occur in many unrelated code bodies [Kaiser, 2001][Gamma and Beck, 2003]. Most DPSLs and DPVTs target the GoF catalogue and few in fact address patterns not contained in this catalogue [Heuzeroth et al., 2003][Stencel and Wegrzynowicz, 2008][Shi, 2007a].

As a number of problems relating to software verification are undecidable in general [Landi, 1992][Ramalingam, 2000][Reps, 2000], it is not possible to design a tool that will automatically verify conformance or non-conformance correctly in all cases. Verification tool developers have the choice to either require user input to direct the verification process, such as in theorem proving, or introduce various incompletenesses (statements that are true but cannot be proven) into the theory of the tool to guarantee termination without user input. The former option requires the user to be proficient in the formal semantic basis of the tool and methods of mathematical proof. The latter option is often preferred when the target user of the tool is a software engineer or developer and the focus is more on code understanding than on the number of provable statements. Though some software tools, such as compilers that guarantee termination of their code optimization algorithms, are not complete, they are sound, i.e., do not prove any statement to be true that is actually false. They do this, for example, by abandoning optimizations that are not provable. We, along with the vast majority of DPVTs, choose the latter option.

In order to give the work a feasible scope, a number of software constructs that are challenging to specify and verify have not been addressed. The two foremost among these are concurrency and exception handling. Concurrency complicates software verification as it greatly expands the state space of the program under analysis by removing assumptions that may be made about sequential programs. There is a large body of work focused on dealing with concurrency alone [Clarke et al., 1986][Qadeer and Rehof, 2005][Andrews et al., 2004] and numerous state-of-the-art specification languages, formalisms and tools are aimed at sequential programs only [Parkinson and Bierman, 2008][Leavens et al., 2007][Shi, 2007a]. Similarly, exception handling increases the state space of the program under analysis by increasing the number of potential flows of control, and hence data, through the program. Most DPVTs do not address exception handling [Shi, 2007a][Blewitt et al., 2005].

1.4 Key Contributions of the Thesis

The first contribution of this thesis is a thorough analysis of the GoF pattern catalogue [Gamma et al., 1995] that identified a set of thirteen design pattern invariant types that are necessary to express the intent of design patterns precisely. We classify these invariant types within five design pattern invariant categories: cardinality, dependency, control flow, object state and data structure. The capability to express this set of pattern elements can

be seen as a requirement on any DPSL. The pattern invariant categories form part of a classification framework for DPSLs and DPVTs.

The second contribution of this thesis is the classification of existing DPSLs and DPVTs using our novel framework. Of the five invariant categories identified, three were found to have invariant types that are either not expressible or imprecisely expressible in existing languages, or not verifiable by existing tools. These categories are dependency, object state and data structure. We identify specification and program analysis techniques that are suitable to express and verify the poorly-supported invariant types.

The third contribution of this thesis is the development of a DPSL called Alas, which is capable of expressing precisely and concisely all of the design-pattern invariant types identified. Alas is also capable of describing variants of design patterns, each of which are specified explicitly and combined in a disjunction that forms the specification of a given pattern. The language is based on UML 2.0 Class and Sequence diagrams and OCL [OMG, 2009] with modified and extended syntax and semantics. UML is the de facto standard for graphical object-oriented software modelling and is widely taught and used in industry. However, in its current form it is not suitable for design pattern specification for a number of reasons. For example, Le Guennec et al. [2000] describes how UML is unsuitable for the specification of cardinality invariants due to the binding semantics of UML Templates. Also, the semantics of some of the Combined Fragments introduced in UML 2.0 have been shown to be ambiguous [Lund and Stølen, 2006]. We clarify existing syntax as well as our extensions using operational semantics.

The fourth contribution of this thesis is the development of AVT, which can compare source code in the Java programming language to Alas specifications and identify whether the code conforms or does not conform to the specification. State-of-the-art DPVTs compute the potential values of variables on the stack only, while the verification of object-state and data-structure invariants requires an accurate model of the graphs of objects stored on the runtime heap. We implement a static analysis known as shape analysis, novel in the area of design pattern specification and verification, to enable the verification of these invariant types. We demonstrate in Chapter 6 that while it is not capable of verifying all invariants within the insufficiently addressed invariant categories in general (due to the inherent limitations of software verification discussed above) it is capable of verifying many of the most common cases that occur in design-pattern implementations accurately.

The fifth contribution of this thesis is the creation a benchmark of identified pattern

instances from three code bodies, for use in the evaluation of DPVTs. Despite numerous authors advocating a shared benchmark of identified pattern instances in code bodies for use by the community [Wegrzynowicz and Stencel, 2009][Fulop et al., 2008][Petterson et al., 2009][Arcelli et al., 2008], the currently available benchmarks are inadequate for a number of reasons. The pattern benchmarks where pattern instances are identified by manual inspection of the source code identify a small number of instances. This is most likely evidence of incomplete coverage of the code body, i.e., manual inspection that does not cover all the classes in the code body. Automated analyses, where complete coverage is more attainable, lack manual validation and often include large numbers of false positives. Also, benchmarks are not always accompanied by complete and unambiguous specifications of the patterns, making independent corroboration of the results difficult. We aggregated information from two existing benchmarks: one manual [Guéhéneuc, 2007] and one fully automated [Shi, 2007b]. Including information from an automated benchmark provided better coverage than existing manual benchmarks without introducing a large number of false positives because we performed manual code inspection on the instances identified automatically. We identified instances not included in any of the benchmarks by performing a keyword search based on variable naming conventions in design pattern implementations. Also, to our knowledge, this is the only sizeable benchmark to include variants of patterns.

The code bodies selected for inclusion in the benchmark were chosen for their widespread and (incompletely) documented use of patterns. They are three of the bodies most commonly analyzed by the literature, had existing publicly available benchmarks and cover a quite broad range of size in terms of number of classes and lines of code (LOC). This range of code body sizes has been a priority in the design pattern verification literature [Tsantalis, 2009][Petterson et al., 2009]. Small extensions to the code bodies were made to exercise more of the code signature of each pattern, increasing the generality of any results obtained from an analysis of the benchmark.

The sixth and final contribution of this thesis is an evaluation of Alas and AVT on the benchmark of identified pattern instances. Specifications of 13 of the 23 GoF patterns in Alas involve invariants taken from the insufficiently-addressed invariant categories identified, demonstrating the added expressiveness of the language. AVT is shown to verify instances of novel or insufficiently-addressed patterns and pattern roles. In particular, we identify instances of the Prototype and Command pattern, and the Client role in the Abstract Factory pattern. Properties of design pattern implementations that were pre-

viously not expressible or verifiable are verified by AVT. For example, we demonstrate data-structure invariants in the context of the numerous instances of the Composite and Decorator patterns in the code bodies analyzed. By specifying and verifying design pattern variants, we are able to identify valid pattern implementations not included in existing benchmarks, as well as distinguishing between variants indistinguishable by state-of-the-art DPSLs.

1.5 Roadmap of the Thesis

The remainder of the thesis is organized as follows: Chapter 2 presents a novel classification framework for DPSLs and DPVTs and classifies existing approaches using the framework. Chapter 3 describes the syntax and semantics of Alas, using examples drawn from specifications of GoF design patterns. Chapter 4 presents AVT. Both Chapters 3 and 4 provide a rationale for the design decisions involved in creating the language and tool respectively. Alas specifications and a benchmark based upon those specifications is given in Chapter 5. The evaluation of AVT is provided in Chapter 6, including an evaluation plan and its rationale. Finally, Chapter 7 presents specific and general conclusions of the thesis and outlines potential future work.

Chapter 2

Design Pattern Specification and Verification Classification

In this chapter, we present a comprehensive survey of the large body of work on specification and verification of object-oriented design patterns. A thorough analysis of the widely-used Gang of Four pattern catalogue yielded a number of invariant types essential to the precise specification of design patterns. These invariant types are separated into five invariant categories that provide the basis for a novel classification framework for design pattern specification languages and verification tools. We classify a large number of languages and tools from the literature using our framework and identify invariant categories that are poorly supported by the state-of-the-art. We identify the program analysis techniques required to support the verification of the poorly-supported invariants. Also, we assess the level of support in the literature for design pattern variant specification and verification.

2.1 Introduction

Before reviewing the large body of literature relating to the specification and verification of design patterns, a thorough analysis of the GoF pattern catalogue [Gamma et al., 1995] was performed, to identify a set of invariant categories and types that are sufficient to specify the GoF patterns precisely. The capability to express this set of pattern invariants can be seen as a requirement on any design pattern specification language. While the focus of the analysis was on the pattern's intent, we address the specification of different variants of the same pattern that share that common intent. The GoF catalogue was chosen for its popularity, and because the vast majority of specification languages in the literature

focus on supporting elements of these patterns. While the identified invariants were derived from a single catalogue, it is expected that they are applicable to a wide variety of object-oriented design patterns at a similar level of abstraction, covering numerous application domains. Some invariant types could be expanded or changed to support language concepts or properties that do not appear in the GoF catalogue, for example, threads or different types of aggregation relationships. While it is difficult for such a set of invariants to be exhaustive, the set should be large enough to capture the essential properties of the pattern, so that a specification can be created that will not be too vague and yield many false positives during verification. For instance, a specification language that only allows structural relations between classes to be expressed overlooks the behavioural invariants important for precisely representing patterns, e.g., conditional branching in the Flyweight and Singleton patterns.

Invariants are not the only elements required to specify design patterns: the required elements identified are categorized as either syntactic, invariant or conceptual elements. Syntactic elements represent elements of OOPL syntax. A conceptual element is an abstract concept that leaves no consistent *signature* on the application code, and is thus difficult or impossible to verify by automated analysis. For example, the distinction between intrinsic and extrinsic state in the context of the Flyweight pattern seems to defy concise formalization. An invariant element describes some property that is always true in a correct implementation of a pattern. Invariant elements are further sub-divided into cardinality, (inter-object) dependency, control-flow, data-structure and object-state invariants.

The set of invariant elements forms the first part of a classification framework for design pattern specification and verification approaches that also addresses verification issues such as the conformance relation supported (i.e. what is the required relation between the specification and a satisfying implementation), how instances are classified as either conforming or non-conforming, and the program analysis type. The conformance relation is concerned especially with what behaviour can be inter-leaved with pattern behaviour in a correct pattern implementation. Program analysis is split into three types: structure-only, dynamic and static. Structure-only ignores behavioural properties and analyses static structure alone. Dynamic analysis involves the program being executed with a number of different inputs. Static analysis builds an abstract model of the potential control and data-flow in a program at compile-time. Finally, program analysis techniques that can address the poorly-supported invariant categories and types are discussed within the context

of related work classification.

The remainder of the chapter is organized as follows: Section 2.2 outlines the scope of the review, including a list of languages and tools classified. Section 2.3 discusses some high-level issues regarding design pattern specification. Sections 2.4 and 2.5 present the language and tool sections of the classification respectively.

2.2 Scope and related work

We begin this section by briefly introducing some terminology used throughout this review and classification. The reviewed languages are referred to as *design pattern specification languages* (DPSLs). A pattern description in one of these languages is referred to as a *pattern specification*. Patterns are discussed at three levels: the pattern, its variants and its implementations. Pattern and pattern variant names are capitalized throughout the text. A pattern (e.g. Proxy) defines a collection of incomplete (or generic) classes and objects with associated invariants that allow many possible *variants* or *realizations*. A variant of a pattern fills gaps left at the generic level, by choosing between trade-offs and optional features in the generic pattern, as well as by potentially adding its own invariants. The GoF book presents multiple variants of patterns to be used in different situations. The Observer pattern, for example, can be realized as a Push-based or Pull-based variant, depending on whether the Subject object provides the state that was updated to the Observer pro-actively or reactively after an update notification. Hofer describes three variants of the Visitor pattern: a Visitor-Controlled, Structure-Controlled and Direct Visitor variant, each with different flows of control between the Visitor and an object structure. Pattern variants are independent of source code, and may be described informally in plain text or formally, using a specification language. A pattern *implementation* or *instance* is the pattern expressed in OOPL code.

Pattern specifications are said to dictate a number of *roles* that must be filled by conforming implementations. These roles may refer to, for example, classes, objects or methods. The entities in the implementation that fulfill these roles are referred to as *actors*. An *event* is any computational step that affects the abstract state (control flow or data) of the program, and will be referred to when discussing pattern behaviour. We define *Invariant types* as language elements that place constraints on an implementation of an object-oriented design pattern. *Invariants* or *clauses* are formed by instantiating some invariant type and

refer to one or a number of OOPL constructs. Invariants are combined using logical conjunction or disjunction, to form a pattern specification. Classification dimensions will be illustrated throughout the chapter using tree diagrams. Each of these diagrams contains mutually exclusive alternatives, i.e., only one path may be selected from the root to the leaf of the tree. For example, a tool may implement both a static and a dynamic analysis, but any particular program analysis in isolation can be classified as either static or dynamic.

DPSLs may be distinguished from other specification languages, such as Architecture Description Languages (ADLs) [Garlan and Shaw, 1994], by their level of granularity: DPSLs may describe interactions at the object level, while ADLs address software components and connectors. ADLs describe global invariants, which must be satisfied by all the components of a program: in the Pipe and Filter architectural style, all components are either Pipes or Filters. DPSLs describe local invariants, which need only to be satisfied by one or a small number of classes in a program [Eden and Kazman, 2003]. DPSLs are capable of representing some of the syntax of a programming language, but a DPSL is not a programming language. The objects, attributes and methods specified in DPSLs are placeholders or role specifiers for actual programming language entities. For example, in the Observer pattern, the method role `attach(Observer)` may be filled by any method that accepts an Observer object as a parameter and adds it to a list of objects that subsequently receives notifications of state updates. The actor that fills this role could have any name (e.g., `attachObserver` or `addSubscriber`). A design-pattern specification omits the details that are not relevant to the pattern, for example, it may specify that a method signature must have a particular return type and parameter list, but omit the entire method body. A design-pattern specification abstractly describes a set of possible implementations, that each conform to the design pattern. To be considered a DPSL by our classification framework, an approach must address the higher level of abstraction of design patterns relative to concrete classes and methods by providing some means of binding or linking entities at these two levels of abstraction. Binding roles to actors is dealt with in more detail in Section 2.5.7

There are a large number of approaches that support one or a number of the following invariant types (defined in detail in later sections): structural cardinality, interface dependency, sequence and method calls. We include in our classification of related work only languages and tools that support some of the other invariant types (they may also support some of the types listed above), to limit the scope and length of the review. Approaches

excluded by these criteria will still be discussed, where they provide a useful example of some language or tool feature. As numerous interesting pattern verification approaches do not have a corresponding DPSL or are excluded by our first criterion, verification tools that support the most powerful conformance relation are also included. Approaches which meet the first criterion are DisCo [Mikkonen, 1998], OC/VDM++ [Lano et al., 1996], BPSL [Taibi and Ngo, 2003], Lauder and Kent [1998], Le Guennec et al. [2000], LePUS [Eden, 2008], RBML [France et al., 2004], DPML [Mapelsden et al., 2002], FUJABA [Wendehals and Orso, 2006], Dong et al. [2007], GEBNF [Bayley and Zhu, 2010], DVP [Knudsen et al., 2007], Hofer [2009], Shetty and Menezes [2011], Ammour et al. [2005] and Contracts [Helm et al., 1990]. Verification tools that meet the second criterion are D³ [Stencel and Wegrzynowicz, 2008], SanD [Heuzeroth et al., 2003], PINOT [Shi, 2007a], Hedgehog [Blewitt et al., 2005], Columbus [Ferenc et al., 2005], PEC [Lovatt et al., 2005], Peng et al. [2008], DeMIMA [Guéhéneuc and Antoniol, 2008] and MoDeC [Ng et al., 2010]. Some approaches are relevant to, and are discussed in, both the specification and verification sections, while some appear in only one section. Also, some approaches are based upon existing formal languages that are more expressive than the subset used to define a DPSL by the reviewed approach. These formal languages are not considered in their full generality, but evaluated according to their application to the specific problem of design-pattern specification.

Two approaches included in the DPVT section only (PINOT [Shi, 2007a] and Hedgehog [Blewitt et al., 2005]) have an associated specification language. Both specification languages provide sophisticated composite invariants as single predicates: for example, lazy instantiation. Such predicates require that behaviour such as selection and properties such as aliasing must be verified, but do not provide the means to specify these invariants in isolation. According to our classification framework, both tools are more expressive than their associated specification language, and the languages have been excluded for the sake of brevity. Another classification approach would be to give each language and their associated tools equivalent expressiveness: this could have been done without altering any of the major findings of the review.

Some approaches included in the classification are actually the combination of a number of publications that build on each other. This explains why these approaches occasionally have two or more mutually-exclusive classifications. This is especially true of FUJABA, which is the combined work of Wendehals [2004] and later von Detten [2011].

2.2.1 Other DPSL and DPVT reviews in the literature

Dong et al. [2009] provide a classification framework for DPVTs only, with a focus on a reverse engineering use case. We include some of the classification dimensions of their framework in the framework provided in this chapter. A number of DPVTs included in their review are also included in our review (though neither review is a super-set of the other, as we exclude numerous tools they include by applying our inclusion criteria). Novel features of our classification framework and review relative to Dong et al. are the inclusion of DPSLs and a more detailed classification dimension regarding the expressiveness of approaches (namely, our invariant categories and types). We also include more recent work that has emerged since the publication of Dong et al.

Rasool et al. [2011] describe a direct performance comparison between six DPVTs, focusing on the number of pattern instances found by each tool in a number of commonly-analyzed bodies of code such as JHotDraw and JUnit. We do not directly compare the analysis results of different DPVTs, as different DPVTs are based upon different pattern specifications (more detail is provided on this issue in Chapter 5). We review DPSL/DPVT evaluation methodologies in Chapters 5 and 6, Section 5.1. Baroni et al. [2003] review six DPSLs without an extensive classification framework. As it was written in 2003 and addresses a field of research that is still active, it requires updating. Nonetheless, it is a good source of information on the earlier approaches in this field. Shi [2007a] provides a brief critique of 13 verification tools including a discussion of the GoF patterns and OOPLs they support. Taibi's (ed.) book [2008] was useful to this review, as it collects more up-to-date information on many of the foremost approaches in the area, though it makes no attempt at a comparison or classification of language or tools. Eden's website [2012] also contains links to many useful resources in the area of design-pattern specification and verification.

2.3 Abstraction Levels in Design Pattern Specification

Approaches to design-pattern specification may be separated into two categories based on the genericity of the specifications that they support. One category advocates the use of one generic specification to describe all possible variants of a pattern, while the other advocates the precise specification of each pattern variant (see Figure 2.2). The advantages and disadvantages of each approach are discussed in Section 2.3.1 and 2.3.2 respectively.

2.3.1 Generic pattern specification

One key decision when developing a language for describing design patterns is the level of abstraction with which to represent the pattern. Many of the reviewed approaches seek to capture a pattern in its most generic form, which has been called the patterns ‘essence’ or ‘leitmotif’ [Eden, 2001][Mak et al., 2004]. A leitmotif includes only invariants that are common to all valid implementations of a particular pattern. Approaches that favour generic specifications tend to specify structure only, and ignore behaviour completely. While the distinguishing features of pattern variants are more likely to be seen clearly at the level of behaviour, numerous pattern variants also differ in their structure. Thus, specifying structure only is not a complete solution to the problem of specifying pattern variants. The approaches that refer to a pattern ‘leitmotif’ [Eden, 2001][Mak et al., 2004][Le Guennec et al., 2000] only specify pattern structure using, for example, UML Class diagrams. The problem with specifying only the static structure is that it completely overlooks the behaviour required to satisfy the pattern’s intent. The intent of the Singleton pattern, for example, is to ensure that only one instance of a particular class is ever created. This dictates that any method that returns an instance of the Singleton always returns the same object, which can only be verified by analyzing the method’s behaviour.

Structural descriptions are not suitable for reverse engineering (a common use case for DPSLs [Shi, 2007a] [Le Guennec et al., 2000][Heuzeroth et al., 2003]). Using only a structural description, it is difficult to distinguish structurally-similar patterns such as the State and Strategy patterns. Even when behavioural invariants are included at the generic pattern level, the specification may still be too vague to be very useful during verification. In our analysis of the GoF State pattern, for example, only two invariants (an inheritance relationship and a method call) were found to be common to all variants of the State pattern, meaning that many class definitions in a body of code may conform to this specification that were not intended instances of the pattern. The GoF catalogue defines two major variants of the State pattern, one where objects of State subclasses instantiate each other, and one where a Context object that holds a reference to a State instantiates State subclasses. Once a variant is formed, by choosing which role instantiates State subclasses, the specification captures much more of the original designer’s intent and is thus much better at uncovering implementation errors.

2.3.2 Variant-specific pattern specification

In many cases, it is useful to have a variant-specific description of a pattern, with a detailed behavioural specification that can be compared to the source code to verify that the designer's intent is met by an implementation. This can help to avoid mistakes during initial development, as well as architectural drift during maintenance. For example, a Virtual Image Proxy should only create the image object after the `draw` method has been called by the document editor. This invariant is what is important to the developers of the document editor. Both Hedgehog and D³ verify multiple pattern variants (D³ verifies the Eager Initialization, Lazy Initialization, Delegated Construction and Limiton variants of the Singleton pattern) to reduce the occurrence of false positives described above, but much of the analysis is hard-coded. This leads to the possibility of false negatives, when variants not thought of by the tool developer occur in the analyzed code. Anticipating all possible variants of a pattern when developing a verification tool is difficult. For this reason, allowing the user to specify their own variants is preferable to hard-coding. However, user-created specifications may be more computationally expensive to verify. For the reasons discussed above, we believe that both generic and variant-specific pattern specifications can be useful, depending on their intended use.

Pattern variants can be specified all together as a single pattern specification or can be specified separately in multiple pattern variant specifications. An advantage of having different specifications for each variant is that it limits the visual and logical complexity of a specification, by avoiding the need to combine the variant-specific clauses using disjunction. When variants of the same pattern are specified separately, however, there is no distinction between a pattern and a pattern variant: the specification of both involves creating a new and independent set of invariants. Combining all variants in a single specification makes the points of variability between variants more explicit and maintains a close relationship between all the variants of the same pattern. Having a single specification may also speed up verification, where some complicated behaviour can be verified once for all the variants that share the behaviour.

To make pattern variants explicit in specifications, it is necessary to have some facility for naming variants and associating a name with each alternative at a point of variability. This is illustrated in Figure 2.1 taken from Bayley and Zhu [2010], who describe two variants: `Single factory method` and `Multiple factory methods`, applying some clauses to both variants and having other clauses differ for each variant. The `IN CASE OF` phrase identifies

STATIC CONDITIONS

1. **DEPENDS ON ALTERNATIVES OF Components Declaration 2:**
 - (a) **IN CASE OF Single factory method, ALTERNATIVES:**
 - i. Stronger condition: $factoryMethod.isAbstract$
 - ii. Weaker condition: $\neg factoryMethod.isLeaf$
 - (b) **IN CASE OF Multiple factory methods, ALTERNATIVES:**
 - i. A: $\forall fm \in factoryMethods \cdot (fm.isAbstract)$
 - ii. B: $\forall fm \in factoryMethods \cdot (\neg fm.isLeaf)$
 - iii. C: $\forall fm \in factoryMethods \cdot (\neg fm.isLeaf \vee fm.isAbstract)$
2. for each creator subclass there is one product subclass
$$\forall C \in subs(Creator) \cdot \exists! P \in subs(Product)$$
3. furthermore, denoting witness P by $f(C)$, then f is a total bijection.

Fig. 2.1: Bayley and Zhu’s [2010] specification of the Abstract Factory pattern, including two variants: `Single factory method` and `Multiple factory methods`

each alternative at a variation point clearly, and prefixes the variant name associated with the alternative.

We classify approaches according to whether they allow for all variants to be included in a *combined specification* or whether they require a *separate specification* for each pattern variant (See Figure 2.2). We also classify approaches based on whether they are capable of associating the *explicit name* of a variant with an alternative at a variation point, or whether variants are *anonymous*. GEBNF [Bayley and Zhu, 2010] is in fact the only approach included in this classification that is capable of providing a combined specification for all variants and associating a variant name with an alternative at a variation point, though it only supports structural variation between variants. Hofer [2009] specifies three variants of the Visitor pattern separately using an extension to Spec#. D³ verifies multiple variants of the Singleton pattern, as stated above. Overall, the specification and verification of design pattern variants is poorly supported in the literature, in particular, the specification of variants that differ in terms of behaviour.

2.4 Specification of Design Patterns

In order to precisely specify object-oriented design patterns, a language is required that can represent all the object-oriented programming language constructs referred to by the pattern as well as constraints on object interconnection and interaction. The parts that can be combined to create a design-pattern specification are referred to as *elements*. The

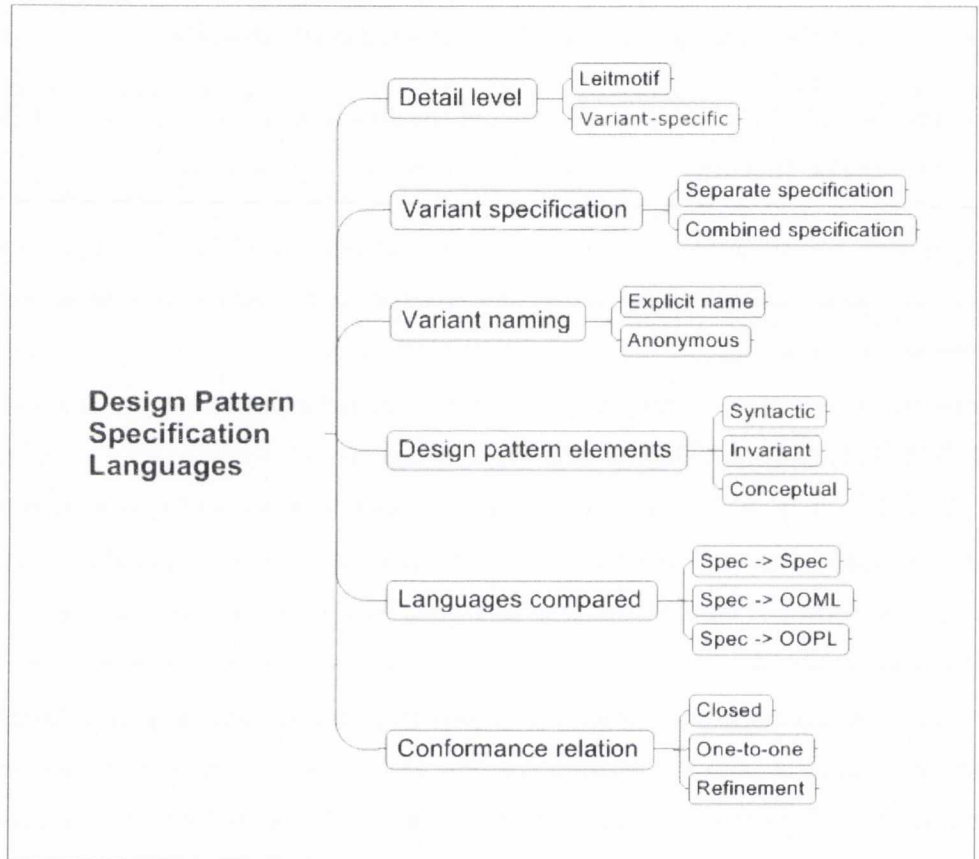


Fig. 2.2: DPSL classification framework dimensions

elements of design-pattern specifications that refer to programming language constructs, e.g., inheritance, method signatures, and lists, are termed *syntactic* elements. Constraints on the connection and interaction of structural and behavioural entities, as well as the allowable flows of control within a program, are termed *invariant* elements. Finally, pattern elements that have a specific meaning for developers, but are difficult or impossible to automatically identify in OOPL source code are termed *conceptual* elements (see Figure 2.2). Each of these three element categories are dealt with in turn below. The elements identified by this classification were extracted from the GoF book [Gamma et al., 1995], as this is the catalogue supported by all the reviewed specification languages. We generalize these elements in some cases, so they may be applied to other object-oriented design patterns. The section ends with a brief discussion of specification language syntax. This section forms the specification half of our classification framework.

2.4.1 Syntactic elements

Design patterns can represent any reusable solution to a commonly occurring design problem. As such, they could be expected to contain any language construct of the class of programming language within which they are applicable. As the GoF catalogue [Gamma et al., 1995] focused on demonstrating how to use object-orientation to its full potential, however, it focuses on a small subset of constructs (related to class relationships, object composition and interaction) and how they can be used to create reusable and extensible software designs. Figure 2.3 lists the syntactic elements that we discovered in the 23 patterns catalogued in the GoF book. Syntactic elements that occur less frequently in the GoF catalogue have a list of patterns where they occur in square brackets after their name. Syntactic elements are divided into two categories: *structural* syntactic elements and *behavioural* syntactic elements. Structural syntactic elements are visible at the interface level (or form part of the structural relationship between classes, e.g., references and inheritance), and are illustrated in the Structure section of each design-pattern description in the GoF catalogue. Behavioural syntactic elements are visible only at the implementation level, and are sometimes shown in informal notes in the Structure section and described in more detail, for example, in the Sample Code section of the GoF catalogue. While the GoF catalogue used C++ and Smalltalk as their implementation languages, it can be seen that the syntactic elements listed are also to be found in currently-popular object-oriented languages such as Java and C#.

- **Structural syntactic elements**
 - 1. Class
 - 2. Inheritance
 - 3. Method signature
 - 4. References (including reference to self) [Singleton, Visitor, Prototype]
 - 5. Access modifiers
 - 6. Attributes
 - 7. Abstract
- **Behavioural syntactic elements**
 - 8. Object
 - 9. Object creation
 - 10. Conditional branches (including boolean) [Singleton, Flyweight]
 - 11. Loops [Observer, Iterator]
 - 12. Collections (adding to, removing from and iteration) [Observer, Composite, Interpreter, Iterator]
 - 13. Method invocation
 - 14. Assignment (especially assignment of an instance of a concrete subclass to an abstract superclass variable) [Bridge, State, Strategy]

Fig. 2.3: Syntactic elements of GoF design patterns

The only element that requires more explanation is the final one. The intent of numerous patterns is to allow clients not to prematurely commit to a particular implementation. This manifests itself in code as an assignment of an instance of a concrete subclass to an abstract superclass. It is also worth noting the common syntactic elements or concepts of OOPs *not* within scope of GoF design patterns. These include: arithmetic operations, exception handling, package import, casting and concurrency. Operators overlooked include algebraic operators and bitwise operators. The omission of these concepts simplifies the verification problem, though some features may need to be handled to provide sound verification nonetheless. Multiple inheritance is only required for the Class Adapter variation of the Adapter pattern, and, as it is not supported in more recent popular object-oriented languages, such as Java and C#, it has been omitted from the list. To our knowledge, it is only supported by DisCo [Mikkonen, 1998]

Structural syntactic elements (elements 1-7) receive more widespread support than behavioural syntactic elements (See Table 2.1). Only OC/VDM++ [Lano et al., 1996], RBML [France et al., 2004] and FUJABA [Wendehals and Orso, 2006] provide support for the majority of syntactic pattern elements. Conditional branches, loops and assignment statements are the most poorly supported elements. Approaches that are based upon temporal logic (e.g. DisCo [Mikkonen, 1998]) perform poorly in this dimension of the classification. The reason for this is that temporal logic-based approaches only describe relations between a system at different times in terms of logic, and not in terms of programming language constructs. An additional mapping stage is required in these approaches to connect temporal logic operators and operands, and the programming language code constructs that realizes them. DPSLs have been classified here according to only the syntactic elements described in the literature, which may not include all the syntax supported by the languages.

2.4.2 Invariant elements

A pattern implies one or a number of invariants, each of which is an instance of a particular invariant element or type. A pattern invariant is defined as something that is always true in a correct implementation of a pattern. Key invariants can often be identified from a description of the pattern's intent, for example, 'separate the construction of a complex object from its representation' or 'Ensure a class only has one instance' [Gamma et al., 1995]. Other important invariants become clear from a careful reading of the remaining sections of a pattern description in a catalogue. The GoF book also discusses pattern realization

DPSL	Class	Inheritance	Method signature	References	Access modifiers	Attributes	Abstract	Object	Object creation	Conditional branches	Loops	Collections	Method calls	Assignment
DisCo	✓	✓	✓	✓	×	✓	×	✓	×	×	×	×	✓	×
OC/VDM++	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
BPSL	✓	✓	✓	✓	✓	✓	✓	✓	✓	×	✓	✓	✓	×
Lauder & Kent	✓	✓	×	✓	✓	✓	✓	×	✓	×	×	×	✓	×
Le Guennec et al.	✓	✓	×	✓	✓	✓	×	×	×	×	×	✓	×	×
LePUS3	✓	✓	×	✓	×	×	✓	×	✓	×	×	×	✓	×
RBML	✓	✓	×	✓	×	✓	✓	✓	✓	✓	✓	✓	✓	×
DPML	✓	✓	✓	✓	✓	✓	✓	×	×	×	×	×	×	×
FUJABA	✓	✓	✓	✓	✓	✓	✓	✓	×	✓	✓	×	✓	✓
Dong et al.	✓	×	×	×	×	✓	×	×	×	×	×	×	×	×
GEBNF	✓	✓	✓	✓	✓	✓	✓	✓	✓	×	×	×	✓	×
DVP	✓	✓	✓	✓	×	✓	×	✓	×	×	✓	×	✓	×
Hofer	✓	✓	✓	✓	✓	✓	✓	×	×	×	×	×	✓	×
Shetty & Menezes	✓	✓	✓	✓	✓	✓	✓	×	×	×	×	×	✓	×
Ammour et al.	✓	✓	✓	✓	✓	✓	✓	×	×	×	×	×	×	×
Contracts	✓	✓	✓	✓	✓	✓	×	✓	×	×	×	✓	✓	×

Table 2.1: Classification of the support of each DPSL for each of the syntactic elements

alternatives, where each realization may offer a different trade-off between non-functional properties. In a pattern realization, a choice must be made between each alternative, and regarding each open issue, and this choice, when made, should be captured in an invariant, if it is important to the correct functioning of the system. For example, in the context of the State pattern, it is necessary to decide whether the Context or the State subclasses define the state transitions, and the existence of this necessity means there are at least two significant variants of the State pattern, each with different non-functional properties. This review includes all the key invariant types, along with those regarding realization alternatives, identified by the authors. Each of the invariants listed contributes to providing a precise formal description of a design pattern, with the potential to aid program understanding as well as automated verification.

This section separates the 13 identified invariant types into five categories and describes each category in turn while providing examples. A description of the each invariant type, along with concrete examples, are given below in Figures 2.4, 2.5 and 2.6. Invariant categories may depend upon each other. In particular, cardinality invariants often refer to dependencies between structural entities and both object-state and data-structure invariants can be applied to different stages of the control-flow. Structural entities in the following discussion are either classes or methods. Behavioural entities are typically objects, but may also be control-flow events. Invariants are also classified according to their dynamism. An invariant is *behavioural* if its truth value depends upon the state of the computation at a particular point. *Structural* invariants can be verified independently of the computational state. Some invariants have been taken directly from the literature, while others have been generalized. To the knowledge of the authors, invariants 2, 11, 12 and 13 have not previously been identified (in the generality given here) in the context of design-pattern specification and verification.

2.4.2.1 Dependency invariants

Dependency invariants are defined as invariants that place some constraint on the level of coupling allowed between classes. As one of the main focuses of the GoF catalogue is on reducing the coupling between classes and objects, dependency invariants are key to capturing a pattern's intent. The key invariant of many of the GoF patterns can be expressed informally as 'class A shouldn't need to have knowledge of class B'. More specifically, in the Façade and Mediator patterns, it is important that instances of some class or set of

- **Dependency invariants:**
 - **1. Interface dependency:** A structural entity depends, or does not depend, on the interface of another structural entity.
 Positive example: (Proxy) A Proxy has a reference to a RealSubject.
 Negative example: (Mediator) No Colleague has a direct reference to any other Colleague.
 - **2. Implementation dependency:** A structural entity commits, or does not commit, to a particular implementation of a class.
 Negative example 1: (Abstract Factory) A Client does not initialize a ConcreteProduct directly, instead it calls a Factory Method.
 Negative example 2: (Command) An Invoker does not initialize a ConcreteCommand.

- **Cardinality invariants:**
 - **3. Universality and Existence:** There is a relationship between separate sets of structural or behavioural entities.
 Structural example: (Abstract Factory) There must be one Factory Method in each ConcreteFactory for each AbstractProduct class.
 - **4. Uniqueness:** An element of a set of structural or behavioural entities performs a unique role in that set.
 Behavioural example: (Visitor) Each accept method in each ConcreteElement must call a unique member of the set of visit methods in a ConcreteVisitor class.

Fig. 2.4: Invariant categories and types: Part 1: Dependency and cardinality invariants

- **Control-flow invariants:**

- **5. Sequence:** An event is always followed by another event.

Example: (Template Method) The primitiveOperation methods are called in a particular order in the templateMethod.

- **6. Selection:** An event occurs if some condition is satisfied, or a choice is made between mutually-exclusive events.

Example: (CoR) A ConcreteHandler's handleRequest method should either handle a request or forward it to a successor.

- **7. Iteration:** An event occurs repeatedly, often once for each element in some set of behavioural entities.

Example: (Observer) A Subject's notify method calls the update method on all Observer objects in its list of Observers.

- **8. Method call:** A method calls another method.

Example: (Adapter) The Adapter's request method calls the Adaptee's specificRequest method.

Fig. 2.5: Invariant categories and types: Part 2: Control-flow invariants

- **Object-state invariants:**

- **9. Basic state:** An object is in a particular basic state (not initialized, initialized, marked for deletion).

Example: (Singleton) If a reference variable that is intended to point to the single instance of the Singleton class is null when the Singleton's getInstance method is called, a new instance of the Singleton is created and assigned to the reference variable.

- **10. Aliasing:** The same object is involved in a number of events, or two reference variables are aliases of each other.

Example: (Factory Method) The ConcreteProduct object initialized in the Factory Method is the same object that is returned by the Factory Method.

- **11. Copying:** One object is a 'deep copy' of another.

Example: (Memento) A Memento object is a deep copy of the original object, so that the original can be manipulated without affecting the Memento.

- **Data-structure invariants:**

- **12. Object position:** An object (of a particular type) is or is not in a collection, or is at a particular position within a collection.

Example: (Observer) A Subject's attach(Observer) method adds the Observer object argument to its list of Observers.

- **13. Shape:** A data-structure has a particular shape, or has an object of a particular type at a particular position.

Example: (Composite) A Composite object structure is free from cycles, and each object is reachable via only one path (i.e., the structure is also free from sharing).

Example: (Decorator) A chain of Decorator objects is terminated by a ConcreteComponent object.

Fig. 2.6: Invariant categories and types: Part 3: Object-state and data-structure invariants

classes do not hold a direct reference to instances of some other set of classes. These are examples of *interface dependency invariants*, where the holding or not holding of a reference is important. A more subtle requirement that occurs in many patterns including the Abstract Factory and Bridge patterns dictates that the holder of a reference to an object (Abstraction in Bridge, Client in Abstract Factory) must not initialize the object itself but must delegate the initialization of the object to, for example, a Factory object. These are instances of *implementation dependency invariants*, where a client should not commit to a particular subclass by calling the constructor directly. This distinction allows fine-grained statements about inter-class dependency to be made. Invariants that require a dependency to exist are termed *positive* dependency invariants, while invariants that forbid particular dependencies from existing are termed *negative* dependency invariants.

While a number of approaches support interface dependency invariants, few make the distinction between interface and implementation dependency (see Table 2.2). Contracts and FUJABA support a restricted (positive) form of interface dependency invariants (Approaches supporting positive dependency invariants only have a P in parentheses in their dependency column in Table 2.2). It is possible to specify that a class references or calls another class, but not that a class should *not* reference another class. Numerous approaches, including LePUS and RBML support specification of restricted interface and implementation dependency. LePUS, for example, has a specific ‘creates’ relation, to specify that a class has the responsibility of instantiating another class. RBML can specify direct calls to constructors in its interaction diagrams. Finally, BPSL supports all dependency invariants in their full generality, with ‘Reference-to-one’ and ‘Creation’ relations and a logical negation operator.

Ammour et al. [2005] clearly has the intention of describing the absence of an implementation dependency with their `hiddenSubclasses` predicate, which states that a class does not ‘access’ the set of all subclasses of some class. However, accessing is defined in terms of any use of the name of the class within the other classes’s definition, e.g., a variable, a cast, a constructor call, and this is their only form of dependency. For this reason, they cannot distinguish between interface and implementation dependency.

2.4.2.2 Cardinality invariants

Invariants in this category place constraints on the relationship between sets of entities, or elements within a set of entities. Sets are a natural way to describe groups of entities (classes

DPSL	Dependency	Cardinality	Control flow	Object state	Data structure
DisCo	All (P)	Univ		Aliasing	-
OC/VDM++	All (P)	-	All	Aliasing	-
BPSL	All	-	Sequence and choice	Aliasing	-
Lauder & Kent	All (P)	-	Sequence and calls	-	-
Le Guennec et al.	-	All (S)		Basic state	-
LePUS3	All (P)	Univ (S)	-	-	-
RBML	All (P)	Univ (S)	Basic	-	-
DPML	All (P)	-	Sequence and calls	-	-
FUJABA	Interface (P)	-	Sequence and iteration	-	-
Dong et al.	-	-	All	Aliasing	-
GEBNF	All (P)	All	Sequence and calls	-	-
DVP	-	-	All	Aliasing	-
Hofer	-	Univ	Calls	Aliasing	-
Shetty & Menezes	Interface (P)	Univ	Calls	-	-
Ammour et al.	All (P)	-	-	-	-
Contracts	Interface (P)	-	Sequence and calls	Aliasing	Object position

Table 2.2: Classification of the support of each DPSL for each of the invariant types

or methods in this case) that share a particular characteristic, and will be used throughout this thesis. Examples of sets in pattern specifications include the set of all classes that inherit from `AbstractFactory`, the set of all classes that initialize a `ConcreteImplementor` (Bridge) and the set of Factory Methods. An example of a cardinality invariant from the Abstract Factory pattern, is that the number of Factory Methods in each `ConcreteFactory` should be equal to the number of `AbstractProduct` classes. Taking an instance from the GoF catalogue, a `MotifWidgetFactory` must be capable of creating (have one Factory Method for) all the concrete Motif widgets that inherit from an `AbstractProduct`, e.g., `MotifWindow` and `MotifScrollBar`.

Cardinality invariants received a lot of attention in the earlier literature on design pattern specification. DPSLs that are based on set semantics [Le Guennec et al., 2000][Mak et al., 2004][Eden, 2001] [Mapelsden et al., 2002], or first-order logic, [Bayley and Zhu, 2010][Shetty and Menezes, 2011] as expected, handle cardinality invariants well (Table 2.2). Figure 2.7 from Mak et al. shows their specification of the Visitor pattern. Pattern roles are modelled using UML ClassifierRoles (e.g., `/Visitor`, `/AcceptOp`), which may be filled by classes, interfaces or methods. “The number at the right upper corner of each ClassifierRole denotes the number of its instances in one pattern instance” [Mak et al., 2004]. The specification states that for every concrete `Element` class, there needs to be a `VisitElement` method that is capable of ‘visiting’ that `Element`, i.e., in any instance of the Visitor pattern, there is n `VisitElements` and n `Elements`. If there is only one `ConcreteVisitor`, there will be n `VisitConcreteElement` ClassifierRole instances. However, every new `ConcreteVisitor` that is added must be capable of visiting every concrete `Element`, i.e., it must implement every `VisitElement` operation. If the number of `ConcreteVisitors` is denoted by m , then the required number of `VisitConcreteElements` in the inheritance hierarchy must equal $m \times n$.

Most approaches that specify cardinality invariants (some of them excluded from this classification) focus on structural cardinality (constraints on sets of structural roles), but a few also specify behavioural cardinality (constraints on sets of entities performing some behaviour). Shetty and Menezes, as well as Hofer, specify that every element of some set of methods should call some element of another set of methods. Technically, both can specify uniqueness constraints, as they use first-order logic, but neither approach demonstrates this capability. GEBNF is capable of specifying both structural and behavioural instances of both cardinality invariant types. In Table 2.2, and Universality and Existence is abbreviated

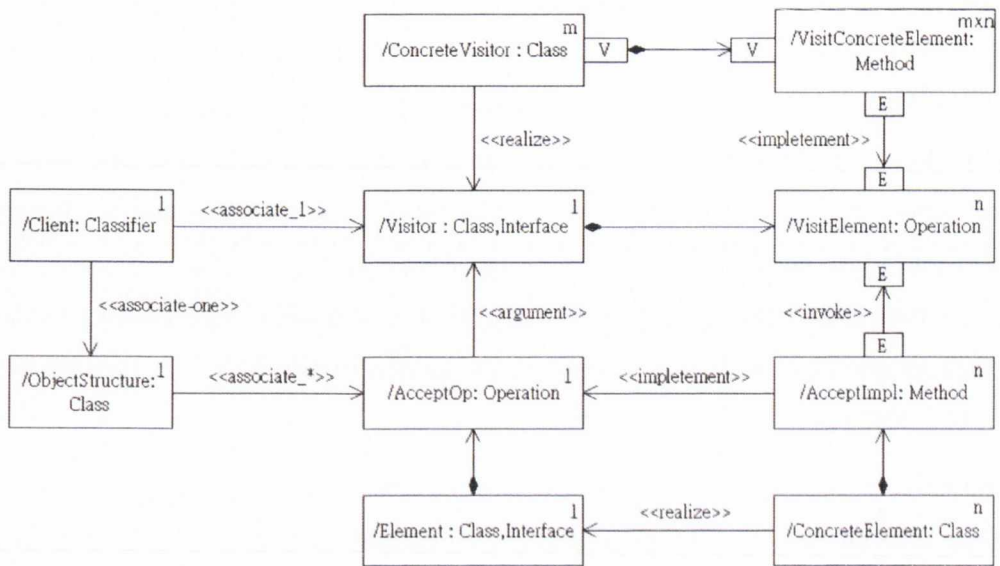


Fig. 2.7: The specification of the Visitor pattern given in Mak et al. [2004]

to *Univ.*

2.4.2.3 Control-flow invariants

Control-flow invariants are defined as invariants that place constraints on the control-flow in a pattern. The four types of flows required are sequencing, selection, iteration and method calls. In pattern specifications, a particular action, such as a *method call*, may be specified to occur before another (*sequencing*), a choice between alternative actions might be made (*selection*), or a particular action should be performed repeatedly (*iteration*). While the static elements of design patterns are represented by Class diagrams in the GoF book, the control flow (especially important in the behavioural patterns of the GoF catalogue) is represented by interaction diagrams.

The Strategy and Template Method patterns define a sequence of events within a single method, while the Observer and Visitor patterns are characterized by fixed inter-object protocols, i.e., sequences of events spanning multiple methods. In the Observer pattern, a call to the Subject's `setState` method should always be followed by a call to its `notify` method, which in turn should call the `update` method of all attached Observers. Depending on whether the pattern implementation follows the Push or Pull variant, each Observer may then call the Subject back, to acquire the necessary state information. The Visitor pattern

$$\begin{aligned} & \text{Exists}(\text{Flyweight}[i]) \wedge \text{GetFlyweight}(\text{client}, i) \rightarrow \text{Return}(\text{Flyweight}[i]) \\ & \neg \text{Exists}(\text{Flyweight}[i]) \wedge \text{GetFlyweight}(\text{client}, i) \rightarrow \text{Create}(\text{Flyweight}[i]) \end{aligned}$$

Fig. 2.8: Flyweight pattern conditional branching expressed in BPSL

realizes double-dispatch in languages that support only single-dispatch by following a call by an `ObjectStructure` to a `ConcreteElement`'s `visit` method with a call to a suitable `Visitor`'s method. Thus the method which is executed depends upon the dynamic type of two objects, `ConcreteElement` and `Visitor`.

While the sequence of invocations is linear in the examples above, the Singleton pattern demonstrates selection between alternatives in a pattern's dynamics. When a client invokes the `getInstance(Key)` method, it should either return a reference to the existing Singleton object, or if it does not exist, create it and then return a reference to it. Other examples of GoF design patterns that include selection are Proxy, Flyweight and Chain of Responsibility. The Observer and Composite pattern both include iteration when calling a method on every element of a list.

DPSLs are classified based on their support for the four control-flow invariant types (see Table 2.2). Most approaches that specify behaviour of any kind support sequence and method calls, as these are central to the intent of design patterns: many patterns insert an intermediary (Decorator, Proxy, Mediator, Façade) between a Client and its initial delegate before the pattern was applied. Both DisCo [Mikkonen, 1998] and BPSL [Taibi and Ngo, 2003] are based upon the Temporal Logic of Actions (TLA) [Lamport, 1994], and support the specification of sequencing and choice in control-flow. Figure 2.8 shows how the conditional branch in the Flyweight pattern might be expressed in BPSL. Such an expression is less readable than if/else constructs for pattern users who are familiar with object-oriented programming but not formal logic.

A number of approaches support all control-flow invariant types. OC/VDM++ [Lano et al., 1996] uses linear temporal logic operators from the Object Calculus to express sequencing. Object-oriented programming constructs are supported by using VDM++ as a pattern specification language, which includes if/then/else and for all/do constructs. DVP is based on a number of different formal languages, but derives its ability to specify control flow from Communicating Sequential Processes (CSP) [Hoare, 1985]. RBML [France et al., 2004] and FUJABA [Wendehals and Orso, 2006] use UML 2.0 Sequence Diagrams, which

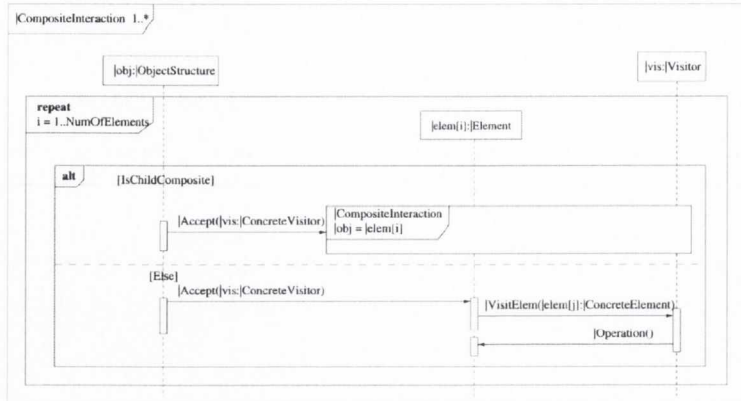


Fig. 2.9: RBML [France et al., 2004] behavioural specification of the Visitor pattern

adds syntax for expressing conditional branching and loops. A Boolean guard condition can be placed at the beginning of a sequence diagram fragment, to allow specification of the selection between alternatives or a loop condition. A repeat fragment specifies that the sequence of events inside the fragment is repeatedly executed. Figure 2.9 [France et al., 2004] shows the use of the new *Combined Fragment* syntax element in UML 2.0. It is visualized as a rectangle enclosing a subsequence of the Sequence diagram with a label in the top left corner. The outer rectangle (labelled **repeat**) specifies a loop that continues until all the elements contained in the `ObjectStructure` are visited. The inner (**alt**) rectangle performs a selection based on whether the current element is a composite structure or not. The use of the vertical bar prefix (`|`) indicates that these are role names, and may be filled by elements with different actual names.

2.4.2.4 Object state invariants

The invariants in this category specify that an object should be in a particular basic state: not initialized, initialized or marked for deletion, or that a particular relation (aliasing, deep copying) should hold between the state of two objects. Object state invariants generally also have temporal properties and for this reason often depend upon control-flow invariants. The interaction of the Singleton pattern that involves comparing an object to the value `NULL` was described above in Section 2.4.2.3. A similar interaction occurs in the context of the Flyweight pattern, except that a `FlyweightFactory` creates instances of not one but numerous different `ConcreteFlyweight` classes.

Numerous interactions between GoF design patterns assume the same object is involved

in a number of different events. A simple example of such an invariant is provided by the Factory Method pattern. The Factory Method should initialize a ConcreteProduct object within its body and return it, i.e., the object returned from the Factory Method is the same object returned from the ConcreteProduct constructor within the Factory Method. Also, the `accept()` method of each ConcreteElement class in an instance of the Visitor pattern should accept a ConcreteVisitor object as an argument and perform a callback to that same ConcreteVisitor object. The Observer pattern involves a similar interaction. These interactions are examples of *aliasing* invariants.

The intent of the Memento pattern is to ‘capture and externalize an object’s internal state so that the object can be restored to that state later’. Typically, a Memento is created from the state of some object before an operation that mutates the state of that object but should also be undoable. For this requirement to be satisfied, the Memento object itself should not be mutated by the operation, i.e., none of the Memento’s variables should be aliased with variables of the original object. This *deep copying* invariant type is discussed in the GoF catalogue in the context of the Prototype pattern [Gamma et al., 1995, p.221], where it is also relevant. Performing a deep copy of an object is challenging, especially when object structures contain cyclic references [Gamma et al., 1995]. Also, only the state that is composed by the object, i.e., that is modelled by a ‘has a’ relationship from object to composed state, should be copied. Other objects that are associated with the object, but have a separate lifetime, should not be copied (e.g., while copying an Observer, the associated Subject should definitely not be copied, as it is important that each Observer observes the *same* Subject). For this reason, to precisely specify deep copying behaviour, it is necessary for a DPSL to be capable of making a distinction between associated and composed state.

RBML, Le Guennec et al., and OC/VDM++ support the specification of basic state invariants, as they are each capable of testing for object initialization by comparing an object to the value `NULL`. The approaches that specify only a single generic ‘equality’ relation [Dong et al., 2007][Mikkonen, 1998][Taibi and Ngo, 2003][Hofer, 2009], without defining what equality means in detail, are classified as supporting the aliasing invariant type (abbreviated to *alias* in Table 2.2). These approaches do not make explicit the distinction between object identity and value equality. Contracts defines an equality as well as a less-strict `relates` operator between the state of two objects, but does not define it in detail, though the intention is likely that the state of one object is some function of the state of another,

e.g., some subset of the states are equal or an integer is converted to an equivalent string of characters. The approaches that are based upon UML are capable of distinguishing between an **association** and **aggregation** relationship, though these relationships are not precisely defined in terms of object initialization and lifetimes. The value equality of primitive variables required to perform a deep copy are not specifiable by any of the reviewed DPSLs. This is not surprising, as the GoF catalogue focuses on relations between user-defined classes and objects such as inheritance, class dependency and object composition. In summary, basic state invariants are supported by a small number of approaches, aliasing is well-supported, but deep copying is not supported by any of the reviewed DPSLs.

2.4.2.5 Data-structure invariants

Data-structure invariants place constraints on the objects in, and the position of objects within a collection. They also constrain the contents and shape of user-defined recursive data structures. Both the Observer and Flyweight patterns involve an object (of class Subject and FlyweightFactory respectively) that holds a collection of objects of some other class (Observer and ConcreteFlyweight respectively). Both patterns involve methods that insert objects into, or otherwise mutate the contents of the collection. In the context of the Iterator pattern, a ConcreteIterator has a reference to an aggregate object, and should be capable of performing operations such as accessing the first object in that aggregate. These are examples of *object position* invariants.

Data structure invariants also place constraints on the *shape* of data structures at runtime, such as whether the structure contains cycles, or whether a particular object is reachable (transitively) via a particular reference. A number of GoF design patterns either describe the use of recursive data structures or are often applied to them. A desirable property of the Chain of Responsibility pattern is that every request eventually gets handled by some Handler. This is often ensured by providing a root Handler that is placed at the end of every chain of Handlers, which can provide some default response. The Composite pattern ‘compose[s] objects into tree structures to represent part-whole hierarchies’. To ensure correct traversal behaviour, the Composite object structure should be free from cycles and sharing, so that the traversal terminates and visits each object only once.

Some approaches define ‘attach’ events in the context of the Observer pattern [Taibi and Ngo, 2003], but these events have little or no associated semantics. Only Contracts defines an ‘attach’ event in terms of an object being inserted into a collection (or, in their

case, a logical set). Dong et al. provide a `First()` and `Next()` function that may be used to specify iteration over a list, but do not describe any syntax for describing the data structures themselves.

With regard to shape invariants, both Krishnaswami et al. [2009] and DiStefano [2008] specify a number of design patterns using Separation Logic [Reynolds, 2002], which allows the disjointness of portions of the heap to be specified. Separation logic also allows shape invariants such as cycle-freeness to be guaranteed through an ownership model (though neither approach demonstrates this), where an owner object encapsulates all of its state. However, Leavens et al. [2007] identify invariants of the Composite pattern that cannot be verified using an ownership model, showing the model is not applicable to data-structure invariants in general. Neither Krishnaswami et al. nor DiStefano et al. provide a DPSL, and their specifications must be verified manually using a theorem prover. In summary, object position invariants have been given limited attention in the context of design pattern specification, while shape invariants have been almost completely ignored.

2.4.3 Invariant dynamism

It should be clear from the previous section that some invariants depend upon the state of the computation, while others do not. The former category may be classified as *behavioural* invariants, and are defined as those invariants whose truth value depends upon the state of the computation at a specific point in the execution. From a verification perspective, these invariants require control- and data-flow information to verify that they are satisfied. Those invariants that are computation state-independent are classified as *structural* invariants. From a verification perspective, these invariants can be checked more easily by inspection of the source code.

Dependency invariants are structural, as they depend upon the properties of single declarations or expressions in isolation. Cardinality invariants, as discussed, may be both structural and behavioural. Control-flow, data structure and object state invariants truth-value clearly differs depending on the values that flow into and out of different points in the program execution. They are all thus classified as behavioural invariants.

2.4.4 Conceptual elements

Conceptual elements are defined as higher-level elements of design patterns that are difficult or impossible to verify by automated analysis of source code. This difficulty is not due to

the complexity of the analysis required, but to the fact that the pattern element is an abstract concept that leaves no consistent *signature* on the application code. Both the Interpreter and Strategy pattern rely on some abstract concept to describe their intent: the class hierarchy in the Interpreter pattern is intended to implement a *language*, while each of the concrete Strategy subclasses implements a *related algorithm*. The difference between some patterns is not always obvious when syntax alone is considered, or even after control- and data-flow analysis. For example, the Composite and Interpreter pattern are both, syntactically, an operation applied to every member of an aggregated object hierarchy.

A common difference between the DPSLs reviewed is the set of concepts that they consider to be conceptual or unverifiable elements. For example, Hedgehog cannot distinguish between State and Strategy and rules out Command for being too vague. PINOT considers the Builder and Memento patterns as ‘Generic concepts’ that ‘lack definite structural and behavioural aspects for pattern detection’, while the Interpreter and Command patterns are classified as ‘Domain-specific patterns’, that require domain-specific knowledge for verification [Shi, 2007a]. We have attempted to identify, in the previous section, a number of additional invariants to those currently occurring in the literature to make it possible for more patterns to be distinguished during verification. However, it is not practical to provide a language expression for some more sophisticated concepts that have no obvious signature in the source code. A list of these conceptual elements of patterns is given in Figure 2.10. Again, the pattern that contains the conceptual element is given in brackets after the element name.

The authors of the GoF book themselves observed this problem with such generic description: “Considered in its most general form (i.e., an operation distributed over a class hierarchy based on the Composite pattern), nearly every use of the Composite pattern will also contain the Interpreter pattern.” However, an implementation should only be considered to implement the Interpreter pattern in “those cases in which you want to think of the class hierarchy as defining a language” [Gamma et al., 1995] - a hard-to-formalize distinction. This makes reverse engineering of patterns from source code difficult and prone to error, where these conceptual elements are involved.

2.4.5 Summary

In this section, we presented a novel classification framework for DPSLs that includes a set of invariant types that are necessary for the precise specification of design patterns. De-

- *Intrinsic* and *extrinsic* state (Flyweight)
- Object represents a *request* (Command, CoR)
- Class hierarchy represents a *language* (Interpreter)
- Each member of a class hierarchy implements a *related algorithm* (Strategy)

Fig. 2.10: Conceptual elements of GoF patterns

dependency invariants place constraints on the level of coupling between classes. Cardinality invariants describe relationships between different sets of structural or behavioural entities. Control-flow invariants constrain the allowable flows of control through an implementation. Object-state invariants describe properties that must hold on one or a number of related objects, while data-structure invariants constrain the shape or contents of data-structures at runtime.

OC/VDM++ supports all syntactic elements, while RBML, BPSL and FUJABA each support a majority of syntactic elements. Dependency invariants are supported in full by BPSL, which is the only approach addressing implementation dependency in its full generality. Structural cardinality invariant elements are addressed by a number of approaches, with fewer addressing behavioral cardinality. All control-flow invariants are supported by OC/VDM++ and RBML. Object state invariants related to basic states and aliases are supported by some approaches, but deep copying is not supported fully by any approach. Data-structure invariants are possibly the most poorly-supported category, with object position not supported in its full generality, and shape invariants have been largely ignored. Finally, a number of DPSLs provide a concise graphical syntax for describing structure, with fewer also providing syntax for behavioural invariant specification. RBML provides perhaps the most expressive and one of the most intuitive graphical syntaxes overall.

2.5 Verification of Design Pattern Implementations

Verifying that an implementation conforms to a pattern specification involves two main steps: binding elements of the implementation to elements of the specification to which they correspond and verifying that the elements of the implementation are capable of performing the actions outlined by the specification elements to which they correspond. The issues

involved when a verification tool evaluates whether an implementation conforms to the roles outlined in the specification are discussed in this section. We classify DPVTs according to their support for the invariant categories described in Section 2.4.2, as well as introducing some tool-specific classification dimensions. The distinction between language and tool and tool-specific classification dimensions is not strict, as, for example, the conformance relation of an approach is dependent on the semantics of the language and also the features and soundness of the tool. Likewise with the mapping from a specification to an implementation language: this could be defined precisely by a DPSL, or a DPSL could lack a semantic mapping, which would then have to be performed by the associated tool. This section forms the second half of the classification framework.

Unlike DPSLs, we do not classify DPVTs according to the OOP syntax they support. How a DPVT addresses, for example, method calls, involves many design decisions such as how to represent the local stack and global heap, polymorphism and calling context. This makes a simple yes/no classification of support for OOP syntax quite meaningless.

2.5.1 Invariant elements

Similarly to DPSLs, we classify DPVTs according to the invariant types they support. Where DPSLs have an associated DPVT, we classify these tools also: a tool may verify only a subset of the invariants expressible by the language, or it may verify some invariants in an unsound and incomplete manner. DPVTs associated with a DPSL included in this review will be referred to mostly using the language's name, to avoid confusion.

2.5.1.1 Dependency invariants

The verification of dependency invariants involves searching for associations between classes that are manifest as either reference variables or expressions such as methods calls. Comparing the type of all the variables of a class to some pattern role is straightforward and is performed by a number of DPVTs [Ammour et al., 2005][Dong et al., 2007][Taibi and Ngo, 2003][Maplesden et al., 2007] (see Table 2.3). Making a full exploration of the abstract syntax tree of a method or all methods in a class searching for a particular kind of expression is more challenging, but likewise well documented in the literature [Lovatt et al., 2005][Guéhéneuc and Antoniol, 2008]. DeMIMA in particular includes a rich set of dependency relations and classifies relations into categories such as association, aggregation and composition using dynamic analysis to determine, for example, the lifetime of different

objects and whether associations between objects are exclusive or not. As stated above in Section 2.4.2.1, however, each of the classified approaches (apart from BPSL) only allows the specification of the existence of certain dependencies and not their non-existence. BPSL’s [Taibi et al., 2009] verification tool compares TLA+-based specifications to other TLA+-based specifications only. Interface dependency invariant verification is well-supported and evaluated in the literature, though this cannot be said for implementation dependency. In particular, there is a lack of tools that can verify implementation dependency invariants against implementations in OOPLs and a lack of evaluations on well-documented benchmarks.

2.5.1.2 Cardinality invariants

The verification of cardinality invariants simply involves counting the number of entities satisfying a dependency, control-flow or other type of invariant and is thus not particularly challenging. MaramaDPTool is implemented as an eclipse plugin and is capable of verifying the cardinality invariants specifiable by DPML [Maplesden et al., 2007] (universality cardinality invariant, structure only). Similarly, Shetty and Menezes [2011] describes the implementation of a tool to support the verification of the same invariant type, though its operation is not documented. LePUS3 has an associated tool capable of verifying Java code against LePUS3 specifications involving both universality and uniqueness invariants (structure only). LAMBDES-DP supports the verification of both the structural and behavioural invariants specifiable in GEBNF through the use of an automatic theorem prover that does not guarantee termination in general. Zhu et al. [2009] describe the use of LAMBDES-DP to compare GEBNF specifications to their own benchmark of UML models. In summary, the verification of cardinality invariants is not particularly challenging in isolation (cardinality invariants may contain complex sub-clauses involving object-state etc.) and has been demonstrated by a number of approaches in the literature. However, these approaches have tended to analyze benchmarks that have a small number of pattern instances, were developed by the authors, or both.

2.5.1.3 Control-flow invariants

Control flow invariants constrain the valid paths that a program may take at runtime. Verifying whether implementations conform to control-flow specifications involves relating each potential path through the implementation to a path in the specification. As the

DPSL	Dependency	Cardinality	Control flow	Object state	Data structure
D ³	-	-	Sequence, selection and calls	Aliasing	-
SanD	Interface (P)	-	Sequence and calls	-	Object position
PINOT	All (P)	-	All	Aliasing	-
Hedgehog	All (P)	-	All	Basic state and aliasing	-
Columbus	All (P)	-	Sequence, iteration and calls	-	-
PEC	Interface (P)	-	-	Aliasing	-
Peng et al.	-	-	Sequence and calls	-	-
DeMIMA	All (P)	-	-	-	-
MoDeC	Interface (P)	-	Sequence, iteration and calls	-	-

Table 2.3: Classification of the support of each DPVT for each of the invariant types

implementation may perform additional behaviour beyond what the specification requires, multiple paths through the implementation may satisfy a single path in the specification. In a typical OOPL, there are numerous ways to implement selection and iteration. In Java, for example, any statement placed inside an `if`, `if...else`, or `switch` block is conditional, but so is any statement that may be bypassed by a jump statement (`break`, `return` etc.). Iterative behaviour may be implemented in Java using loop statements or recursive calls. Such flexibility is challenging for a verification tool. It should also be noted that precise verification of control-flow invariants requires knowledge of the valid data flows through a program: a control-flow invariant may require not just that a sequence of calls are made through the same reference variable, but that they dispatch to the same object.

MoDeC [Ng et al., 2010] uses bytecode instrumentation to verify control-flow invariants regarding sequence, iteration and method calls. It handles loops by ‘[identifying] every branch instruction whose target is indexed before its own position’. DeMIMA [Guéhéneuc and Antoniol, 2008] performs Prolog queries on program traces, but it ignores the coverage problem by assuming that a set of unit tests is pre-generated for it. SanD [Heuzeroth et al., 2003] can verify that the same object received a sequence of method calls by tracking object identity through its dynamic analysis.

Hedgehog [Blewitt et al., 2005], PINOT [Shi, 2007a] and D³ [Stencel and Wegrzynowicz, 2008] all verify all four of the control-flow invariant types as well as sophisticated combinations of control-flow invariants as atomic invariants, such as ‘lazy initialization’ (see Table 2.3). However, PINOT classifies a method as a Factory Method even when a new object is not returned over every path through the method. It also misses some method calls, for example, those that occur as arguments to other method calls. Finally, it performs ‘limited loop analysis’ which occasionally classifies an infinite loop as conforming to the specification of the Observer pattern, where the loop should iterate over the list of Observer objects attached to the subject. Hedgehog assumes loops execute once, which is not the typical approach to analyzing loops in the iterative fixed-point algorithms that perform data-flow analysis (DFA) and is unsound. Similarly, D³ performs an ‘arbitrary number of iterations’ of the data-flow analysis, rather than iterating until convergence. Calls to constructors are treated context-insensitively and the call graph is computed before the data-flow analysis algorithm commences. Both of these design decisions introduce inaccuracy, though they may both be sound (context-sensitive analysis and call graph construction will be discussed in more detail in Chapter 4).

Control-flow invariants are well supported in the literature and are addressed by some of the most sophisticated DPVTs. However, each tool has a number of features that address control-flow invariants that are either inaccurate, unsound or both.

2.5.1.4 Object-state invariants

To verify each of the object state invariants identified in our GoF analysis it is necessary to compute the potential values of variables at different points in the program at runtime and compare those values to each other or to some fixed value. Basic state invariants compare variable values to the undefined value (in Java: `null`). In a static analysis, the value of each variable could be represented as one of two alternatives: `null` or $\neg \text{null}$ [Cousot and Cousot, 1977]. In a dynamic analysis, the value in each trace could be compared to `null` directly. Aliasing invariants specify that two reference variables are or are not aliased. Alias analyses are a common form of data-flow analysis, with a large body of associated literature [Hind, 2001]. Similarly to basic state invariants, dynamic analyses could compare the value of reference variables in a trace directly. Deep copy invariants are more challenging, as they involve graphs of related objects (the clone and all its associated state). An analysis must compute an accurate model of the heap to verify such invariants precisely. Such an analysis is discussed in the next section on data-structure invariants. Also, invariants of this type require the value of primitive variables to be compared. Primitive variables are rarely modelled accurately by program analyses, as to do so causes a state-space explosion. However, as a method performing a copying operation would be expected to perform direct assignments from the state of the original to the state of the copy, a reaching definitions data-flow analysis could identify that the required assignment had occurred. Finally, a cloning or copying method may in turn call clone or copy methods on its composed state, so an inter-procedural analysis is required to track the flow of values into and returning from method calls.

Hedgehog [Blewitt et al., 2005] performs a data-flow analysis that tracks the values `null` or $\neg \text{null}$. A variable has non-`null` value if it satisfies one of three criteria: (1) it is initialized with a non-`null` expression, (2) all constructors initialize the variable or the variable is assigned the value of a parameter and (3) the assignment is guarded by a conditional that tests if the parameter is non-`null`. The second criterion overlooks the possibility that the variable will be re-assigned the value `null` after initialization, so is unsound. The third criterion involves path-sensitive data-flow, which is undecidable in

general, though in this case, the criterion is a useful heuristic that is sound. Lano et al. [1996] verify similar invariants, but only manually and against VDM++ code.

Numerous approaches, both based on static and dynamic analysis, track object identities and thus can compute aliasing relationships [Heuzeroth et al., 2003][Blewitt et al., 2005]. Each of these approaches could perform reaching definition analyses for primitive variables with minor extensions, though, to our knowledge, none of them do so. Hedgehog and PINOT both perform a limited inter-procedural analyses, but only for certain patterns and invariant types. The traces generated by Heuzeroth et al. are inter-procedural also, but it is not clear whether the heap, or just the local stack is modelled at any particular stage of the trace. In summary, basic state and aliasing invariants are supported by the literature, with some incompleteness or unsoundness common to program analysis tools while deep copying invariants is not supported by any DPVT, though some tools provide some prerequisites for the verification of invariants of this type.

2.5.1.5 Data-structure invariants

Data structure invariants are computationally expensive to verify, as they deal with structures that are of a potentially unbounded size. Despite this, proving properties of recursive data structures at runtime is an active area of research. Shape analysis [Sagiv et al., 2002][Berdine et al., 2007] is an advanced form of alias analysis that aims to statically determine the complex data structures built in the heap at runtime. Questions about heap-allocated data structures that a shape analysis can answer include aliasing, heap-sharing, reachability, the disjointness of two data structures or the presence/absence of cycles in a recursive structure. As heap-allocated data structures are theoretically unbounded in size, some part of the structure is represented exactly, while the rest is only approximated, depending on the properties of interest to the verifier. Size information, such as the length of lists, is typically lost, and information regarding approximated or *summarized* contents of the heap are *conservative* (produce a false outcome when a true outcome cannot be proven, i.e., generate false negatives).

With regard to object position invariants, SanD verifies that an object is inserted into a collection by identifying the argument passed to a collection’s insert method (see Table 2.3). Using this facility, they can identify that an object is added to the Observer list held by a Subject object, in the context of the Observer pattern. Taibi et al. [2009] verify an ‘attached’ event, though this event is defined only in terms of temporal logic and

not programming language code. As stated above, both Krishnaswami et al. [2009] and DiStefano et al. [2008] specify some shape invariants in the context of detailed, pattern instance-specific invariants. These specifications must be verified manually using a theorem prover. Rosenberg et al. [2010] use an ownership model to verify invariants of the Composite pattern. Limitations of the ownership model were discussed in Section 2.4.2.5. Finally, Bierhoff et al. [2008] verify limited shape invariants in the context of the Composite pattern given a number of assumptions and limitations (e.g., a Composite has a maximum of two children). Verification is also manual and the specification and verification language is the same (Spec#). In summary, some but not all invariants of object position are verifiable by DPVTs in the literature, while no approach can verify shape invariants without making limiting assumptions, and even then, invariants are not verifiable automatically and in their full generality.

2.5.1.6 Summary

Cardinality, interface dependency and control-flow invariants are well supported by DPVTs, though there are some unsound or sporadically-applied methods in the analyses that support control-flow invariants. Positive implementation dependency invariants are addressed by numerous tools, but negative implementation dependency invariants are largely overlooked. The invariant types that were poorly-supported by DPSLs, such as deep copying and all data-structure, were also poorly-supported by DPVTs, though there is some work in the area of verification of shape invariants that did not meet the inclusion criteria of this review.

2.5.2 Languages compared during verification

During verification, a design-pattern specification can be compared to its realization or implementation in a number of different language forms. Firstly, a design-pattern specification could be compared to another, perhaps more detailed, specification in the same language, to prove, for example, that a Pluggable Factory realization is a refinement of the Abstract Factory pattern. Secondly, a specification can be compared to a model in some object-oriented modelling language (OOML) that has a small semantic gap to code (e.g., UML [OMG, 2009]). Finally, a specification can be compared to OOPL code. This requires a semantic mapping between DPSL and OOPL that allows statements in the two languages to be related to one another. Figure 2.2 visualizes these alternatives while Table 2.4 outlines the mappings provided by the verification tools supporting the DPSLs reviewed, while

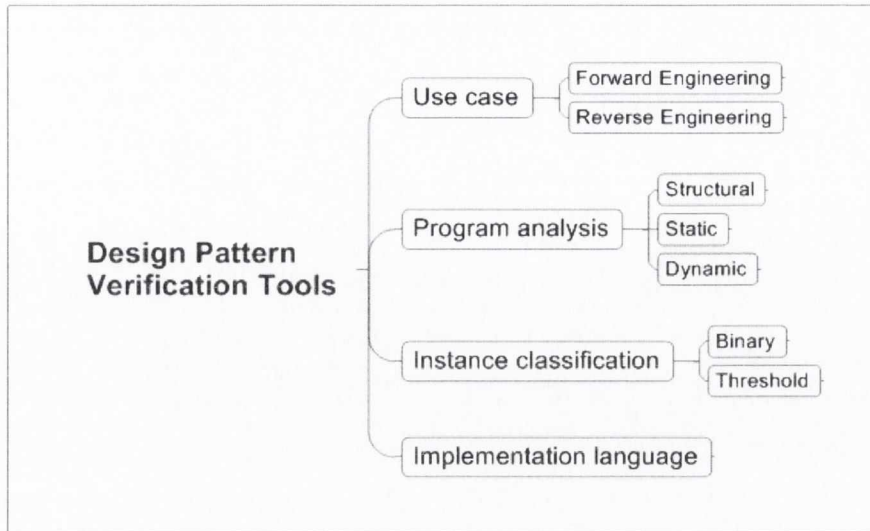


Fig. 2.11: Classification framework dimensions specific to DPVTs

Table 2.5 outlines the same for DPVTs without an associated DPSL included in the review. We also classify DPVTs based on the OOML or OOPL they analyze (see Figure 2.11), as shown in the final column of Tables 2.6 and 2.7.

2.5.3 Conformance relation

Conformance relations define how the specification and implementation are compared during verification. We distinguished three conformance relations based on their handling of behaviour that occurs in the implementation but does not appear in the specification (see Figure 2.2). We refer to this behaviour as *non-pattern functionality* or *unspecified functionality* from this point onwards. Patterns do not exist in isolation, but perform a particular function for the rest of the application. The unspecified functionality can nonetheless refer to pattern actors and interfere with the proper functioning of the pattern. For example, a pattern actor may be aliased, and the alias may be used to perform an action that breaks, for example, an object-state invariant. Such a violation may go undetected without a sufficiently powerful alias analysis. Also, a method actor may call unspecified methods that update pattern variables in unintended ways. A relation is categorized as safe if it considers all functionality (specified and unspecified) that may affect the correct execution of a pattern. A relation is categorized as unsafe if it considers only specified functionality.

The most restrictive relation is termed *closed*. The closed relation disallows non-pattern functionality completely in the implementation, i.e., the specification and implementation

are required to be equivalent. Such a relation is safe but impractical, as the implementation is required to be a closed system that does not have any other interactions and performs only the function specified. This requirement could be relaxed to only require the dynamic part of the specification, e.g., a control flow invariant describing a method call and a loop, to be closed while the static specification is open. This would allow other dependencies and methods to exist without violating the specification. Even with this relaxation, the closed relation remains very restrictive, as the developer who implements the pattern is forbidden from interleaving non-pattern and pattern functionality, which may be useful or necessary in many pattern realizations.

The *one-to-one* conformance relation is satisfied if the structure and behaviour of the specification is included in the implementation, while non-pattern functionality is ignored. The relation identifies a one-to-one mapping between elements in the specification and implementation and then terminates. As should be clear from the previous section, this relation may overlook a violation due to non-pattern functionality, and is thus categorized as unsafe.

The third conformance relation is named *refinement*. An implementation is a refinement of a specification if, when placed in the same environment, the implementation behaves exactly like the pattern when the pattern behaviour is involved, while it may also perform unspecified behaviour when required. This unspecified behaviour is verified also and must not violate any of the pattern invariants. This definition of refinement is derived from the one in Hoare [1985]. The objects and methods playing the roles dictated in a pattern specification do not constitute a closed system, but are inevitably involved in many other interactions, that may or may not be governed by any pattern specification. The performance of these other operations may delay the pattern behaviour, but should not cause the pattern behaviour to never occur or violate any of the pattern invariants.

It should be noted that the relation supported by a particular approach depends mainly upon the verification tool, but also depends on the semantics of the mapping from specification language to OOPL. A correct classification of each approach can only be done where the verification stage and/or OOPL mapping is described explicitly in the literature. Tables 2.4 and 2.5 summarize the conformance relation for each of the reviewed approaches. The cells belonging to the DPSLs that do not have a clearly defined semantic mapping to an implementation language and do not have an associated verification tool are left blank.

The DPSLs based on temporal logics (Dong et al., BPSL and OC/VDM++) are capa-

Name	Implementation language type	Conformance relation	Instance classification
DisCo	Spec \rightarrow Spec	Refinement	Binary
OC/VDM++	Spec \rightarrow OOML	Refinement	Binary
BPSL	Spec \rightarrow Spec	Refinement	Binary
Lauder & Kent	Spec \rightarrow OOML	One-to-one	Binary
Le Guennec et al.	Spec \rightarrow OOML	One-to-one	Binary
LePUS3	Spec \rightarrow OOPL	One-to-one	Binary
RBML	Spec \rightarrow OOML	One-to-one	Binary
DPML	Spec \rightarrow OOML	One-to-one	Binary
FUJABA	Spec \rightarrow OOPL	Refinement	Threshold, Binary
Dong et al.	Spec \rightarrow Spec	Refinement	Binary
GEBNF	Spec \rightarrow OOML	Refinement	Binary
DVP	Spec \rightarrow Spec	Refinement	Binary
Hofer	Spec \rightarrow OOPL	Refinement	Binary
Shetty & Menezes	Spec \rightarrow OOPL	Refinement	Binary
Ammour et al.	Spec \rightarrow OOML	One-to-one	Binary
Contracts	Spec \rightarrow Spec	Refinement	Binary

Table 2.4: Relation between specification and implementation language for each DPSL

ble of specifying a refinement relationship, as they use temporal operators that can state something is always true, i.e., true for the entire execution of the program, after a given event or between events. DPVTs that perform static or dynamic analyses are also classified as supporting the refinement relation, where it is clearly documented that object identities and other data values are tracked throughout the analysis and used to reason about conformance.

2.5.4 Pattern instance classification

The classification of candidate pattern instances as conforming or not conforming to a pattern specification by most DPVTs is a binary yes or no decision. However, a number of approaches use a scoring system that includes a *threshold* value, where a score above

Name	Implementation language type	Conformance relation	Instance classification
D ³	Spec \rightarrow OOPL	Refinement	Binary
SanD	Spec \rightarrow OOPL	Refinement	Binary
PINOT	Spec \rightarrow OOPL	Refinement	Binary
Hedgehog	Spec \rightarrow OOPL	Refinement	Binary
Columbus	Spec \rightarrow OOPL	One-to-one	Threshold
PEC	Spec \rightarrow OOPL	Refinement	Binary
Peng et al.	Spec \rightarrow OOML	Refinement	Binary
DeMIMA	Spec \rightarrow OOPL	Refinement	Binary
MoDeC	Spec \rightarrow OOPL	Refinement	Binary

Table 2.5: Relation between specification and implementation language for each DPVT

the threshold value indicates conformance (Figure 2.11). Tsantalis et al. [2006] calculate a similarity score between graphs of classes, where the edges are associations and inheritance relations. FUJABA [Wendehals and Orso, 2006] collects a number of traces of a program, and counts the number of traces that reach an *accepting* or *final* state of a petri net representation of the design pattern specification. Finally, Columbus [Ferenc et al., 2005] counts the number of loops and recursive calls in a method, and classifies the method as the Strategy’s algorithmInterface method if the number is above a particular threshold. We classify the instance classification method of approaches as either ‘binary’ or ‘threshold’ (see Tables 2.4 and 2.5).

2.5.5 DPVT use cases

Tools for design pattern verification fall into two categories depending on their intended use (see Figure 2.11). These categories separate tools that aim to support a forward engineering (FE) process [Blewitt et al., 2005][Peng et al., 2008], from those that aim to support a reverse engineering (RE) process [Shi and Olsson, 2006][Bergenti and Poggi, 2000][Guéhéneuc and Antoniol, 2008][Smith and Stotts, 2003] (see Tables 2.6 and 2.7). A FE process guides the development and maintenance of software systems by allowing the developer to check that their new implementation or modification conforms to the designer’s original intent. A RE process takes existing source code and attempts to discover patterns in source code, to aid

program understanding. A FE process that incorporates design-pattern specification and verification involves three stages: specification of the properties/invariants that characterize each of the supported patterns, manually matching roles in the specification to actors in the source code, and verification. A RE process shares the first stage with a FE process, but must then make an automatic search of the source code for pattern implementations.

Both FE and RE tools have the potential to produce false positives and false negatives during verification, but in FE these errors are caused by either a fault in the specification, or the tool. RE tools have the added problem of producing false positives that were never intended to be patterns. This cannot happen in a FE process, as the user identifies explicitly which entities are supposed to constitute a pattern implementation. This raises the issue of the level of detail of pattern specifications again, as a generic description of a static class structure is likely to produce a large number of false positives. Indeed, numerous designers of RE approaches complain that some patterns have specifications that are too vague to provide meaningful results during verification. For example, PINOT [Shi and Olsson, 2006] does not support the Builder and Memento patterns for this reason.

Performing a detailed program analysis, required for the verification of dynamic pattern invariants (e.g., object-state invariants), on the entire source code is prohibitively expensive for a practical verification tool. For this reason, most RE tools perform a first pass to identify pattern candidates, applying data-flow analysis only to those candidates identified on the first pass. SanD [Heuzeroth et al., 2003] and IDEA [Bergenti and Poggi, 2000] identify candidates by their static structure first, then analyze behaviour. PINOT ‘begins its detection process for a given pattern based on what is most likely to be most effective in identifying that pattern (e.g., declarations, associations, delegations)’.

In summary, design pattern verification tools have two major use-cases: forward and reverse engineering. Reverse engineering provides a number of extra challenges that make it a more difficult problem to eliminate false positives and false negatives during verification.

2.5.6 Program analysis

From the perspective of this review, tool support for DPSLs is vital for two main reasons: it can speed up the proof of conformance to a pattern specification, saving time that is precious in a software development life cycle, and secondly, it has the potential to shield much of the complexity of the formal specification from the software developer, who is principally concerned with applying patterns and developing well-designed systems. In

Name	Use case	Program analysis	Implementation language
DisCo	-	-	-
OC/VDM++	FE	-	VDM++
BPSL	RE	Static	TLA+
Lauder & Kent	FE	UML	UML
Le Guennec et al.	FE/RE	Static	UML
LePUS3	FE/RE	Structural	Java
RBML	FE	Static	UML
DPML	FE	Static	UML
FUJABA	RE	Static, Dynamic	Java
Dong et al.	FE	Static	TLA
GEBNF	RE	Static	UML
DVP	-	-	-
Hofer	FE	Dynamic	Spec# (C#)
Shetty & Menezes	RE	Static	C#
Ammour et al.	FE	Structural	UML
Contracts	-	-	-

Table 2.6: Analysis use case, program analysis and implementation language classification of each DPSL's associated DPVT

Name	Use case	Program analysis	Implementation language
D ³	RE	Static	Java
SanD	RE	Dynamic	Java
PINOT	RE	Static	Java
Hedgehog	RE	Static	Java
Columbus	RE	Static	C++
PEC	FE	Dynamic	Java
Peng et al.	-	-	-
DeMIMA	RE	Dynamic	Java
MoDeC	RE	Dynamic	Java

Table 2.7: Analysis use case, program analysis and implementation language classification of each DPVT

order to verify the behavioural invariants, it is necessary to analyze the behaviour of a candidate implementation at runtime. Program analysis techniques [Nielson et al., 1999] are intended to perform this task, and the use of program analysis techniques to verify DPSL specifications is reviewed in this section. Program analyses can be classified as either static or dynamic (see Figure 2.11).

Static program analysis (SA), as defined by Nielsen et al. ‘offers static compile-time techniques for predicting safe and computable approximations to the set of values or behaviours arising dynamically at run-time when executing a program on a computer’ [Nielson et al., 1999]. SA techniques build an abstract representation of the entire program in terms of control- and data-flow. As a number of problems in program analysis are undecidable in general [Landi, 1992], SA techniques produce safe or conservative results: they output false negatives or “don’t know” when the answer is undecidable [Sagiv et al., 2002].

There are many uses of the term static analysis in the literature, and they do not all have the same meaning. Static analysis is a term often applied to the analysis of the static class structure of inheritance and dependency, but it is used here with respect to the more powerful definition from Nielson et al., given above. This definition of static analysis is equivalent to some definitions of program analysis in the literature. We use the term program analysis in a more general sense to describe any technique that aims to understand the runtime behaviour of a program, and is not limited to compile-time

analyses.

Dynamic program analysis (DA) involves executing the system with a set of test cases. The challenge of developing a DA system is generating adequate inputs to attain sufficient code coverage (e.g., to execute all paths through conditional statements that are reachable). This is known as the code coverage problem, and is a limitation of dynamic analyses, as it is difficult to identify the set of test cases that will exercise every potential path, or even every line of code in a program. Dynamic analysis is more appropriate for use in the context of specific requirements, and for identifying defects caused by interactions that are too complex to be uncovered by static analysis [Evans, 2005]. Design-pattern verification tools exist that use both the static [Shi and Olsson, 2006] and dynamic [Wendehals and Orso, 2006][Heuzeroth et al., 2003] approaches to program analysis. We provide a brief overview of the two approaches here using an illustrative example. The classification of the program analysis method of each of the reviewed DPVTs is given in Tables 2.6 and 2.7.

2.5.6.1 Static analysis

Data-flow analysis (DFA) is the only form of static analysis used by existing approaches to the verification of DPSL specifications ([Aho et al., 1986] Section 10.5) ([Nielson et al., 1999] Ch.2). Data-flow information can describe which variables are live on exit from a block and the variables that are aliased at a particular point in the program execution. An analysis progresses by solving a system of data-flow equations that refer to the entry and exit of a basic block. A basic block is ‘a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without the possibility of branching’.

PINOT [Shi and Olsson, 2006] describes the use of data-flow analysis to verify implementations of the Singleton pattern. The Singleton pattern has proven problematic for design pattern specification and verification approaches, as it involves conditional behaviour and object initialization. The Singleton pattern is represented in PINOT as a class whose methods either (a) return an instance or (b) implement lazy instantiation. This is a tractable data-flow problem, as it only involves two values: `null` and $\neg \text{null}$. The Singleton’s `getInstance()` method (Figure 2.12) is separated into two basic blocks. `BasicBlock0` is shown to be the only block to create an instance of the class, and will do so only if the object was `null` on entry to the method. `BasicBlock1` does not assign `null` to the object variable and returns it unchanged. So for both input values: `null` and $\neg \text{null}$, the output value is verified to always be $\neg \text{null}$. The same analysis also proves that only one instance can

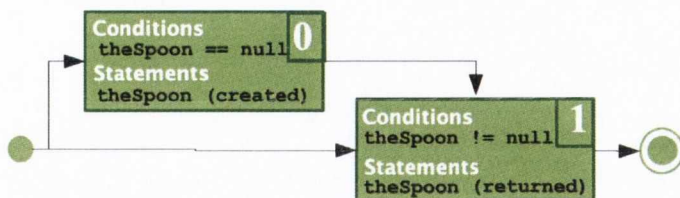


Fig. 2.12: Control-flow graph for an implementation of the Singleton's `getInstance()` method (PINOT)

be created. This analysis demonstrates a refinement relation between PINOT specification and Java code, as any non-pattern functionality that subsequently updated the value of the instance would be identified by the data-flow analysis.

2.5.6.2 Dynamic analysis

FUJABA [Wendehals and Orso, 2006] uses dynamic analysis to verify that sequences of method calls occur in the correct order using method call traces captured from Eclipse using a tool that leverages the debugging interface of the Java Virtual Machine (JVM). Sequences of method calls are transformed into a Deterministic Finite Automata (DFA). A single call is transformed into two states with a transition between them. The transition is labelled with the caller object, the call symbol(\rightarrow) and the callee object. When two calls occur in sequence in the pattern specification, the end state of the first call is merged with the start state of the second call, and so on. A DFA for the State pattern is shown in Figure 2.13. The Petri net concept of tokens are used to count the method traces that pass correctly through each of the states of the pattern DFA. A trigger is any method that labels a transition from the initial state e.g. `context.request` and `context.setState` in Figure 2.13. The execution of a trigger method call adds one token to the initial state. The token moves through the DFA as the method trace follows the correct behaviour of the pattern specification. Method calls that occur in the wrong order move to rejecting states and are evidence of non-conformance. When all tokens are collected the number of tokens that have passed through an accepting state (a state at which the DFA has performed its full procedure) are compared to the number of tokens ending in a rejecting state. This example describes the verification of control-flow invariants, but dynamic analysis has also been used to verify invariants that require data-flow information, such as aliasing invariants, as

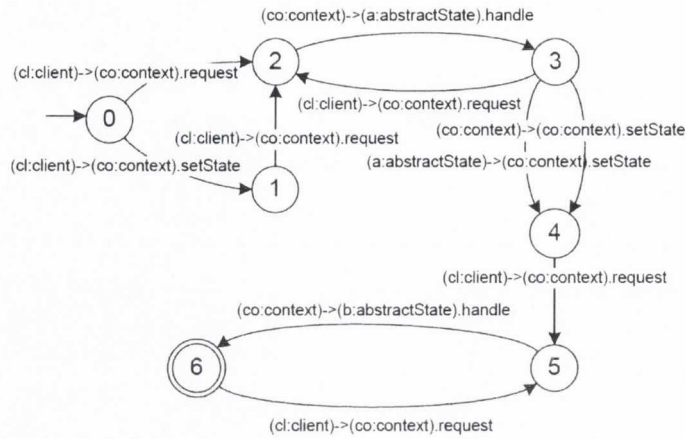


Fig. 2.13: FUJABA [Wendehals and Orso, 2006] DFA for the State pattern

discussed above. To our knowledge, none of the reviewed DPVTs that perform DA address the code coverage problem in detail: DeMIMA, for example, assumes the existence of a set of test cases that will provide good code coverage, rather than generating the set of test cases itself.

2.5.7 Assigning pattern roles to implementation actors

Assigning roles, as described in the design pattern specification, to corresponding actors in the implementation involves creating a binding between the two, in order for the actor to be checked for conformance to be checked. Binding actors to roles is a relatively straightforward process compared to specification and checking conformance to dynamic pattern specifications, but there are a few issues that deserve a brief discussion. A number of approaches use queries to compare implementations to specifications. Dong et al. pass implementation-specific names as parameters in a query to a Prolog rules database. Mak et al. and Le Guennec et al. use OCL queries as part of their specifications, which are then applied to UML (Class) diagrams. Most of the approaches that use a graphical syntax for specification require a separate diagram of the pattern realization. LePUS3 uses shaded instead of blank shapes to distinguish *constants*, i.e., actual method signatures or classes from *variables*, i.e., class or method roles. LePUS3 supports specifications that contain only variables (pattern specifications), a mix of variables and constants (frameworks) and only constants (concrete programs). DPML uses solid outlines for its specification symbols, and identical shapes with dashed outlines for symbols in an Instantiation diagram. Verification

in these two approaches involves a check for equivalent symbols in the two diagrams being compared, i.e., a one-to-one conformance relation. A further link is required, however, between instance model and code. Lauder and Kent use three layers of models, with abstract state and behaviour at the highest (*role*) level and concrete semantics at the lowest (*class*), with an explicit user-defined mapping between each layer. The role level suffers from the genericity problem as discussed in Section 2.3.1.

Most approaches perform some consistency checking while linking is being performed. Invariants that can be checked simply, such as cardinality, are often supported directly in the linking process. DPML also guides the user through the process by highlighting valid candidates for certain pattern roles. Allowing an implementation actor to play roles in multiple patterns, e.g., an Observer that is also a Singleton, is also desirable and can be supported easily. Finally, Mak et al. support flexible links between specification and implementation. Taking the example of a combination of Abstract Factory with either Factory Method or Prototype: “Whether the instantiation takes place locally as the factory method in [the] Factory Method pattern, or delegates to other methods such as the clone method in the Pluggable Factory pattern is left open”. This allows for more flexibility in the conforming implementation, but adds an extra verification challenge.

2.5.8 Summary

In this section, we presented the dimensions of the classification framework specific to DPVTs, and used these dimensions, along with the invariant types identified in the specification section of the classification framework, to classify existing work in the literature. Of the invariants supported by DPSTLs, implementation dependency is found to be poorly-supported by verification tools. Cardinality invariants are addressed by some tools that are based on a DPSTL. A single approach addresses data-structure invariants regarding object position, and no approach supports shape invariants. The Object-state invariant type relating to deep copying behaviour is also overlooked by existing DPVTs. Control-flow invariants are well supported, but with some inaccuracies and unsoundness common in software verification.

Numerous DPVTs are capable of addressing the most sophisticated conformance relation (refinement). DPVTs were identified that perform both static and dynamic analysis. Static analysis and dynamic analysis have different advantages and disadvantages, and are suitable in different contexts. The state-of-the-art tools in design pattern verification are identified

as PINOT [Shi, 2007a], Hedgehog [Blewitt et al., 2005] and D³ [Stencel and Wegrzynowicz, 2008], as they support the most sophisticated conformance relation and verify some of the more challenging behavioural invariant types. Most DPVTs address the Java programming language, but some also target C++ and C#. DPVTs in the literature are quite evenly split between applying a forward engineering and a reverse engineering use case, while some do not define a particular use case. Most DPVTs classify implementations as either conforming or non-conforming, while a few use a scale of conformance levels along with a threshold.

2.6 Conclusions

A multitude of approaches to the formal specification and verification of object-oriented design patterns exist in the literature. This chapter presents our novel classification framework for DPSLs and DPVTs. Our inclusion criteria focus on expressiveness and also try to limit the scope of the review without excluding any approach that has a unique solution for some invariant type. Some issues specific to design pattern specification such as the levels of abstraction at which to specify and the re-usability of specifications were discussed.

We identified the OOP syntactic elements, as well as 13 invariant types, within 5 invariant categories, that are required for precise design pattern specification. We classified the DPSLs according to their support for each of the syntactic elements and found that one approach supported all elements, while a few others supported a majority of elements. With regard to invariant elements, it was found that three invariant types in particular were either poorly-supported or entirely overlooked in the literature. These are implementation dependency, deep copying (object-state) and data-structure invariants regarding data-structure shape. This lack of support is manifest as either DPSLs lacking the syntax and semantics to express the invariant type or DPVTs lacking the ability to perform a required analysis, or both.

Other classification dimensions include the conformance relation, use case, targeted implementation language and program analysis applied. The most powerful conformance relation was found to be addressed by a number of approaches, some of which applied static and some of which applied dynamic program analyses.

We conclude that a novel DPSL is required that is capable of specifying invariants from the three poorly-supported invariant categories, and also capable of specifying design pattern variants, which vary in terms of structure and/or behaviour. The remainder of this

thesis describes the design and evaluation of such a DPSL and its associated DPVT. The associated DPVT implements a static analysis, for reasons discussed above. The DPVT targets the Java programming language, due to its popularity, and the fact that most of the code bodies analyzed by existing DPVTs are written in Java.

Chapter 3

Alas

In the previous chapter, we identified a set of syntactic and a set of invariant elements necessary to precisely specify object-oriented design patterns. In this chapter, we present Alas (Another Language for pAttern Specification), a DPSL capable of expressing each of the syntactic and invariant elements identified in the previous chapter¹. In particular, we focus on the three invariant types poorly supported in the DPSL and DPVT literature:

- Implementation dependency invariants
- Deep copying object state invariants
- Data-structure shape invariants

Also, it was found in the previous chapter that the specification of structural and behavioral variants of patterns is poorly supported. We describe the variant specification features included in Alas.

In Section 3.1, we consider major design decisions inherent in the definition of a DPSL and provide a rationale for the decisions we made in designing Alas. A brief illustrative example of specification in Alas is also provided. Section 3.2 outlines structural specification in Alas, while Section 3.3 describes behavioural specification. We conclude the chapter with a brief summary in Section 3.4.

¹Alas might be more accurately described as a ‘notation’ for design pattern specification, as it has not been formally demonstrated that each statement that can be made in Alas is meaningful and consistent.

3.1 Introduction

This section discusses the design alternatives considered and the salient design decisions made in the definition of Alas. The main contributions of Alas in the context of design pattern specification are briefly summarized and the section finishes with an illustrative example of an Alas specification.

3.1.1 Language basis rationale

As a large body of work on design pattern specification exists, we decided to base our new language on an existing language that has either been developed specifically for design pattern specification, has been modified and applied in that context or has some characteristics that make it suitable for specification of design patterns. While the ability to express each of the design-pattern aspects outlined above is the primary design goal of Alas, other characteristics are also desirable and relevant, namely: precision, the popularity of the existing language in academia and industry, the availability of tool support, and the semantic gap to Java. A language is precise if each of its elements has a clearly-defined meaning that is free from ambiguity. The related concept of consistency requires that the meaning of different language elements never contradict one another. Precision in a language definition removes the number of design decisions left open to the developer of analysis tools conforming to the language definition, improving tool interoperability. The removal of ambiguity also improves communication among the users of the language and increases the level of confidence in the analysis results of language-conforming tools. The semantic gap in this context is the difference between the meanings of constructs in the specification and implementation language. For example, the semantic gap between Java and some specification language with object-oriented constructs may be easier to bridge than the gap to a language without such constructs. Each of these properties were considered when choosing a basis for Alas. This section considers first DPSLs then non-DPSLs as a basis for Alas.

Unfortunately, two of the three more precisely-defined DPSLs reviewed (LePUS [Eden, 2008] and DisCo [Mikkonen, 1998]), were also two of the least expressive with respect to the pattern invariant categories and types, especially the behavioural invariant categories. The third more precisely-defined DPSL (OC/VDM++ [Lano et al., 1996]) supports all control-flow invariants as well as some restricted dependency and object-state invariants, but does not address cardinality or data structure.

Architecture Description Languages such as Wright [Allen, 1997], and other languages based on process algebras are precisely defined and supported by industrial tools but address a different level of abstraction to DPSLs, as discussed in Chapter 2. Separation logic has also been used to specify design patterns [Distefano and Parkinson J, 2008], though it does not meet the criteria for inclusion in our classification framework in Chapter 2. It is also ideally suited to data-structure and some object-state invariant specification. Its formality gives it a steep learning curve and it does not address the other three invariant categories. Object-oriented specification languages have received much attention in the research community recently. JML [Leavens et al., 2006] and Spec# [Barnett et al., 2005] are relatively expressive, but do not provide a convenient mechanism for control-flow invariant specification and do not address cardinality or dependency invariants at all. UML is used (in extended and/or constrained forms) by a number of DPSLs [Lauder and Kent, 1998][Le Guennec et al., 2000][Mak et al., 2004], including the most expressive DPSL according to our classification framework, RBML [France et al., 2004]. Also, UML is the de facto standard for object-oriented software modelling, and as such, it is widely taught in universities and widely used in industry [Fowler and Scott, 2000][Cheesman and Daniels, 2001]. UML 2 Sequence Diagrams, in combination with OCL, provide a convenient mechanism for control-flow and some object-state invariant specification. UML has also been extended to address cardinality invariants by a number of DPSLs [Le Guennec et al., 2000][Mak et al., 2004].

A related but separate issue is how to translate each of the syntax elements of the DPSL into constraints on object-oriented programming language (OOPL) code (in this case, Java code). A number of formal definitions of Java exist, varying in semantic framework and Java language coverage [Börger and Schulte, 1999][Farzan et al., 2004][Parkinson, 2005]. These definitions could be reused in the definition of Alas to improve the precision of the overall model, and may improve tool support, if the definitions have associated tools. Model-driven engineering [Selic, 2003] describes a methodology for transforming models from one meta-model (i.e., language definition) to another, and meta models for parts of UML, OCL, Java and other OOPLs exist [AtlanMod, 2010]. Transformations are written in an operational language, making the meaning of the mapping hard to identify and understand. Also, meta-models are often informal or incomplete.

VDM++ [Durr and van Katwijk, 1992] is a specification language based on the VDM formal method, but extended to support object-oriented and concurrent systems. Some

tools exist to reverse engineer VDM++ from Java and also to generate Java/C++ from VDM++, and the mappings in both directions are documented [VDMTools, 2010][VDM-Tools , 2010]. OCL is used to define the semantics of UML, but its operational style lends itself to verbose specifications [Vaziri and Jackson, 2000], a number of OCL expressions evaluate to an undefined value by design, and there are a number of UML features OCL is not able to formalize [OMG, 2010b]. Finally, another alternative approach to providing a translation from DPSL to Java, is to create one from scratch using some widely-used existing framework, such as operation or denotational semantics.

Ultimately, we decided to make some syntactic extensions to UML 2.0 (subsequently referred to as UML 2) and provide more detailed semantic definitions (using an operational semantics) where required for design pattern specification. UML was chosen as it has already been demonstrated to be well suited to design pattern specification, especially by RBML. Other benefits of choosing UML are its popularity and the small semantic gap between elements of its syntax and constructs in OOPLs.

However, UML is widely criticized for its lack of a formal semantics [Lund and Stølen, 2006][France et al., 2006][Jackson, 2002] and, possibly because of this, lacks supporting program analysis tools. Also, UML in its current form is unsuitable for design-pattern specification for a number of reasons, each of which is listed here and described in more detail later in the chapter:

- Le Guennec et al. [2000] note that UML, despite its templates and parameterized binding, is not suited to expressing cardinality invariants. This is due to a lack of control over the number of bindings that can be made between classes and roles.
- The binding semantics of lifelines are unsuitable for specifying iterations over lists or other collections of unbounded size, and also for specifying generic pattern behaviour.
- The operators introduced in UML 2 to describe conditional behaviour are ambiguous [Lund and Stølen, 2006].
- OCL lacks a transitive closure operation and OCL queries and constraints in the context of a particular object may only refer to objects that are navigable from the contextual object via association. These two features, as well as others, make data-structure invariants expressed in UML/OCL verbose, error-prone and inefficient to verify [Vaziri and Jackson, 2000] [Baar, 2010].

The rest of this chapter describes how each of these deficiencies of UML in the context of the specification of design patterns are resolved in Alas.

3.1.2 Alas design decisions

An Alas specification consists of a single structural specification and a behavioural specification, which refers to entities defined in the structural specification. The structural specification is made up of a Structure Diagram (SD), with syntax based on UML 2 Class Diagrams (subsequently Class Diagrams), as well as supporting text, which is placed inside UML 2 constraint boxes (e.g., cardinality invariants are specified textually). The behavioural specification consists of zero or more Behaviour Diagrams (BDs), with syntax based on UML 2 Sequence Diagrams, which are a good fit for the interactions between objects described in the GoF catalogue (the catalogue itself uses OMT Interaction Diagrams, a forerunner of Sequence Diagrams, but lacking constructs to describe selection and iteration [Booch, 1994]).

Structural pattern roles (classes, methods and references) are specified either graphically or textually in the UML Class Diagram-based SD. Languages based on UML, and UML Class Diagrams in particular, can leverage the popularity of UML, making them easier to learn and use. Also, the GoF catalogue itself uses OMT Class Diagrams [Rumbaugh et al., 1991], a forerunner of UML Class Diagrams, to describe pattern structure, indicating that Class diagrams are a suitable means of describing the structural constraints imposed by design patterns.

Cardinality invariants in Alas are expressed using first-order logic, similarly to some DPSLs [Bayley and Zhu, 2010][Shetty and Menezes, 2011]. First-order logic enables the specification of the universality and existence, and uniqueness cardinality invariant types. As Class Diagrams are used to specify relationships that should exist and not relationships that should not exist, they are suitable for specifying *positive dependency invariants*, but not *negative dependency invariants*. In Alas, dependency invariants are specified textually. We could have defined negated versions of the standard UML relationships such as association and inheritance, but this would preclude the combination of cardinality and dependency invariants in a single invariant (as cardinality invariants are text only). Another approach would be to use OCL at the meta-model level, but we decided against this due to the verbosity of the specification.

BDs support the description of the behavioural syntactic elements identified in Chapter

2, as well as *behavioural pattern roles* (i.e., object roles). *Control-flow invariants* are expressed within the BDs, which include syntactic elements taken from UML 2 that support the specification of *Sequence*, *Selection*, *Iteration* and *Method calls*. Both *object-state* and *data-structure* invariants are specified in constraint boxes accompanying a BD.

BDs provide a concise way to specify design pattern *variants* that differ in terms of behaviour. This is done by clarifying existing Sequence Diagram operators and adding an additional operator (see Section 3.3.5). Alas object-state and data-structure invariants can be expressed in Hoare logic-style post-conditions and invariants similar to class invariants used by many object-oriented software specification languages [OMG, 2010a][Meyer, 1997][Liskov and Wing, 1994][Leavens et al., 2006]. Emerson [Emerson, 1990] states that Hoare logic is a simpler formalism than temporal logic for sequential terminating programs and it has the advantage that invariants in Hoare logic are modular, simplifying verification [Müller et al., 2006][Chin et al., 2008]. Finally, generic structure and behaviour in design patterns is specified in Alas using sets, flexible role-to-actor binding rules and modified roles and operators.

3.1.3 UML as a basis for design pattern specification

The UML standard provides two extension mechanisms: a ‘lightweight’ extension mechanism (UML Profiles) that provides a means to constrain but not contradict existing UML constructs and a ‘heavyweight’ extension mechanism (the Meta-Object Facility [MOF]), that allows the definition of UML-like languages (UML itself was defined, retrospectively, using the MOF). As Alas requires non-standard UML syntax and semantics, UML Profiles are not applicable. The MOF, as a tool for language definition, has been criticized for its inflexibility in modelling instances and types [Volz and Jablonski, 2010][Génova, 2009]. The limitations of OCL for semantic definition have been discussed above. We choose not to use MOF or OCL in the definition of Alas, instead defining our novel extensions in terms of an operational semantics.

The function of the most well-known UML diagrams (Class, Sequence, Activity) is to describe concrete software architectures, that is, actual *actors* (classes, methods and objects) and their association and interaction with one another. *Roles* in design patterns are not actors but placeholders for many potential actors that may conform to the constraints imposed by the role. UML’s mechanism for describing class structure at the level of roles is the UML Collaboration diagram. UML diagrams may also be parameterized by the use of

UML Templates. Collaborations and templates have a number of characteristics that make them suitable for the definition of pattern roles. Both allow actors to be bound to multiple different roles in different contexts and both allow an actor to contain other structural and behavioural actors unspecified by the role: ‘A bound element may have multiple bindings, possibly to the same template. In addition, the bound element may contain elements other than the bindings’ [OMG, 2010b] pp642.

However, in design pattern specification it is necessary to specify that some roles are filled by only one actor in one context (e.g., **operation** in the Decorator pattern) and that some roles may be filled by multiple actors (e.g., **visit** methods of a Visitor class) in the same context. In UML collaborations and templates, a template may have many bindings and a collaboration may have many uses, but we take this to assume that each binding or use is in a different context. The alternative is that a role may be filled by one or more actors in a given context. In either case, there is no mechanism for making the required distinction between single and multiple actor roles. Likewise, it is not clear whether an actor may fill multiple different roles in the same context.

In summary, Alas diagrams correspond to the level of abstraction of UML collaborations and templates, but this mechanism is not applicable without modification due to ambiguous role-actor binding semantics.

3.1.4 Sample Alas specification

Before describing the Alas syntax and semantics for addressing each of the design pattern aspects in detail, we present a simple example Alas specification here. This example provides an intuitive sense of how Alas specifications are composed, while glossing over semantic issues and usage rules. The invented example we take is called the FalseFaçade pattern, which is similar to the GoF Façade patterns, but with some extensions to exercise features of Alas not needed to specify Façade.

The structural specification of the FalseFaçade pattern is shown in Figure 3.2, and consists of a SD and three constraint boxes². The SD specifies that there are three class roles (Client, Façade and SubSystemClass), three reference roles with multiplicities (**facade**, **ssc** and **next**) and each class role has one method role. Thus, a conforming implementation is required to have three classes with these inter-relationships. The three reference roles

²The diagrams in this thesis are drawn using Rational Software Architect (RSA) [Rational®, 2007], but the expressiveness and meaning of Alas is independent of diagram editor.

are declared using unidirectional associations and the methods are all public³. Each SD has a *pattern label* that associates the SD with a pattern name. Pattern labels are specified within constraint boxes using the syntax `pattern <patternName>`.⁴

The specification of Figure 3.2 can also be expressed textually. The `class` operator is used to declare and name a class role. The `refVar` operator is used to declare and name reference roles and takes two operands: a class name from a previously-defined class role and a reference name. The `isMethod` operator is used to declare a method and can be used to constrain the method's modifier, actor name, parameters and return type (No constraints are indicated by a `*`). Role specifications can place constraints on actor names to capture naming conventions for a particular pattern or within a particular code body. Factory Methods, for example, often begin with the prefix `create`. Methods are associated with classes using the `hasMethod` operator, which takes a class and a method operand. Reference roles are associated with class roles using the `hasRefVar` operator, which takes a class and reference role operand. The equivalent textual specification to the graphical part of Figure 3.2 is given in Figure 3.1.

The first constraint box in Figure 3.2 contains a dependency invariant using the Alas `hasRef` operator, which specifies that a class has some reference variable of a particular class. The `hasRef` operator takes two operands, in this case, two class roles. The invariant also uses the logical operator `NOT` with the usual meaning. The invariant states that the Client class should not hold a direct reference to a SubSystemClass instance. The second constraint box declares a named variant (`backReference`) that requires that the SubSystemClass has a reference to the Façade. If an actor is found to bind to this reference role, then the implementation is an instance of the `backReference` variant. Otherwise, it is an instance of the pattern's 'core' variant, the variant that includes none of the invariants applied by structural and behavioural variants. The third constraint box declares a data structure (`sscChain`), which is a linked list of SubSystemClass objects. While the `next` reference role is declared in the SD, it is still necessary to define the data structure separately, as a data structure definition can specify that some subset of the SD is a relevant data structure.

The behavioural specification in the case of the FalseFaçade pattern consists of only one BD (Figure 3.3), while other pattern specifications may require two or more. The BD contains three object roles, indicated by the three lifelines. Each of the lifelines is an object of a class role defined in the SD. Note also that the object names for the Façade and

³indicated in RSA by the green circle before the method name.

⁴Pattern labels are typically omitted from the Alas SDs in this thesis to avoid visual clutter.

```
pattern AbstractFactory
class Client
class Facade
class SubSystemClass
refVar Facade facade
refVar SubSystemClass ssc
refVar SubSystemClass next
doStuff isMethod public, *, *, void
delegatingMethod isMethod public, *, *, *
receivingMethod isMethod public, *, *, *
Client hasMethod doStuff
Facade hasMethod delegatingMethod
SubSystemClass hasMethod receivingMethod
Client hasRefVar facade
Facade hasRefVar ssc
SubSystemClass hasRefVar next
```

Fig. 3.1: A textual specification of the FalseFaçade pattern, equivalent to the graphical specification in Figure 3.2

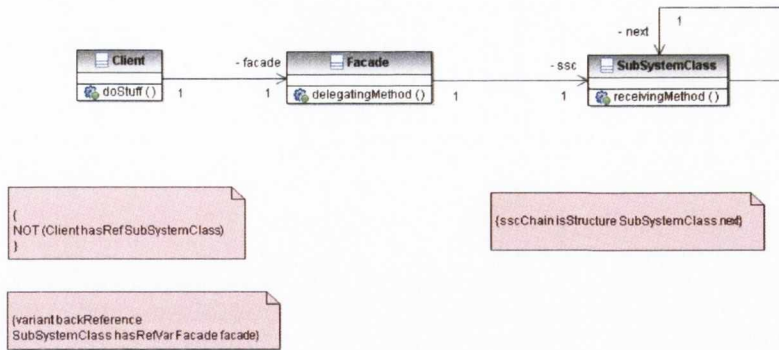


Fig. 3.2: The structural specification of the FalseFacade pattern, including a structure diagram with three class roles, three method roles and three reference variable roles. The constraint boxes define a dependency invariant, a pattern variant differing in structure from the structure diagram and a data-structure definition.

SubSystemClass correspond to the reference roles defined in the SD⁵.

The specification states that the Client’s `doStuff` method calls the Façade’s `delegatingMethod`. The call from the Façade to the SubSystemClass object role is contained in an `opt` operator, indicating that the call is conditional. A constraint box is attached to the return arrow of the `receivingMethod`, indicating a post-condition on it. The post-condition in this case states that the linked list of SubSystemClass objects is always cycle free at the end of the `receivingMethod`, which may mutate the list and potentially violate the invariant.

3.2 Structural specification in Alas

Structural specifications in Alas allow the definition of structural invariants and also define the roles that are the basis for behavioural specification. Structural specifications also allow the definition of structural variants. Two invariant categories contain invariants that can be expressed using structural specification alone: dependency and cardinality. Section 3.2.1 discusses dependency invariant specification in Alas. Section 3.2.2 outlines structural variants and Section 3.2.3 deals with cardinality invariants.

⁵The numbering placed before the method name that labels a method call is inserted by RSA and is not Alas syntax.

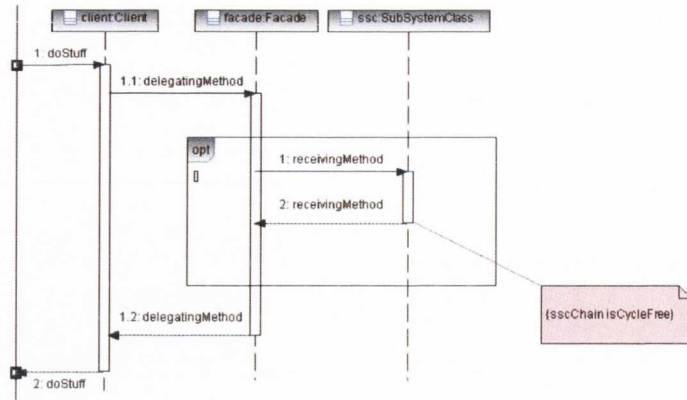


Fig. 3.3: The behavioural specification of the FalseFacade pattern. The behaviour within the `opt` operator is conditional. A data structure invariant is attached to the end of `receivingMethod`, indicating a post-condition.

3.2.1 Dependency invariants

While every design pattern specification defines required associations between classes that can be described as dependencies, the focus of many dependency invariants imposed by the GoF patterns is on associations that should not exist, i.e., negative dependencies. We distinguish between interface and implementation dependencies. The former refers to the situation when one class knows the interface of another class, for example, by holding a reference to that class. The latter refers to a situation where one class commits to a particular implementation of another class by calling its constructor directly. As discussed in Chapter 2, interface dependency invariants are widely supported by DPSLs, a number of DPSLs support a restricted form of implementation dependency and only BPSL is capable of expressing both (though this capability is not demonstrated in the literature [Taibi and Ngo, 2003] [Taibi and Mkadmi, 2006]).

3.2.1.1 Interface dependency

Many of the GoF design patterns promote loose-coupling and extensibility, to allow maintenance and the addition of new functionality to be performed more easily. The intent of numerous patterns can be generalized to ‘Clients shouldn’t need to know about Class X’ or ‘Clients shouldn’t be hard-coded to use a particular subclass of Class X’ where ‘know about’ and ‘hard-coded’ describe the way that the Client refers to the other object. Interface dependency invariants are expressed in Alas using the interface dependency operators: `hasRef`, `depends` and `calls`, which are typically applied to two class operands, with the

meaning that the first is constrained in its relationship to the second. Single dependency clauses may be combined and modified with the standard logical connectives: **AND**, **OR**, **XOR** and **NOT**, with their usual meanings. **AND**, **OR** and **XOR** are binary operators and are placed between individual clauses, while **NOT** is unary and occurs before the clause to which it is applied.

The intent of the Façade pattern is to ‘minimize the communication and dependencies between subsystems’ by introducing a Façade class as a single access point to a number of subsystem classes, so that clients do not have to refer to them directly. The intent of the pattern is captured in the Alas invariant below:

```
NOT Client hasRef SubSystemClass
```

hasRef is a binary operator that states that the first operand has a reference variable whose class is the second operand. The variable must be at class scope, i.e., an attribute in UML and Java terminology. By default it takes two class operators but the first operator may also be a method allowing for more fine-grained statements about where dependencies occur to be made. **hasRef** differs from **hasRefVar** in that it does not define and name a role, it only specifies the existence of some relationship.

By default, the second operand of a **hasRef** clause applies to a class or any of its subclasses (in this example, a Client must not have a reference to a SubSystemClass or any of its subclasses). This can be over-ridden by another clause that states otherwise, as shown in Section 3.2.1.2. This latter case is actually common in design pattern specification: a class should have a reference to another class, but should not have a reference to any subclass of that other class. The reference may be inherited, e.g., if Client above inherited a reference to a SubSystemClass from its superclass AbstractClient, this would violate the invariant above.

While **hasRef** constrains only the variables of a class, the **depends** operator, when applied to two class operands, means that the first does not have an attribute or local variable whose type is the second operand. Finally, the **calls** operator is useful in cases where an explicit reference is not required to make a method call, e.g., when calling static methods or in method call chaining. The first and second operand of the **calls** predicate may be either a class or method. For example, the Caretaker in the Memento pattern has the responsibility of creating a Memento, but should never access or write to its state. This

could be described using the Alas invariant below:

```
Caretaker hasRef Memento AND (NOT Caretaker calls Memento.GetState())
AND (NOT Caretaker calls Memento.SetState())
```

Both `hasRef` and `calls` are not transitive relations, so, for example, even though `Director hasRef Builder` and `Builder hasRef PartA`, `Director hasRef PartA` evaluates to false for a correct implementation of the Builder pattern. Parentheses are included above for readability purposes only, the `NOT` operator has higher precedence than `AND` (and `OR`), so the brackets may be omitted here and the meaning is the same.

3.2.1.2 Implementation dependency

A number of GoF design patterns focus on flexibility in the creation of objects. A summary of their common intent might be that ‘a client holds a reference to an object, but is not hard-coded to a particular implementation (subclass).’ Alas provides a single implementation dependency operator, `isInitializer`, that is used similarly to the interface dependency operators described above, taking two class operands or a method and class operand. It states that the first operand calls the constructor of the second operand explicitly somewhere in its definition. The ability to distinguish between interface and implementation dependency, and positive and negative dependency, is part of the first major contribution of Alas.

The Abstract Factory pattern ‘provide[s] an interface for creating families of... objects without specifying their concrete class.’ Thus, a client should never contain the code: `Maze aMaze = new Maze()` or `BombedMaze bMaze = ...` as the first performs the initialization itself, and the second commits to a particular subclass. Instead, creation of the object is delegated to a factory object: `Maze aMaze = factory.makeMaze()`. This is partially described in the Alas invariant below:

```
Client hasRef Product AND NOT (Client hasRef ConcreteProduct) AND
NOT (Client isInitializer Product)
```

where `ConcreteProduct` inherits from `Product` and there are no intermediate classes between the two roles in the inheritance hierarchy defined in the SD of the Abstract Factory

pattern (not shown here). The first two clauses state that the Client has a reference variable of superclass (`Product`), but not of a subclass of `Product`. The third predicate states that the Client does not initialize any object of class `ConcreteProduct`.

3.2.2 Structural variant specification

There are a number of trade-offs or alternatives to consider when implementing design patterns, which create a number of potential pattern variants, each requiring their own specification. As stated in Chapter 2, named variants, especially those differing in terms of behaviour, are not well supported by the state-of-the-art in DPSLs and variant specification is one of the main contributions of Alas. In Alas, variants may differ in terms of structure, behaviour or both. The structure and behaviour that is common to all variants can be specified only once and shared between each variant. Variant-specific structure is specified inside constraint boxes in the structural specification. The first line in the constraint box defines the variants that the following constraints apply to using the `variant` keyword followed by a comma-separated list of variant names. In Behaviour diagrams, variant-specific control flow is specified using the `var` Combined Fragment that expresses potential choice (see Section 3.3.5). Variant-specific object-state and data-structure behaviour are placed within constraint boxes that are labelled similarly to those included in structural specifications described above. Variant specifications can place additional constraints on other variants or remove the conditions described in the shared or *core* variant pattern specification, to allow for more concise specifications. Structural roles are removed from the core (or other variants) by repeating the role definition exactly in the variant definition, and preceding it with the keyword `removes`. Removing a structural role from a specification also removes all clauses that refer to that role.

A trade-off to consider with respect to the Composite pattern is where to declare child management methods (`add`, `remove` and `GetChild(int)`). If they are defined in the Component superclass, it adds transparency, as all Components can be treated uniformly. Defining them in the Composite subclass, however, offers safety, as a client cannot do something meaningless such as trying to add children to a Leaf. Another variation point is whether each child has a reference back to its parent. Figure 3.4 shows the Alas specification of the safe variant as the generic pattern, with the `unsafe` and `parentLinks` variants specified in separate constraint boxes. An equivalent specification could be obtained by adding the three child management methods to the `Component` class in the diagram, changing the

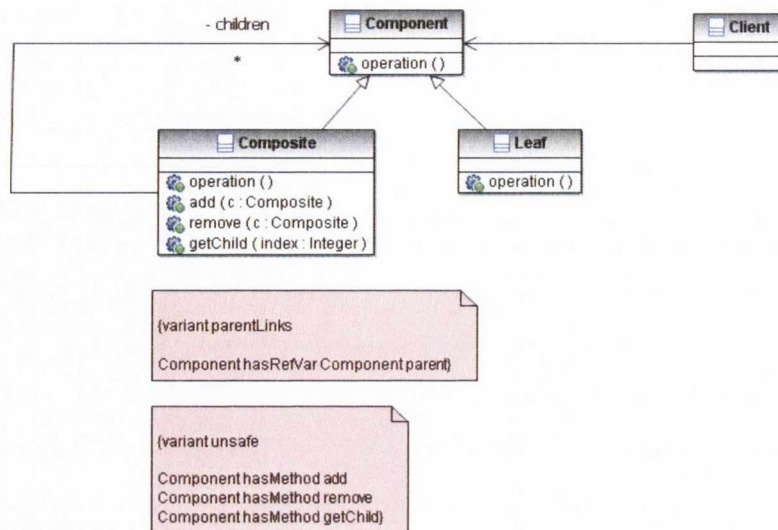


Fig. 3.4: Composite Structure diagram with two structural variant definitions, both of which add static roles to the core specification. This allows for four valid variants of the Composite pattern.

variant name from `unsafe` to `safe` and prefixing each of the `hasMethod` statements with `removes`. For simplicity, the textual descriptions of the child management methods (defining return type etc.), have been omitted. The differences between variants can be inferred during verification. In this case, if no actor is found to fill the role of `add`, `remove` and `GetChild(int)` methods in the `Component` class, but all other roles are correctly filled, the implementation will be identified as the ‘safe’ variant of the Composite pattern.

The `replaces` keyword is a concise way to combine in a single statement additions of new invariants with the removal of existing invariants. It is especially useful for changing some property of a method signature, as the signature specification, defined using the `isMethod` keyword, can be separated into its constituent sub-clauses (`hasReturnType`, `hasParam`). For example, to specify the No Abstract Factory variant of the Abstract Factory or Factory Method patterns, it is necessary to change the return type of the Factory Method from `AbstractFactory` to `ConcreteFactory`. This is specified as:

```

variant No Abstract Factory
removes AbstractFactory
replaces ConcreteFactory.factoryMethod hasReturnType AbstractFactory
with ConcreteFactory.factoryMethod hasReturnType ConcreteFactory
  
```

Note that in a sequential interpretation of the Alas specification above, `factoryMethod` could not be referred to in line 3 of the specification above, as the `AbstractFactory` structural role is removed in line 2 and `factoryMethod` is removed by extension, as it refers to `AbstractFactory`. We define a more flexible interpretation that allows structural role definitions to be ‘brought back’ into the specification after their removal is triggered by having the part of their definition that has been removed replaced by a still-present role. In this case, the new definition of `factoryMethod` no longer relies upon `AbstractFactory`, and thus, does not need to be removed⁶. The specification of the No Abstract Factory variant also illustrates qualified names of structural roles. Both `AbstractFactory` and `ConcreteFactory` have a `factoryMethod` role (the method is over-ridden in the subclass), so the specific method referred to must be identified by prefixing the method with the class role defining it and a dot.

Sometimes a variant of a pattern may have more differences from the structure of the core variant (subsequently referred to simply as core) than similarities, but is still considered a valid variant of the pattern. To address this issue, Alas provides a scoping operator (`::`) to allow the specification of variants in separate SDs to be linked back to their core. The scoping operator takes two operands, the first operand is the pattern name given in the pattern label, the second operand is the variant name. See Figure 3.5 for an example of the scoping operator in the context of the Abstract Factory pattern. The constraint box is an example of a *variant label*, which uses the scoping operator to associate the variant with its core variant, in this case, the Abstract Factory pattern. Constraints from the core are not automatically applied to the variant specified in the separate SD, as the names used in the two SDs may not be equal. To include invariants from the core in the variant specification, the substitution operator (`->`) must be used to map names in the two SDs. The substitution operator takes two operands, the first of which is a name from the core SD and the second of which is a name from the variant SD that is being substituted for the first in the core. Invariants from the core are only included in the variant specification when *every* role in a clause or sub-clause has a mapping defined for it.

⁶A strictly sequential interpretation of Alas specifications with the `removes` and `replaces` keywords would require variants to be ordered with respect to each other, to identify the roles that are present or removed. This interpretation complicates the semantics of the language and also makes specifications less readable, as all variants must be read in order to understand any variant in isolation.

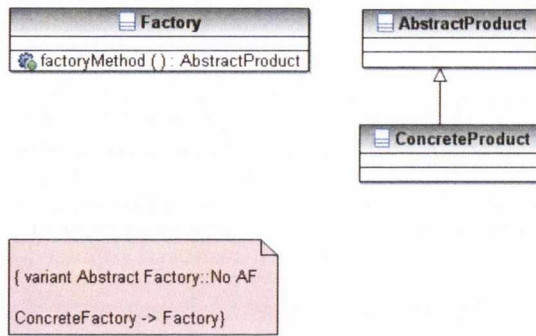


Fig. 3.5: A variant specified in a separate SD, illustrating the use of the scoping (`::`) and substitution (`->`) operators

A second use for the scoping operator is to define sub-variants: variants that further relax or constrain another variant and are only meaningful or relevant in the context of the this other variant. In this case, the first operand to the scoping operator is a variant name, or comma-separated list of variant names, the second operand is the sub-variant name. The variants that are affected by the constraints in another variant could be inferred, but the scoping mechanism allows for more fine-grained control of the scope of a variants constraints. For example, a sub-variant could be applied to only a single variant, while it could validly be combined with two or more variants. An example of a sub-variant is provided in the section on behavioural variants (section 3.3.5).

3.2.2.1 Semantics

A pattern specification includes a set of structural roles defined in a SD, a set of invariants defined in associated textual clauses and constraints from one or more BDs. We refer to all these definitions and invariants generically as *constraints* in the following discussion. A pattern specification is a conjunction of each of these individual constraints from each of the included diagrams. The `removes` keyword can be used to remove both structural role definitions and textual clauses such as dependency invariants from a specification. In both cases, the entire constraint must be repeated in the removes clause. This avoids ambiguity but makes it verbose to remove long constraints such as cardinality invariants. Removal of a structural role also removes all clauses that refer to the role.

In a conjunction or disjunction of multiple clauses, a single sub-clause can be removed, and the effect of removing the sub-clause is to replace the clause in the specification with

the value `true`. For example, a variant that removes the second sub-clause of:

```
NOT (A isInitializer B) AND NOT (B isInitializer C)
```

results in the clause:

```
NOT (A isInitializer B) AND true
⇔ NOT (A isInitializer B)
```

`NOT` is considered as part of a sub-clause, while `AND`, `OR` and `XOR` are the connectives between clauses. When removing textual clauses (or sub-clauses) that include `NOT`, it is necessary to remove the `NOT` in the clause included after the `removes` keyword, otherwise it is possible to create a `NOT (true)` clause, which is unsatisfiable. A variant that removes all structural definitions (and, by extension, all textual clauses) from the core variant is an invalid definition.

Defining a set `cons` as the set of all constraints, with a size `n`, a pattern specification (the core variant) can be defined as:

$$\text{core} = \text{cons}_1 \wedge \text{cons}_2 \wedge \dots \wedge \text{cons}_n$$

A variant specification containing `m` additional constraints (i.e., no `removes` constraints) is defined as:

$$\text{variant} = \text{cons}_1 \wedge \text{cons}_2 \wedge \dots \wedge \text{cons}_m$$

The combination of a core variant with the above variant results in the overall pattern specification:

$$\text{pattern} = \text{core} \vee (\text{core} \wedge \text{variant})$$

A variant specified in a separate SD and linked using the scoping operator does not by default include constraints from the core variant. Defining the set `cores` as the set of constraints taken from the core for which the other variant provides complete substitutions, the specification of the overall pattern can be represented as:

$$\text{pattern} = \text{core} \vee (\text{core}_s \wedge \text{variant})$$

Returning to variants specified in the same SD as the core, we define variant_r as the set of removes clauses and variant_a as the set of additional constraints, the pattern specification becomes:

$$\text{pattern} = \text{core} \vee ((\text{core} \setminus \text{variant}_r) \wedge \text{variant}_a)$$

where the hiding operator, \setminus , removes both structural definitions and any clause or sub-clause that refers to them. Multiple variants are not mutually-exclusive by definition. For example, the inclusion of two variants leads to the pattern specification as defined below (angled braces $\langle \rangle$ have been used to denote the outermost parentheses of each sub-clause, to improve readability):

$$\begin{aligned} \text{pattern} = & \text{core} \vee \langle (\text{core} \setminus \text{variant}_r^1) \wedge \text{variant}_a^1 \rangle \vee \\ & \langle (\text{core} \setminus \text{variant}_r^2) \wedge \text{variant}_a^2 \rangle \vee \\ & \langle \text{core} \setminus (\text{variant}_r^1 \wedge \text{variant}_r^2) \wedge ((\text{variant}_a^1 \setminus \text{variant}_r^2) \\ & \wedge (\text{variant}_a^2 \setminus \text{variant}_r^1)) \rangle \end{aligned}$$

The four clauses combined in a conjunction represent the core variant, the first variant, the second variant and the combination of the first and second variant respectively. Note on the last two lines of the definition that the set of removes clauses of each variant applies not just to the core variant but to all other variants with which it may be combined also, i.e., variants may remove definitions and clauses from other variants. Similarly to a single variant, a combination of multiple variants that removes all structural definitions from the core is also invalid.

3.2.3 Structural cardinality invariants

Cardinality invariants place constraints on the number of elements in a set, usually by comparing the number of elements in two or more sets to each other. As described in Chapter 2, cardinality invariants are well supported in the literature [Le Guennec et al., 2000, Mak et al., 2004, Eden, 2008], though the languages that address them tend to focus on structure and lack support for behavioural specification. Structural cardinality invariants (cardinality invariants that are described completely in the structural specification) can be expressed in Alas using sets and quantification. Up until now, each role has been an *individual* role, i.e., a role that is played by one actor only. In this section we introduce *set roles*, roles that may be filled by one or more actors. Sets can be created using two approaches: explicitly using

Alas built-in operators or implicitly in a quantified expression. Sets of classes are always created implicitly while sets of methods are always created explicitly. The `isMethodSet` operator is identical to the `isMethod` operator except that it defines a set role rather than a single role. Similarly, `hasMethodSet` is identical to `hasMethod`, except that it associates a class to a method set and not a single method role. An example of a definition of a method set, which we will use later to specify a cardinality invariant in the context of the Visitor pattern is shown below:

```
VisitMethodSet isMethodSet
    public NOT static NOT abstract, visit*, ConcreteElement ce, *
    Visitor hasMethodSet VisitMethodSet
```

The above specification excerpt states that the `VisitMethodSet` is a set of methods that are all public and not abstract or static. `isMethod` clauses that do not include `static` or `abstract`, mean that the role may be filled by an actor that is either static or non-static, abstract or non-abstract. The specification also states that each member of the set must have a name that begins with 'visit', have a parameter that is a member of the `ConcreteElementSet`, and may have any return type. Role names specified so far have simply been placeholders for the actual actor names in a candidate implementation and place no constraints on the actual name of the actor. With the `isMethod` operator, it is possible to constrain the names of actors in a conforming implementation, using a limited form of regular expressions.

Alas supports the universal, existential and unique quantifiers of first-order logic, represented by the keywords `FORALL`, `EXISTS` and `EXISTSONE`, with their usual meanings. The general form of quantified first-order logic statements in Alas is:

```
Q x in set [where constraint] @ predicate
```

where `Q` is a quantifier, `x` is a bound variable drawn from `set` and `predicate` is a valid Alas clause or another quantified statement. Constraints are optional, as indicated by the brackets and are used to exclude elements of the set that do not meet some criteria from the evaluation of the invariant. The `in` keyword is used to designate that a variable is drawn from a particular set (as in the Python programming language). The `@` symbol substitutes

for the bullet symbol from Z notation (\bullet) [Woodcock and Davies, 1996], which can be read as ‘it holds that’ or ‘such that’, and is used to separate quantification and predicates.

Sets of classes are defined by designating a previously-defined single class role as the set in some quantified statement. This overrides the role’s initial definition, and makes the role a set role. The Mediator pattern ‘promotes loose coupling by keeping objects from referring to each other explicitly’ by placing a Mediator object between a set of objects that co-operate. The Mediator pattern differs from the Façade pattern in that it defines a multi-directional protocol, while the Façade handles unidirectional communication. The Alas invariant below states that no two (non-identical) members of a set of classes (`ConcreteColleague`) that inherit from a common parent (`Colleague`, whose definition is not shown) refer to each other explicitly:

```
NOT EXISTS cA, cB in ConcreteColleague
  where NOT (cA == cB) @ cA hasRef cB
```

where `cA` and `cB` are arbitrary names. Note that `cA` and `cB` are quantified variables representing classes, that may be substituted here for class roles. Quantified variables may take the place of roles in any Alas clause, provided they are of the type that the operator that is applied to them expects. The `ConcreteColleague` role is over-ridden here by its use as a set in a quantified statement and is made a set role. Finally, this example illustrates the use of the `==` operator that may compare two classes or methods for equality (a class or method is equal to itself only).

The Visitor pattern will be used to demonstrate universal quantification in Alas structural specifications. A Visitor ‘represent[s] an operation to be performed on the elements of an object structure’. Each Visitor class should be capable of visiting every element type (e.g., a code generating visitor must be able to generate code for every type of expression). Using the definition of `VisitMethodSet` above, and the universal quantifier `FORALL` (and omitting some definitions for brevity), this invariant can be specified as:

```
FORALL ce in ConcreteElement, v in ConcreteVisitor @
  EXISTS visitMethod in v.VisitMethodSet @
    visitMethod hasParameter ce
```

where `hasParameter` is a convenience method defined in terms of `isMethod`, where only the parameters in the method signature are constrained. Note the use of the dot notation (`v.VisitMethodSet`) to indicate the set of methods belonging to a particular class. In natural language this states that every class that inherits from `Visitor` contains at least one method that accepts each element of the `ConcreteElementSet` as a parameter. This example is an instance of the *universality and existence* invariant type. Invariants of the *uniqueness* invariant type are expressible directly using the `EXISTSONE` quantifier.

Finally, cardinality invariants can also place constraints on the multiplicity of relationships between classes. A `Decorator` object should be associated with, or ‘decorate’, only one `Component` object, so the `Decorator` class should define only one reference variable attribute of type `Component`. `Alas` provides the `refVars` keyword to represent the set of all reference variable attributes held by a class role. Similarly, the `methods` keyword represents the set of all methods defined in a class role⁷. The class of a reference variable can be accessed using the `class` keyword. `refVars`, `methods` and `class` keywords are all associated with a role using the dot notation: `<roleName>.<keyword>`. Inheritance relationships can be specified textually using the `inherits` keyword, which states that the first class operand is required to inherit from the second class operand. The invariant in the context of the `Decorator` pattern may be specified as:

```
EXISTSONE var in Decorator.refVars @
    var inherits Component OR var.class == Component
```

3.3 Behavioural specification in `Alas`

In this section, we describe behavioural specification in `Alas`. `Alas` behavioural specifications are capable of expressing behavioural cardinality invariants, control-flow, object-state, and data-structure invariants. In particular, `Alas` is capable of expressing object-state invariants regarding deep copying and data-structure invariants regarding data-structure shape. `Alas` BDs also provide a concise syntax for expressing design pattern variants that differ in terms of their behaviour. Finally, generic control flow can be specified using BDs (see Appendix E).

BDs describe a sequence of *events* that are initiated or performed by roles. BDs are

⁷Both `refVars` and `methods` refer to the set of all structural entities occurring in the class *actor* bound to the class role, not just the structural entities specified in the role.

based on UML 2 Sequence diagrams and OCL but introduce new constructs and differing semantics for some existing constructs. This section introduces the syntax required to specify each of the invariant categories in turn, drawing examples from the GoF catalogue. An overview of the semantics of each concept is given in a short section after that concept is introduced. Detailed semantics and more subtle issues that are less critical to basic understanding are covered in Appendix A.

3.3.1 Control-flow invariants

Delegation of responsibilities between objects is a central concern in design patterns and the ability to express delegation is thus a basic requirement of a DPSL. Numerous GoF design patterns describe two or more events that should always occur in sequence, while a number of patterns also include conditional and iterative behaviour. As described in Chapter 2, a number of DPSLs are capable of expressing all control-flow invariants [Lano et al., 1996, France et al., 2004]. This section describes how invariants of the types `Sequence`, `Selection`, `Iteration` and `Method call` are specified in Alas.

Lifelines are the foundation of BDs and represent interacting object roles. Lifelines are indicated by a labelled box, called its `head`, with a vertical line below it, along which events are specified. The labels within lifeline heads take two operands, which are typically an object name and a class name, separated by a colon (though some exceptions are discussed later). Objects interact by exchanging synchronous method calls. Every call event has a corresponding return event. A call event is indicated by an arrowed line between caller and callee with the callee being the target of the arrow. The line is labelled with the method role name (or other method identifier) and may optionally include arguments. Arguments specified must conform to the type specified by the parameters of the method. Every BD begins with a *found* call, a call event with no calling object role. This indicates that the method receiving the found call does not need to be called anywhere in the implementation, i.e., no role needs to be bound to the source of the found call.

The object name given to a lifeline receiving a call in a BD may match a reference role belonging to the class of the calling role. This indicates that the object to which that reference is currently pointing is the intended receiver of the call (issues regarding object identity are discussed later). For example, Figure 3.3 shows the Client calling the Façade object using the reference role (`facade`) defined above in Section 3.1.4. Alternatively, the object name may match no previously defined role. Subsequently, we will use the term

object role sometimes as a catch-all term for all roles representing objects and sometimes in the more specific sense of roles that do not match previously defined reference variable roles. In each case, it should be clear from the context.

3.3.1.1 Sequencing

A DPSL should be capable of specifying two related but separate issues: that some set of events should occur in a particular *order* and that they should *always* occur together. The Command pattern, for example, requires that the execute method of the ConcreteCommand (Command subclass role) should always call the action method of the Receiver. The first issue, ordering, is indicated in Alas by the vertical position of events in a BD: events occur after all events above them and before all events below them. As there is no concurrency in Alas, BDs define a total order of events.

We will refer to the second issue, that the events always happen, as *universal* behaviour. The default semantics of UML Sequence diagrams, however, is that a diagram defines only a trace that should be possible, while allowing the existence of other traces that are different: ‘there are other legal and possible traces that are not contained within the described interactions’[OMG, 2010b, p.473]. Thus, a Sequence diagram defines by default *existential* behaviour. An assert operator can be used to state that ‘all other continuations [besides the one specified] result in an invalid trace’ *ibid.* p.486. This assert operator approximates the universal behaviour specification that we require but is not formally defined elsewhere. We have chosen the universal behaviour semantics for Alas diagrams. Though this limits expressiveness, as context-specific behaviour is not expressible, it does not affect the specification of conditional behaviour.

Existential behaviour specification is useful in the early stages of system design, when the design is being ‘fleshed-out’ and is typically replaced by universal specification when the requirements of the system are better understood [Damm and Harel, 1998]. With regard to design patterns, control flows are described that must be satisfied in all contexts, so the universal meaning is desired. The Alas meaning is similar to a Sequence diagram contained entirely within an assert operator or a Live Sequence Chart (LSC) universal chart [Damm and Harel, 1998].

3.3.1.2 Selection

A number of design patterns involve conditional behaviour. The Flyweight pattern describes the initialization of an object on the condition that the object has not yet been initialized (lazy initialization), while in the Proxy pattern, the Proxy object delegates only if the intended target of the client (the RealSubject object) has been initialized. These examples involve conditional behaviour, but a number of alternative or mutually-exclusive conditional behaviours are required in other patterns. A Handler in the CoR pattern, for example, should either handle a request or forward it, but not both or neither. The Parameterized Factory variant of the Abstract Factory and Factory Method patterns describes the use of a parameter to choose between the initialization and return of multiple alternative types of object.

The `opt` and `alt` operators, taken from UML 2, are used in Alas to specify single, and multiple mutually exclusive conditional behaviour respectively. *Composite* operators describe events that can themselves contain sequences of events. Composite operators accept one or more operands, each of which is a sequence of events. Similarly to UML, the `opt` operator is a composite operator that contains one sequence of events and states that ‘either the sole operand happens or nothing happens’ [OMG, 2010b] p.484, while the `alt` operator takes composes multiple sequences of events and states that ‘at most one of the operands will be chosen’. Lund and Stølen [Lund and Stølen, 2006] show that these operators are ambiguous. It is not clear whether mandatory or potential choice is intended, i.e., whether two alternative behaviours must be possible in the implementation or whether the implementor has the option to implement one or the other behaviour. We choose the mandatory choice semantics, as the intention is that some conditional branching occurs in the implementation, as shown in the informal examples in the GoF catalogue. As we will see later, the `var` operator, introduced in Alas for pattern variant specification, is an operator that expresses the potential choice semantics.

All composite events are mutually recursively composable, meaning that any composite event may contain an instance of any other composite event including an event of its own type, e.g., an `opt` may contain an `opt`. Every operand of a composite event has an optional guard, specifying the condition under which the operand is chosen. Guards can be specified completely, though anticipating the guard condition for all valid implementations in all contexts is typically not practical or possible. For this reason, Alas provides a means to specify *generic guards* (see Appendix E), where only a sub-clause of the complete guard

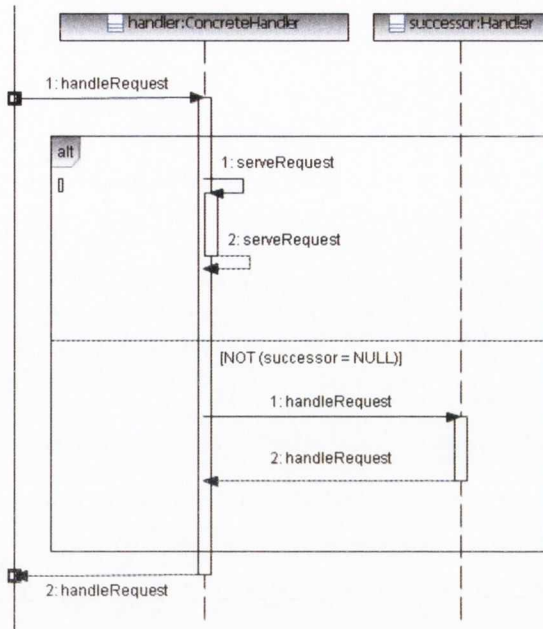


Fig. 3.6: Behavioural specification of the CoR pattern involving a two-operand `alt` with an operand guarded by a basic state invariant on an object role. A single path in a valid implementation may not contain the behaviour of both operands. The call event involving `successor's handleRequest` only occurs if the successor has been initialized.

condition is specified.

Object roles have their basic state (uninitialized, initialized, garbage collected) specified by comparing the role to the `NULL` role (`NULL` is a keyword that is equal to any object role that is either uninitialized or garbage collected), and a constraint on the basic state of an object role may be placed in the guard of an `Alt` control-flow operator. Figure 3.6 gives an example usage of the `alt` operator and a complete guard imposing a basic state invariant on an object role. The specification in the figure states that a `ConcreteHandler` can either handle a request or forward to its successor, on the condition that its successor is initialized.

3.3.1.3 Iteration

Both the `Composite` and the `Observer` pattern involve a method that iterates over an unbounded collection of objects and calls the same method on each element of the collection (collections are dealt with in more detail in Section 3.3.2.4). To express iterative behaviour, `Alas` provides the `loop` operator (taken from UML 2), which, similarly to `opt`, takes a single operand containing a sequence of events and has a guard. To express iteration and interaction with an unbounded collection, we require two further mechanisms: a way to

relate the number of iterations of a loop with the size of a collection and a lifeline that may represent different objects at different times.

While a number of non-standard idioms have been proposed to express this behaviour [France et al., 2006][Larman, 2005], it is not currently supported in the UML standard. In UML Sequence diagrams, a lifeline represents one and only one object: ‘While Parts and StructuralFeatures may have multiplicity greater than 1, Lifelines represent only one interacting entity...’. An object role name in a UML diagram may be replaced by a collection name followed by square braces. A *selector* is an integer value placed within the square braces to act as an index into the collection, selecting an element at a particular position. Also, the selector may be omitted allowing an arbitrary object to be bound, but these bindings are still to a single object and immutable.

In Alas, the selector is not constrained to being an integer only, and may be a string. A quantified statement, quantifying over some collection role, may be placed in the loop guard of the loop operator (also in UML). When the quantified variable’s name matches the selector string, the desired meaning is obtained: that the loop iterates once for each element of the collection and that each element of the collection is involved in the interaction on one iteration. An example from the Observer pattern as shown in Figure 3.7 (all valid uses of selectors and their meanings are given in a table in Appendix A Section A.5). The order that each element of the collection is involved in the interaction is not constrained in this case, e.g., a loop interaction may begin with the first element and progress forwards or begin with the last element and progress backwards.

3.3.1.4 Non-pattern structure and behaviour

A pattern actor will rarely *only* perform a pattern role and is typically involved in a number of other interactions that are not part of the specified pattern behaviour. Thus, requiring that a method in a candidate implementation does not call any other method except the ones explicitly specified is very restrictive, and may lead to false negatives during verification. However, a (specified) call event occurring twice instead of once in a candidate implementation may have a significant affect on behaviour, as many methods are not idempotent. We use the term *conformance relation* to denote the allowable non-pattern structure and interleaved non-pattern behaviour that an actor may contain/perform and still satisfy a role. The default conformance relation of Alas, which we call *Non-role refinement*, states: all interaction between object roles in a pattern specification should be explicitly specified,

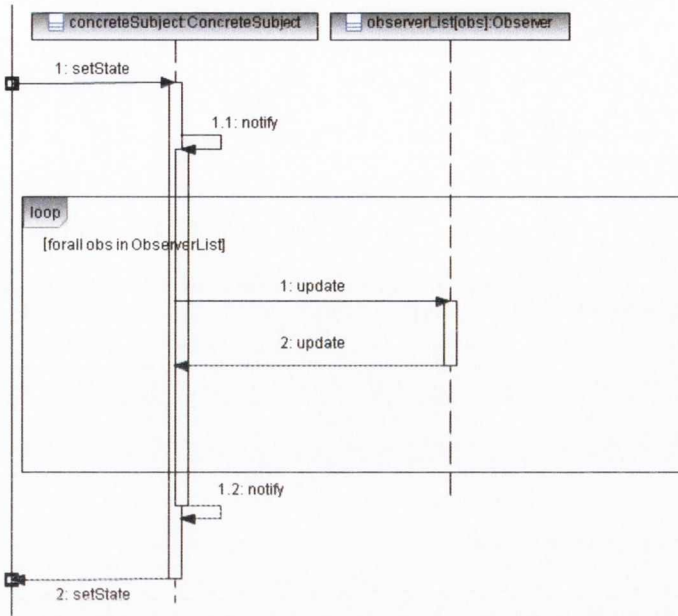


Fig. 3.7: Specifying iteration over and interaction with an unbounded collection by matching the name of the quantified variable in the loop guard (`obs`) and the selector string in the lifeline.

but a role may have other interleaved interactions with unspecified objects. A structural role may have any number of non-pattern structural relations provided they do not violate some specified invariant. This is a special case of the refinement conformance relation defined in Chapter 2

The conformance relation only relates to method calls, (i.e., interactions between roles) and does not affect other control branchings such as loops and conditionals. Unspecified interleaved loops and conditionals are allowed, as are local method calls. One side-effect of this relation is that it means that a conforming implementation may contain self calls that are specified and also self calls that are unspecified. Non-role refinement is similar to the conformance relation supported by LSC [Damm and Harel, 1998]: “If the chart is a universal chart, none of the events in [the set of events visible to the chart] will be allowed to occur in between the events appearing in the chart itself... Events and variables not visible in a chart are not constrained by the chart.” However, the scope from which events are drawn is slightly different: In LSC, a set of visible events is defined for each diagram while in Alas (and UML) the set of methods to consider is defined in the Structure (or Class) diagram to which the Behaviour (or Sequence) diagram corresponds.

However, a Factory Method specification, for example, may include only a call from

the Factory to the Product constructor while an implementation may also call a mutator method (also included in the structural specification) on the Product before returning in order to initialize some state, violating the non-role refinement relation. For this reason, Alas includes a *Refinement* relation that allows arbitrary interleaving of behaviour with specified pattern behaviour. The default relation may be overridden by including a constraint box, not attached to any lifeline, stating `ConformanceRelation = Refinement`. Alas also provides the `Equivalence` conformance relation, that forbids a method actor to have any other method calls except those included in the specification. The conformance relation is set for the entire diagram.

3.3.2 Object-state invariants

In Alas, invariants may be placed on the state of objects or the relationship between the states of interacting objects at a particular point in the execution of a program using Alas BDs and attached constraints. Object-state invariants can be used to specify (1) whether an object has been initialized, (2) whether two objects are copies or aliases of one another and (3) the state or contents of data structures and collections. Each of these types of object-state invariants will be discussed below, using examples. In particular, we focus on the specification of deep copying invariants as one of the major contributions of Alas.

The UML Standard describes a concrete syntax for constraints that involves a rounded rectangle that spans potentially numerous lifelines that are required to ‘synchronize’ on this constraint. Rational Software Architect does not provide this syntax, but instead provides constraint boxes that do not impinge upon lifelines but may be attached to them using a dotted line. In Alas, as there is no concurrency, it is sufficient to connect the constraint to a single lifeline, though the constraint may refer to the state of multiple objects (i.e., there is no synchronization requirement). We use the term *anchoring* to refer to the connection of a constraint to a particular point in a lifeline. We use the term *constraint* for the text within a constraint box and *invariant* for the combination of constraint and anchoring position.

Constraints may be anchored to one of two positions in an Alas BD: at a method return they specify pre- and post-conditions. The second position to which a constraint may be anchored is at any other control branch, such as the end of a loop or conditional. This second positioning is treated like a post-condition: it states that an invariant should hold at that point in the interaction. Alas does not provide pre-conditions as they are were found to be less useful than post-conditions on methods and control-flows, and complicate the

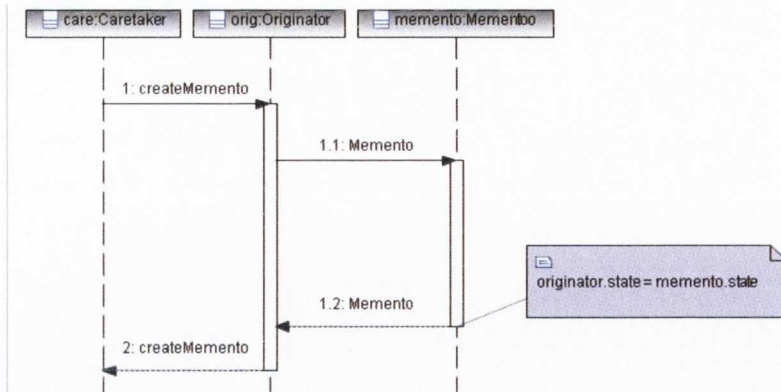


Fig. 3.8: A potential (non-Alas compliant) Memento pattern specification where the objects pointed to by reference variable roles are strictly equal when the Memento’s constructor returns.

semantics of the language.

3.3.2.1 Deep copying behaviour

In the Memento pattern, a memento object ‘capture[s] and externalize[s] an object’s internal state so that the object can be restored to this state later.’ The state of the Memento (or some subset of it) is a function of the state of the Originator at two points in the execution: when the Memento is created and when the Originator is restored to the state held by the Memento later. Thus, the Memento is a copy of (a subset of) the state of the Originator, encapsulated within an object of a different type. A potential specification (non-Alas compliant) of a particular instance of the Memento pattern is given in Figure 3.8, where *state* is a reference variable role. There are two problems with this diagram in the context of design pattern specification. Firstly, in most implementations of the Memento (and Observer) pattern we encountered during the creation of the benchmark used in Chapter 5 of this thesis, the state of the two objects is not strictly equal (same runtime type and same value), as the value stored in the Memento, for example, is some function of the Originator’s state (e.g., a Java integer array in one object converted to a string and prepended with an identifying tag in another object).

Secondly, the relevant state to be copied (subsequently, *copy state*) differs between pattern implementations, while each subclass may add extra state that should be copied. The copy state is some combination of primitive and reference variables (subsequently *variables*) and objects, that may themselves contain other variables and objects, and so on

recursively. A generic method for specifying this bundle of state is necessary to describe the object-state invariant imposed by the Memento pattern.

To deal with the first issue, Alas includes a `relates` operator (inspired by the `reflects` keyword in Contracts [Helm et al., 1990]) that takes two pre-defined reference variable or object roles as operands and specifies that one operand is some function of the state of the other.

With regard to the second issue, Alas provides the keywords `CopyState` and `copystate`, that allow the definition of a special role that is defined identically to a reference variable role except that its role type and role name must be `CopyState` and `copystate` respectively. The role may be bound to a set of variable and reference variable actors, defined in the containing class, and also to further state recursively reachable from them. An example of the usage of the role is given in Figure 3.9, in the context of the Memento pattern.

The copy state of two or more roles may be compared in a BD using the operators `isCopy` and `isRCopy`. `isCopy` takes two copy state operands and states that the two are strictly equal at the point in the execution where the constraint is anchored (constraints are always anchored at control-flow events, which can be matched between specification and implementation). `isRCopy` is equivalent to `relates`, except that it takes two copy state operands. An example of the usage of `isCopy` is given in Figure 3.10. These mechanisms collectively address the deep copying invariant type in our DPSL classification framework of Chapter 2. Introducing an `isCopy` operator to OCL has been proposed elsewhere, but with shallow copy semantics [Marković and Baar, 2005]. The ability to define the copy state of class roles in Alas gives it the ability to describe the Prototype and Memento patterns, two of the most poorly supported GoF patterns in the DPSL literature.

The anchoring of the constraint in Figure 3.10 is equivalent to a post-condition in OCL. The specification states that the Originator must call the Memento object that was passed to it as a parameter to `setStateToMemento` and ensure that the copy state of itself and the memento are equal at the end of the method execution. Note that this demonstrates a parameter role, where the parameter role name matches an object role name.

3.3.2.2 Copy state definition

Alas provides two categories of approaches to defining the copy state: user-defined and relation-defined. The copy state to be included during verification can be defined by the user by selecting reference and primitive variables to be included from the set of classes

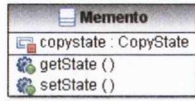


Fig. 3.9: Illustration of the usage of the CopyState reference variable role in a structural specification. CopyState may be bound to a different set of primitive and user-defined reference variables in each implementation of the Memento

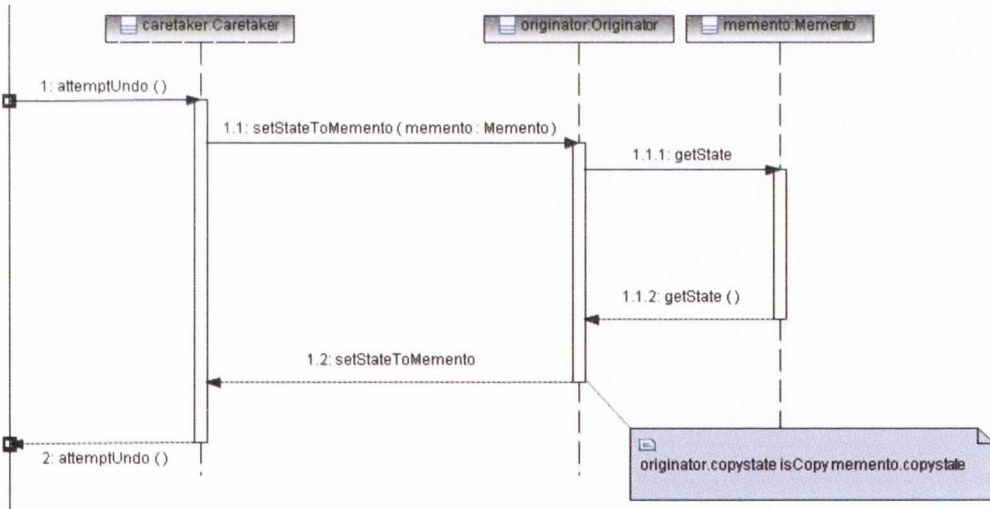


Fig. 3.10: Illustration of the isCopy operator, relating the copystate of two objects at a particular point in the execution. Note also the matching parameter and lifeline object role name. The diagram states that the Originator object calls the Memento passed to it as a parameter.

reachable from the class actor. This approach is only available in a forward-engineering use case, where the original developer can capture their intent precisely at development time by binding actors in the implementation to the copy state role manually. To enable a reverse engineering use case, and to provide a generic specification that can be used in multiple contexts, Alas also provides copy state definitions based on the relation between composing object and composed copy state that is both formal and can be automatically verified.

The distinction between association, aggregation and composition relations can help to identify the copy state. The issue may be phrased as: should we follow this object reference and include all the objects reachable transitively from it in the `copystate`? Association represents the ability to send a message from one object to another. Aggregation restricts this to message sending within a whole/part hierarchy, while composition adds the further requirement that the lifetime of the part ends with the destruction of the whole or aggregating object. The state that is in a composition or aggregation relation to the composing object might be expected to be copied, but the state of all objects with which the object is associated is neither suitable nor practical for inclusion in a copy.

The UML Reference Manual [Rumbaugh et al., 1999] defines the composition relation relative to an aggregation relation as adding the constraint that ‘an object may be part of only one composite and that composite object has responsibility for the disposition of all its parts - that is, for their creation and destruction’. Ambler [2005] states that whole and part should have ‘coincident lifetimes’. The key concepts in defining a composition relation are ownership and lifetime. An ownership relation exists between whole and part objects if the part is not included in any other whole, and a strict ownership relation exists if the part *cannot* be included in any other whole. A composition relation also requires that the lifetime of the whole and part, defined by their creation and destruction time, are related.

Alas defines the `comp` (*composition*) copy state as all the states that satisfy a strict ownership relation with the composing object. A close relation between the lifetimes of whole and part is implied by a strict ownership relation, as the whole must initialize the parts, as no other object is capable of referring to it, and the parts should be destroyed (or garbage collected) when the whole is destroyed, also because no other object has a reference to it. In a class actor definition, strict ownership is guaranteed by initializing the copy state somewhere in the class definition and not providing any accessor method that returns a reference to any object of the part. An ownership relation, by contrast, is satisfied by a *program* where the class initializes the copy state and no accessor method is called on

```

class A {
    B b;
    D d;
}

class B {
    C c;

    C getC(){
        return c;
    }
}

class C {}

class D {}

```

Fig. 3.11: Class A satisfies the strict ownership relation with respect to class B, but class B does not satisfy the strict ownership relation with respect to class C

any object of the part, though an accessor method may be available.

The definition of `comp` is recursive: each recursively composed class must satisfy the strict ownership relation between it and its own state variables. For example, in Figure 3.11, class A owns the object referred to by `b` and `d` but also `c`: even though B provides an accessor method for its variable `c`, no client is aware that an instance of class A has a reference to an instance of B, so has no means to gain access to the recursively composed C object. However, the relation between B and C does not model a strict ownership relation, so C is not included in a `comp`-based copy state definition. The relation between B and C is only association.

The candidates for inclusion in the copy state of any object are all the objects transitively reachable via the reference variables defined in the object's class. These objects can be imagined to form a graph, where the nodes are objects and the edges are reference variables. Such a graph may include cycles. A relation-defined copy state constrains the graph of objects included in the copy state to those objects that satisfy the composition relationship with the object that *directly* composes them, i.e., whole and part objects are opposite ends of a single edge. Figure 3.12 illustrates the graph formed by the example in Figure 3.11. Black nodes and edges are included based on the `comp` definition of copy state, while red nodes and edges are reachable but excluded from the copy state.

As accessor and mutator methods are ubiquitous in object-oriented programming, developers may choose to include not just the state that is composed by an object in its copy state, but also the state that it merely aggregates. For example, a `Tyre` object may be

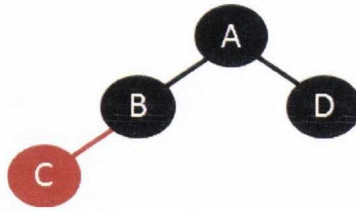


Fig. 3.12: Graph of all reachable state and the included copy state for the classes defined in Figure 3.11 using the `comp` copy state definition

initialized by a `TyreManufacturer` and added to a `Car` after its construction, but when copying a `Car` it is also necessary to copy its aggregated `Tyre` objects. Alas provides two other relation-defined copy state definitions that relax the constraints imposed by `comp`. `iComp` (*initialization-only*) copy state requires only that a whole initializes its parts, and does not require ownership or coincident destruction of whole and part. Secondly, Alas provides the `dComp` (*destruction-only*) copy state definition, that requires that the part is not shared in the heap. This allows clients to initialize a part and pass it to the constructor or some mutator method of the whole. Note, this definition forbids sharing of a part between any two objects in the heap, not just objects of the same class or kind. For example, not only can an `Engine` not be shared between two `Car` objects, it also cannot be shared between a `Car` and a `ScrapYard`, as this may well break some key invariant of the `Car` or `ScrapYard` class. All three of the relation-defined copy states include all primitive variables in their graph of relevant state.

3.3.2.3 Object identity

The role of a Factory Method is to return a newly-created instance of a Product class. Thus, a key invariant of the Factory Method pattern is that a new object is returned, i.e., the object created by the call to the Product constructor in the Factory Method is the same object that is returned by the Factory Method. A related creational pattern is the Prototype pattern, where new objects are created by copying a prototypical instance. One invariant of the Prototype pattern is that the object returned by the Prototype's `clone()` method is *not* the same object as the prototype, but should have identical values for some subset of its state, similarly to the Memento example above. Thus, to specify the Factory

Method and Prototype patterns precisely, it is necessary to be able to express the distinct concepts of object identity and value equality respectively.

The OCL Standard [OMG, 2010a] defines operators informally using natural language. The definition of the equality operator, for example, is: ‘The equality of values of the same type can be checked with the operation $=_t$ ’ *ibid.* p195, where a value ‘can be either an object, which can change its state in time, or a data type’ *ibid.* p.98. The interpretation of this depends on the meaning of the word ‘equality’, which is not defined. Object identity is discussed briefly in the OCL Standard, Appendix A, Section 1.2.1: ‘Objects are referred to by unique object identifiers’ [OMG, 2006]. The OCL set `oid(c)` is also defined as the set of object identifiers for a class. However, this set is not used in the definition of any of the relevant OCL operators. The implementation of Dresden-OCL’s [Demuth and Wilke, 2009] equality operator calls the Java `equals()` method. The implementation of the `equals()` method is not constrained in Java and could provide either object identity or value equality semantics. In this case, using the equality operator in OCL has different meanings in different contexts, and the meaning can not be anticipated when the specification is created.

The UML Standard also makes little reference to object identity. A `DataType` is described as being ‘similar to a Class. It differs from a Class in that instances of a `DataType` are identified only by their value.’ However, the meta-class `Class` has no attributes or associations that could be used to store its identity, and both `Class` and `DataType` occur at the same level of the UML meta-inheritance hierarchy, inheriting directly from `Classifier`, and nothing else. Thus, it is not clear how the unique identify of objects is represented in UML.

Object identity and value equality are distinguished explicitly in Alas using the `isAlias` and `isCopy` binary operators respectively. `isAlias`, similarly to `isCopy` described above, takes two reference variable or object roles. It evaluates to true where the two roles are bound to the same object actor. Both operators are defined precisely in terms of object identifiers and values below in Section 3.3.2.3. Note that $x \text{ isCopy } y \Rightarrow \text{NOT } (x \text{ isAlias } y)$ thus `isCopy` is a mechanism for specifying *deep* copies (the values of corresponding reference variables in both operands are not identical objects) and `isAlias` is a mechanism for specifying *shallow* copies (the operands themselves, and the values of corresponding reference variables in both operands are identical objects). As pattern invariants referring to copied state and the distinction between copies and aliases is poorly addressed in existing DPSLs, the mechanism described above is one of the major contributions of Alas.

`returnval` is a keyword in Alas that refers to the object returned by a method. It may be used in constraint boxes that are anchored to method call return events, and in this case refers to the object returned by the method. The value returned by methods of other lifelines in the behavioural specification can also be accessed using qualification (`<MethodName>.returnval`). When a method is called more than once in a BD, its calls may be distinguished using an occurrence number, appended to the method name after a colon in the form `<MethodName>:<OccurrenceNumber>.returnval`.

In fact, the state of any interacting object may be referred to at any control-flow event during an interaction. The object pointed to by a reference variable role belonging to some other lifeline can be referred to by qualifying it with the owning lifeline's object role name. Similarly to method calls above, object state at conditional branchings may be accessed by attaching an occurrence number to the control-flow operator (e.g., `opt`). For example, `opt:2.handler.successor` accesses the object pointed to by the successor reference variable of the handler lifeline at the end of the third `opt` operator occurring in the specification. This mechanism in Alas provides a means to express detailed relationships between the states of objects at different stages of the interaction.

Figure 3.13 illustrates the use of the `isAlias` operator and `returnval` keyword in the context of the Factory Method pattern. The specification states that the object returned by the Factory Method is the same object that was returned by the Product constructor. The distinction between object identity and value equality and the `isAlias` operator addresses the aliasing invariant type of our classification framework.

Semantics

To define object state invariant syntax, the state of a program is represented as a transition system. In each state ($s \in S$), there is a set of objects (O) that can grow and shrink between states as objects are created and destroyed, but has a fixed size in any one state. Each object has a unique identity, which can be accessed using the function $id(o)$. Each object has a set of variables (A), each element of which is referred to using the notation $obj.a$ (including variables of primitive and user-defined types), and also a subset of variables CA (i.e., $CA \subseteq A$) that represents the subset of variables bound to its `copystate`. The value of variables in each state can be obtained using the function $Val(a)$. Each object may be bound to a set of role names (N), and the function $obj(n)$ maps a role name to its object.

We can now define the Alas operators `isAlias` and `isCopy`:

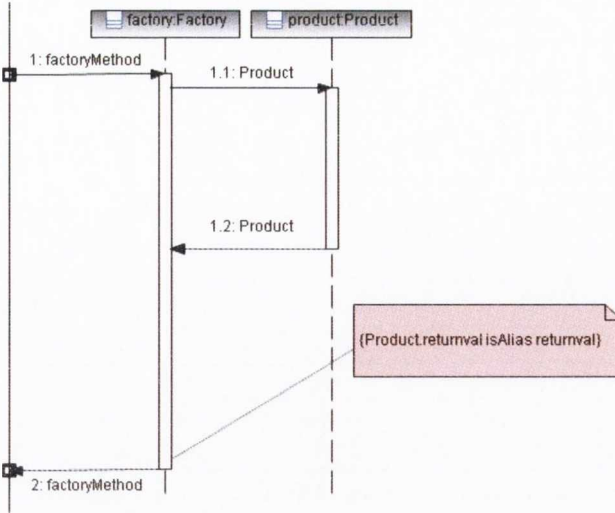


Fig. 3.13: Factory method specification using the `isAlias` operator and the qualified and unqualified version of the `returnval` keyword. The `factoryMethod` is required to return the object returned by the constructor of `Product`.

```

name isAlias otherName →def id(obj(name)) = id(obj(otherName)) .
name isCopy otherName
  →def
    ∀ca : CA •
      (Val(obj(name).ca) = Val(obj(otherName).ca)) ∧ ¬((name).ca isAlias (otherName).ca)

```

3.3.2.4 Collections

Alas supports the four kinds of collections provided by OCL: Bag, Sequence, Set and OrderedSet. Their meanings are summarized in Table 3.1, and are compatible with OCL. Collections are typed, i.e., their contents have a definite type that is specified using a class role. Collection roles are defined and related to their containing class roles similarly to method and reference variable roles. For each collection kind, there is an associated operator for its definition (e.g., `isBag`) that takes two operands: the collections role name and its content type. Its content type must be a previously-defined class role. An example of the definition of a Sequence in the context of the Observer pattern (the Sequence used above in Figure 3.7) is given below:

	Ordering	Element uniqueness
Bag	Unordered	Non-unique
Sequence	Ordered	Non-unique
Set	Unordered	Unique
OrderedSet	Ordered	Unique

Table 3.1: Ordering and element uniqueness of each of the Alas (and OCL) collection kinds

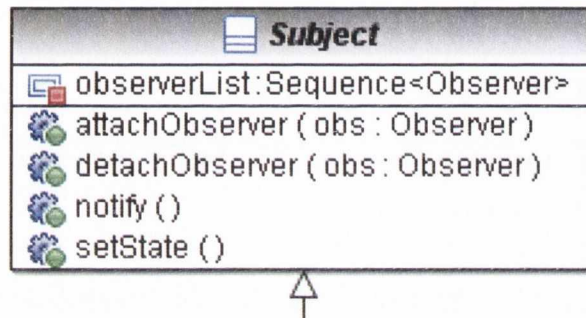


Fig. 3.14: A graphical definition of a collection. Subject contains an ordered collection of potentially non-unique Observer objects

```
class Observer
observerList isSequence Observer
Subject hasCollection observerList
```

In structure diagrams, collections are defined either with associations of multiplicity \star (with optional `ordered` and `unique` keywords attached to association ends, as in UML) or in the attribute compartment of a class. The type of the contents of a collection is given in angled braces after its kind. A graphical equivalent of the textual specification above is given in Figure 3.14.

Alas supports a number of operations that refer to the states of collections. These include `includes`, `excludes` and `union` and are taken directly from OCL. Particular positions in ordered collections may also be specified, such as `first`, `next` and `at`. The suffix `.old` is used to refer to the value of some variable at the beginning of the method in a condition placed elsewhere in the control flow of the method, and is borrowed from the Eiffel programming language [Meyer, 1997]. Applying the equality operator (`=`) to two

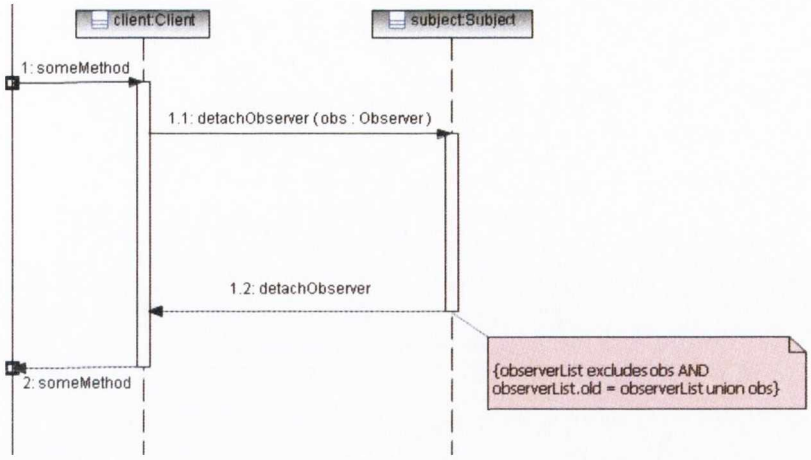


Fig. 3.15: Subject’s `detachObserver` specification illustrating collection operators and matching parameter and constraint role names. The method should remove the parameter from the list and have no other side effects.

collection operands states that every element of one operand is also an element of the other operand and, for ordered collections, that the elements occur in the same order. Both the Observer and Composite patterns include an object role (Subject and Composite respectively) that adds to and removes from a Sequence of objects. The Alas specification of the `detachObserver` method is given in Figure 3.15. Note the matching of the parameter role with an object role used in the constraint box. The specification states that the parameter passed to the `detachObserver` method has been removed from the collection and no other object has been added or removed.

3.3.3 Data structure invariants

As the specification of data-structure invariants is poorly-supported in the DPSSL and DPVT literature, the ability to specify data-structure invariants, in particular, shape invariants, is one of the major contributions of the Alas language. A number of design patterns describe the use of recursive data structures that can be modified at runtime. For example, the CoR pattern describes the use of a linked list of Handler objects and the Composite pattern involves a part-whole hierarchy structured as a tree. Each of these data structures has desirable properties relating to their ‘shape’, e.g., the absence of cycles in a linked list, the violation of which can lead to runtime exceptions, deadlocks, and logic errors. To be able to describe these properties, it is necessary to be able to express the transitive closure operation, so all elements in an unbounded structure can be related to one another logically.

There is no primitive operator in OCL for expressing transitive closure directly and it is not discussed in the latest OCL Standard [OMG, 2010a] [Baar, 2010]. To obtain the transitive closure of following a reference variable `predecessor`, for example, the user may write a recursive function similar to:

```
allPredecessors = self.predecessor
  → union (self.predecessor.allPredecessors) .
```

This statement, however, may not provide a valid closure, as it may recurse infinitely if the data structure has cycles and, in this case, evaluates to an undefined value [Vaziri and Jackson, 2000]. Some tools supporting OCL, such as Eclipse, provide a safe closure operation, by building a collection using an iterative fixedpoint algorithm [Damus, 2007].

In OCL queries and constraints, it is possible only to refer to objects that are navigable from the contextual object via associations. In a singly-linked list, for example, this corresponds to all the objects occurring later in the list than the contextual object. When defining data-structure properties, however, it is often more convenient to navigate a structure in the opposite direction to association links: whether heap sharing occurs can be expressed succinctly by evaluating if an object has two or more immediate predecessors (see Section 3.3.3.1). In OCL, it would be necessary to begin from the root of the data structure and attempt to identify two (potentially very long) paths to the object.

In Alas, Data structure roles are defined textually and data-structure operators are applied to the roles to form constraints. These constraints may be anchored to BDs, similarly to object-state invariants, or may be used unanchored in interaction invariants (see Section 3.3.4). Data structure roles are defined using existing reference variable or collection roles (*link roles*, in this context), where the variable or collection is the same class or a superclass of the class role that contains it. The data structure is defined as the transitive closure of following the link roles (which are potentially 1-to-n multiplicity relations, creating multiple *link paths*) from some root object until a terminating object is encountered over every link path. A terminating object may either be of a class that does not include any link roles or the link roles are uninitialized or empty. For example, in the Composite pattern, a Composite object holds a list of Component (its superclass) objects called `children`. Some of the `children` may themselves be Composites, while others are Leafs. Transitively including the `children` list of all Composites starting from the root

defines a tree of Composite objects at any particular stage in an interaction. A particular link path terminates either when a Leaf is encountered (no link roles) or when a Composite has no objects in its `children` list (empty link role).

A data structure is defined using the `isStructure` operator, which takes two operands. The first operand is a structure role name and the second operand is a class role, a dot and a comma separated list of link roles, all of the same class or a superclass of the class role and where all link roles are pre-defined reference variable or collection roles associated with the class role. Including a role in a data structure definition has no effect on existing object-state invariants that refer to that role. The data structures used in the CoR and Composite pattern are defined below. Note that there is syntactically no difference between including a reference variable (`Handler.successor`) and a collection (`Composite.children`) role:

```
chainOfResponsibility isStructure Handler.successor
compositeTree isStructure Composite.children
```

Alas provides a number of data-structure operators: `isCycleFree`, `isReachableFrom`, `isDisjointFrom`, `isShared` and `isSharingFree`. `isCycleFree` takes a single data structure operand and states that the data-structure should be free from cycles. `isReachableFrom` takes two object operands, and states that the first object may be reached from the second object by following one of the link roles in the associated data-structure definition. When `isReachableFrom` is used within a behavioural cardinality invariant, its operands may be bound variables representing objects drawn the data structure being quantified over (an example of this use is given in Figure 3.16). When the data structure that the operands is not clear from the context, the data structure can be specified explicitly using the *in* keyword as:

```
x isReachableFrom y in chainOfResponsibility
```

`isDisjointFrom` takes two data-structure operands and states that no object should be contained in both structures. `isShared` takes a single object operand and states the operand is reachable via at least two links in a data structure. Finally, `isSharingFree` is based on `isShared`, takes a single data-structure operand, and states that the structure is free from

objects that are shared. The operators are defined in more detail in the next section. Data structure invariants may be attached to lifelines, but are more often used in our GoF specifications in the context of interaction invariants, as described in Section 3.3.4. The ability to specify data-structure invariants is the second major contribution of Alas, as the entire invariant category is overlooked in the DPSL literature. These invariants are challenging to verify, and are the focus of an active area of research in software verification [Rondon et al., 2008][Kim and Rinard, 2011].

3.3.3.1 Semantics

A recursive data structure is defined as a directed graph, where the nodes are objects (with unique identifiers) and the edges are references labelled by their variable name. NULL is a valid value for a node, indicating that the object pointed to by a reference has not been initialized, or has been garbage collected, and objects may occur more than once in the same structure. The extent of the data structure stretches from some root node until all paths from the root encounter a NULL node or a node that defines no out edges. We define *hasSuccessor** as a transitive (non-reflexive, non-commutative) binary operator taking two object operands that evaluates to true if it is possible to navigate along the direction of the references from the first operand to the second operand. *hasPredecessor** is a similar operator, though it navigates in the opposite direction to the references (this operator distinguishes Alas from OCL in this context). Both operators have a non-starred counterpart, that indicates navigation is only performed for one step. Thus, *o hasSuccessor p* is true iff one of *o*'s immediate successors is *p*. The Alas data-structure operators are defined using equivalences (\Leftrightarrow) to statements using these basic operators and first-order logic. Below, the Alas data-structure predicates are defined in terms of equivalences:

$$\begin{aligned}
ds \text{ isCycleFree} &\Leftrightarrow \\
&\forall x, y : ds \mid x \text{ hasSuccessor* } y \bullet \neg x \text{ hasPredecessor* } y . \\
x \text{ isReachableFrom } y &\Leftrightarrow x \text{ hasSuccessor* } y . \\
x \text{ isShared} &\Leftrightarrow \\
&\exists y, z : ds \bullet x \text{ hasPredecessor } y \wedge x \text{ hasPredecessor } z . \\
ds \text{ isSharingFree} &\Leftrightarrow \\
&\forall x : ds \bullet \neg x \text{ isShared} \\
x.\text{last} &\Leftrightarrow
\end{aligned}$$

$\neg \exists y : ds \bullet y \text{ isPredecessor } x$

where ds represents some data structure, and x , y and z are objects. The $|$ symbol indicates a constraint (similarly to the `where` keyword in Alas). All the above definitions are qualified with the constraints that x , y and z are non-identical, but this constraint is omitted for the sake of brevity.

3.3.4 Interaction invariants

The object-state and data-structure invariants introduced in previous sections are invariants on the state of pattern roles at explicitly-specified stages of an interaction (e.g., when a method returns). However, it is necessary to express invariants on pattern roles that apply at *any* point at which interaction with the role may occur, e.g., a chain of Decorator objects should always terminate with an object that is a ConcreteComponent, i.e., not a Decorator. As classes can be subclassed, and these subclasses may include additional methods that mutate the pattern role's state, the group of methods that affect a pattern invariant is not known at specification time. To address this issue, Alas provides *interaction invariants*, similar to the history invariants introduced in Liskov and Wing [1994]. Interaction invariants must hold when any method belonging to a class actor returns. Interaction invariants introduce no new syntax to Alas, but are distinguishable syntactically from object-state and data-structure invariants as they are not anchored to any lifeline.

As objects may become aliased and be mutated via an alias, the object that holds a reference cannot always maintain an invariant on the state of an associated object alone. Rather than require a class to withhold access to its state (e.g., provide no accessor method), interaction invariants require all actors that may gain access to an object referred to by an invariant to cooperate to maintain that invariant. In the object-oriented specification literature, this is referred to as a *visibility-based* invariant [Leavens et al., 2007], as the invariant is 'visible' to multiple class roles. The other approach, where a class is required to be the only access point for some state, is known as an *ownership-based* invariant. We chose to specify visibility-based invariants as they are more generally applicable: ownership-based invariants require the ownership relation to be satisfied.

The CoR pattern decouples the sender and receiver of a request by creating a chain of objects, each of which has the option to handle the request or pass it on. A desirable property of the CoR pattern is that every request is eventually handled by some Handler.

This is often ensured by providing a `DefaultHandler` that is at the end of every chain of `Handlers`, where this `DefaultHandler` provides some default response to every type of request. Informally, the constraint is that there exists a handler that is capable of providing a response to this request type and this handler is reachable from every other handler.

The specification of this invariant in Alas uses the `isReachableFrom` data-structure operator, which takes two object operands and states that the first operand may be reached by the second operand by transitively following the link roles. `isReachableFrom` is defined in Section 3.3.3.1 above. The Alas specification is given in Figure 3.16. The BD states that the role `default`'s `HandleRequest` method will always call its `ServeRequest` method, i.e., it will never forward the message without handling it. The interaction invariant states that every element in the `chainOfResponsibility` data structure has a `DefaultHandler` further along the structure from it that will guarantee its request is handled. Note, the definition of the `chainOfResponsibility` data structure is given in Section 3.3.3. Clients manipulate the chain by inserting new `Handlers`, altering the shape of the structure and potentially violating the invariant. The interaction invariant of Figure 3.16 is shown with a specification of the `default` `DefaultHandler` role for illustration purposes, but could equally well be included within the SD.

Note that two quantified variables (`default` and `handler`) are drawn from the same set of objects (the data-structure `chainOfResponsibility`). During evaluation, all combinations of candidate actors for the two roles will be tested in sequence, with the two roles being bound to the same actor in some combinations. For this reason, if the roles are mutually-exclusive (as in the example of Figure 3.16), this must be specified explicitly using negation (`NOT`) and `isAlias`. While most roles are mutually-exclusive, we chose to allow multiple bindings to a single actor for consistency with classical first-order logic.

The Composite and Decorator pattern have similar 'well-configuredness' interaction invariants, as shown below. The `isSharingFree` data-structure operator takes one data-structure role as an operand and states that the data structure should have no object in it that is referenced via a link role from two other objects. The definition of `isShared`, on which `isSharingFree` is based, is given in Section 3.3.3.1. The first class invariant below states that no object is shared within the tree structure, though some object outside the structure may have a reference to it, i.e., non-sharing within the data structure does not imply ownership by a single composing object. Finally, the Decorator invariant is similar to that of the CoR above, but more specific. It specifies that a chain of Decorators must

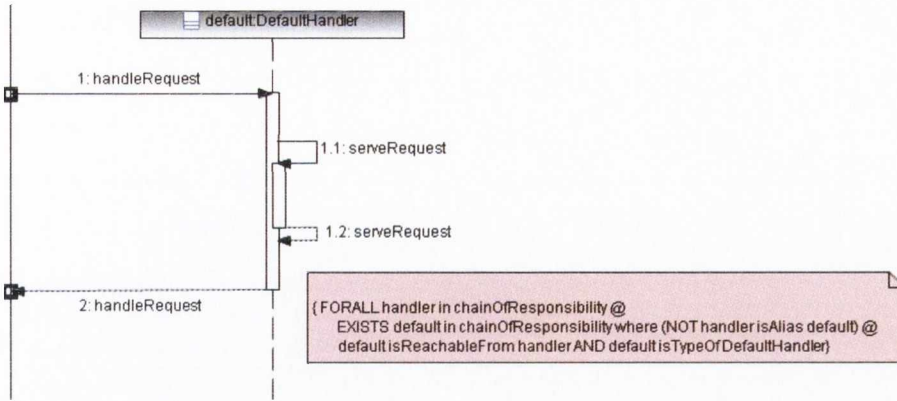


Fig. 3.16: CoR Behaviour diagram along with an interaction invariant using Alas data-structure operators. The specification states that a `DefaultHandler` should be reachable from every `handler` in any valid `chainOfResponsibility`. This constraint must be satisfied at the end of every method that refers to and/or mutates the `chainOfResponsibility`.

terminate in a `ConcreteComponent` (or one of its subclasses) and not a `Decorator`. The operations for ordered collections (`first`, `last`, `next` and `at`) are also applicable to data structure definitions, where the data structure definition defines a structure with a total order of elements.

```
componentTree isSharingFree
decoratorChain.last isKindOf ConcreteComponent
```

The invariant defined above in the context of the CoR pattern is largely equivalent to the `Decorator` invariant stated above: it will be satisfied if the last element in a structure has a particular type. However, this is not the case if each `Handler` in a chain had potentially multiple successors (the `chainOfResponsibility` had multiple link roles or a link role that is a collection).

3.3.5 Behavioural pattern variant specification

The `var` operator is used in Alas to specify variant-specific control-flow. It may take one or more operands. Each operand is labelled with a variant name in its guard. A `var` operand's guard may contain nothing but a variant name. When one operand is supplied, potential choice is intended: the variant's behaviour may or may not occur in an implementation

and the implementation can still satisfy the core or some other variant. When multiple operands are provided, one of the alternatives that these operands represent *must* occur in every valid implementation (i.e., there is no core variant in this case). The semantics of the single and multiple operand var operators are very similar to the definitions of `opt` and `alt` above respectively, and have been omitted here for the sake of brevity. A BD referring to a structural role that has been removed from a variant does not apply to that variant. If the variant is required to perform a similar interaction, it must be specified in full and wholly contained within a `var` operator, naming the relevant variant.

In the Composite pattern, a trade-off to consider is whether to allow explicit links from child Component's to their parent, which can simplify traversal and Component deletion. An important invariant in this context is that the child-parent reference is kept consistent with the parent-child reference. In the Alas specification of Figure 3.17, this responsibility is given to the parent Composite's `add` method, which could, for example, call a `setParent(Composite c)` method on the Component parameter `c`. The `this` keyword can be used to refer to the object whose lifeline has the constraint anchored to it. The invariant of Figure 3.17 states that at the end of the `add` method, the Component that has now been added to the Composite's `children` list has its parent reference set to the identity of the Composite.

An alternative to consider in the context of the unsafe variant of the Composite pattern defined above is whether the child management methods declared in the `Component` simply do nothing or throw an exception. Using the scoping operator of Alas, we can define a sub-variant of the unsafe variant called the *do nothing* variant, where each child management method does not call any of the other method roles defined in the specification. The 'do nothing' requirement can be captured precisely using the `Equivalence` conformance relation, as shown in Figure 3.18.

One of the implementation issues to consider with respect to the CoR pattern is the implementation of the successor reference variable. The variable can be declared in the `Handler` superclass or the `ConcreteHandler` subclass. In the first case, delegation to the next object in the chain is performed by the `ConcreteHandler` calling the request handling method on its superclass, and the superclass performs the forwarding. In the latter case, the subclass object delegates directly. These alternative behaviours are specified in Figure 3.19, which extends Figure 3.6 to support two variants: `viaSuperDelegation` and `directDelegation`. Note that the two roles named `successor` do not create a naming conflict, as they refer to

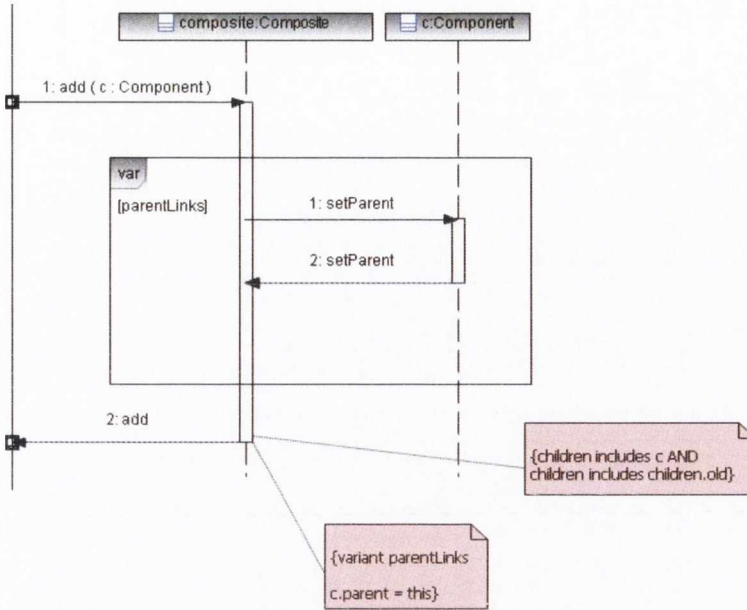


Fig. 3.17: Composite Behaviour diagram illustrating the use of the `var` operator and variant-labelled constraint boxes to specify variant-specific behaviour. The same variant is named in both the `var` operator and constraint box, meaning that both of these constraints must apply in a valid `parentLinks` variant of the Composite pattern.

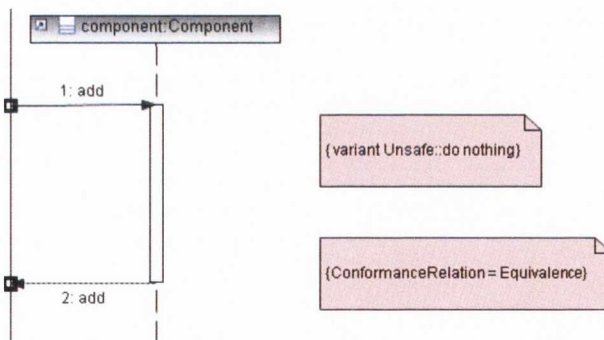


Fig. 3.18: BD illustrating the use of the scoping operator for the definition of a sub-variant and the alternative conformance relations provided by Alas. The do nothing variant may only be satisfied by implementations that satisfy the unsafe variant

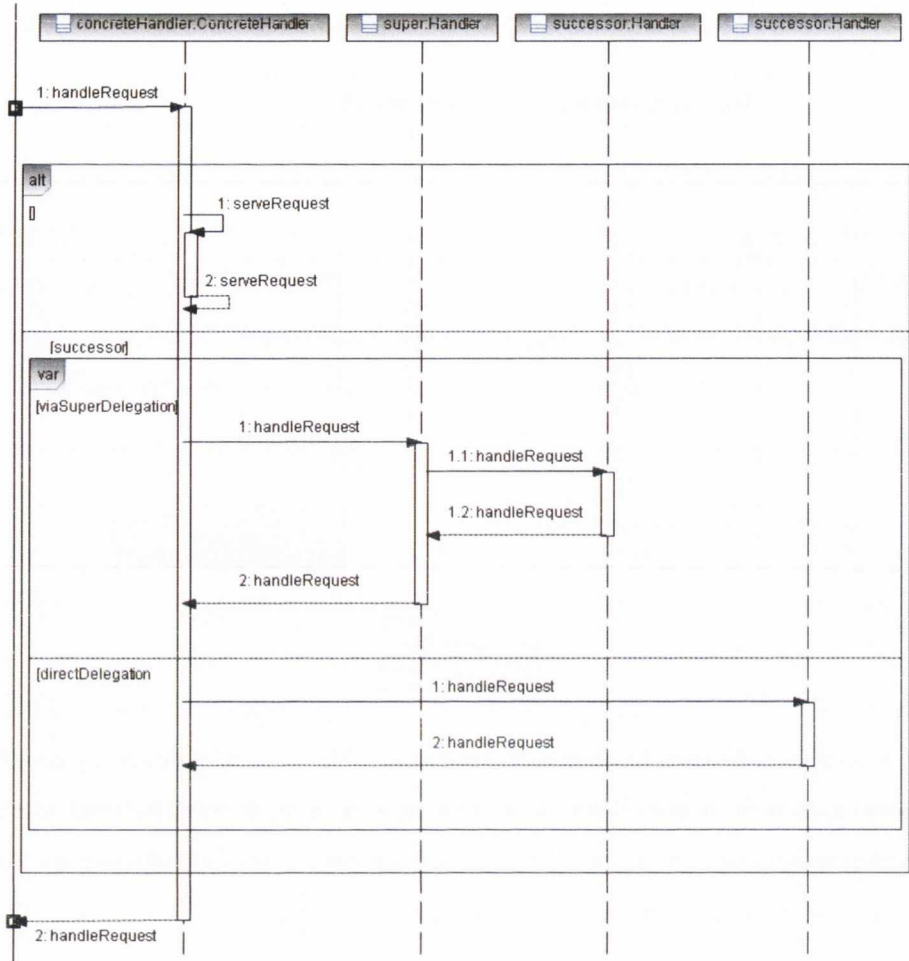


Fig. 3.19: Specification of two behavioural variants of the CoR pattern using a multiple-operand `var`. Each valid CoR implementation must perform the behaviour specified in `viaSuperDelegation` or `directDelegation`, but not both or neither.

reference roles belonging to two different class roles in two separate variants.

3.3.6 Behavioural cardinality invariant specification

The behavioural specifications introduced up until this point describe invariants over interactions between individual roles. These roles may be filled by multiple actors in the implementation. The verification of each class actor can be done in isolation from other class actors, as the conformance of one does not affect the conformance of the other. However, some behavioural specifications place constraints on a set of conceptually-related classes or methods, and these use quantification, similarly to the structural specifications in Section 3.2.3. These types of invariants are referred to as behavioural cardinality invariants.

Quantified statements do not need to be anchored to any lifeline, as they apply to the entire specification.

As in structural cardinality invariants, quantified variables may take the place of structural and behavioural roles in BDs. A variable may substitute for both a class or object role name given in a lifeline head or a method role name that labels call events. Each variable used as a role name substitute must be bound within some quantified statement in the same BD. Variables may occur in a quantified statement and not in the accompanying BD. In cases where two or more variables are drawn from the same set, and may be bound to a single lifeline, the substitution operator (\rightarrow) can be used to map one or more variable names in a quantified statement to a single role occurring in the accompanying BD. Intuitively, two actors can be imagined as being bound to the same role at the same time.

The Abstract Factory pattern describes an exclusive relationship between ConcreteFactory and ConcreteProduct roles: (1) some subclass of each AbstractProduct should be initialized by some Factory Method in each ConcreteFactory but (2) no two Factory Methods in a ConcreteFactory should initialize the same ConcreteProduct. The two conditions above describe a surjective (onto) and an injective (one-to-one) relationship between sets respectively. When combined, this is a bijective relation. These relations are specifiable in Alas using first-order logic.

Figures 3.20 and 3.21 show the specifications of the two invariants described above. It uses the `context` keyword, taken from OCL, to indicate the class role in the context of which the invariant is evaluated. An equivalent specification could be obtained without using the `context` keyword, but with a more complex quantified statement. Note that both variables `fm1` and `fm2` are substituted for the same method variable (`fm`) in Figure 3.21, which specifies the injective relation (that no two ConcreteFactories create the same ConcreteProduct). Note also that the variable `ap` that occurs in the quantified statement in both figures does not occur as a role name substitute in either BD. Also, the `FactoryMethodSet` does not need to be associated with a class role: it is by default identified as the set belonging to the contextual class.

3.4 Summary

This chapter described Alas, a DPSL capable of specifying invariants from each of the five invariant categories identified in Chapter 2, including instances of the three invariant types

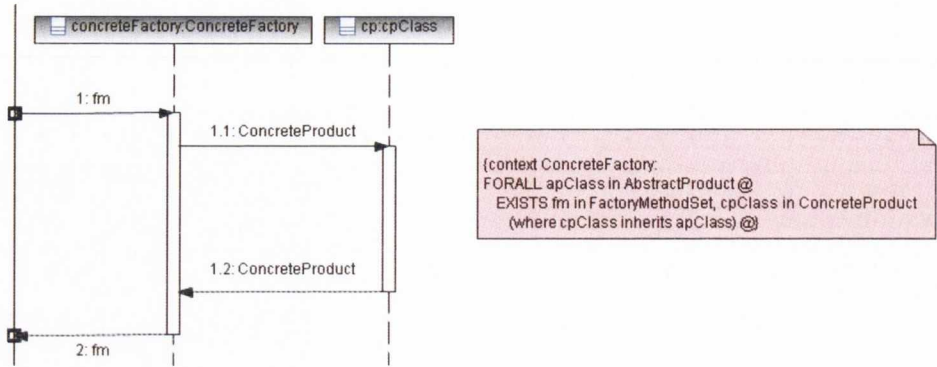


Fig. 3.20: Structural cardinality invariant specifying a surjective relation between Factory Methods and AbstractProducts in each ConcreteFactory: every FactoryMethodSet contains some method that initializes some subclass of each of the AbstractProduct classes. Both the method role representing a Factory Method and the class role representing a ConcreteProduct have been substituted with variables bound in the quantified specification.

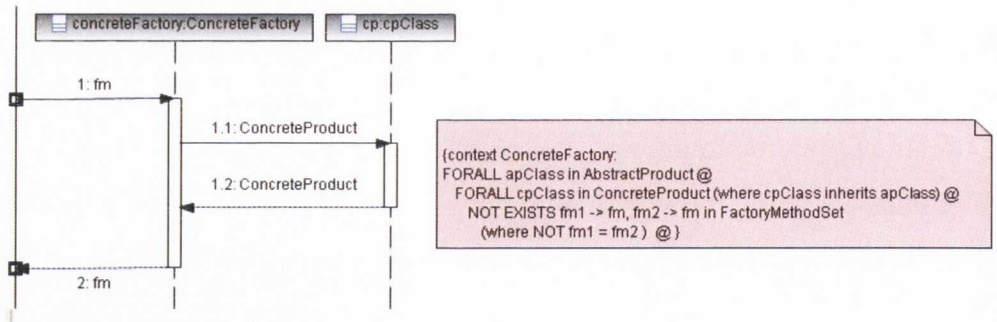


Fig. 3.21: Structural cardinality invariant specifying an injective relation between Factory Methods and ConcreteProducts in each ConcreteFactory: no two Factory Methods create the same ConcreteProduct. Two quantified variables are bound to a single variable substituting for a method role.

that are poorly supported in the DPSL and DPVT literature. Alas includes syntax to describe each of the syntactic elements identified in Chapter 2. Also, Alas is capable of expressing variants of design patterns, that differ in terms of structure and/or behaviour. In fact, variation points can include invariants from any of the five invariant categories. Multiple named variants may be combined into a single specification, and rules are defined that identify combinations of variants as valid or invalid. Alas is based upon and extends UML 2, which provides diagrams that are suitable for both structural and behavioural specification and has a small semantic gap to object-oriented programming languages. The description of a DPVT that is capable of verifying the novel invariant types provided by Alas is described in Chapter 4. An evaluation of GoF design pattern specification in Alas, in the context of code bodies commonly analyzed in the design pattern specification and verification literature is provided in Chapter 5.

Chapter 4

AVT Implementation

The design of the Alas Verification Tool (AVT) is described in this chapter, with occasional implementation details being presented where these are relevant. AVT is designed to be capable of verifying Java source code against specifications written in the Alas design pattern specification language (DPSL). We focus on the verification of novel invariants and invariant categories in Alas. We present the compiler framework used as a basis for AVT below. We then discuss the requirements imposed by Alas on a supporting verification tool. We describe the program analysis algorithm implemented by Alas and list some minor implementation issues. Finally, we outline AVT behaviour specific to each invariant category.

State-of-the-art DPVTs [Shi, 2007a][Blewitt et al., 2005][Stencel and Wegrzynowicz, 2008] all verify properties of local variables or attributes in the context of a single method, such as whether a variable has a null or non-null value (Singleton pattern) or points to a new object (Singleton, Factory Method). The analyses that these tools implement compute the values of variables on the stack only, while the verification of object-state and data-structure invariants requires an accurate model of the graphs of objects stored on the runtime heap. We implement an analysis known as shape analysis, novel in the area of design pattern specification and verification, to enable the verification of these invariant types. A static analysis approach was chosen instead of a dynamic analysis, as the invariants to be analyzed are generic, i.e., not application-specific [Evans, 2005], and it facilitated comparison to the state-of-the-art DPVTs, which all implement a static analysis.

AVT is built upon the Jikes compiler [IBM, 2005]. The Jikes compiler was chosen as it is open source, fast, and because it is the basis for the PINOT tool [Shi and Olsson, 2006]. One disadvantage of choosing Jikes is that it is no longer under development and has limited

support for Java 5 and any subsequent Java versions. Initially, we planned to extend the PINOT tool and perform a direct comparison between AVT and PINOT as part of the evaluation of AVT. However, the PINOT source code is modularized by pattern instead of by language element or invariant category and this made it difficult to extend. Chapter 5 explains why a direct tool-to-tool comparison was not performed. While its source code did not prove useful for our initial intention, PINOT did provide useful examples of how to extract information from Jikes to perform data-flow analysis.

4.1 Verification requirements imposed by Alas

Each invariant category imposes its own requirements and constraints on a supporting verification tool. We address each of the invariant categories separately here

- Verification of implementation dependency invariants requires the complete exploration of the abstract syntax tree (AST) of a method and the identification of constructor calls to classes in a particular inheritance hierarchy.
- Verification of deep copy object-state invariants requires the comparison of the values of objects of the same class, at a particular point in the control flow of a method. The possible values of both primitive and reference variables need to be computed. The concept of a copy involves not just a single object and its variables on the stack, but a complex object structure involving all of the heap that may be referenced transitively from the object. Both the values of reference and primitive variables are relevant to the deep copy invariant.
- Verification of data-structure invariants involves computing the values of theoretically-unbounded recursive data-structures stored in the heap and identifying properties of these structures, for example, that they are free from cycles. As structures are theoretically unbounded, and may grow to prohibitively-large sizes for verification in even medium-sized programs, it is necessary to summarize information about a structure that is not necessary for verification, but represent properties of the structure necessary for verification precisely.
- As objects typically encapsulate their internal state in object-oriented programs, clients usually operate on the internal state of an object by calling methods on the object. The same method, for example, an instance of the Composite class role's

addChild method, may be called multiple times in a program on different objects in different contexts. To verify properties of object structures precisely, it is necessary to distinguish the different calling contexts of methods which operate on the structures.

- Similarly to most DPVTs in the literature, AVT is a fully-automated verification tool. As software verification is undecidable in general, any software verification tool must introduce some inaccuracy into its analysis to guarantee termination in general. It is important that these inaccuracies are sound and conservative, i.e., they do not allow statements to be proven that are untrue. As Alas clauses may or may not be negated, Alas provides a particular challenge for conservative verification, as will be discussed later in the chapter.

The remainder of the chapter is structured as follows: Section 4.2 describes the shape analysis algorithm implemented by AVT. Shape analysis is a precise form of data-flow analysis suitable for the verification of object-state and data-structure invariants, as discussed in Chapter 2. Section 4.3 describes the functionality specific to the verification of each of the different novel invariant categories addressed by AVT. Section 4.4 discusses some miscellaneous implementation issues. A summary of the chapter is given in Section 4.5.

4.2 Shape analysis algorithm

Software verification aims to prove that pieces of software demonstrate some desirable or undesirable property. While there are a number of largely-independent communities working on the problem of software verification, there are many similarities between these communities and the solutions that they have developed. Schmidt [1998], as well as Steffen [Steffen, 1991], have demonstrated that there are close parallels between the capabilities of data-flow analysis and model checking. Nielson et al. [1999] have demonstrated the parallels between a number of verification approaches such as type and effect systems, data-flow analysis, and constraint-based systems. Each of these approaches may be applicable to the verification of the invariant categories and types identified in Chapter 2. We chose data-flow analysis, and in particular, shape analysis, as it has been demonstrated as being applicable to the verification of object-state and data-structure invariants similar to those expressible by Alas [Calcagno et al., 2007][Rinetzky et al., 2005][Berdine et al., 2007]. Data-flow analysis is also the approach used by the current state-of-the-art DPVTs [Blewitt et al., 2005][Shi and Olsson, 2006][Stencel and Wegrzynowicz, 2008]. Data-flow analysis (DFA)

involves setting up and solving a system of equations that involve the program variables and their values before and after each statement in the program [Aho et al., 1986]. The solutions to data-flow equations after each statement are a set of *data-flow facts*. DFAs are commonly implemented as iterative fixed-point algorithms which guarantee termination if a number of well-documented and well-studied conditions are met [Nielson et al., 1999][Aho et al., 1986]. Shape analysis is a DFA technique whose equations describe a finite characterization of the shape of data structures [Sagiv et al., 2002]. Analyses are intra-procedural if operation calls are ignored or if the language under analysis does not include operation calls. Analyses that handle operation calls are termed inter-procedural¹. As part of the implementation of AVT, we developed an inter-procedural shape analysis based on Rinetzky et al. [2005], with a number of differences which we document in this chapter. In particular, the representation of the data-flow facts is closer to [Nielson et al., 1999] than Rinetzky et al. [2005].

DFA is performed on a program representation known as the control-flow graph (CFG). A CFG is a directed graph representing all the paths through the method or program. Nodes in the graph are basic blocks: sequences of consecutive statements that do not contain the possibility of branching or jumping [Aho et al., 1986]. Edges are valid flows of control between blocks and depend upon the type of branch or jump statement used. The transformation of the AST provided by Jikes to a CFG performed by AVT is not documented here in detail, but is based upon the algorithm of Aho et al. [1986, pp529] and the Java Language Specification [Gosling et al., 2005]. The representation of inter-method control flow, referred to as the call graph, is a point of variability in DFA tools, and is discussed in Section 4.2.2.

The DFA fixed-point algorithm calculates the output set of data-flow facts for each basic block in the CFG by applying the transfer function of each statement in the basic block in sequence to the input set, which is itself the output set of each of the block’s predecessors in the CFG. The algorithm continues to iterate until no output set changes since the previous iteration. The transfer functions for each type of assignment are dealt with formally by Nielson et al. [1999], we simply implement the semantics given there for Java.

In this chapter, we present a running example to illustrate the features of the shape analysis algorithm implemented in AVT. The source code for the example is given in Figure 4.1. It contains an assignment of a parameter with an unknown value to an attribute, **a**, and two assignments of newly-created objects to another attribute, **b**, over mutually-

¹As we analyze Java, all procedures are methods, but we retain the term procedure, as it is the prevailing term used in the DFA literature

```

public void someMethod( A aParam ){
    a = aParam;
    if( c != null ){
        b = new B();
    }
    else {
        b = new B();
    }

    b.setRealSubject( a );
}

```

Fig. 4.1: The running example used to illustrate the features of the shape analysis algorithm implemented in AVT

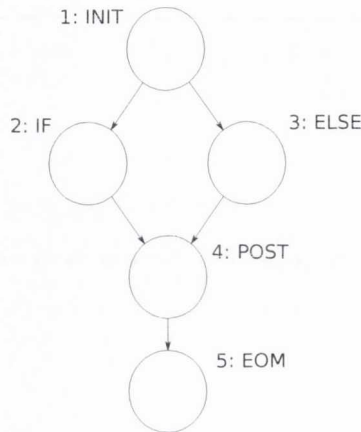


Fig. 4.2: Control-flow graph for the example of Figure 4.1

exclusive control-flow paths. Finally, the two attributes are associated with one another via a call to `setRealSubject`, the source of which is not shown, `setRealSubject` simply assigns its argument to an attribute, `realSubject` of the callee object.

A CFG for the example code of Figure 4.1 is given in Figure 4.2. Data-flow facts flow along edges in the CFG in the direction of the arrows in the figure. For example, the data-flow facts input to the analysis of the final (POST) block is the combination or `meet` of the data-flow facts output from its predecessor blocks (IF and ELSE). How data flow facts are combined at the meeting point of control-flow paths is defined by the `meet` operator. The output of each iteration of the fixed-point algorithm is the result of applying the transfer functions of each of the individual statements in the POST block to its input data-flow facts in sequence.

As a number of Alas invariants are placed on individual methods, AVT performs a program segment analysis instead of a whole-program analysis. This is significant for reasons that will be discussed later in the chapter. The functionality required for a program segment analysis can be seen as a super-set of the functionality required for whole-program analyses, as the whole program is contained within a single method (`main` in Java). We refer to the source code under analysis throughout as the program segment under test (PUT), which refers to the method under analysis and the transitive closure of all the methods called from it.

We present first the intra-procedural part of the shape analysis algorithm, then the inter-procedural part before discussing some issues regarding convergence and scalability.

4.2.1 Intra-procedural analysis

The data-flow facts operated on by a shape analysis are known as shape graphs, and the output of a shape analysis is a set of shape graphs. Each shape graph contains a number of *core predicates* that describe the value of program variables at that point in the analysis. Also, optional additional predicates describe properties of individual or groups of core predicates, such as whether a particular value is shared between references of multiple other values or occurs on a cycle of values. These optional predicates are referred to as *instrumentation predicates* in Sagiv et al. [2002]. The level of precision with which we model core predicates means we do not require any instrumentation predicates, as only minimal information about the precise shape of data structures is discarded by our algorithm (in particular, in the presence of recursive calls). Core predicates are further subdivided into *unary* and *binary* predicates. Unary predicates involve variables that would be stored on the stack at runtime, i.e., local variables and attributes of ‘this’. Unary predicates are represented as pairs of `Var` x `Val`, where `Var` is a variable drawn from the set of all variables in the PUT, and `Val` is a value drawn from the set of all values occurring during the analysis. Binary predicates represent the runtime heap and are triples of `Val` x `Var` x `Val`, where the first value is the *source* object holding the reference `Var`. Unary predicates could be considered a special case of binary predicates where the source object is ‘this’. Values for reference variables are locations, as described below.

The shape graphs resulting from separate control-flow paths are combined in a union when the paths meet, but no information is discarded: the result of two or more paths meeting includes all the graphs resulting from each path. This allows ‘strong updates’ to

be performed: the previous value of a variable is removed or ‘killed’ when a new value occurs [Nielson et al., 1999]. This results in a powerful analysis and is required for Alas verification, as discussed later in Section 4.3.1. The combination of shape graphs at the meeting of control-flow paths is maximally precise (as it does not summarize or discard any information), but means the set of shape graphs is exponential in the number of branches and jumps in the PUT.

4.2.1.1 Locations

Following Sagiv et al. [2002], we use the term *location* to refer to a memory location in the heap. Shape analysis represents the theoretically infinite number of locations with a finite set of abstract locations. In our algorithm, we distinguish between two main types of locations: *allocation locations* and *unknown locations*. Allocation locations represent objects created within the PUT. A new allocation location is created once for every allocation site (expression containing the `new` keyword) in every calling context. Allocation locations occurring within loops are transformed into a single *multi-allocation location*, representing one or more locations (*Multi-unknown locations* also occur in the context of collections and loops). Unknown locations represent objects that are referenced but not allocated within the PUT, including the formal parameters of the method under test, as well as any class attributes referenced before being assigned a value. A new unknown location is created the first time such a variable is referenced in the PUT. The use of multiple unknown locations instead of a single one for each type or for all types allows us to calculate more accurate aliasing information. For example, in Figure 4.3, before the body of the method `setMethodWithTwoParams` begins execution, formal parameters `dsParam` and `objParam`, and attribute `dsAttribute` *may* all be aliases, but none can be shown to *definitely* be aliases. However, after line 1, our algorithm can determine that `dsParam` and `dsAttribute` are aliases, while using a single abstract location for all unknown values, similarly to Sagiv et al. [2002], all three variables can only be shown to be possible aliases.

Both types of locations are uniquely identified by a quadruple of integers: calling context number, block number in method, line number in block, and occurrence number in line. Each unique node in the call graph is assigned a number as an identifier. Call graph construction is dealt with in Section 4.2.2. For allocation locations, this quadruple represents where the `new` expression occurs and for unknown locations, this quadruple represents where the name expression that first references the variable occurs. The value of primitive

```

void setMethodWithTwoParams( DummySubject dsParam,
                             Object objParam ){
    dsAttribute = dsParam;
}

```

Fig. 4.3: Simple set method example with two formal parameters and one reference variable that may be all aliased when the method call dispatches

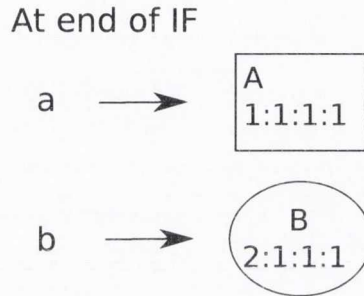


Fig. 4.4: Shape graph resulting from the analysis of the first path through the example of Figure 4.1, until the end of the IF block

variables are represented exactly like unknown locations: as a quadruple representing where they are first referenced. Similarly to most data-flow analysis tools, exact literal values of literal integer and floating-point numbers is not part of the data-flow facts, as this greatly multiplies the size of the state space of the analysis [Cousot and Cousot, 1977].

Figure 4.4 illustrates the shape graph output after analyzing the IF block. Allocation locations are denoted by circles, while summary locations are denoted by squares. Inside the location's outline is its class and its unique identifier. Figure 4.4 shows two unary predicates (i.e., pairs of variable and value), denoting the names and values of two attributes of 'this' (local variables are also represented as unary predicates). `a` has been initialized to the summary location representing the unknown value of the parameter, which is first accessed in the first statement of the first block of the method. While `b` has only one potential value at the end of the IF block: the object created in the block. Note that the new object's context number (the first number in its unique identifier) is different from the that of the summary location, as it has been allocated in a different context: the constructor of `B`. The predicate representing `aParam` is omitted, it has the same value as the predicate representing `a`, due to the assignment.

Figure 4.5 illustrates one of the two shape graphs output at the end of the analysis. This

Shape graph 2
At EOM

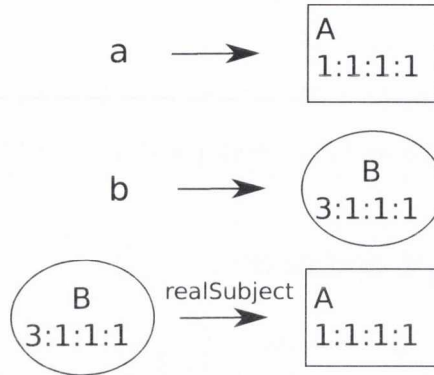


Fig. 4.5: Shape graph representing the second path through the method of Figure 4.1, through the ELSE block

graph represents the second path through the method, through the ELSE block. Note that the context number of the location pointed to by **b** is different from the context number of the location pointed to by **b** in Figure 4.4. This illustrates the distinction made by our shape analysis algorithm between separate calling contexts to the same method. The receiver of the call to `setRealSubject` is identified, and the value of its `realSubject` attribute is set to the argument to the method. This is shown in the figure as the binary predicate, represented by the triple of location, variable and location. The link in the heap between the locations pointed to by **b** and **a** is maintained after the method returns, precisely representing paths in the heap.

4.2.1.2 Collections

Collections are complex to handle in data-flow analyses for a number of reasons. Two of the most challenging features of collections are their theoretically-unbounded size, and accesses and insertions that do not occur at the extremities of the collection. In program segment analysis, a further complication is that the existing contents and properties of a collection at the start of the method execution may not be known.

Updates to collections are handled similarly to updates to any object: with unary and binary predicates. When a new element is added to a collection, a binary predicate is created with the collection as the source object, the new element as the target object and a dummy variable is created to link the source collection and the target element. These dummy

variables are known as *selector* variables and are numbered according to their abstract position in the collection. Collection insertions within loops are handled by making the target of the corresponding predicate a multi-allocation or multi-unknown location. The selector becomes a *multi-selector* also. Multi-selectors and multi-unknown locations are also useful in a second case when a collection is used that has not been initialized in the PUT, and hence has an unknown number of elements already in it. In this case, the collection itself is represented as an unknown location and its original contents are represented as one element: a multi-unknown location accessed through a multi-selector variable, representing the unknown number of elements in the collection.

Accesses of, insertions to and removals from a collection which are indexed by an integer value present a problem, as the exact value operated on cannot be known. We refer to these operations as *complex* accesses, insertions and removals. Complex accesses may access any element in the collection. This is handled in AVT by producing, at a statement where a complex access occurs, an output shape graph for every input shape graph and potentially-accessed location (i.e., multiplying the number of input shape graphs by the abstract size of the collection). Within loops, accesses may occur multiple times within the same calling context. This is handled by having a shape graph store the results of previous accesses in the same context and the analysis produces a different result on each iteration until all potential contents have been accessed. This, along with the use of multi-locations and multi-selectors, allows collection complex access within and outside of loops to be handled conservatively when the exact value accessed on each iteration is not known. Special considerations for complex removals and complex insertions are omitted here for the sake of brevity.

4.2.2 Inter-procedural analysis

Context-sensitive analyses calculate separate results for each separate calling context of the same method, improving analysis precision in languages with operation calls [Nielson et al., 1999]. Object-sensitivity is a form of context-sensitivity that distinguishes calling contexts based on the receiver object at the call site. Object-sensitivity has been shown to be especially suitable for object-oriented languages such as Java [Milanova et al., 2005], providing precise and efficient analyses relative to other approaches to context-sensitivity. For example, the same mutator method may be called multiple times in a program, on numerous different objects of the same class. Analyses that fail to distinguish between contexts merge all the data-flow facts that reach a particular method or method call site.

In the terminology of the object-sensitivity community, we implement an object-sensitive analysis with an infinite context depth [Smaragdakis et al., 2011], though in practice the contexts are finite, as described below.

The *call graph* is a directed graph that represents the potential callee objects and methods at each call site in the entire PUT [Grove et al., 1997]. We compute the call graph simultaneously with the shape analysis (*on-the-fly* call graph construction), so both the call graph and output shape graphs must converge to a fixed point for the algorithm to terminate. On each iteration of the algorithm, all shape graphs reaching a statement containing a call are compared to the call graph edges already created for that statement on previous iterations. If a new call receiver (callee) location occurs in some shape graph, a new call edge is added to the call graph. Following Rinetzky et al. [2005], the input shape graph to a call graph node includes only the portion of the heap reachable from the callee location and call arguments to the target call graph node. This is done by accessing first the values of the call arguments, and then searching through the binary predicates for predicates where the call argument values are the source of the predicate. The targets of each of the binary predicates identified are added to the shape graph. The targets are then used as the sources for the next search through the set of all binary predicates, and so on until all values reachable from the arguments have been included. The resulting shape graph at the end of the statement is a combination of the input graph at the caller and the output graph at the callee. The merging of these two graphs in our algorithm follows Rinetzky et al. [2005], and depends upon whether the call is local or not, and requires conversions of some unary predicates to binary predicates and vice versa, to model the runtime stack and heap accurately.

The call graph edges resulting from the analysis of the call to `setRealSubject` in Figure 4.1 is shown in Figure 4.6. Note, separate call graph nodes have been created for each callee *location* occurring in the shape graphs that reach the call in the analysis. If a new call graph node was added only for every callee *class*, only one node would be created for this graph, as both potential callees in this example are of the same class. This latter approach is highly conservative and discards information necessary to verify object-state and data-structure invariants: all potential callee locations of the same class are merged creating spurious relations between locations. Figure 4.6 also illustrates the mutually-recursive relationship between call graphs and CFGs, the CFG of `someMethod` contains two call graph edges leading to `setRealSubject`, and call graph nodes contain the CFGs of the

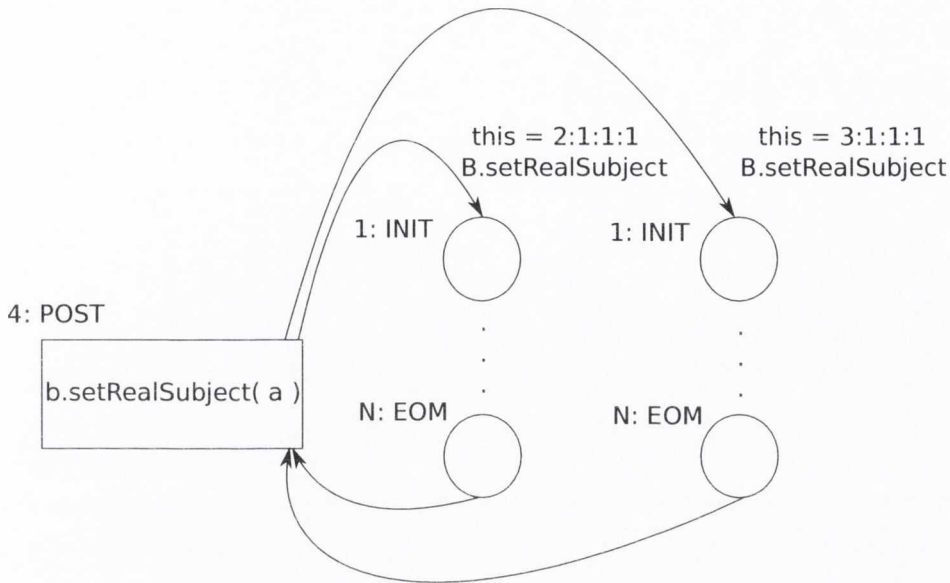


Fig. 4.6: The call graph edges resulting from the analysis of the call to `setRealSubject` in Figure 4.1

methods they represent.

The data-flow facts flow across call edges similarly to how they flow across a method's CFG edges, with the added complication that values can move from the stack to the heap and vice versa when the 'this' predicate changes. To handle these flows, our algorithm converts core predicates to binary predicates and vice versa at call entry and return edges. The identified callee location in the calling context becomes the 'this' predicate in the called context. The state reachable from the parameters of the call, including the values of attributes of the callee location, are included in the shape graphs passed to the call graph node². The callee location is compared to the source location of all binary predicates to identify the values of the callee attributes. The attributes of the callee's attributes are identified in turn using the same method, until the entire graph of objects reachable from the callee is included in the input shape graph. The same procedure is followed for method arguments.

A call stack is maintained throughout the analysis to allow for the identification of recursive and mutually-recursive methods. Each recursive call site is given a new call graph edge the first time it occurs in the stack only (this approach is termed *virtual unrolling* in

²This approach is more efficient than passing the entire heap to every method call on the assumption that the graph of objects reachable from a callee and call parameters are a small subset of the entire heap

Rinetzky et al.	AVT
Single summary location per class	Single summary location per referenced variable name (maximum)
Merge shape graphs where control-flow paths meet	Maintain separate shape graphs for each path through a program

Table 4.1: Distinguishing features of the shape analysis algorithms implemented by Rinetzky et al. and AVT

Theiling and Ferdinand [1998]). For example, a method with two recursive call sites (`site1` and `site2`) will result in a call graph including two separate subgraphs including the edges: `site1` \rightarrow `site2` and `site2` \rightarrow `site1`. Subsequent occurrences of the same recursive call sites in the analysis are not given their own node in the call graph. When these subsequent occurrences are encountered, the analysis result is given as the result of analyzing the call site’s call graph node on the previous iteration.

4.2.3 Termination

The conditions that need to be satisfied for a DFA to be guaranteed to terminate for all inputs are dealt with at length elsewhere in the literature [Aho et al., 1986][Nielsen et al., 1999], and our analysis is based upon approaches that have been shown to converge [Nielsen et al., 1999][Sagiv et al., 2002]. One important condition is that the algorithm does not allow the data-flow facts to grow indefinitely in general. We restrict ourselves to demonstrating that the distinguishing features of our algorithm relative to the algorithm of Rinetzky et al. [2005] cannot lead to infinite data-flow facts. This guarantees that AVT terminates in general. These extensions are summarized in Table 4.1.

The number of variables and occurrences of variable names in a program’s source code are of course finite. The number of unknown locations occurring during the shape analysis is bounded by the number of variables occurring in the PUT and the number of contexts. The number of allocation locations is bounded by the number of allocation sites and the number of contexts. The number of contexts is bounded by the number of call sites and the number of locations. There remain two theoretical sources of infinite state spaces. Firstly, allocations within loops are a potential source of infinite objects. This is avoided by creating a single multi-allocation location in these cases. Such cases can be identified

using the loop-back data-flow from loop exit to loop entry. Secondly, recursive calls are a potential source of infinite contexts. This is avoided by only creating a new context for a recursive call site the first time it occurs in the call stack, as described above. Thus, the interaction between contexts and locations is finite: only one new location is created for each occurrence site (either object allocation or variable reference) in each context, and the number of contexts stops growing in all cases.

There are a finite number of paths through a method. The number of paths through a program is a function of the number of paths through each method in the program and the number of calling contexts in total. As both of these are finite, the number of paths represented by our shape analysis is finite. There is a maximum of one shape graph for each unique path through the program. In many cases, there are less than one, as the analysis disposes of duplicate shape graphs.

The shape analysis algorithm outlined in the previous sections is simple, powerful and precise, but computationally expensive. We chose to develop a simple and precise algorithm as it reduced the implementation effort required to create a proof-of-concept research tool. Also, the methods that AVT analyzes, such as Prototype’s clone method and Abstract Factory’s Factory Method are expected to be quite short in the most common cases. However, interaction invariants may be applied to much larger methods, including the whole program. For this reason, we consider some steps that could be taken to improve the scalability of the algorithm here.

4.2.4 Efficiency

Our algorithm is unlike most in the literature in one main respect: it retains the value of all binary predicates precisely throughout the analysis. This makes our algorithm computationally expensive. In this section, we consider approaches to making our algorithm more efficient, while still being able to provide the information necessary to verify object-state and data-structure invariants.

A salient point of variability among tools in the literature is how they *summarize* the heap: retaining some information regarding the heap precisely while only approximating some other information. Some approaches create a single location for each allocation site [Chase et al., 1990], i.e., they are context-insensitive. Other approaches limit the length of paths in the heap [Jones and Muchnick, 1979]. A third category summarizes all binary predicates by setting their value to be a single *abstract summary location* [Sagiv et al., 2002]

unless they can be distinguished by the value of the instrumentation predicates. In this way, properties, such as a linked list being cycle free, can be verified though information on the exact size of the list or the exact identity of its contents is not available. Implementing one of these three approaches would reduce the number of binary predicates occurring in shape graphs.

Also, Sagiv et al. [2002] provide a technique to merge sets of shape graphs into a single summary shape graph, removing the exponential factor of our analysis involving control-flow branching and meeting. Rinetzky et al. [2005], among others [Cherem and Rugina, 2007], calculate reusable method summaries, reducing the number of times the same method must be analyzed. Finally, Milanova et al. [2005] describe a parameterizable analysis that may represent the values of some variables more precisely, while aggressively summarizing information about other variables. This approach also reduces the number of predicates in each shape graph.

To summarize:

- The number of binary predicates per shape graph can be reduced by predicate summarization. The most suitable approach for our purpose may be some instantiation of the framework described by Sagiv et al. [2002], as some of their instrumentation predicates closely parallel Alas data-structure predicates. However, choosing a set of instrumentation predicates that is accurate while also being scalable is challenging.
- The number of shape graphs could be reduced by combining graphs at the meet point of control-flow paths.
- The number of times a method is analyzed can be reduced by generating and reusing method summaries.
- The number of predicates in each shape graph can be reduced by representing some subset of the variables in the program more precisely.

4.3 Alas clause verification

Verification of two of the three invariant categories requires the shape analysis algorithm described in the preceding section. Object-state and data-structure clauses are verified by querying the set of shape graphs output by the analysis. Dependency invariants refer to

$$\begin{aligned} \text{obj1 isAlias obj2} & \quad \Leftrightarrow \text{obj1 'must alias' obj2} \\ \text{NOT (obj1 isAlias obj2)} & \quad \Leftrightarrow \text{NOT (obj1 'may alias' obj2)} \end{aligned}$$

Fig. 4.7: The relationship between conservative Alias predicate verification and the choice of meet operator

the existence or non-existence of declarations or expressions in a class or method and do not require data-flow information for their verification.

4.3.1 Conservative verification

As software verification is an undecidable problem in general [Landi, 1992][Ramalingam, 1994], it is necessary that any analysis that guarantees termination in the general case must introduce some conservative approximations of the ‘true’ result, the result that would be produced by some ideal unrealizable verification approach. A central concern in software verification is ensuring that the conservative approximations are sound, i.e., that they preserve the program’s semantics and do not allow something that is false to be proven as true. Analyses are classified into two categories based upon their *confluence* or *meet* operator (the operator that defines how different data-flow facts are combined at control-flow meeting points). These two categories are ‘must’ and ‘may’ analyses. Must analyses require a property to be satisfied over all execution paths and may analyses require that a property is satisfied by at least one path only. For must analyses, the output is conservative if it is a subset of the true result. For may analyses, the output is conservative if it is a superset of the true result.

As Alias clauses may or may not be negated, the conservative result may fall into either category of analysis. For example, a negated `isAlias` clause is conservatively verified by a may alias analysis, while a non-negated `isAlias` clause is conservatively verified by a must alias analysis. This situation is summarized in Figure 4.7.

We address the need for must and may alias information in AVT by not merging shape graphs at control-flow meeting points, i.e., all paths are modelled explicitly and there is a shape graph for each unique path through the PUT. The output of the analysis contains the necessary information for a must and may analysis, similarly to Nielson et al. [1999]. The advantage of this approach is that a single algorithm could be used to produce the data-flow facts required to verify both negated and non-negated Alias clauses. The disadvantage of

this approach is that the analysis is exponential in the number of control-flow paths through the PUT.

4.3.2 Dependency

This section focuses on the `isInitializer` implementation dependency invariants, as a major contribution of Alas. An `isInitializer` clause states that a class or method should or should not contain a call to the constructor of a class or any of its subclasses. Verification of clauses containing the `isInitializer` predicate can be performed by exploring the AST of the method actor (or all methods of the class actor in turn), and comparing each constructor call to the class that is the second operand of the clause.

We implemented a single `BlockScanner` class that performs AST parsing and calls different `visit` methods on an `Analysis` class for each type of statement and expression. Each `Analysis` class overrides one or a number of `visit` methods to perform its function. The `InitializerAnalysis` class implements the `isInitializer` predicate and overrides the `visit` method for constructor call expressions only. The `TypeSymbol` class in Jikes represents classes and interfaces and provides an `IsSubType` method, which implements the most generic sub-typing relation, handling transitive closures of superclasses and superinterfaces as well as an array's compatible types. `InitializerAnalysis` calls `IsSubType` to perform its class comparison

4.3.3 Object state

This section focuses on `isCopy`, the predicate in Alas that specifies deep copying relationships. An `isCopy` clause states that the objects referenced by the two reference variable operands should be 'deep' copies of one another at a particular point in the control-flow of the method. The `isCopy` clause in the context of the Prototype pattern's clone method states that 'this' and the special variable 'returnval', which represents the value returned by the method, should be copies at the end of the method. This invariant is simpler to verify than the general `isCopy` clause, as it is applied to the final output of the analysis, but an invariant on any control-flow branching or meeting could be verified by stopping the analysis at that point and applying the invariant once the analysis for the sub-graph leading to that point had converged.

A pseudocode for the `isCopy` verification algorithm is given in Figure 4.8. The mechanism for accessing values by variable name (and object source, in the case of binary

```

GET the class of the current locations being compared
FORALL attribute of this class
  IF attribute is reference variable
    IF attribute is in copystate
      IF values of this attribute for the current locations are
      aliases
        FAIL
      ELSE
        GOTO line 1 with pair of values for this attribute
    ELSE (attribute is primitive)
      IF values are non-equal
        FAIL
  IF no FAILs encountered
    PASS

```

Fig. 4.8: Pseudocode for isCopy verification on a single shape graph

predicates) is straightforward and has been omitted. Aliasing is handled naturally in the analysis: variables with the same value are aliases. When primitive variables of different objects have the same value, one must have been assigned the value of the other, or both assigned the same value from some third variable or expression.

4.3.4 Data-structure

Due to the precision of heap modelling in our shape analysis, no instrumentation predicates are required to verify data-structure invariants. All the data-structure language elements below require the traversal of data structures conforming to data-structure definitions given in the Alas pattern specification. Traversals of data structures are handled naturally in AVT as the set of binary predicates in a shape graph can be searched by source location, and variable name and type: this is exactly the information provided in an Alas data-structure definition. Source locations in binary predicates whose class matches the class role defined in the data-structure definition are potential roots of data structures. If the selector variable of the binary predicate matches the link role of the data-structure definition, then the binary predicate does represent a link in the relevant data-structure. The target of the binary predicate can then be used to search for potential subsequent links in the structure using

the same procedure, and so on transitively, until no successor location can be found.

Verification specific to each of the data-structure invariant types supported by AVT are dealt with in turn below:

- **last**: a sequence of binary predicates is followed until a target location is found that is not the source location for any other predicate satisfying any link role in the data-structure definition. The target location at the end of the sequence is the last element in some data-structure.
- **isCycleFree**: sequences of binary predicates are traversed as above for **last**, but the value, i.e., identity, of each target location (as well as the initial source location) is recorded and compared to each subsequent target. Any two occurrences of the same location in a sequence is evidence of a cycle. This traversal is done depth-first, to allow a source to lead to multiple targets, as is the case when a collection is part of a data-structure definition.
- **isSharingFree**: the set of all binary predicates is iterated over, and those that satisfy any link role included in the **isSharingFree** clause have their target location recorded. If the same target location is added to this record twice, this is evidence of sharing.

4.4 Miscellaneous implementation issues

4.4.1 Synthetic method CFGs

When AVT encounters a node in the call graph, it accesses the source of the called method and analyzes the source directly to produce a CFG. This is not possible for some methods where the source is unavailable, such as some methods in class `Object` in Java. In these cases, we generate our own CFGs, which we refer to as **synthetic CFGs**, as they are generated without analyzing Java code. Synthetic CFGs are created based on documentation and their semantics as described in the Java Language Specification [Gosling et al., 2005]. We will describe the semantics of some of the more relevant synthetic CFGs here.

Object.clone

`Object.clone` creates a ‘shallow copy’ of the callee object. It creates a new object of the runtime type of the callee object and initializes each of the object’s attributes with the corresponding attributes of the callee object as if by assignment. Any reference variable of the callee object will be aliased with the corresponding variable in the clone. Inherited

attributes have their values copied also. Our synthetic CFG generates a direct assignment between all variables of the callee object and clone.

java.util methods

To limit compilation time and space requirements, only Swing was compiled instead of the entire Java 2 Software Development Kit [Microsystems, 2010]. Thus, a synthetic method CFG was required for each call to methods outside of Swing. Many of these were to utility methods, due to the widespread use of collections such as `Lists` and `Vectors`. Creating CFGs for these methods required some additional implementation effort, but added the advantage that collections could be handled without implementing array handling code.

4.4.2 Unimplemented features

A number of features were not implemented in AVT. Neither concurrency nor exception handling are handled by AVT. As concurrent programs can jump after any statement, and even in the middle of statements that contain multiple expressions, concurrency greatly increases the size and complexity of the call graph, while at the same time reducing the likelihood that properties can be decisively verified. We limit ourselves to demonstrating properties in the context of sequential execution, as discussed in Chapter 2. Similarly, exception handling mechanisms increase the complexity of the intra-method CFG, as jumps may occur from any potentially exception-throwing statement to its corresponding catch statement. Computing the potential exceptions thrown by each statement, and the catch block to which control would transfer in each case is not particularly complex, but would require a considerable implementation effort while adding numerous shape graphs to the output from which few properties could be verified (an exception will often cause some of the behaviour required to satisfy the invariant[s] to be skipped completely). Control flow in AVT CFGs proceeds unconditionally from a try block, to the finally block (if available) to the next statements immediately after the exception handling code, so not every potential path through the actual program is represented in the CFG produced by AVT.

Handling array mutation in general requires path-sensitive data-flow analysis, which is challenging and existing approaches either apply heuristics, are unsound or both [Dillig et al., 2008]. AVT does not perform path-sensitive data-flow analysis. Much of the use of arrays in programs depends upon indexing with integer literals or variables, and handling literal values precisely greatly increases the potential size of the data-flow facts [Cousot and Cousot, 1977]. As such, the handling of arrays is likely to be highly conservative

and imprecise. For these reasons, arrays in general are not supported by AVT. Despite not handling arrays, AVT is capable of handling collections through the use of synthetic method CFGs that abstract the underlying array representation.

Finally, support for a number of small features were not implemented due to time constraints, though most of the behaviour required to handle them was implemented to support other features already. Anonymous inner classes complicate the computation of correct dynamic dispatch as they may over-ride one or a number of their superclasses methods. This dispatch behaviour is not implemented in AVT, which can handle dynamic dispatch in non-anonymous contexts. Neither the instance nor static initialization blocks are analyzed when a new object is created. There is nothing particularly challenging about analyzing either, as they contain the same types of statements as normal methods, but they are encoded slightly differently in Jikes. Finally, the composition relation in the context of the `copystate` keyword is not computed by AVT, but it could be with minor extensions: each constructor of a class could be analyzed in turn and the variables that have allocation locations assigned to them could be added automatically to the `copystate`. Already, constructors are analyzed as part of the shape analysis when each new object is created.

4.5 Summary

This chapter described the design and implementation of AVT, with particular focus on the shape analysis algorithm required to verify object-state and data-structure invariants. We outlined the features of the algorithm that address each requirement imposed by Alas. These features are summarized in Table 4.2. We also considered how the algorithm could be made more scalable before describing the verification of novel Alas invariants at a high level.

Requirement	AVT design approach
Implementation dependency invariant verification	AST exploration, class comparisons
Object-state invariant verification	Shape analysis, special ‘locations’ for primitive variable values, recursive copy state comparisons of locations
Data-structure invariant verification	Shape analysis, Alas shape predicate queries on shape graphs
Mutation of encapsulated state via method calls	Context-sensitive call graph construction, context-sensitive allocation and summary locations
Conservative verification	Meet operator that maintains a shape graph for every unique path through the program under test, multiple summary locations for the same class

Table 4.2: A summary of the requirements imposed by the novel invariant categories of Alas and features of object-oriented programming languages, and the AVT design approaches addressing each requirement

Chapter 5

Pattern Specification and Benchmark

In this chapter, we specify each of the patterns that benefit from the novel invariant categories supported by Alas and create a benchmark based upon those specifications. The benchmark is used in Chapter 6 to evaluate AVT. A specification is provided to make the resultant benchmark reproducible. Creating the pattern specifications and benchmark address two evaluation objectives, namely:

- Expressiveness: Alas should be capable of describing the novel and poorly-supported design pattern invariants identified in our GoF analysis, as well as design pattern variants;
- Usefulness: The novel properties and variants specifiable by Alas should actually occur in existing code bodies.

The remainder of the chapter is structured as follows. Section 5.1 reviews the evaluation methodology of other approaches and outlines our own approach. Section 5.2 provides the actual benchmark, as well as the specification of invariants of patterns not provided in Chapter 3. Finally, Section 5.3 provides a summary of the chapter.

5.1 Benchmark Construction Methodology

This section discusses some key concerns when developing an evaluation methodology for DPSLs and DPVTs, with a particular focus on DPVTs. In each subsection, we first consider

aspects of DPSL and DPVT evaluation methodologies in the literature, before presenting our own approach. The evaluation of most DPVTs involves analyzing some piece of software using the tool and reporting the results of the analysis. We refer to the piece of software analyzed as the *code body*. Usually prior to analysis, a number of pattern instances are identified in the code body. One or a set of code bodies with a set of pattern instances identified in them is defined as a *benchmark*.

5.1.1 Code Body Selection

Tsantalis et al [Tsantalis, 2009] state that desirable characteristics of code bodies are that they ‘rely heavily’ on design patterns (i.e., have a large number of pattern instances per class or method), have documented pattern instances [Ng, 2008] and have publicly-available source [Pettersson et al., 2009]. Another desirable characteristic suggested in the DPVT literature is that the code bodies within a benchmark should vary in size [Tsantalis, 2009][Pettersson et al., 2009]. Blewitt [Blewitt et al., 2005] states that the code body should include ‘real world examples’, i.e., should not be written by the tool developers themselves. Finally, Guéhéneuc and Antoniol [2008] state that being able to compare results against other DPVTs is desirable and affects the choice of code body.

Shadish et al. states that “most experiments are highly local but have general aspirations” [Shadish et al., 2002](p18). An experiment may implement and test only one instance of some class of system or analyze only a sample of data from a much larger dataset. The problem of *external validity* relates to the ability to infer whether a causal relationship holds over variations in application, such as a different instance of some class of systems or a different dataset. Demonstrating the external validity of an experiment improves the level of confidence in a tools applicability, one of the evaluation goals we stated above.

There is not a one-to-one mapping from pattern specifications to implementations: there is often a number of ways to correctly implement a behaviour. For example, in Java, mutually-exclusive control-flow could be implemented using `if...else` blocks; jump statements, such as `break` or `return`, nested within conditional blocks or with `switch` statements. We refer to the set of all correct implementations of a particular pattern specification as the pattern’s *signature*.

It is possible that the implementation of the DPVT is tuned to produce correct results for just the code body or pattern instances analyzed or that instances were chosen that show the tool in a good light. It is possible that such a DPVT would perform badly on other

Code body	Occurrences in DPVT evaluations
JHotDraw	12
AWT	7
JRefactory	5
JUnit	4
QuickUML	3

Table 5.1: Code bodies and how often they occur in the evaluation of existing DPVTs

code bodies. Each code body has its own implementation idioms (termed *implementation variation* in [Antoniol et al., 2001]), as well as common design pattern variants. For example, the widespread use of the `ObserverList` class in Swing leads to many instances of the encapsulated observer list variant of the Observer pattern in that code body. JHotDraw makes widespread use of iterator classes. To help address the problem of external validity, and to demonstrate the applicability of a DPVT, we believe that the code bodies selected for inclusion in a benchmark should not just include a large number of valid pattern instances, but should include a large number of different members of the pattern’s signature. We added this selection criterion to the criteria from other DPVT evaluations.

From 19 DPVT evaluations, we identified the most commonly-used code bodies. Our findings are summarized in Table 5.1. Unfortunately, the same version of these code bodies was not used by all evaluations. The most commonly used versions of JHotDraw were versions 5.1 [Guéhéneuc and Antoniol, 2008][Tsantalis, 2009][Kaczor et al., 2006][De Lucia et al., 2009] and 6.0b1 [Shi and Olsson, 2006] [Stencel and Wegrzynowicz, 2008][Alnusair and Zhao, 2010] [Ng et al., 2010]. The most commonly used version of AWT was version 1.4 [Beyer et al., 2003] [Pettersson et al., 2009]. We selected JHotDraw version 6.0b1, Java Swing v1.4.2 and JUnit v3.7 as the code bodies for our benchmark. JHotDraw is well known for its use of design patterns, some pattern instances are documented, it has a relatively large number of pattern instances and is the most commonly used code body in DPVT evaluations. Swing v1.4.2 has a large number of pattern instances, and is analyzed by two other approaches [Shi and Olsson, 2006][Pettersson et al., 2009]. JUnit v3.7 has a large number of pattern instances for its size and some of these instances are documented. The three code bodies vary in size: Swing is the largest with 1,540 classes, JHotDraw has 744 and JUnit is the smallest with 156. All three have source code that is publicly-available.

After creating our benchmark, it was decided that these three code bodies did not completely exercise the full code signature of each of the pattern specifications we wished to focus on, so we developed our own small benchmark, in the same way that Stencil [Stencil and Wegrzynowicz, 2008] provided their own implementations for variants of the Singleton pattern in their benchmark. Similarly to SanD [Heuzeroth et al., 2003], we included in this benchmark a number of ‘fault patterns’, true negatives similar to correct implementations, to increase the external validity of the benchmark.

5.1.2 Inspection method

A benchmark may be constructed manually (by code inspection by a human) or automatically (by a tool), but most are constructed with a combination of the two approaches. Some evaluations involve no benchmarking being performed before analysis, but rather allow the analysis to form a starting point for the benchmark, with false positives being removed after manual inspection [Shi and Olsson, 2006][Guéhéneuc and Antoniol, 2008]. Such an approach has the advantage of being fast and having complete code coverage (if the tool is run over the entire code body). A significant disadvantage of this approach is that it ignores false negatives completely, artificially improving the recall results. Antoniol et al [Antoniol et al., 2001] use a version of their tool with slightly relaxed pattern specifications to build a benchmark, reducing but not completely removing the disadvantage described above.

SanD [Heuzeroth et al., 2003] and Kramer and Prechelt [1996] perform a keyword-based search of the code body. Numerous patterns have common naming conventions for roles (e.g., clone method in the Prototype pattern) or common coding idioms (many Factory Methods in Java will contain the code segment `return new`). Candidate actors identified by keyword-based searching can then be classified as instances using manual inspection. Keyword-based searching is another fast method of covering all the code within a code body, but is reliant on the existence of naming conventions and idioms.

Manual inspection could be performed on every class, considering each class for all possible pattern roles. Such an approach would take a prohibitively long time for all but the smallest benchmarks, and to our knowledge, has not been performed in any DPVT evaluation. In summary, each benchmark construction method has its own advantages and disadvantages, but none of the methods are mutually-exclusive and we consider them complementary when used in conjunction with each other, as some mitigate the disadvantages

of others.

One reason for naming entities is that it facilitates the comparison of results in different studies if the studies refer to the same name. Shadish et al. [Shadish et al., 2002] use the term *construct validity* to describe the problem of making valid inferences from the particular instances on which data was collected to the higher order constructs that those instances aim to represent. In the context of the evaluation of DPVTs, the key constructs of interest are design patterns. However, languages and tools differ in the roles and invariants that represent a given design pattern name. One tool might require, for example, an abstract Observer superclass in a valid instance of the Observer pattern, while another might not. One tool might require a Composite implementation to iterate over a list, while another may only require a 1-to-n aggregation relationship.

Comparing the results of two tools while analyzing the same pattern name with different representations of the pattern is not meaningful: a true positive according to one approach may be classified as a false positive by another approach. Thus, inferences about the relative merit of DPVTs that differ in their representation of patterns lack construct validity. Most new DPSLs and DPVTs innovate not just by changing pattern specifications but also by adding new roles and invariant categories to pattern specifications. Requiring that each tool addresses the same specification would preclude innovation. The problem of construct validity limits the re-usability of benchmarks, as the pattern specifications used for the benchmarks differ between approaches. This is a major barrier to the goal of a shared benchmark within the DPVT literature [Wegrzynowicz and Stencel, 2009][Fulop et al., 2008][Pettersson et al., 2009][Arcelli et al., 2008]. Rasool et al. [2011] identify instances in their benchmark which are also contained in other benchmarks. Selecting and analyzing the same pattern instances as existing approaches facilitates the comparison of analysis results between tools and also indicates that the benchmark is not biased to suit a particular tool’s capabilities.

We aggregated the information from two existing benchmarks: one manual [Guéhéneuc, 2007] and one fully automated [Shi, 2007b]. These will be referred to in the remainder of the chapter as the P-MARt [Guéhéneuc, 2007] and PINOT [Shi, 2007b] benchmarks. The benchmark of Pettersson [2010] could not be used, as it does not address any of the patterns that we focus on. We found that some but not all of the pattern instances identified therein satisfied the corresponding Alas specification. Similarly to Rasool et al. [2011], we identify the pattern instances that are common to our benchmarks and the aggregated

benchmarks. We also performed a keyword-based search of the code bodies, which yielded some pattern instances not included in the other benchmarks. All pattern instances were manually inspected for correctness. With this approach, we obtained the main benefit of an automated benchmark while avoiding its main disadvantage. To our knowledge, only one other DPVT evaluation in the literature performed both a tool-based and keyword-based search [Heuzeroth et al., 2003].

We construct a benchmark for only the patterns that demonstrate a major contribution of Alas and AVT. In some cases, we identified instances where comments, variable naming, or relationships between classes suggest a pattern instance was intended by the code body developer, but the instance does not satisfy the Alas specification. For example, all methods called ‘clone’ and returning an instance of the method’s containing class were recorded as intended instances of the Prototype pattern. These intended or failed instances are useful as they are true negatives that test AVTs ability to identify specification violations as well as specification conformance.

5.2 Benchmark

In the remainder of this chapter, we present the benchmark created based on our Alas specifications. We separate the results for the three invariant categories where Alas provides a contribution in their own section, and within these sections, each pattern relevant to that invariant category is given its own subsection.

In each section, we specify the structure and behaviour of each pattern that we address in detail, to demonstrate the capabilities of Alas, especially with regard to pattern variant specification. The subsections on individual patterns are organized as follows: first we describe our specifications of each of the pattern variants that were included in the benchmark and analysis and outline the keywords used in the keyword search. The actual number of instances of each pattern found is given in Appendix D. Each invariant category section ends with a brief discussion of findings.

Some interesting observations made during the benchmark that are not central to the presentation of the evaluation, and some additional specifications based on those observations, are provided in Appendix B. Appendix D discusses the benchmarks we aggregated, the similarities and differences between our specifications and theirs, and enumerate the total number of instances in the Alas benchmark as well as the instances common to Alas

Design pattern	Invariant type
Creational patterns	
Abstract Factory	Implementation dependency
Builder	Implementation dependency
Factory Method	Implementation dependency
Prototype	Deep copying, Implementation dependency
Structural patterns	
Composite	Data-structure
Decorator	Data-structure
Behavioural patterns	
CoR	Data-structure
Command	Implementation dependency
Iterator	Implementation dependency
Memento	Deep copying
Observer	Deep copying
State	Implementation dependency
Strategy	Implementation dependency

Table 5.2: Novel invariant categories in Alas required for each of the patterns in GoF design pattern catalogue specifications

and the aggregated benchmarks.

5.2.1 Dependency

As shown in Figure 5.2, the Abstract Factory, Builder, Command, Prototype, Factory Method, State, and Strategy patterns benefit from the novel dependency invariants expressible by Alas. The Prototype pattern is dealt with in the section on object-state (Section 5.2.2). The Client class roles in the Prototype pattern instances in the benchmark are similar to the classes tested in the context of the other patterns, so analysis of the Prototype pattern in the context of dependency invariants has been skipped to avoid repetition. The Abstract Factory, Factory Method and Command patterns are dealt with below, while the remainder are discussed in Appendix B, as few instances of them were found.

5.2.1.1 Abstract Factory/ Factory Method

We address the Abstract Factory and Factory Method patterns together, as each instance of the Abstract Factory pattern must contain one or a number of Factory Method instances. The structure required for a Factory Method instance is also a subset of the structure required for an Abstract Factory instance, as can be seen in the GoF pattern catalogue [Gamma et al., 1995, p.88,108].

We specified four variants of the Abstract Factory pattern. The first is analagous to the GoF OMT diagram for the pattern (*GoF* variant, the core variant) [Gamma et al., 1995, p.88], requiring an `AbstractFactory` and a `ConcreteFactory` role, as well as `AbstractProduct` and `ConcreteProduct` roles. The second variant removes the `AbstractFactory` class roles, allowing a single class (`ConcreteFactory`) to define the Factory Method interface and provide an implementation (*no AF* variant). Both of these variants are specified in Figure 5.1. The third and fourth variants remove the `AbstractProduct` and `ConcreteProduct` roles, and have the Factory Method return an instance of the `ConcreteFactory` class. Similarly to the first two variants, one requires an `AbstractFactory` class (*Self factory* variant) and the other does not (*Self factory no AF* variant).

The behaviour of the Factory Method role is the same in all variants, and was given in Figure 3.13 of Chapter 3. A client is any class that calls the Factory Method or a constructor of the Product or any of the `ConcreteProduct` classes¹. A client that only calls the Factory Method is referred to as a ‘good’ client (as it satisfies the key dependency invariant), while

¹The `isInitializer` clause can be applied to individual client methods also.

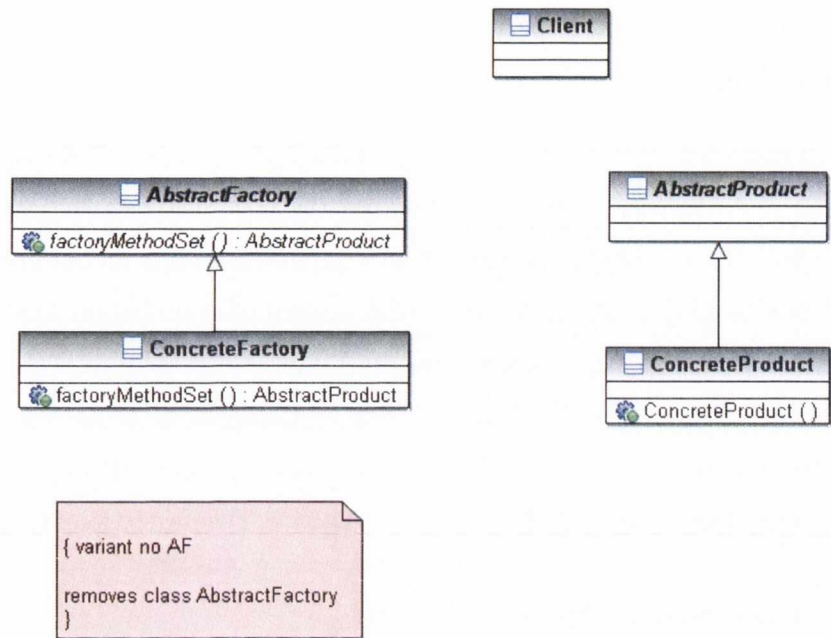


Fig. 5.1: Alas specification of the GoF (core) and No AF variants of the Abstract Factory pattern

a client that calls the constructor of a ConcreteProduct class directly is referred to as a ‘bad’ client.

While some Factory Methods are intended to be used by the entire code body, others are only intended for use by some small set of clients, or even just the Factory Method defining class itself. The decision on how widely a Factory Method should be used within a code body is orthogonal to the evaluation of a DPVT. We classify as a ‘bad’ client any class or method that creates an instance of a Product class for which there exists a Factory Method within the code body, i.e., we imply that the entire code body should use the same Factory Method if it wishes to create an instance of any Product class². We choose this widest possible scope (whole code body) for each Factory Method, as this provides us with more instances of ‘bad’ clients for AVT to analyze.

Keyword search and pattern instances

The keywords used to search for Abstract Factory instances were: ‘factory’, ‘kit’ and ‘make’. Due to the size of the Swing code body, it was not possible to complete a search for the

²This means that some instances of the Factory Method clash, i.e., both return instances of classes within the same Product inheritance hierarchy and are thus ‘bad’ clients of each other.

‘create’ and ‘return new’ strings.

As we are concerned only with the dependency invariant between the Client and Product roles, we are concerned only with instances that have clients. For the AF and all other patterns, tables of instance counts are given in Appendix D. In JHotDraw, we identified 45 Clients of 23 separate Abstract Factory pattern instances, with 26 Clients that satisfy the dependency invariant and 19 that violate it. In JUnit, we identified 10 Clients of 10 separate Abstract Factory instances, with 8 Clients that satisfy the invariant and 2 that violate it. In Swing, we identified 32 Clients of 12 separate Abstract Factory instances, with 17 Clients that satisfy the invariant and 15 of which violate it. The large number of classes violating the invariant demonstrates that Factory Methods, even when created, are not used consistently throughout code bodies. The addition of the novel implementation dependency invariants in Alas could benefit the non-functional properties of these systems by highlighting when a Factory Method could be used.

5.2.1.2 Command

The intent of the Command pattern is to ‘encapsulate a request as an object, thereby letting you parameterize clients with different requests’. This allows requests to be made ‘without knowing anything about the operation being requested or the receiver of the request’ [Gamma et al., 1995, p.233]. The object encapsulating the request is the Command, and it separates the Invoker and Receiver roles. A Client class initializes both ConcreteCommand and Receiver, decoupling Invoker from ConcreteCommand and also ConcreteCommand from Receiver. The Command pattern is not widely supported in the design pattern specification and verification literature. Blewitt et al. [2005] places the Command in the category of ‘semantic patterns’, which are beyond the capabilities of the SPINE language they define. However, we were able to identify a number of implementation dependency invariants that specify the intent of the pattern.

Our specification of the Command pattern is shown in Figure 5.2, with the three key implementation dependency clauses in the upper left constraint box. The first variant (*Command forms Façade*), defined in the lower left constraint box, adds the extra constraint that the Invoker role does not hold a reference to or call the Receiver. The other two variants relax the allowed coupling between the Command and Receiver. The *Command is Receiver* variant removes the Receiver role altogether, along with all structural definitions and clauses

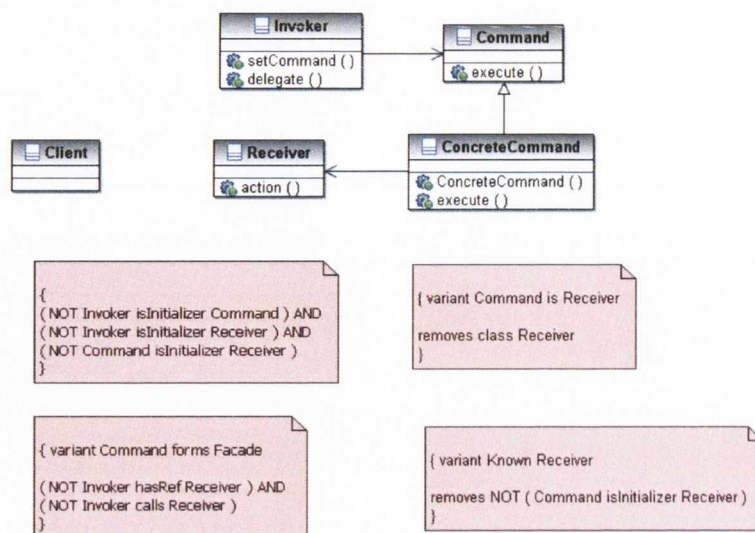


Fig. 5.2: Alas specification of the structure of all variants of the Command pattern

that refer to it. The final variant (*Known Receiver*) removes only the requirement that the Command does not initialize the Receiver. The inclusion of the last variant was prompted by our experience creating the benchmark. An intended instance of the Command pattern can be configured to use one of a set of Receivers, though it initializes these Receivers itself. The behavioural specification of the Command pattern is given in Figure 5.3, and simply involves two stages of delegation from Invoker, to Command, to Receiver. The behavioural specification of the Command is Receiver variant, which removes the Receiver role, involves simply the first of the two delegations described in Figure 5.3.

Keyword search and pattern instances

The keywords used to search for instances of the Command pattern were: ‘Command’, ‘Invoker’, ‘Receiver’ and ‘Execute’. The lack of an automated benchmark for this pattern is unfortunate, as it is likely a number of valid instances of the Command pattern were not recovered. We speculate numerous instances exist that do not follow the naming scheme of the pattern, because, despite a number of roles and implementation dependency invariants, the pattern’s signature is not very strong, as the behavioural specification involves straightforward unconditional delegation.

While Guéhéneuc [2007] identify a single instance of the Command pattern in the JHot-Draw code body (for all classes inheriting from the `AbstractCommand` class), we separate this instance into 5 different instances, as described in Appendix D.1.2.

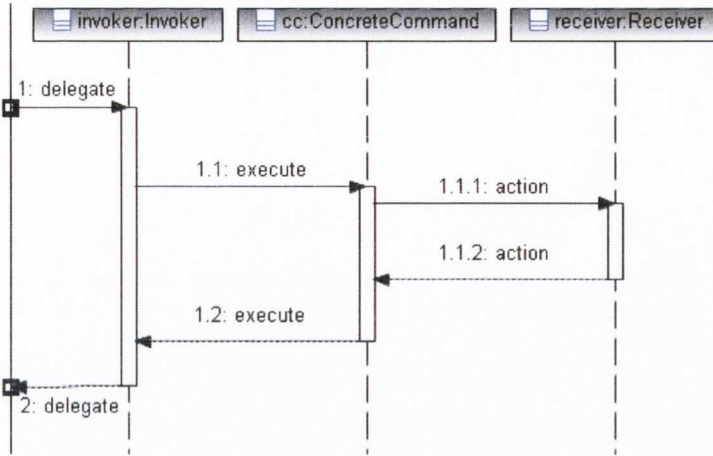


Fig. 5.3: Alas specification of the behaviour of the Command pattern

5.2.1.3 Discussion

With regard to the evaluation objectives:

- Expressiveness: Alas is expressive enough to describe implementation dependency invariants in the context of numerous patterns.
- Usefulness: We demonstrated that Alas is capable of specifying multiple variants of patterns, and, in the context of the Abstract Factory pattern especially, that these variants actually occur in existing code bodies

Abstract Factory

To our knowledge, we are the only approach in the design pattern specification and verification literature in placing constraints in the Client role of the Abstract Factory pattern. Alas specifications distinguish between variants that are combined in other benchmarks: we separate instances of the GoF and Self Factory variants listed by Shi [2007a], and we separate instances of the GoF and No AF variants provided by Guéhéneuc [2007]. With the exception of the Self Factory, No AF variant we find numerous instances of each of the variants specified. Also, in some cases, the inclusion of multiple variants allows us to identify more pattern instances in total than the aggregated benchmarks.

Numerous instances of clients were found that bypassed a provided Factory Method and instantiated the Product class directly, violating the implementation dependency invariant. This demonstrates that the application of this novel invariant in Alas could improve the architecture of software systems, by identifying when these dependencies are unnecessarily

created.

Command

Along with DeMIMA [Guéhéneuc and Antoniol, 2008], Alas/AVT is one of the few approaches that address the Command pattern. Unfortunately, without an automated benchmark, it was not possible to identify many instances of the pattern in code bodies. Of the instances found, it was clear that there were significant variations in the implementation of the pattern, even within the same inheritance hierarchy.

5.2.2 Object State

As shown in Figure 5.2, the Prototype, Memento and Observer patterns benefit from the novel object state invariants expressible by Alas. The following subsection deals with the Prototype pattern, as more instances of it that benefitted from the novel invariant categories were found. The Memento and Observer patterns are dealt with in Appendix B.

5.2.2.1 Prototype

The Prototype pattern ‘specif[ies] the kinds of objects to create using a prototypical instance, and create[s] new objects by copying this prototype’. A key invariant of the Prototype pattern is that the copy and original can be mutated without affecting each other, i.e., the values that are not intended to be shared between the copy and original should not be aliased.

We specified two variants of the Prototype pattern, one with a Prototype class declaring a clone method that returns an instance of the Prototype class, and a ConcretePrototype classes inheriting from Prototype and implementing the clone method (*GoF* variant). The second variant does not require the Prototype class role (*No AP* variant). We created the second variant to increase the likelihood of pattern instances to analyze. The behaviour of the clone method is specified in Figure 5.4. The clone method’s post-condition requires the value returned to be a copy of ‘this’, where the meaning of copy for each Prototype instance is defined by the `copystate` definition.

Keyword search and pattern instances

The keywords used in the search for the Prototype pattern were: ‘Prototype’ and ‘Clone’. We identified each method with the prefix ‘clone’ that returned an instance of its containing class. Also, during the creation of the Abstract Factory benchmark for Swing, the `create` method `DebugGraphics` was found to have the intention of creating a deep copy of ‘this’.

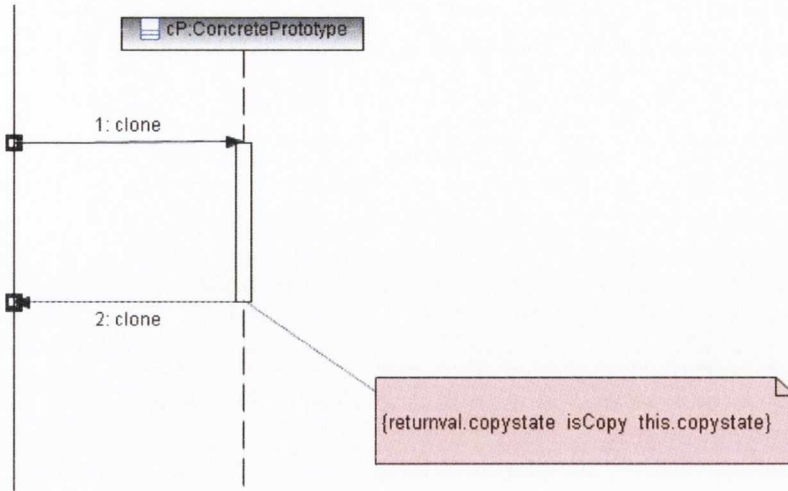


Fig. 5.4: Alas specification of the Prototype’s clone method. The clone method is required to make and return a copy of ‘this’

variant	JHotDraw	Swing
GoF	2	2
No AP	2	7

Table 5.3: Intended instances of variants of the Prototype pattern in the Alas benchmark for each code body

Table 5.3 summarizes the intended instances of each variant identified in the code bodies.

To identify if the relation-based `copystate` definitions described in Chapter 2 match deep copying behaviour occurring in practice, we compare the `copystate` included by the definition and the actual state copied by each candidate clone method instance identified in our benchmark using manual inspection. It was found that `iComp` (`copystate` definition that requires initialization of `copystate` only) was the best indicator of association relationships: relationships where the variable should be aliased between original and copy objects. `iComp` correctly predicated aliased state in 71.43% of candidate instances. However, it predicated aliased *and* deep copied state in only 42.86% of cases. In numerous cases, variables are initialized with newly-created objects that are not assigned any state from the original. An example of this in the Swing code body is the `clone` method of `OptionListModel`, shown in Figure 5.5. `OptionListModel`’s `BitSet` variable `value` is deep copied by calling `clone` on it, while the `ListenerList` variable `listenerList` is merely initialized

```

/**
 * Returns a clone of the receiver with the same selection.
 * <code>listenerLists</code> are not duplicated.
 *
 * @return a clone of the receiver
 * @exception CloneNotSupportedException if the receiver does not
 * both (a) implement the <code>Cloneable</code> interface
 * and (b) define a <code>clone</code> method
 */
public Object clone() throws CloneNotSupportedException {
OptionListModel clone = (OptionListModel)super.clone();
clone.value = (BitSet)value.clone();
clone.listenerList = new EventListenerList();
return clone;
}

```

Fig. 5.5: OptionListModel’s clone method in the Swing code body. The ListenerList variable is not made an alias or assigned a deep-copied object

with a new object. The comments shown indicate that this is the intended behaviour: ‘listenerLists are not duplicated’. None of the relation-based copystate definitions consistently distinguished between variables that have their values deep copied and variables that are assigned to new objects. Thus, the relation-based definitions do not closely match the behaviour of deep copy methods, and may provide inaccurate results when used in a reverse engineering use case.

We classify candidate instances as conforming or non-conforming based on their satisfaction of the *iComp* copystate definition. The choice of copystate definition does not affect the number of reference variables that AVT is required to analyze: each copystate definition classifies all reachable state as satisfying an association or composition relation with the object that contains it. Comments and documentation are not available for all candidate instances, so a user-based copystate definition cannot be inferred from the source code. In JHotDraw, 2 clone methods conform to the definition, and 2 do not. In Swing, 4 methods conform to the definition, and 6 do not.

5.2.2.2 Discussion

Instances of both variants of the Prototype pattern were identified in the code bodies included in our benchmark, though there is a small number of instances of the Prototype in total, which is not surprising, due to its specific intent and strong signature. With respect to the evaluation objectives:

- Expressiveness: Alas is capable of expressing deep copying invariants through the use

of the `isCopy` keyword and the relation- and user- based definitions of `copystate`.

- **Usefulness:** The inclusion of deep copying invariants in Alas allows for the precise specification of the Prototype pattern: a pattern with a weak code signature in existing DPSLs and DPVTs. The three relation-based `copystate` definitions address concepts such as ownership and initialization, which are the most frequently-used concepts used to define composition relationships. However, these `copystate` definitions only partially matched the copying behaviour occurring in the benchmark. Either a more sophisticated definition of relation-based `copystates` is required, or there is no consistent code signature for the state of an object that requires deep copying, re-initialization or aliasing in general. The latter case would suggest that reverse engineering instances of the clone method of the Prototype class, and deep copying invariants in general, is fundamentally limited, and a forward engineering use case, with a user-based `copystate` definition is more suitable in this context.

5.2.3 Data structure

As shown in Figure 5.2, the Composite, Decorator and Chain of Responsibility (CoR) patterns benefit from the novel object state invariants expressible by Alas. The following subsection deals with the benchmarking and analysis of the Composite and Decorator patterns. Few CoR pattern instances were identified, and it is discussed in Appendix B.

5.2.3.1 Decorator

The Decorator pattern ‘attach[es] additional responsibilities to an object dynamically’ [Gamma et al., 1995] in a way that is transparent to clients. This is achieved by inserting a decorator object between decorated object and client, with the same interface as the decorated object.

We specify three variants of the Decorator pattern. The first variant (*GoF* variant) has a similar structure to that given in the GoF catalogue, including a Decorator and ConcreteDecorator role. As in other patterns, we remove the requirement that Decorator is abstract: we found this restriction greatly reduced the number of conforming instances found in code bodies in most cases³. The second variant (*No sub* variant) removes the ConcreteDecorator role. This variant is proposed in the GoF catalogue for cases when the

³The key role of the abstract Decorator class, as with other abstract classes in the GoF catalogue, is to define a common interface for subclasses, and this can be achieved by a non-abstract class as well

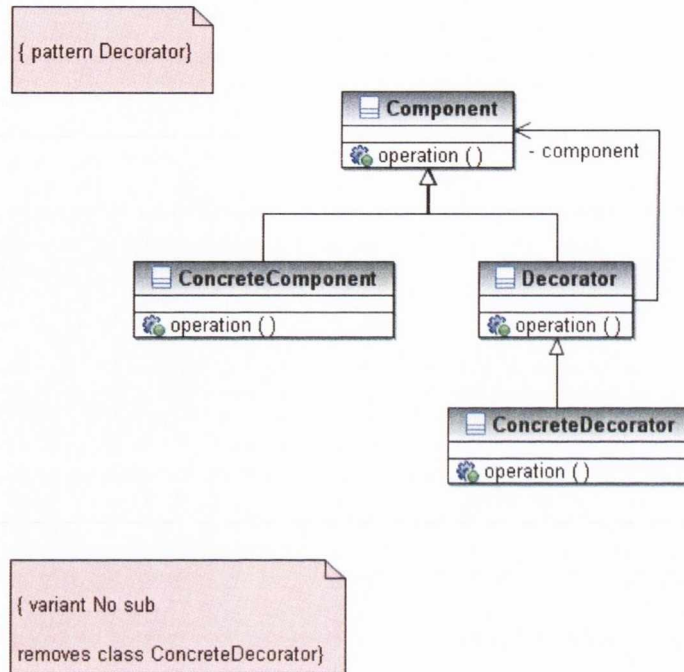


Fig. 5.6: Structural specification of the Decorator pattern, along with a single structural variant

Decorator class ‘add[s] one responsibility’ only. The structural specification of the Decorator pattern is shown in Figure 5.6.

We found that some intended instances of the Decorator pattern in both JHotDraw and Swing forward to a Component only on the condition that the Component is not null. We created a *Forward if not null* variant to handle this case, as described in Appendix B.

The data-structure invariant included in the Alas specification of the Decorator pattern was given in Chapter 3 Section 3.3.3, but is reproduced here, along with the data-structure definition.

```

decoratorChain isStructure Decorator.component
decoratorChain.last isKindOf ConcreteComponent
  
```

Informally, it states that every instantiated chain of Decorators and Components should terminate with a ConcreteComponent object. Every chain terminating in a ConcreteComponent is known to be well-configured (WC), every chain terminating in a Decorator is known to be badly configured (BC) and chains terminating in Components of unknown

type (as they are not initialized within the method under test's body) is not provably well or badly configured (this third case is abbreviated to UC in the following).

Keyword search and pattern instances

The keywords used to search for Decorator pattern instances were: 'Decorator', 'Decorate', 'Delegate', 'Real', 'Underlying', 'Wrap' and 'My' (JHotDraw only). The convention of naming the decorated reference variable 'my...' is applied throughout much of the JHotDraw code body. This did not hold for the other two code bodies, and a search of their source code using this keyword was not performed. Similarly to the Abstract Factory benchmark above, we only include instances with clients that instantiate chains of Decorator and Component objects. In JHotDraw, we identified 13 Decorator instances, with 23 methods instantiating Decorator chains. Of these, 7 are provably well-configured. In Swing, we identified 10 Decorator instances, with 16 methods instantiating Decorator chains. Of these, one is provably well-configured. As the benchmark included no badly-configured chains, we added a 'fault pattern' instance by modifying a method from JHotDraw to badly-configure a chain. We also added two methods that instantiated well configured chains of Decorators drawn from the Swing code body.

5.2.3.2 Composite

The Composite pattern's intent is to 'compose objects into tree structures to represent part-whole hierarchies.' The Composite pattern involves a Composite object that contains a collection of objects of its Component superclass. The Composite is commonly referred to as the 'parent' and the Component objects it aggregates are its 'children'. Classes that inherit from Component but not Composite are Leaf classes. An `operation` method is declared in the Component class, and over-ridden in the Composite class. The `operation` method is important to the intent of the Composite pattern as it 'lets clients treat individual objects and compositions of objects uniformly.' In the context of the Composite pattern, the GoF catalogue considers a number of trade-off or variation points. One of these variation points is whether a child has a reference back to its parent. We specify a *parentLinks* variant to capture this variation point. The structural specification of the Composite pattern was given in Chapter 3, Figure 3.4. An Alas behaviour diagram specifying the `operation` method of the Composite role is given in Figure 5.7. The parent reference variable is declared in the Composite, as suggested by the GoF catalogue, but it could also be declared in the Composite subclass. An essential invariant of the *parentLinks* variant is that 'all children

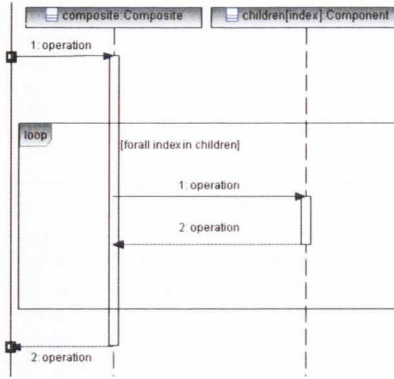


Fig. 5.7: Alas specification of the operation method of the Composite subclass

have as their parent the composite that in turn has them as children' [Gamma et al., 1995, p.166]. This can be ensured most safely by updating a Component object's parent only when it is added or removed from the Composite. We specify this invariant in the context of the `add` method in Chapter 3, Figure 3.17.

The second variation point we consider is where the child management methods, such as `add` and `getChildAt`, are declared. The GoF catalogue suggests a transparent or *unsafe* variant that declares the child management methods in the Component superclass. This variant is also specified in Figure 3.4. This approach has the advantage of allowing instances of all Component classes to be treated uniformly by clients, but has the disadvantage that a client may do something meaningless, for example, trying to add a child object to a instance of a Leaf class. Variants *No operation* and *No remove* are described in Appendix B.3.3.

The Composite pattern is used to build trees of objects, as each parent may have multiple children. To ensure correct traversal of tree structures, it is important that each object may be reached by only one path from the root object, i.e., no child should be shared by two parents. The data-structure definition and `isSharingFree` clause is given below.

```

componentTree isStructure Composite.children
componentTree isSharingFree
  
```

Keyword search and pattern instances

The keywords used to search for Composite instances were: 'Composite', 'Component', 'Parent', 'Tree', 'Leaf'. In total, 4 instances were found in the Swing code body, 2 in JHotDraw and 1 in JUnit. More details are provided in Appendix D.4.1.

To evaluate the verification of Alas `isSharingFree` clauses, we require non-trivial instantiations of trees of Composite and Component objects. In Swing, we identified four trees, instantiating instances of two different Composite pattern instances. In JUnit, we found 12 instantiations of the single Composite pattern instance identified, while no non-trivial tree instantiations were found in JHotDraw. Each of these instances are well-configured: they contain no cycles. Though this is a small number, it is perhaps unsurprising, as each of the code bodies are frameworks, that focus on providing a structure for applications to instantiate. Creating a benchmark from applications that use the code bodies included in our evaluation could extend the number of instances of Composite trees (as well as Decorator and CoR chains) to analyze, and would be an interesting direction for future work.

5.2.3.3 Discussion

With respect to the evaluation objectives:

- Expressiveness: Alas data structure invariants are sufficiently expressive to describe invariants of both linked-list-like structures and trees, such as the absence of cycles and sharing.
- Usefulness: Structural and behavioural invariants were identified in the context of both the Composite and Decorator pattern. Instances conforming to four different Composite variants were identified across the entire benchmark, while instances of three separate Decorator variants were identified. Some of these instances had non-trivial instantiations of data-structures, suitable for analysis by AVT.

Decorator

We address three variants of the Decorator pattern, two taken directly from the GoF catalogue and one based upon a common variation in the code bodies, in particular, Swing, included in our benchmark. Numerous instantiated Decorator chains were identified.

Composite

The specification of multiple variants allowed us to capture important design trade-offs discussed in the GoF catalogue. We were able to distinguish variants of patterns undistinguished in existing benchmarks and each of the variants was found to occur in some code

body. Even in our largest code body, Swing, only a few valid instances of the Composite pattern exist due to its strong signature and specific intent.

5.3 Summary

This chapter demonstrated the specification of the novel invariant types supported by Alas. We identified a list of evaluation objectives, as well as reviewing the DPSL/DPVT evaluations in the literature. We discussed properties of our benchmark and analysis that improve the generality of the results obtained. The remainder of the chapter treated specification of the three invariant categories that contain novel invariant types in Alas, and a benchmark based upon those specifications. Design pattern variants were discussed on a per-pattern basis.

Implementation dependency invariants were included in the specification of numerous GoF patterns. As one of the main focuses of the GoF catalogue is reducing coupling, implementation dependency invariants allow more of the intent of GoF patterns to be captured during specification. Instances of the Abstract Factory and Command pattern were identified. Numerous classes were found that violated the dependency invariant of the Abstract Factory pattern.

Deep copying invariants were evaluated mainly in the context of the Prototype pattern. This pattern has a weak code signature in existing approaches that are not capable of expressing deep copying invariants, as it is difficult to distinguish the Prototype's clone method from a Factory Method without the concept of composed state and copying. A small number of intended 'clone' methods were identified in the code bodies, some of which satisfied our specification and some of which violated it. The relation-based `copystate` definitions only partially matched the copying behaviour occurring in the benchmark.

Data-structure shape invariants specifiable in Alas allowed invariants novel in the literature to be expressed. The use of data-structure invariants in interaction invariants expanded the realm of design pattern specification and verification, from single methods performing a specific function in isolation or sequences of method calls, to potentially the entire program. Instances of the Composite and Decorator pattern were identified, as well as associated non-trivial instantiations of linked-list and tree shaped structures.

The ability of Alas to specify design pattern variants allows it to capture the intrinsic variation in GoF patterns. Structural and behavioural variation points are explicit in

Alas specifications, highlighting the trade-offs involved in implementing design patterns. Variants of GoF patterns were found to differ in terms of structural roles, dependency and object state. In numerous cases, the specification of variants allows us to identify more pattern instances than existing benchmarks, and also distinguish between variants that are indistinguishable in other approaches.

Chapter 6

AVT Verification Evaluation

In chapter 3, the features of the Alas language that enable the specification of invariants from the novel invariant categories were presented. Chapter 5 presented Alas specifications of patterns that benefit from the novel invariant categories, and a benchmark based upon these specifications. There are two main objectives in the evaluation of AVT. Firstly, to demonstrate that the novel invariant categories provided by Alas are actually verifiable and secondly, to demonstrate that the design of AVT presented in chapter 4 is appropriate for the verification of those categories of invariants in Java code. Appropriateness is represented using three properties: applicability, accuracy, and practicality. In more detail:

- **Verifiability:** AVT must demonstrate that the novel Alas invariants are actually verifiable, i.e, that it is possible to compare Java code to an Alas specification and obtain a result consistent with the Alas semantic model;
- **Applicability:** AVT is capable of verifying Alas design pattern specifications implemented using a variety of different implementation styles and idioms;
- **Accuracy:** AVT is capable of classifying pattern instances correctly, producing false negatives only for implementations containing highly-challenging features (the incompleteness inherent in all software verification approaches), and largely avoiding false positives (as this is the more serious case of unsoundness);
- **Practicality (and Scalability):** We document the running time needed and computing hardware used by AVT to verify each conforming or non-conforming implementation.

The evaluation objectives stated above are not addressed by different experiments, but rather different aspects of the same experiment. Including some number of true positives

and negatives addresses verifiability, while the further inclusion of false positive and false negative statistics addresses accuracy. As discussed in detail above, the characteristics of the code bodies and benchmarks address applicability.

6.1 Methodology

Two major methodological issues are discussed in this section. The first discusses which invariants from the specifications are actually verified by AVT. The second relates to the metrics used to evaluate accuracy and how they are applied.

6.1.1 Scope

For each pattern, we perform verification of the novel invariant type or types only. Other invariants imposed by each of the patterns addressed were not verified by AVT. As stated in the previous chapter, pattern instances were identified using manual (human) inspection, after some automated filtering. Interface dependency, cardinality and control-flow invariants have already received considerable attention in the literature, and the verifiability, applicability, accuracy and practicality of the tools addressing those invariant categories has already been demonstrated [Shi, 2007a][Blewitt et al., 2005][Tsantalis, 2009][Guéhéneuc and Antoniol, 2008]. The verification of structural characteristics such as method signatures and interface dependency has been demonstrated by many existing DPVTs, and much of the functionality required to verify these invariants, e.g., variable or return type checking, is required to verify the more challenging invariants verified in later sections of this evaluation also.

6.1.2 Metrics

The most commonly used metrics for DPVT evaluation are precision and recall [Guéhéneuc and Antoniol, 2008]. Both of these metrics are taken from the field of information retrieval and measure the accuracy of the tool. In the context of DPVTs, precision is the ratio of true positive instances obtained by the analysis to the true positives and false positives obtained. Similarly, recall is the ratio of true positives to true positives and false negatives. A false positive in this context is a non-pattern instance that is wrongly classified by the tool as a pattern instance. A false negative is a true pattern instance that is wrongly classified by the tool as a non-pattern instance. Some evaluations only identify true positives [Stencel and

Wegrzynowicz, 2008] [Heuzeroth et al., 2003], while others present only recall [Tsantalis, 2009][Wendehals and Orso, 2006] or only analyze known pattern instances (guaranteeing 0% or 100% recall) [Blewitt et al., 2005][Ng et al., 2010]. Shi [2007a] analyzes the entire code bodies but does not categorize identified instances as positives or negatives.

We use precision and recall to measure the accuracy of AVT analyses. Only inspected true positives and negatives are analyzed, instead of the entire code body¹. This does not preclude the occurrence of false positives or negatives, unlike some evaluations in the literature.

Where there are a lack of true positive or true negative instances in the benchmark based on existing code bodies, we add instances to our own benchmark to address the gap. However, precision and recall statistics for the external code bodies only are presented, to avoid skewing results.

6.2 Results

In the following section, we present the results of the AVT analysis of the benchmark outlined in the previous chapter. Results are grouped by invariant category, and by pattern within each invariant category. All analysis results were obtained using a desktop computer with two Intel Core 2 E 6600 @ 2.4GHz CPUs and 2GB of RAM running the Ubuntu Linux 10.04 operating system.

6.2.1 Dependency

In this section, we evaluate AVT’s ability to verify implementation dependency invariants. In the context of the Abstract Factory pattern, this involves analyzing the ‘good’ and ‘bad’ clients from the benchmark and comparing the result with the benchmark’s classification. Similarly with the Command pattern, except that three separate implementation dependency invariants exist between pattern roles (See Figure 5.2).

As described in Chapter 4, verifying implementation invariants involves searching the abstract syntax tree (AST) of a method or methods for constructor calls of a particular type. AVT performs an extensive exploration of the AST, so we expect its accuracy to be high. As AVT does not require data-flow analysis to verify dependency invariants, we expect it to scale well to larger class definitions. In summary, we expect AVT to verify

¹All data-structure instantiations are analyzed, so full coverage is achieved in that category

Code body	TP	TN	FP	FN	Precision	Recall
JHotDraw	26	19	0	0	100%	100%
JUnit	8	2	0	0	100%	100%
Swing	17	13	2	0	93.75%	100%

Table 6.1: AVT analysis results for the Abstract Factory pattern

implementation dependency invariants in an accurate and scalable manner.

6.2.1.1 Abstract Factory

Table 6.1 summarizes the results of the AVT analysis of the Client invariant of the Abstract Factory pattern for all three code bodies. Recall was 100% for all three code bodies. Precision was 100% for both JHotDraw and JUnit, and 93.75% for Swing. Both false positives in Swing were due to overlooked initializations within the instance initialization block, and both occur in the class `BasicTextUI`. For example, `BasicTextUI` creates an object of the class `DefaultEditorKit` directly in its instance initialization block, instead of calling the Factory Method `createDefaultEditorKit` of class `JEditorPane`. These results meet, and slightly exceed, our expectations, as we expected more relevant constructor calls to occur in the initialization blocks of class definitions.

Some Client classes are quite large, for example, `HTMLDocument` in the Swing benchmark, which was analyzed in full, contains around 1,400 lines of code and over 50 individual methods. It was analyzed by AVT in under a second. An example of an AVT analysis of a dependency invariant is given in Appendix C.1.

6.2.1.2 Command

Table 6.2 summarizes the results of the AVT analysis of all Command instances. Precision and recall are 100%, as AVT correctly classifies the four true positives and one true negative contained in the benchmark. None of the Command instances suffer from the issues that affected the results of the Abstract Factory analysis.

6.2.1.3 Discussion

This subsection presented the results of the benchmarking and AVT analysis of dependency invariants. We discuss here how well the language and tool performed with regard to each

Code body	TP	TN	FP	FN	Precision	Recall
JHotDraw	4	1	0	0	100%	100%

Table 6.2: AVT analysis results for the Command pattern

of the evaluation objectives outlined at the beginning of the chapter.

- The verifiability of dependency invariants was demonstrated through the correct classification by AVT of numerous true positives and true negatives.
- The classes to which dependency invariants apply cover a broad spectrum in terms of size and implementation styles. Some, such as `CommandButton`, are small and simple to analyze while others such as `HTMLDocument` contain around 1,400 lines of code, and were analyzed in full. Constructor calls in the benchmark occur within a number of different contexts within the AST, within return expressions, as the parameters to other calls and on the right-hand side of assignments. AVT was able to locate the call within the AST in each case.
- The accuracy of the AVT analysis was good. This matches our intuition that 100% precision and recall is achievable in the context of dependency invariants, as the verification involves the parsing of an abstract syntax tree (all of which is available) and the comparison of two types (for which complete name and scope information is available from the compiler).
- The verification of dependency invariants by AVT was demonstrated to be practical, as all analyses took less than a second running on the modest hardware platform described above.

6.2.2 Object State

In this section, we evaluate AVT’s ability to verify object-state invariants, specifically, deep copying invariants in the context of the Prototype pattern. AVT is used to compute whether or not the intended clone method instances identified in the benchmark correctly perform a deep copying operation. The results of the AVT analysis is compared to the benchmark’s classification, which contains both true positives and true negatives.

```

public Object clone() throws CloneNotSupportedException {
DefaultListSelectionModel clone = (DefaultListSelectionModel)super.clone();
clone.value = (BitSet)value.clone();
clone.listenerList = new EventListenerList();
return clone;
}

```

Fig. 6.1: DefaultListSelectionModel’s clone method in the Swing code body

6.2.2.1 Prototype

We illustrate the operation of AVT in the context of an example before providing the overall results. A more complex example is provided in Appendix C. The implementation of DefaultListSelectionModel’s clone method is shown in Figure 6.1. The first line calls Object.clone, which creates a new instance of the class as if by directly assigning all the variables of the original to the same variables of the clone. This is adequate ‘deep’ copy for all the primitive variables contained in the class. Lines two and three create new objects and assigns them to the two reference variables of the class. The first is created by a call to BitSet’s clone method, while the second simply creates a new instance of an EventListenerList.

An abridged version of the shape graph produced by AVT on analyzing this method is given in Figure 6.2. Core predicates (variables directly reachable from ‘this’) are displayed as pairs of variables and values while binary predicates are triples of: source object, variable name, and target object. As stated in Chapter 4, values are uniquely identified by the quadruple of integers: context, block, line, occurrence, so all variables with the same quadruple must point to the same value. The second core predicate ‘clone’ in the listing is the intended copy. It is the same value as the one returned by clone, as the special variable ‘returnval’, the last in the listing of core predicates, has the same quadruple: (1,1,0,0). The source of all binary predicates in the listing is the ‘clone’. It can be seen that the value of all primitive variables, while unknown, is the same in both the original, ‘this’, and the copy: the quadruples for ‘anchorIndex’, ‘leadIndex’ and ‘firstAdjustedIndex’ are identical in the core and binary predicate listings. The values pointed to by the two reference variables ‘value’ and ‘listenerList’ are different, due to the assignments in line 2 and 3 of clone.

The output of applying the isCopy predicate to the shape graph of Figure 6.2 is shown in Figure 6.3. AVT correctly asserts that all the primitives variables of the original and clone are equal and that the two reference variables are not aliased. A violation is identified as the two attributes of ‘listenerList’ have not been copied: two Object arrays called


```

Analysis result:
Number of ShapeGraphs: 1      Graph number:0

Core Predicates:
Variable: Name: this, Type: DefaultListSelectionModel, DUMMY
, Value: Type: DefaultListSelectionModel, Method: clone, Context: 0, Block Number: 0, Line Number: 0, Occurrence Number: 0
Variable: Name: clone, Type: DefaultListSelectionModel, REAL
, Value: Type: DefaultListSelectionModel, Method: clone, Context: 1, Block Number: 1, Line Number: 0, Occurrence Number: 0
...
Variable: Name: anchorIndex, Type: int, REAL
, Value: Unknown primitive value of type: int, Method: clone, Context: 1, Block Number: 1, Line Number: 7, Occurrence
Number: 0
Variable: Name: leadIndex, Type: int, REAL
, Value: Unknown primitive value of type: int, Method: clone, Context: 1, Block Number: 1, Line Number: 8, Occurrence
Number: 0
Variable: Name: firstAdjustedIndex, Type: int, REAL
, Value: Unknown primitive value of type: int, Method: clone, Context: 1, Block Number: 1, Line Number: 9, Occurrence
Number: 0
Variable: Name: value, Type: BitSet, REAL
, Value: Summary location for type: BitSet, Method: clone, Context: 1, Block Number: 1, Line Number: 14, Occurrence Number:
0
Variable: Name: listenerList, Type: EventListenerList, REAL
, Value: Summary location for type: EventListenerList, Method: clone, Context: 1, Block Number: 1, Line Number: 15,
Occurrence Number: 0
Variable: Name: returnval, Type: Object, DUMMY
, Value: Type: DefaultListSelectionModel, Method: clone, Context: 1, Block Number: 1, Line Number: 0, Occurrence Number: 0

Binary Predicates:
Source object: Type: DefaultListSelectionModel, Method: clone, Context: 1, Block Number: 1, Line Number: 0, Occurrence
Number: 0, Selector: Name: anchorIndex, Type: int, REAL
Target object: Unknown primitive value of type: int, Method: clone, Context: 1, Block Number: 1, Line Number: 7, Occurrence
Number: 0
Source object: Type: DefaultListSelectionModel, Method: clone, Context: 1, Block Number: 1, Line Number: 0, Occurrence
Number: 0, Selector: Name: leadIndex, Type: int, REAL
Target object: Unknown primitive value of type: int, Method: clone, Context: 1, Block Number: 1, Line Number: 8, Occurrence
Number: 0
Source object: Type: DefaultListSelectionModel, Method: clone, Context: 1, Block Number: 1, Line Number: 0, Occurrence
Number: 0, Selector: Name: firstAdjustedIndex, Type: int, REAL
Target object: Unknown primitive value of type: int, Method: clone, Context: 1, Block Number: 1, Line Number: 9, Occurrence
Number: 0
...
Source object: Type: DefaultListSelectionModel, Method: clone, Context: 1, Block Number: 1, Line Number: 0, Occurrence
Number: 0, Selector: Name: value, Type: BitSet, REAL
Target object: Type: BitSet, Method: clone, Context: 2, Block Number: 1, Line Number: 0, Occurrence Number: 0
Source object: Type: DefaultListSelectionModel, Method: clone, Context: 1, Block Number: 1, Line Number: 0, Occurrence
Number: 0, Selector: Name: listenerList, Type: EventListenerList, REAL
Target object: Type: EventListenerList, Method: clone, Context: 0, Block Number: 1, Line Number: 2, Occurrence Number: 0

```

Fig. 6.2: Shape graphs output by AVT after analyzing DefaultListSelectionModel's clone method

```

===== this isCopy returnval =====
Graph number: 0
'this' is not an alias of 'returnval'. Good!
'this -> selectionMode' has a primitive value equal to: 'returnval -> selectionMode'. Good!
'this -> minIndex' has a primitive value equal to: 'returnval -> minIndex'. Good!
'this -> maxIndex' has a primitive value equal to: 'returnval -> maxIndex'. Good!
'this -> anchorIndex' has a primitive value equal to: 'returnval -> anchorIndex'. Good!
'this -> leadIndex' has a primitive value equal to: 'returnval -> leadIndex'. Good!
'this -> firstAdjustedIndex' has a primitive value equal to: 'returnval -> firstAdjustedIndex'. Good!
'this -> lastAdjustedIndex' has a primitive value equal to: 'returnval -> lastAdjustedIndex'. Good!
'this -> isAdjusting' has a primitive value equal to: 'returnval -> isAdjusting'. Good!
'this -> firstChangedIndex' hdfgdfgas a primitive value equal to: 'returnval -> firstChangedIndex'. Good!
'this -> lastChangedIndex' has a primitive value equal to: 'returnval -> lastChangedIndex'. Good!
'this -> leadAnchorNotificationEnabled' has a primitive value equal to: 'returnval ->
leadAnchorNotificationEnabled'. Good!
'this -> MIN' has a primitive value equal to: 'returnval -> MIN'. Good!
'this -> MAX' has a primitive value equal to: 'returnval -> MAX'. Good!
'this -> value' is not an alias of 'returnval -> value'. Good!
'this -> listenerList' is not an alias of 'returnval -> listenerList'. Good!
listenerList undefined reference variable in both 'this -> listenerList' and ' returnval -> listenerList'.
No copying performed. Fail!
NULL_ARRAY undefined reference variable in both 'this -> listenerList' and ' returnval -> listenerList'.
No copying performed. Fail!
Graph number: 0: FAIL!

FAIL!

```

Fig. 6.3: Result of applying the isCopy predicate to the shape graphs of Figure 6.2

Code body	TP	TN	FP	FN	Precision	Recall
JHotDraw	2	0	2	0	50%	100%
Swing	2	5	2	1	50%	66.6%

Table 6.3: AVT analysis results for the Prototype pattern

'NULL_ARRAY' and 'listenerList'. The comments attached to DefaultListSelection-Model indicate that the 'listenerList's are not to be copied, so this is actually correct behaviour in the context of this particular instance. Nonetheless, this is a useful example to illustrate the recursive verification of the isCopy predicate in AVT.

The results of analyzing both the JHotDraw and Swing benchmarks is shown in Table 6.3. As the definition of both precision and recall has as numerator the number of true positives, both overlook the number of true negative instances identified by the analysis. The percentage of candidate instances that were correctly classified by AVT for the Swing benchmark (the only benchmark containing true negatives) was 70%.

The limited handling of arrays means AVT fails to verify that DefaultTreeSelection-Model correctly copies its selection array. Both DebugGraphics and FigureAttributes instances require handling of polymorphic calls to variables whose runtime type is unknown, and whose compile type is abstract and does not provide an implementation of the method. A solution to this issue is to add a call edge for each method implementation in a subclass

and analyze each of these. This is not difficult and involves no technique not already included in AVT. However, this approach has the potential to greatly multiply the number of shape graphs in the analysis, decreasing scalability. The copying performed by `AbstractFigure` is based on serialization, which is not supported by AVT. In a simple example such as this one, where the entire object is both written and read again in a single statement, the input and output streams could be modelled as if they contain objects of the written/read class. A synthetic summary of `readObject` could be provided that has a similar effect to `Object.clone`. Handling serialization in general, however, would require the analysis tool to track the ordering and contents of reads and writes to streams where one attribute is read or written at a time. Finally, to verify `AbstractAction.ArrayTable`, AVT would require the application-specific knowledge that its `table` attribute can be downcast to an `Object` array or `Hashtable` and that, in array form, it stores key and value pairs contiguously.

6.2.2.2 Discussion

This subsection presented the results of the benchmarking and AVT analysis of object-state invariants. With regard to each of the evaluation objectives outlined at the beginning of the chapter:

- The analysis of true positive and true negative instances demonstrate the verifiability of Alas object-state invariants. The shape analysis converged in all cases.
- Despite the small number of instances, many of the implementation idioms and language features that may be expected to occur frequently in implementations of the clone method are present. Instances were identified that: call `Object.clone` to create a new object (`OptionListModel`), do not call `Object.clone` (`DebugGraphics`), access state directly via 'this' (`SmallAttributeSet`), call clone methods on reference variables (`ElementIterator`), and call clone methods defined in their superclasses (`TriangleFigure`). In terms of language features, there are instances that create deep copies of lists, by iterating over each element in the list (`ElementIterator`) and perform copying using serialization (`AbsractFigure`). Also, there are instances that make polymorphic (`FigureAttributes`) and potentially-recursive calls (`DebugGraphics`). For this reason, we believe our benchmark of the Prototype pattern is representative of Prototype instances that occur in practice, and the analysis results obtained will generalize to other code bodies.

```

protected Tool createDragTracker(Figure f) {
    return new UndoableTool(new DragTracker(editor(), f));
}

```

Fig. 6.4: SelectionTool’s createDragTracker method from the JHotDraw benchmark

- Accuracy over the two code bodies was good. As a proof of concept tool, it does not implement all the language features of Java, which is a flexible high-level language. Some of the incorrect results are due to issues that could be solved without fundamental redesign of the shape analysis algorithm realized in AVT. Other features, such as the path-sensitive analysis required to verify `AbstractAction.ArrayTable` requires a different and challenging analysis approach. We believe this illustrates the challenge of verifying implementations in high-level languages in the general case.
- The analysis of every candidate pattern instance completed in under three seconds.

6.2.3 Data Structure

In this section, we evaluate AVT’s ability to verify data-structure invariants, in the form of interaction invariants. AVT is used to compute whether the Decorator chains and Composite trees identified in the benchmark are well- or badly-configured. The results of AVT’s analysis is compared to the benchmark, which contains both true positives and true negatives.

6.2.3.1 Decorator

We analyzed each of the Decorator chains in the benchmark with AVT. Results are discussed in terms of the client method that instantiates the Decorator chain and of the Decorator class. We illustrate the verification of data structure invariants in the context of an example before providing the overall results. A second example is provided in Appendix C. The source code for SelectionTool’s createDragTracker method from the JHotDraw code body is shown in Figure 6.4, and the corresponding AVT output is shown in Figure 6.5. AVT correctly identifies that the last object in the chain is an instance of a ConcreteComponent class (`DragTracker`), and the chain is thus well-configured.

Table 6.4 summarizes the results of analyzing the Decorator chain instantiations identified in JHotDraw and Swing using AVT. UC chains are classified as *negatives*, as they cannot be proven to be well configured. Failures or errors when analyzing UC chains are thus

```

Analyzing:
=====
->Tool myWrappedTool . LAST isKindOf UndoableTool OR
->Tool myWrappedTool . LAST isKindOf Tool
=====
ShapeGraph::getCorePredicateByName: couldn't find core predicate:myWrappedTool. Returning NULL. Danger here!
LastOperand::evaluate: Binary predicate number: 0 is valid link role
LastOperand::getLastInChain: Path ends
InitUtility::equalsString: DragTracker == UndoableTool ?
InitUtility::equalsString: mis-match
AbstractTool <: UndoableTool ?
Object <: UndoableTool ?
InitUtility::equalsString: DragTracker == Tool ?
InitUtility::equalsString: mis-match
AbstractTool <: Tool ?
InitUtility::inheritsString: type match via interface
Last element in chain is of ConcreteComponent type. Data structure is well configured.

```

Fig. 6.5: Result of applying the data-structure invariant to SelectionTool’s createDragTracker method

Code body	TP	TN	FP	FN	Precision	Recall
JHotDraw	7	12	2	2	77.7%	77.7%
Swing	2	9	1	1	66.6%	66.6%

Table 6.4: AVT analysis results for the Decorator pattern

reported as false positives. As the definition of both precision and recall has as numerator the number of true positives, both overlook the number of true negative instances identified by the analysis. The number of decorator chains correctly classified by AVT includes both true positives and true negatives: AVT correctly classifies 82.6% of decorator chains in JHotDraw, and 84.6% of decorator chains in Swing. In total, six instantiations were not correctly classified by AVT. Both the analysis of the `GraphicalCompositeFigure.createInstance` client method in JHotDraw and the `BasicTreeUI.createDefaultCellEditor` client method in Swing fail as AVT does not handle calls between different constructors using the syntax `this(...)`. This is a failure of the control-flow graph building code, and not the shape analysis: constructors are handled soundly in other cases. In Swing, `DefaultFocusManager`’s constructor, which is a client of the `LegacyGlueFocusTraversalPolicy` Decorator class, requires analysis of the instance initialization block, which is not analyzed by AVT, to produce the correct output.

In JHotDraw, `FigureAttributes.read`, which is a client of the `MapWrapper` Decorator class, requires more precise handling of polymorphic calls to abstract methods, where the method is implemented in one or more subclasses, as discussed in the context of the

Prototype pattern above. Finally, the analysis of two instantiations, of separate Decorator instances, both of which occur in the `JavaDrawApp.createTools` client method cannot be completed by AVT within a runtime of 5 minutes. The method contains the instantiation of 14 separate decorator chains, as well as numerous other objects of `ConcreteComponent` types, which are not summarized even in the parameterized-precision version of the shape analysis. Numerous `ConcreteComponent` objects are created over conditional paths, leading to a set of data flow facts including many shape graphs. Examples such as `JavaDrawApp.createTools` represent a worst-case scenario for the shape analysis algorithm implemented in the current version of AVT, though larger program segments, such as an entire program, may be expected to present similar challenges. The focus of future development of AVT could be on improving scalability in cases such as these.

One of the largest examples that does complete is the analysis of `DrawApplication.createEditMenu` method, which contains the instantiation of 10 well-configured decorator chains, each of length 2. The output of AVT contains a single shape graph with 45 core predicates and over 80 binary predicates. The program segment contains around 100 unique lines of code, some of which are analyzed many times, as there are a total of over 300 calling contexts: the method contains 27 constructor calls of different types, some of which call multiple local methods to perform further initialization; each of the newly-instantiated decorator chains is then added to a `HashMap` via a call to a mutator method on `CommandMenu`. This example demonstrates that AVT can precisely compute the runtime value of more than 100 program variables, created in numerous separate calling contexts, scaling well to program segments containing fewer conditional branches.

No badly-configured Decorator chains were encountered in either code body. A single badly-configured Decorator chain was added to our own benchmark by instantiating a chain of `UndoableCommands`, which terminates with an object of type `UndoableCommand`, i.e., the Decorator class. `UndoableCommand` allows this type of configuration by providing a mutator method for its delegate that performs no type checking on the argument that is set as its delegate. AVT correctly classifies the chain as being badly configured.

6.2.3.2 Composite

The results of the analysis are given in Table 6.5, first for the unchanged code body, and below it the starred version of Swing represents the version of the code body re-implemented with `Vectors`. Results for which AVT failed to complete due to scalability issues are

Code body	TP	TN	FP	FN	Precision	Recall
JUnit	7	0	0	5	100%	58.3%
Swing	1	0	0	3	100%	20%
Swing*	4	0	0	0	100%	100%

Table 6.5: AVT analysis results for the Composite pattern

classified as false negatives: AVT is incapable of demonstrating that the tree is sharing free. AVT correctly identifies that all instances in the modified code body are free from sharing. The source code for the modified version with added sharing is shown in Appendix C.3. Appendix C.3 also presents an example of AVT verifying an `isSharingFree` invariant in the context of the Composite pattern.

Three of the four instantiations of Composite trees in the Swing benchmark rely on arrays (three `createDefaultRoot` methods in subclasses of `AbstractDocument`). These were re-implemented to use `Vectors` instead of arrays in a semantics-preserving manner. AVT performs better on the Swing examples, as they contain fewer conditional updates. The program segment contained in `JTree`'s `getDefaultModel` contains around 60 unique lines of code (many of which are analyzed multiple times in different calling contexts) and the AVT analysis terminates after 2 seconds outputting a single shape graph containing 31 core predicates and 38 binary predicates. Also, the analysis involves a total of at least 326 calling contexts (this is the largest context number of a variable in the output). Similarly, the analysis of `HTMLDocument`'s `createDefaultRoot` method results in an output of 4 shape graphs, one of which contains 29 core predicates and 47 binary predicates, i.e., the value of 76 different program variables. The analysis handles greater than 176 calling contexts in 2 seconds. We modified `getDefaultModel` to introduce sharing.

6.2.3.3 Discussion

The ability of Alas to specify data-structure invariants novel in the context of design patterns was demonstrated in this section. These data-structure invariants capture critical correctness properties of instantiations of data structures inexpressible in existing DPSLs. These instances are specifiable by Alas, but not verifiable by AVT, as it does not perform path-sensitive data-flow analysis, as discussed in Chapter 4.

This subsection presented the results of the benchmarking and AVT analysis of data

structure invariants. With regard to each of the evaluation objectives outlined at the beginning of the chapter:

- AVT was demonstrated on conforming and non-conforming pattern implementations in the case of each pattern to demonstrate the verifiability of the Alas specifications. In the cases where there were no instances of specification violations, these were added from our own benchmark.
- While few non-trivial instantiations were identified in the code bodies, they are quite representative examples, as there are only a few ways to implement part-whole composition relationships. We identified examples of both arrays and collections intended to have contents that are instances of some superclass. We created our own small code body containing a few ‘fault pattern’ instances, as well as implementation variants of instances from the other code bodies, to more fully demonstrate the capabilities of AVT.
- Accuracy is good, for the analyses which converge. In all cases where the AVT analysis completes without error, it is able to correctly classify the Decorator chain as being WC, BC or UC.
- AVT analyzed many of the methods included in the evaluation in under a second or two. AVT was demonstrated to scale well to increasing numbers of variable updates and calling contexts and increasing lines of code, but was shown to scale poorly when the number of conditional branches increases, where conditional paths differ in the values they assign to variables. This situation increases the number of shape graphs that AVT must include in the data-flow facts propagating through the analysis. AVT scaled less effectively to the Composite examples in this section than the Decorator examples of the previous section, but this is a property of the benchmark rather than the pattern. The program segments instantiating Composite trees in this benchmark tend to involve more elements and include more complex behaviour than the program segments instantiating Decorator and CoR chains.

6.3 Summary

This chapter demonstrated the verification of the novel invariant types by AVT.

Verification of implementation dependency invariants by AVT was both accurate and efficient. The inter-procedural shape analysis algorithm implemented by AVT enabled the verification of deep copy object-state invariants by precisely representing graphs of objects in the heap and identifying sharing of objects between reference variables. Verification of these invariants by AVT was demonstrated to be accurate, where the language features occurring in the benchmark had been supported. The relationship between object structures reachable from different root objects can be computed efficiently. Data-structure invariants were verified accurately in cases where the analysis converged, but the application of AVT to larger code segments exposed a limitation in the scalability of the tool.

Typically, existing DPVTs analyze methods with a particular function that can be expressed as a binary property such as created/not created or null/not null, and are only required to model the method's local stack. Object state and data-structure invariants require precise modelling of the shape of the runtime heap. The verification of data-structure invariants in Alas is accurate, as the entire heap is modelled precisely with limited summarization. However, the analysis is not as scalable as other shape analysis algorithms that perform more aggressive summarization. While tracking all values occurring in a program segment precisely is suitable for typical deep copying object state invariants that are verified typically on small and medium sized methods, scalable verification of similarly complex properties over a larger program segment will require more efficient algorithms. We discussed approaches that would improve scalability in Chapter 4.

Chapter 7

Conclusions

In this chapter, we begin by discussing the specific contributions and conclusions of this thesis. We then consider the implications of our findings in the field of design pattern specification and verification. Finally, we propose future research enabled by this work.

7.1 Specific conclusions

A large body of literature exists that addresses the specification and verification of object-oriented design patterns, especially those outlined in the GoF pattern catalogue [Gamma et al., 1995]. Each design pattern imposes constraints that must be satisfied by a conforming implementation, and most patterns include trade-off or variation points where some alternative must be chosen to fully instantiate the pattern. The first contribution of this thesis is our analysis of the GoF pattern catalogue identified a set of invariant categories, each of which contains a set of invariant types that are necessary to specify the GoF patterns precisely and enable accurate verification of design pattern implementations in object-oriented programming language (OOPL) code. This analysis also identified a number of key variation points discussed in the GoF catalogue.

The second contribution of this thesis is a classification of existing design pattern specification languages (DPSLs) and design pattern verification tools (DPVTs) was performed according to which invariant categories each DPSL and DPVT supported. Some of the DPVTs included in the classification verify specifications written in a DPSL and some others are hard-coded with a pre-defined set of pattern specifications. It was found that numerous approaches addressed the structural roles and interface dependencies that are required. Control flow such as method calls and sequencing of events were also found to

be well supported. Two invariant types were found to be almost completely overlooked by the literature: deep copying behaviour and the shape of data structures at runtime. Some research has been conducted on data structure shape invariants in the context of design patterns, but this has either lacked a DPSL, specified only application-specific invariants or has major limitations in its generality. One invariant type was found to be addressed by a DPSL, but insufficiently supported by any DPVT and insufficiently evaluated on benchmarks of identified pattern instances: implementation dependency. Typically, dependency invariants in the literature focus on required or positive dependencies between pattern roles and not on forbidden or negative dependencies. One of the focuses of the GoF catalogue, however, is reducing the level of coupling between classes to improve extensibility, and because of this, the absence of different kinds of dependencies between roles is a key concept in design pattern specification.

Numerous approaches, especially earlier approaches to design pattern specification attempted to avoid the problem of design pattern variants by specifying only the ‘leitmotif’ of a pattern: the constraints common to all variants of a pattern. These approaches also tended to focus on structure. Such an approach provides a specification that has a weak signature in OOPL code, leading to the production of a large number of false positives or unintended instances being identified during verification. This approach also overlooks the behavioural variation of design patterns. To our knowledge, only a single DPSL in the literature, GEBNF [Bayley and Zhu, 2010], provides support for the specification of multiple pattern variants within a single specification and for the explicit naming of variants at variations points. GEBNF enables structural variation only, and its specifications are verified against object-oriented models only, and not code.

The third contribution of this thesis is the Alas DPSL, which is capable of specifying each of the invariant types, including the invariant types poorly-supported in existing DPSLs. We demonstrate the utility of the novel invariant types identified by our GoF analysis and supported by Alas by enhancing the specification of around half of the GoF design patterns using invariants from the novel invariant types. Alas is also novel in that it supports the specification of structural and behavioural variants of design patterns. In fact, it is possible to specify variations based on all the invariant categories in Alas, as variant specifications can refer to variations in structure, control-flow and the state of objects or data-structures at particular points in the control flow.

The fourth contribution of this thesis is the design and implementation of a DPVT called

the Alas Verification Tool (AVT), capable of verifying the novel invariant types specifiable in Alas. AVT performs an inter-procedural shape analysis of Java code, the output of which can be queried to verify object-state and data-structure invariants. AVT is also capable of verifying both positive and negative implementation dependency invariants through an exploration of the abstract syntax tree (AST) of a method, and the comparison of the type of objects created by expressions in the AST to class actors bound to roles in the specification.

The fifth contribution of this thesis is the creation of the first sizeable benchmark to include pattern variants, as well as providing the specifications used in creating the benchmark that makes its results reproducible independently. The aggregation of an automatic and a manual benchmark, as well as the performance of our own keyword-based search, conveyed a number of desirable properties on our benchmark. The aggregation of an automatic benchmark allowed us to achieve good code coverage relatively quickly. The inclusion of a manual benchmark, as well as the manual inspection of all instances, gave us more confidence in the validity of our benchmark than automated benchmarks, which suffer from errors or oversights in the implementation of the benchmarking tool. Code bodies for inclusion in the benchmark were chosen based on a number of criteria, including how often they were analyzed by other DPVTs, their size and their density of pattern instances. The specification of design pattern variants allowed for the identification of instances overlooked in other benchmarks as well as allowing us to distinguish between pattern instances that were not distinguished, i.e., combined into a single generic specification, in other approaches. The specification of invariants from the novel invariant categories allowed us to address patterns, e.g., Command and Memento, that are often overlooked by other approaches, which consider them as unspecifiable or too generic. Invariants from the novel invariant categories also allowed us to add precision to the specification of roles that are commonly specified in the literature, add roles that are not commonly addressed and specify novel properties that relate to the correct instantiation and use of design pattern roles. The ability to combine control-flow, object-state and data-structure invariants into the same specification allows more sophisticated concepts, such as lazy initialization, to be described based on the atomic set of invariants they impose, though the verification of such concepts in general has not been implemented.

The sixth and final contribution of this thesis was the evaluation of AVT, which demonstrated that the novel invariant types are verifiable accurately and in the context of the

varying implementation idioms occurring within the benchmark. The verification of implementation dependency invariants was found to be efficient and accurate. The development of a single analysis that can verify negated and non-negated clauses, deep copying behaviour and data-structure interaction invariants was found to be challenging. In particular, the shape analysis performed by AVT was found to have scalability limitations when applied to program segments that contained numerous conditional paths that differ in terms of the values they give to some pattern actor. Some approaches to improve the scalability of the tool were identified. The inter-procedural shape analysis algorithm implemented by AVT enabled the verification of deep copy object-state invariants and data-structure shape by precisely representing graphs of objects in the heap and querying those graphs.

7.2 General conclusions

While this thesis describes the ability to specify and verify both variants and the invariant categories poorly-supported in the design pattern specification and verification literature as separate contributions, they are in fact complimentary. Some pairs of variants can only be distinguished using the novel invariant types provided by Alas, and sometimes invariants from the novel invariant types are only relevant in the context of a particular variant. There exists a positive feedback loop whereby increasing the number of invariant types supported by a design pattern specification language increases the number of variants that can be described, and expanding the types of distinctions that can be made between variants (structural, control-flow, etc.) can increase the opportunity to express novel invariants. The combination of pattern variant specification and the novel invariant types serves to increase the precision with which patterns can be specified and benchmarks of pattern instances can be documented. Not all the specifications enabled by the novel features of Alas improve the precision and recall of the Alas Verification Tool, but instead add extra roles that were previously unspecifiable or had a weak signature, or constrain how a client accesses or instantiates already specifiable roles.

While the novel invariant types supported by Alas increase the precision with which patterns can be specified, numerous GoF patterns did not benefit significantly, and were largely excluded from our evaluation. Most of these patterns are structural and have a weak code signature in both their Alas specification and in the specification of other DPSLs. Examples of these patterns are Façade, Adapter and Mediator. DPVTs that have addressed

these patterns have found very large numbers of instances, which is likely evidence of many of these instances being unintended by the code body developer. These patterns resist precise description of their intents and are not amenable to a reverse engineering or ‘pattern mining’ use case.

As evidenced by our experiments, the specification of variants of design patterns increases the number of instances of a pattern that can be identified and captures valid variation points inherent in the design patterns structure or behaviour. However, some types of variations were found to be difficult to capture, especially in their most general sense. For example, the *Encapsulated child list* variant of the Observer pattern found commonly in the Swing code body splits the role of Subject into two objects, one which triggers notification when its state is updated, and one which holds the list of associated Observers (the latter are often instances of the `ListenerList` class). While this variant is specifiable in Alas, theoretically, any number of further levels of indirection could be added, and the implementation may still be considered a valid variant of the Observer pattern. Enumerating every valid splitting or merging of pattern roles would lead to a large and difficult to understand specification. Some work in the literature has considered general rules for allowable splitting and merging of pattern roles using sub-graph isomorphism (SGI) [Zhang et al., 2004], though this is applicable mainly to structural roles. Explicit variant specification and SGI rules can be complementary: SGI rules can reduce the number of variants that need to be explicitly specified and explicit variant specifications can identify allowable variations from a pattern’s core specification that cannot be automatically identified by SGI rules.

The benchmark of pattern instances we created from code bodies commonly analyzed in the literature has few non-trivial instantiations of data-structures. Also, numerous abstract classes and interfaces are defined that suggest by their name that they are intended to be instances of patterns, but lack subclasses or methods implementing behaviour required by our specification. Often, the code bodies commonly analyzed in the literature were chosen for their high density of design pattern instances, or at least interfaces. The code bodies that have tended to have the highest density of patterns are frameworks, which aim to provide an elegant and extensible structure of classes, each of which an application may chose to subclass or instantiate. However, the framework itself lacks implementations of abstract methods, subclasses of abstract classes or interfaces, method calls between associated classes and instantiations of objects and data-structures. The properties of the code

bodies analyzed may have influenced the design and focus of DPSLs. The novel invariant types, though their utility has been demonstrated for frameworks such as JHotDraw and Swing, may be even more useful when used in the context of applications that are built upon these frameworks.

7.3 Future work

The performance of DPVTs is difficult to evaluate due to the lack of construct validity of inferences about the relative performance of tools that encode different specifications of the same pattern name. Our specification of design pattern variants, and the creation of a benchmark based on those specifications, enables a more direct comparison between DPVTs, as it is more likely that the variants addressed by each tool can be identified. We do not claim that our specifications currently represent a superset of all variants supported in the literature, though we have identified the variant or variants of particular patterns addressed by DPVTs in some cases. The set of Alas specifications could be extended to include more of the variants addressed by other DPVTs, capturing the variation in the interpretation of a pattern's intent throughout the literature.

As stated earlier in this chapter, most of the code bodies commonly used to evaluate DPVTs in the literature are frameworks, which provide the structure required for pattern instances, but lack some behaviour that is required to satisfy the Alas specification. The creation of benchmarks from applications that use the commonly-analyzed frameworks would enable further measurement of the utility of the novel behavioural invariant types identified in this thesis, as well as stimulating research into the behavioural aspects of patterns, which have received less attention than structural aspects. To our knowledge, only Heuzeroth et al. [2003] has included a framework and an associated application in their benchmark. Also, the commonly-analyzed frameworks are old, and are coded in an older version of the Java programming language. Future benchmarks could focus on more recent frameworks developed in more recent versions of Java and that also continue to be utilized for application development. The Eclipse Framework [Eclipse, 2012] is a promising candidate, as it has documented pattern instances and a large body of applications based upon it.

Clearly the shape analysis algorithm implemented in AVT has limitations; in particular, its scalability to larger and more complex programs, especially programs with multiple

conditional paths that differ in terms of the values they give to state that is bound to a role in the Alas specification. A number of approaches to improving the scalability of the analysis were discussed in Chapter 4, and these could be implemented in future versions of the tool. Also, the efficiency of the implementation could be improved by tuning the performance of the code. While creating our benchmark, we identified some classes and methods that guarantee shape invariants in all cases, irrespective of the behaviour of clients. Path-sensitivity could be combined with shape analysis to address these interesting but challenging-to-verify invariants.

While the GoF catalogue of design patterns is by far the most popular catalogue specified in the literature, some DPSLs have been applied to other patterns and catalogues [Kim, 2004][Shetty and Menezes, 2011] from different domains. Future work could apply the novel invariant categories and the pattern variant specification mechanisms provided by Alas to the specification of other sets of patterns.

Finally, pattern specifications that specify variants using separate structure diagrams, substitution, and the removal and replacement of structural roles can make it difficult to visualize the specification of an individual variant. An interpreter and visualization tool could be developed to interpret valid and invalid combinations of pattern variants and generate diagrams to facilitate understanding.

Appendix A

Control-flow invariant semantics

The semantics of control-flow invariant specifications in Alas Behaviour Diagrams (BDs) is treated in more detail in the following appendix than in the body of the thesis.

A.1 Sequencing

The semantics of an Alas specifications and Java implementations to which they are compared are defined in terms of control-flow graphs.

A control-flow graph (CFG) is a set of basic blocks and an ordering relation (*successor*). A basic block (subsequently block) is a sequence of consecutive events in which control flow enters the beginning and exits at the end without the possibility of branching [Aho et al., 1986]. The ordering relation between blocks may relate 1-to-n and n-to-1 and may contain cycles. This relation can also be viewed as the edges in the graph of blocks where this view is more intuitive. Each graph contains an initial and a final block. The final block is the only block with no successor and no block has the initial block as its successor.

A path is a series of blocks formed by beginning at the initial block and following the edges of the graph until the final block is encountered (cyclic paths are taken only once, so all paths terminate). All paths through the graph begin at the initial and end at the final block. We define P to be the set of all unique paths, such that no two elements of the set have the same set of blocks, every block is included in at least one path and all valid paths are included. Such a set can be imagined as the output of a depth-first traversal of the control-flow graph where a new element of the set is created each time the final block is encountered.

Every specification, implementation and implementation path has a set of *Events*, ac-

cessible using $\langle Path - path\ set \rangle.Events$. The specification and candidate implementation CFGs are referred to as *Spec* and *Imp* respectively. The ordering relation between events (\rightarrow) indicates that an event occurs before another event. We define universal behaviour and sequencing of events by stating that every event in the specification occurs over every path of the implementation and over every path in the implementation, the events occur in the order defined in the specification.

$$\begin{aligned} &\forall e \text{ in } Spec.Events \bullet \\ &\quad \forall path \text{ in } Imp.P \bullet \\ &\quad e \in path.Events \end{aligned}$$

$$\begin{aligned} &\forall a, b \text{ in } Spec.Events \mid a \rightarrow b \bullet \\ &\quad \forall path \text{ in } Imp.P \bullet \\ &\quad a \rightarrow b \text{ in } p.Events \end{aligned}$$

More detail on verification of implementations against specifications are provided in Chapter 4 , but for an intuitive understanding, verification can be thought of as pattern matching between the CFG of specification and implementation, where the pattern defined by the specification CFG can be satisfied by many implementation CFGs (pattern here is used in a more general sense than design pattern). We define a function *satisfies* that relates implementations to specifications and evaluates to true when the CFG of the implementation conforms to the pattern defined by the specification CFG, or false otherwise.

A.2 Selection

The Alas selection operators `opt` and `alt` do not have a single equivalent in programming language syntax: there are multiple ways to implement conditional and mutually-exclusive conditional flows. For example, in Java, the code after a conditional return block is itself conditional, while multiple conditional return blocks in a single method are mutually exclusive. Instead of defining a set of equivalences, we formalize the selection operators in terms of paths, sub-paths and guarded blocks.

As Alas has composite events and allows the specification of mutually-exclusive behaviour, an event is only required to occur over some subset of paths where it is conceptually ‘reachable’. We define a sub-path as the set of paths beginning at a particular event or block and ending at the end of some other event or block, with the traversal of back edges that loop back to blocks before the beginning of the sub-path forbidden. As multiple events can occur in the same block, a sub-path may begin in the middle of a block. A sub-path may be accessed using $\langle Event - block \rangle.SubPath$. For example, an `opt` operator defines a set of sub-paths defined by the contents of its single operand. A (multi-line) if block in a Java implementation likewise defines a set of sub-paths that is equal to all the valid paths from the opening left-brace to the closing right-brace of the block’s definition. While matching an implementation with an `opt` in the specification, if the end of some conditional sub-path is reached before all the events contained in the `opt` are matched, then this candidate conditional sub-path does not match the `opt` in the specification.

The unique guard of an event, $ug(event)$, is the disjunction of all the guards of the edges entering that block (as a block may be ‘reached’ along multiple paths). The cumulative guard, $cg(event)$, is the conjunction of all the unique guards of each path from the initial to the block containing the event, themselves all combined in a conjunction. The cumulative guard can be thought of as the condition required to reach a particular block or event during ‘execution’. The unique and cumulative guard of the initial block is true. We define the function $container(event)$ to evaluate to the composite event or lifeline containing $event$ and the function $previous(event)$ may be used to access the previous event within the current containing event, or the beginning of the containing event if no such event exists. We abbreviate $previous(opt).SubPath$ to $currentSubPath$ in the following formalization, and similarly for `alt`, for convenience. The events contained in a composite event may be accessed using $\langle CompositeEvent \rangle.Events$.

An empty `opt` operator (one that has no events contained in it) is valid Alas syntax and states that some conditional block must exist within the current sub-path. An `opt` operator with contained events states that there exists some path over which all the contained events occur (1), the cumulative guard on every one of the contained events is the same (2), and is a strict super-set of the cumulative guard of the containing event (3). Also, the ordering relation defined above in the section on sequence applies equally to the contents of an `opt` operand (and `alt` operands), though with the relevant sub-path substituting for P. The ordering relation defined previously can be read as the sequencing requirement for events

at the ‘top level’ of the spec, i.e., within no container except for the lifeline itself.

$$\begin{aligned} &\exists path \text{ in } currentSubPath \bullet \\ &\quad \exists block \text{ in } path \bullet \\ &\quad cg(block) \neq true \end{aligned}$$

$$\begin{aligned} &\forall e1, e2 \text{ in } OPT.Events \bullet \\ &\quad cg(e1) = cg(e2) \end{aligned}$$

$$\begin{aligned} &\forall e \text{ in } OPT.Events \bullet \\ &\quad cg(e) \supset cg(container(OPT)) \end{aligned}$$

Note that this definition allows for the nesting level in the specification and implementation to be non-equivalent. A specification of an `opt` containing two call events, for example, may be matched by the two calls occurring at some arbitrarily deep level of nesting within, for example, three levels of Java `if` blocks. Likewise, a block may not be nested at all, but may still have a non-true guard. For example in:

```
if ( x > 5 ){
    return
}
a();
```

`a` is guarded, with guard `x <= 5`. This code snippet could match a specification with a call to `a` contained within and `opt` event.

With regard to the `alt` operator, each `alt` operand shares with the single `opt` operand the condition of guardedness and equal guardedness for all contained events (first and second equation above) and that the cumulative guard of each operand is greater than the cumulative guard of the container of the `alt` (third equation above+). The `alt` operator adds a fourth condition, that each of its operands are mutually exclusive. Note, this does not say that an event cannot occur in more than one operand of the same `alt`, but that two operands cannot contain *all* the same events in the same *order*. Also, we impose the rule that no `alt` operands in Alas are equal to, or equal to a prefix of some other Alas operand,

to facilitate verification, i.e., operands must be distinguishable. This implies that no `alt` operand is allowed to be empty, as the empty specification is a prefix of all specifications.

$$\begin{aligned} & \neg \exists path \text{ in } currentSubPath \bullet \\ & \exists op1, op2 \text{ in } ALT.Operands \bullet \\ & path \text{ satisfies } op1 \wedge path \text{ satisfies } op2 \end{aligned}$$

This definition of `alt` allows for sub-paths to exist where none of the operands actually occur. We term this type of `alt` as an existentially path quantified `alt` or `alte` for short. Figure 3.6 is an example of an `alte`, as there is no guard condition labelled *true*. In some cases however, we wish to state that one of a number of alternatives must occur. To specify this, *true* must be specified as the guard condition of the final `alt` operand. We term this type of `alt` as a universally path quantified `alt` or `altu`. The `altu` places the further constraint that every sub-path satisfies some operand:

$$\begin{aligned} & \forall path \text{ in } currentSubPath \bullet \\ & \exists op \text{ in } ALT.Operands \bullet \\ & path \text{ satisfies } op \end{aligned}$$

A.3 Iteration

Identifying the number of iterations a loop will make requires path-sensitive data-flow analysis, which, in languages like Java that have dynamic allocation and recursive data structures, is an undecidable problem[Dhurjati et al., 2006][Landi, 1992]. For this reason, we relax the meaning of the matching of quantified variables in loop guards (subsequently, *loop variables*) and selector strings to be that the number of iterations the loop performs is *some function* of the size of the collection.

`loop iteration condition` \simeq `collection size`

As with selection operators, there is no one-to-one mapping to Java syntax, and a `loop` operator may be matched by any of the loop constructs in Java and also by loops in the control-flow graph caused by recursive calls. Also, all information relevant to the number of iterations performed by the loop is not contained in single loop condition in the implementation. A Java `while` loop, for example, often has its loop variable incremented in the body of the loop.

Not only must the loop iterate some number of times proportional to the size of the collection, but the call to the collection element must be indexed in some way by the current loop iteration count. This again requires path-sensitive data-flow information, as the indexing may be indirect, as in the example below:

```
ObserverClass obsObj = collection[ loopIndex ];  
obsObj.methodRole();
```

Similarly to the iteration condition, we relax the indexing requirement from equality to proportionality:

$$\textit{Receiver object position} \simeq \textit{Iteration number}$$

With regard to lifeline role to object actor binding, in the case where the loop variable and selector string match, we allow the binding to be mutable. More specifically, a binding is fixed for the sub-paths defined by `loop` operator, but may be changed each time a back-edge is followed to the start of the `loop` sub-path.

A.4 Lifeline semantics

The lifeline type specified should correspond to the compile-time type intended, even when that type is abstract (thus there are ‘object roles’ of non-instantiable type). A behavioural specification can be thought of as stating ‘the method of this class must perform this particular behaviour’ rather than ‘an object of this class must be instantiable and behave

in this way'. Also, where a lifeline represents a reference variable role, its type in the structural and behavioural specification must match exactly.

A class actor must be *capable* of performing a specified behaviour, but does not necessarily have to *define* the behaviour within its own class definition, i.e., a class actor can satisfy a specification using inherited behaviour. For example, Figure 3.7 specifies the behaviour of the ConcreteSubject's `setState` and `notify` methods, but the `notify` method is often implemented in the abstract Subject. If an abstract superclass *must* implement a behaviour, it should be specified separately using a lifeline of the superclasses' type. The Observer specification in Figure 3.7 could be separated into two BDs: one defining `notify`'s behaviour with a Subject role and one defining `setState` (subclass-specific) with a ConcreteSubject role.

The runtime type of a role may always be equal to or a subclass of the compile-time type, and may change at any stage during the interaction.

Role naming

Duplicate names are allowed, where this does not cause ambiguity. It is a syntax error to produce a specification where a name does not clearly denote one and only one role. In the case of generic use of sets, it is possible that two lifelines or method calls will be indistinguishable from their name alone. To allow the lifelines or calls to be distinguished, their generic name must be suffixed by their occurrence number. Occurrence numbers increase from left-to-right horizontally across lifelines and from top to bottom of the BD for method names. E.g., the second generic lifeline's label might become `refVarRoleName : ClassSetRole[]:1`. Similarly to ordered sets, occurrence numbers begin at zero.

Lifeline naming

The object role name in a lifeline may be an arbitrary string, in the case where interaction is not through an already-defined reference variable role. Class role names, however, must always be predefined single or set roles.

A.5 Generic behaviour

Both generic behaviour and behavioural cardinality invariants affect the role-to-actor binding semantics in contradictory ways. For this reason, the use of roles with selectors (generic) and the use of quantified variables as roles (cardinality) is forbidden within a single BD. Generic behaviour and behavioural cardinality invariants may be specified in different BDs

within the same pattern specification, where the two BDs do not contradict one another.

Valid selector usage and semantics

The table below summarizes the valid combinations of set role types (class set, method set or collection) and selector usages. Parameter roles are a combination of an object and a class role. Both class and object (potentially drawn from a collection) in a parameter may have a selector applied.

	Class	Method	Collection
Empty selector	yes	yes	yes
Integer selector	no	yes	yes
Unmatched string selector	yes	yes	yes
Matched string selector	yes	yes	yes

In summary, the only invalid combination is an integer selector with a class role set, as a class set role is unordered. Object roles (where an object role name matches no reference variable role) are always mutable within a BD, as this is much more flexible and actually required in some cases. The meaning of the different selectors in terms of the element selected from the set and the mutability of the binding is given in the following table.

Selector usage	Element position	Binding	Example usage
Empty selector	Arbitrary	Mutable	Builder (Method role)
Integer selector	Fixed	Fixed	-
Unmatched string selector	Arbitrary	Fixed	-
Matched string selector	Varying, non-arbitrary	Mutable	Observer (Collection role)

Appendix B

Benchmark Observations and Ancillary Specifications

Some interesting observations made during the benchmark that are not central to the presentation of the evaluation, and some additional specifications based on those observations, are provided in this appendix.

B.1 Dependency

B.1.1 Abstract Factory

While creating our benchmark, we found two instances of Factory Methods with unorthodox implementations. The subclasses that implement both `DrawApplication.createStorageFormatManager` and `Figure.handles` differ not in the runtime type of the `ConcreteProduct` object returned, but in the objects composed within the `ConcreteProduct` object. A `PolyLineFigure`, for example, returns a `HandleEnumerator` containing zero or more `PolyLineHandles`, while an `ElbowConnection` returns a `HandleEnumerator` that contains a `ChangeConnectionStartHandle`, a `ChangeConnectionEndHandle` and zero or more `NullHandles` and `ElbowHandles`. Though we cannot specify an invariant such as ‘All `ConcreteProducts` must return a different composition of types of objects’, the shape analysis performed by AVT would be capable of verifying such an invariant. This is a combination of a cardinality and data-structure invariant that we did not consider during the design of Alas and may be an interesting direction for future work.

B.1.2 Command

The instance involving `ConnectedTextTool` is noteworthy, as its conformance to the pattern specification depends on the interpretation of inner classes. The Invoker role is filled by `ConnectedTextTool.DeleteUndoActivity`, which is an inner class of the Client. If an inner class is considered to be part of the outer class, then this breaks the first dependency invariant in Section 5.2.1.2, as the Invoker (which is also the client) initializes a `ConcreteCommand`. However, if they are considered separate classes, then this is a valid instance. The former interpretation (that inner classes are part of the outer class definition) is the one that is consistent with Alas semantics.

As part of future work, AVT could be modified to perform reverse engineering to discover more Command implementations. Also, the improvement in the pattern signature provided by the addition of implementation dependency invariants could be measured by performing reverse engineering on the selected code bodies with two specifications of the Command pattern: one with, and one without implementation dependency invariants.

B.1.3 Builder, Strategy, State

Builder

The Builder pattern aims to ‘separate the construction of a complex object from its representation’ by having a Director object define the construction algorithm and delegating the actual initialization of each part of the complex object to a Builder object, which knows how parts are represented. To satisfy the intent of the pattern, the Director should not initialize parts of the complex object. This requirement can be captured in the following invariant:

NOT (Director isInitializer ProductPart)

The interaction diagram in the GoF catalogue [Gamma et al., 1995, p.99] also describes a Client role configuring a Director with different `ConcreteBuilder` objects (similarly to the interaction of Client and Invoker, and Client and `ConcreteCommand` in the Command pattern). This can also be specified as:

NOT (Director isInitializer Builder)

While creating our benchmark, we found that most classes and methods with ‘builder’ in their name are not called from within a loop, as defined in the GoF catalogue and in our generic specification of Figure E.1. ‘Builder’ classes and methods in the code bodies studied typically resemble Factory Methods, though they often perform some additional state

initialization. We were unable to identify any genuine Builder instances with our keyword search, and the pattern is not included in either of the benchmarks we aggregated.

State/Strategy

It is widely agreed in the design pattern specification and verification literature that the State and Strategy patterns are similar, and difficult to distinguish in code [Guéhéneuc and Antoniol, 2008]. However, some approaches do address both patterns. Both patterns involve an object (Context) with a configurable delegate (a State or Strategy object). Shi [2007a] suggests that the delegate is modified by State subclasses in the State pattern, and by some separate Client role, through the Context’s interface, in the Strategy pattern. Though these alternatives are both considered in the context of the State pattern in the GoF catalogue, they are the typical means used by approaches to distinguish the two patterns. In the informal description of Shi [2007a], State subclasses must know about and initialize instances of each other. This could be forbidden in the Strategy pattern with the following cardinality invariant that uses a dependency clause:

```
NOT EXISTS s1, s2 in ConcreteStrategy @ s1 isInitializer s2
```

In the State pattern, the Context class could be forbidden from initializing State classes:

```
NOT ( Context isInitializer State )
```

Though the PINOT benchmark contains both State and Strategy instances, it is likely many of these are unintended instances, due to the weak signature of the patterns in PINOT’s (and our) definition. For example, their automated benchmark includes 96 instances of the Strategy pattern and 37 instances of the State pattern, in a code body of just over 1,000 classes. Also, with the widespread use of interfaces in the Swing benchmark, numerous classes may be considered members of the same inheritance hierarchy if they, for example, both implemented the `Serializable` interface. However, it should be clear that `Serializable` is not a suitable interface for the State class role.

B.2 Object-state

B.2.1 Prototype

The clone method of `FigureAttributes` (also in Swing) highlighted an issue with the Alas definition of `copystate`. When creating a copy, the `FigureAttributes` object wraps its own `Map` attribute with a `MapWrapper` object and returns both objects as part of the clone.

This satisfies the Alas requirement that a particular variable of the original is not an alias of the *same* variable in the clone, but is nonetheless returning a clone with aliased state. The solution to this issue would be to define a ‘deep’ copy as one that shares no state with the original, irrespective of the variable used to access the state. We found that some intended instances of the Decorator pattern in both JHotDraw and Swing forward to a Component only on the condition that the Component is not null.

B.2.2 Observer and Memento

Observer

The Observer pattern ‘define[s] a one-to-many dependence between objects so that when one object changes state, all its dependents are notified and updated automatically’ [Gamma et al., 1995]. The GoF describes a variation point with regard to the Observer update protocol. If the Subject sends the updated state to the Observer during the initial notification, this is an example of the *push model* variant of the pattern. If the Subject does not send the updated state in the initial notification, the Observer may perform a call-back on the Subject to access its state. This interaction is described as the *pull model* variant. A second variation point relates to whether the Subject or a Client role triggers updates by calling notify. We specify the behaviour of the *Subject triggers update* variant in Chapter 3 Figure 3.7. In the *Client triggers update* variant, some method of the Client role calls `notify` directly, instead of the call from Subject’s `setState` method to `notify` in Figure 3.7.

We specified both these variation points, which create four potential variants of the Observer pattern. We searched for instances where the Observer copies the updated state for processing, but found few instances of this behaviour. In most cases, the state of the Subject is read directly by the Observer via an alias, but no copy is made. In other instances, no state is pushed or pulled: the Subject sends a notification of an update to the Observer, and the Observer performs some behaviour based upon the update method called, but independently of any state of the Subject.

Memento

The Memento pattern ‘capture[s] and externalize[s] an object’s internal state so that the object can be restored to that state later’ [Gamma et al., 1995]. We provided part of the behavioural specification of the Memento pattern in Figure 3.10. While creating our benchmark, we identified very few instances of the Memento pattern. In Swing, Mementos are typically stored using serialization. We considered a serialization-based variant of the

Memento pattern, but there was little variation in the behaviour of instances when compared to instances of the Prototype pattern. For this reason, we focused our evaluation of object-state invariants on the latter pattern.

While deep copying behaviour is relevant to the Memento pattern, as it is explicit in the intent of the GoF description, few instances occur in practice. With regard to the Observer pattern, we identified less inter-dependence between the state of the Subject and Observer roles occurring in practice than the core variant described in the GoF catalogue. Alas is capable of describing two major variation points in the context of the Observer pattern.

B.3 Data-structure

B.3.1 Decorator

We found that some intended instances of the Decorator pattern in both JHotDraw and Swing forward to a Component only on the condition that the Component is not null. This is surprising, as it is counter-intuitive that a decorator object could exist without an object to decorate. However, it is conceivable that a Decorator could act as a placeholder for an object that may or may not exist. This is the role, for example, of the `JComponent.ActionStandin` class in the Swing benchmark. The forward if not null behaviour is not evidence of a CoR pattern, as a `handleRequest` method of a `ConcreteHandler` class is required to have not just one conditional path, but two mutually-exclusive conditional paths, one that serves the request and one that forwards it. The `operation` method for the *Forward if not null* variant is specified in Figure B.1. The `operation` method of the Decorator class for the other two variants simply delegates unconditionally to the `operation` method of its associated Component.

One of the notable features of the Swing code body is that it is most of the Decorator pattern instances make it impossible to badly-configure Decorator chains. This is done by requiring that the Component that the Decorator is decorating must be set in the Decorator's constructor as no mutator method is provided for the Component reference variable. The only way to badly configure a Decorator chain constructed from these classes would be to create an infinite chain of Decorator's. This is a limited instance of an ownership relationship in a context we did not expect and may be an interesting direction for future work.

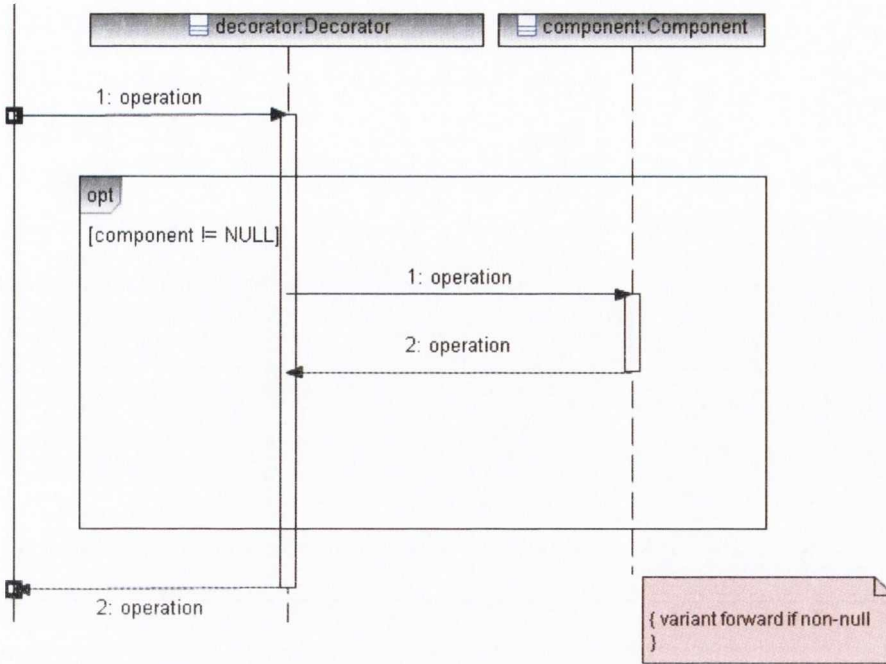


Fig. B.1: Specification of the operation method for the Forward if not null variant of the Decorator pattern.

B.3.2 CoR

The CoR pattern ‘avoids coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. We specified the behaviour of the ConcreteHandler role in Chapter 3 Figure 3.6. The PINOT specification of ConcreteHandler’s behaviour only requires a single conditional path, while our specification requires two mutually-exclusive paths. We found few instances of the pattern in each of the code bodies, and almost no non-trivial instantiations of chains of ConcreteHandlers. As the Decorator pattern has a similar data-structure invariant and more instantiations of chains of Decorators were identified, it was chosen to focus on Decorator pattern instance verification.

B.3.3 Composite

While the specification described in Section 5.2.3.2 follows the GoF catalogue closely, our experience while creating our benchmark is that there are very few instances that satisfy the entire specification. Numerous classes that are clearly intended to fill the Composite role lack one or more of the child management methods, or combine the `add` and `rem-`

ove methods into a single method, and some `operation` methods perform only conditional delegation to child Components. It is common in the literature for DPSLs and DPVTs to relax the constraints of the Composite pattern relative to the GoF specification: Blewitt et al. [2005], for example, require only an `add` and `remove` method. We specify two variants that relax the core Composite specification: one removes the `operation` role (*No operation* variant), and the other omits the `remove` method in the Composite class, to allow instances with a single combined `add/remove` method (*No remove* variant). The specification of these two variants involves simple remove clauses and have been omitted for the sake of brevity. All four variant specifications are mutually-combinable, making a total of 16 valid variant combinations of the Composite pattern according to our specification.

The child management methods defined in the Component class may then either do nothing or throw an exception. We can specify an approximation of ‘do nothing’ using a BD for each child management method that includes a found call to the method and no events within the methods lifeline. By the definition of the conformance relation of Alas, this states that a conforming method actor in a candidate implementation may not call any other method actor (including itself) that is bound to a role in the specification, but is free to call other unbound methods. A more precise specification of ‘do nothing’ could be achieved if Alas supported the other, more restrictive conformance relations discussed in Chapter 2. Supporting multiple conformance relations, however, requires more effort in defining the language semantics, as well as tool support, and we have only found this one application for the other conformance relations in our GoF pattern specifications.

Note, that if the parent reference variable from the parentLinks variant was included in the data-structure definition below, a correctly-configured tree with respect to the matching of parent and child links would be neither cycle nor sharing free. Each parent-child pair would form a cycle, and multiple children may potentially share the same parent.

```
componentTree isStructure Composite.children
componentTree isSharingFree
```

Though we focus on specifying data-structure properties using interaction invariants, we found a few cases where a method in isolation can guarantee a data-structure invariant. The `insert` method of `DefaultMutableTreeNode` guarantees there are no cycles in the parent links by calling an `isNodeAncestor` method, which checks if the parameter is an

ancestor, i.e., transitive closure of the parent link, of `this`. It also checks whether the new child has an existing parent, and if so, removes it from its current parent's children list, ensuring there is no sharing. Verifying that `insert` guarantees these properties requires path-sensitive data-flow analysis that evaluates conditional expressions and also handles looping, and is beyond the capabilities of AVT.

Appendix C

Verification Examples

This appendix provides some further verification examples that illustrate the capabilities and limitations of AVT. For each example, the code under analysis is shown, along with abbreviated output from AVT's analysis.

C.1 Dependency

The source code of `StandardDrawingView`'s `selectionZOrdered` method from the `JHotDraw` code body is shown in Figure C.1. Figure C.2 shows the AVT output when analyzing this method: AVT correctly identifies `selectionZOrdered` as a 'bad' client of the Factory Method `UndoableAdapter.getAffectedFigures`, which returns a Product of class `FigureEnumeration`.¹

¹In this and all subsequent illustrations of AVT output, some debugging output has been removed or reformatted for the sake of readability. However, all output presented is a result of the AVT analysis.

```
public FigureEnumeration selectionZOrdered() {
    List result = CollectionsFactory.current().createList(selectionCount());

    result.addAll(fSelection);
    return new ReverseFigureEnumerator(result);
}
```

Fig. C.1: `StandardDrawingView`'s `selectionZOrdered` in the `JHotDraw` code body is a 'bad' client of the Factory Method `UndoableAdapter.getAffectedFigures` as it creates an instance of a `FigureEnumeration` subclass

```

NOT selectionZOrdered isInitializer FigureEnumeration

MethodName: selectionZOrdered

BlockScanner::scanStatement
Statement: Local variable declaration
Expression is call
BlockScanner::scanExpression: Expression is non-local call
Scanning base_opt
Expression is call
BlockScanner::scanExpression: Expression is non-local call
Scanning base_opt
Expression is name
Expression is call
BlockScanner::scanExpression: Expression is local call
BlockScanner::scanExpression: Method not in call stack. Analyzing now
BlockScanner::scanStatement
Expression is call
BlockScanner::scanExpression: Expression is non-local call
Scanning base_opt
Expression is name
Expression is cast
Expression is name
BlockScanner::scanStatement
Return encountered
Expression is cast
Expression is class creation
InitializerAnalysis::visitClassCreation

ReverseFigureEnumerator equals or sub FigureEnumeration?
InitializerAnalysis::visitClassCreation: Subclass constructor call match found
Creation of object of type: ReverseFigureEnumerator
Finished processing return statement

FAIL! selectionZOrdered initializes FigureEnumeration
=====

```

Fig. C.2: AVT output when applying the implementation dependency invariant of the Abstract Factory Client role to the method of Figure C.1

```

public synchronized Object clone() {
    try {
        ElementIterator it = new ElementIterator(root);
        if (elementStack != null) {
            it.elementStack = new Stack();
            for (int i = 0; i < elementStack.size(); i++) {
                StackItem item = (StackItem)elementStack.elementAt(i);
                StackItem clonee = (StackItem)item.clone();
                it.elementStack.push(clonee);
            }
        }
        return it;
    } catch (CloneNotSupportedException e) {
        throw new InternalError();
    }
}

```

Fig. C.3: ElementIterator’s clone method in the Swing code body

C.2 Object state

This example illustrates the capabilities of AVT in the context of loops, and in particular, loops that mutate collections: a particularly challenging case.

The implementation of the clone method of `ElementIterator` is shown in Figure C.3, and the result of AVT applying the `isCopy` predicate to it is shown in Figure C.4. The three shape graphs correspond to the three paths through the method: entering the if only, entering the if and the nested for, and skipping both completely. As discussed in Chapter 4, the construction of the control flow graph for exception handling was designed but not implemented so the shape graph corresponding to the flow that enters the catch block does not appear in the output. All three graphs include the information that the ‘root’ variables of the original and copy are aliased. In the first graph, AVT correctly identifies that the ‘elementStack’ of the copy is empty and the ‘elementStack’ of the original may not be, so it flags a violation. To verify that the size of the original’s ‘elementStack’ is checked within the for loop’s condition, and that correct copying of the stack is actually performed, requires path-sensitive data-flow analysis, which is not performed by AVT, as discussed in Chapter 4. In the third graph, AVT complains that no copying of the stack has occurred.

The result of applying the `isCopy` predicate to the second shape graph illustrates some of the most sophisticated features of AVT’s shape analysis, as it combines looping (which provides an extra complication to shape analysis algorithm convergence), complex accesses of collection contents, aliasing and cloning. The call to `elementAt` accesses the single summarized selector variable (a selector in a collection that represents zero or more contiguous elements in a collection), and the `StackItem` accessed is then itself cloned and added to the copy’s ‘elementStack’. AVT correctly identifies that the elements in the first position ([0])

```

===== this isCopy returnval =====
Graph number: 0
'this' is not an alias of 'returnval'. Good!
'this -> root' is an alias of 'returnval -> root'. Fail! (isCopy => NOT isAlias )
'this -> elementStack' is not an alias of 'returnval -> elementStack'. Good!
Collection: 'this -> elementStack' has an element at position: 0
    but collection: 'returnval -> elementStack' does not. Fail!
Graph number: 0: FAIL!

Graph number: 1
'this' is not an alias of 'returnval'. Good!
'this -> root' is an alias of 'returnval -> root'. Fail! (isCopy => NOT isAlias )
'this -> elementStack' is not an alias of 'returnval -> elementStack'. Good!
Collections: 'this -> elementStack' and 'returnval -> elementStack' both have elements at position: 0.
    Comparing elements now.
'this -> elementStack -> [0]' is not an alias of 'returnval -> elementStack -> [0]'. Good!
'this -> elementStack -> [0] -> childIndex' has a primitive value equal to:
    'returnval -> elementStack -> [0] -> childIndex'. Good!
'this -> elementStack -> [0] -> item' is an alias of
    'returnval -> elementStack -> [0] -> item'. Fail! (isCopy => NOT isAlias )
Graph number: 1: FAIL!

Graph number: 2
'this' is not an alias of 'returnval'. Good!
'this -> root' is an alias of 'returnval -> root'. Fail! (isCopy => NOT isAlias )
elementStack undefined reference variable in both 'this' and 'returnval'. No copying performed. Fail!
Graph number: 2: FAIL!

FAIL!

```

Fig. C.4: Result of applying the isCopy predicate to the source code of Figure C.3

```

public synchronized Figure replace(Figure figure, Figure replacement) {
    if (!(replacement instanceof AnimationDecorator) &&
        !(replacement instanceof ConnectionFigure)) {
        replacement = new AnimationDecorator(replacement);
    }
    return super.replace(figure, replacement);
}

```

Fig. C.5: BouncingDrawing's replace method from the JHotDraw benchmark

of both 'elementStack's are not aliases, that their primitive attribute, 'childIndex', has an equal value and their reference variable attribute, 'item', is aliased (as the implementation of StackItem's clone method simply calls Object.clone).

C.3 Data structure

C.3.1 Decorator

The source code of BouncingDrawing's replace method (also in JHotDraw) is shown in Figure C.5, and the corresponding AVT output is shown in Figure C.6. In this case, the chain can not be proven to be well- or badly-configured, as the next object in the chain is a parameter of the method under test and its class is Component.

```

Analyzing:
=====
->Figure myDecoratedFigure . LAST isKindOf DecoratorFigure OR
->Figure myDecoratedFigure . LAST isKindOf Figure
=====
...
ShapeGraph::getCorePredicateByName: couldn't find core predicate:myDecoratedFigure. Returning NULL.
LastOperand::evaluate: Binary predicate number: 0 is valid link role
ShapeGraph::getBinaryPredicateByIdentifier: NO MATCH FOUND! RETURNING NULL!
LastOperand::getLastInChain: Path ends
InitUtility::equalsString: Figure == DecoratorFigure ?
InitUtility::equalsString: mis-match
InitUtility::inheritsString
Storable <: DecoratorFigure ?
InitUtility::inheritsString: No interfaces in type: Figure
Cloneable <: DecoratorFigure ?
InitUtility::inheritsString: No interfaces in type: Figure
Object <: DecoratorFigure ?
InitUtility::inheritsString: Found Class Object
No match
IsKindOfClause::verify: Object is not of that kind
InitUtility::equalsString: Figure == Figure ?
InitUtility::equalsString: match
Last element in chain is of Component type. Can't prove structure well or badly configured

ANALYSIS TIME: 2

```

Fig. C.6: Result of applying the data-structure invariant to BouncingDrawing's replace method

C.3.2 Composite

The analysis of the `TestSuite` Composite actor from the JUnit code body provides a useful illustration of the scalability limitations of AVT. `TestSuite` can be initialized with one of three constructors: two simple constructors, one with no parameters, one with a `String` parameter; and a very complex constructor that accepts a `Class` parameter and conditionally adds multiple child `Test` objects (`Test` is the Component class). Around half the trees we identified use a simple constructor, while the other half use the complex constructor. `SuiteTest`'s `suite` method creates a `TestSuite` with the String-parameter constructor and adds 9 `Test` objects to its list of children. AVT's analysis of the program segment converges after two iterations and outputs a single shape graph. Figure C.7 shows the end of the output file generated by the AVT analysis of `SuiteTest`'s `suite` method. The values of the binary predicates which satisfy the link role of the data-structure definition are compared to identify if any sharing occurs in the program segment. AVT correctly identifies that the tree is free from sharing in under a second.

`VectorTest`'s `suite` method contains a single call to the class-based constructor of `TestSuite`. The AVT analysis of the program segment contained in `suite` outputs 19 shape graphs in under two seconds. The `suite` method of `AllTests` in the `tests` package

```

Comparing:
Source object: Summary Collection Location: Summary location for type: Vector, Method: addTest, Context: 4, Block Number: 1, Line Number: 0,
Occurrence Number: 0, Selector: Selector: [ Name: [6], Type: Object, LOCAL VAR, DUMMY
]Target object: Type: SuiteTest, Method: suite, Context: 0, Block Number: 1, Line Number: 34, Occurrence Number: 0
and
Source object: Summary Collection Location: Summary location for type: Vector, Method: addTest, Context: 4, Block Number: 1, Line Number: 0,
Occurrence Number: 0, Selector: Selector: [ Name: [7], Type: Object, LOCAL VAR, DUMMY
]Target object: Type: SuiteTest, Method: suite, Context: 0, Block Number: 1, Line Number: 39, Occurrence Number: 0
IsSharedClause::compareAll: Target objects non-identical. Good!

Comparing:
Source object: Summary Collection Location: Summary location for type: Vector, Method: addTest, Context: 4, Block Number: 1, Line Number: 0,
Occurrence Number: 0, Selector: Selector: [ Name: [6], Type: Object, LOCAL VAR, DUMMY
]Target object: Type: SuiteTest, Method: suite, Context: 0, Block Number: 1, Line Number: 34, Occurrence Number: 0
and
Source object: Summary Collection Location: Summary location for type: Vector, Method: addTest, Context: 4, Block Number: 1, Line Number: 0,
Occurrence Number: 0, Selector: Selector: [ Name: [8], Type: Object, LOCAL VAR, DUMMY
]Target object: Type: SuiteTest, Method: suite, Context: 0, Block Number: 1, Line Number: 44, Occurrence Number: 0
IsSharedClause::compareAll: Target objects non-identical. Good!

Not comparing predicate to itself

Comparing:
Source object: Summary Collection Location: Summary location for type: Vector, Method: addTest, Context: 4, Block Number: 1, Line Number: 0,
Occurrence Number: 0, Selector: Selector: [ Name: [7], Type: Object, LOCAL VAR, DUMMY
]Target object: Type: SuiteTest, Method: suite, Context: 0, Block Number: 1, Line Number: 39, Occurrence Number: 0
and
Source object: Summary Collection Location: Summary location for type: Vector, Method: addTest, Context: 4, Block Number: 1, Line Number: 0,
Occurrence Number: 0, Selector: Selector: [ Name: [8], Type: Object, LOCAL VAR, DUMMY
]Target object: Type: SuiteTest, Method: suite, Context: 0, Block Number: 1, Line Number: 44, Occurrence Number: 0
IsSharedClause::compareAll: Target objects non-identical. Good!

Not comparing predicate to itself
No sharing found in current graph
No sharing found
Result:
Graph number 0:
No sharing found in graph: 0
No sharing found

ANALYSIS TIME: 0s

```

Fig. C.7: Segment of the AVT analysis output when analyzing SuiteTest's suite method.

The Composite tree is correctly classified as being free from sharing

```

protected static TreeModel getDefaultTreeModel() {
    DefaultMutableTreeNode root = new DefaultMutableTreeNode("JTree");
    DefaultMutableTreeNode parent;

    parent = new DefaultMutableTreeNode("colors");
    root.add(parent);
    DefaultMutableTreeNode blueNode = new DefaultMutableTreeNode("blue");

    parent.add(blueNode);
    parent.add(new DefaultMutableTreeNode("violet"));
    parent.add(new DefaultMutableTreeNode("red"));
    parent.add(new DefaultMutableTreeNode("yellow"));

    parent = new DefaultMutableTreeNode("sports");
    root.add(parent);
    parent.add(new DefaultMutableTreeNode("basketball"));
    parent.add(new DefaultMutableTreeNode("soccer"));
    parent.add(new DefaultMutableTreeNode("football"));
    parent.add(new DefaultMutableTreeNode("hockey"));
    parent.add(blueNode);

    parent = new DefaultMutableTreeNode("food");
    root.add(parent);
    parent.add(new DefaultMutableTreeNode("hot dogs"));
    parent.add(new DefaultMutableTreeNode("pizza"));
    parent.add(new DefaultMutableTreeNode("ravioli"));
    parent.add(new DefaultMutableTreeNode("bananas"));
    return new DefaultTreeModel(root);
}

```

Fig. C.8: Modified source code of `JTree`'s `getDefaultModel` method from the Swing code body, with sharing introduced. The original simply omits the addition of the 'blue' node to the parent 'sports'

ordinarily contains 12 calls to the complex Class-parameter constructor of `TestSuite`. We commented out all but two, and the analysis ran for 276 seconds. The Class-parameter constructor of `TestSuite` is close to the worst-case scenario for AVT, in that it conditionally adds numerous children to the structure in separate sequential conditionals. Sequential conditionals are more problematical than nested conditionals, as they create more potential paths through the program than the same number of nested conditionals.

The source code for the modified version of `JTree`'s `getDefaultModel` method with added sharing is shown in Figure C.8 It simply adds the 'blue' node to two parents, while the original only added it to the 'colors' parent. AVT correctly identifies that sharing exists in the tree.

Appendix D

Aggregated benchmarks

As stated in Section 5.1.2, it is important that a benchmark is created objectively, and instances are not cherry-picked to the advantage of one approach or another. In this appendix, we identify the commonalities and differences between our benchmark and the benchmark we used as a starting point, similarly to Rasool et al. [2011]. We demonstrate that numerous instances are common to our benchmark and those created by others, while also highlighting differences and providing a justification for these.

D.1 Dependency

D.1.1 Abstract Factory

We create and analyze the first sizeable benchmark that includes variants of the Abstract Factory pattern. Existing benchmarks and DPVT evaluations in the literature report instances of a single, generic specification [Shi, 2007a][Blewitt et al., 2005][Stencel and Wegrzynowicz, 2008].

Both PINOT [Shi, 2007b] and P-MARt [Guéhéneuc, 2007] provide a benchmark for the Abstract Factory pattern that we aggregated in creating our own. PINOT's specification requires an `AbstractFactory` role, but also adds the constraint that the `Factory Method` returns an object of a subclass of its return type. Some of the instances provided by PINOT do not create a new object over all paths, and some in fact may return the value `null`. This is forbidden by our specification. Guéhéneuc [2007] does not provide a detailed specification of the Abstract Factory pattern, but seems to allow GoF and No AF variants.

The first two columns of Tables D.1, D.2 and D.3 summarize the outcome of aggregating the PINOT and P-MARt benchmarks. In Table D.1, as in all subsequent tables the number

variant	PINOT	P-MARt	Alas
GoF	22(26)	1(1)	22
No AF	0(0)	1(1)	23
Self Factory	3(4)	0(0)	3
Self Factory, No AF	0(0)	0(0)	0

Table D.1: Instances of variants of the Abstract Factory pattern in each benchmark from the JHotDraw code body

variant	P-MARt	Alas
GoF	0(0)	1
No AF	0(0)	12
Self Factory	0(0)	1
Self Factory, No AF	0(0)	0

Table D.2: Instances of variants of the Abstract Factory pattern in each benchmark from the JUnit code body

in each cell under each aggregated benchmark indicates the number of instances of that variant from the benchmark that we include in our own, i.e., the number of instances shared by our benchmark and the benchmark created by others. The second number in brackets indicates the total number of instances of that variant reported in the aggregated benchmark, indicating the number of instances in the aggregated benchmark not included in the Alas benchmark. As can be seen from the table, our classification of instances agrees in all cases with P-MARt, and agrees in a majority of cases with PINOT. The instances from PINOT’s benchmark that we exclude either do not create a new object over all paths through the Factory Method, or are intended instances of clone methods of the Prototype pattern. The one instance of the Self Factory pattern identified in JUnit is also included in the small manual benchmark DEEBEE [Fulop et al., 2008].

D.1.2 Command

Typically, approaches that address the pattern specify only an inheritance relationship (between Command and ConcreteCommand) and a sequence of method calls (Invoker to Con-

variant	PINOT	Alas
GoF	24(39)	24
No AF	0(0)	3
Self Factory	2(2)	2
Self Factory, No AF	0(0)	0

Table D.3: Instances of variants of the Abstract Factory pattern in each benchmark from the Swing code body

creteCommand, followed by ConcreteCommand to Receiver) [Tsantalis, 2009]. Guéhéneuc and Antoniol [2008] distinguish Command from other patterns by identifying separate Client and Invoker roles, though the invariants placed on those roles are not described. In Alas, it is possible to specify the interaction between Client, Invoker and ConcreteCommand as a series of method calls where object and parameter roles match, as demonstrated in the context of the Memento pattern in Chapter 3 Section 3.3.2.1.

Guéhéneuc [2007] provides a benchmark for the Command pattern in both JHotDraw and JUnit. We include in our benchmark the single Command instance they identify in JHotDraw. The GoF pattern classification of Shi [2007a] places Command in the category of ‘domain-specific patterns’, the verification of which require application- or domain-specific knowledge. Domain-specific patterns are not supported by the PINOT tool [Shi and Olsson, 2006].

Without documentation, it was difficult to identify the intended Receiver role, and thus difficult to classify the pattern as an instance of one or another variant. We separate the single instance identified by Guéhéneuc [2007] into 5 different instances. We identify each instance below using the different Invoker actors in the code body, and the different ConcreteCommand actors with which they interact. Each subclass of `AbstractCommand` in the JHotDraw code body differs in how much behaviour its `execute` method performs and how much it delegates. Two subclasses in particular contain complex `execute` methods and delegate only relatively small tasks to the Receiver: `AlignCommand` and `ChangeAttributeCommand`. `AlignCommand` has an abstract inner class `Alignment`, which declares a number of static final instances (`Alignment` is a type of `Limiton` [Stencel and Wegrzynowicz, 2008]). The client configures the Command with one of these particular instances of `Alignment`. Much of the behaviour of Command’s `execute` is performed by

Instance number	Invoker	ConcreteCommand	Client
1	CommandButton	DeleteCommand etc.	DrawApplet
2	CommandMenu	AlignCommand	DrawApplication
3	CommandMenu	UndoCommand etc.	DrawApplication
4	ConnectedTextTool .DeleteUndoActivity	DeleteCommand	TextTool, ConnectedTextTool
5	CommandChoice	ChangeAttributeCommand	DrawApplet

Table D.4: Actor names in candidate Command instances

Instance number	Variant
1	Command forms Façade
2	Known Receiver
3	Command forms Façade
4	Negative
5	Known Receiver

Table D.5: Command instance classification

delegating to the `Alignment`. We classify these two instances as Known Receiver variants. The execute method role still delegates to a `DrawingView`, similarly to other instances, so it could be considered an orthodox GoF example, but it is clear that the main task of the method is performed by delegating to the `Alignment`. The classification of the five instances is summarized in Tables D.4 and D.5.

D.2 Object state

D.2.1 Prototype

None of the aggregated benchmarks address the Prototype pattern. The GoF pattern classification of Shi and Olsson [2006] places Prototype in the category of language-provided patterns, which is not supported by the PINOT tool [Shi and Olsson, 2006]. We argue that the provision of features such as the Cloneable interface in Java does not remove the issue

of deep vs. shallow copies, and the implementation of correct clone methods is a non-trivial task.

D.3 Data structure

D.4 Decorator

PINOT is based on a different specification of the Decorator pattern: all delegation to the Component must be unconditional. Any conditional delegation is a CoR instance. Our specifications follow the GoF catalogue in having a superclass delegate in the Decorator pattern and a same-class delegate in the CoR pattern: PINOT seems to allow superclass delegation in the CoR pattern. Thus, a lot of the forward if not null variant instances of the Decorator pattern in our benchmark are CoR instances in PINOT's [Shi, 2007b]. PINOT's benchmark was very useful in this case, as it provided complete code coverage for two of our three code bodies identifying all superclass delegation instances.

The P-MARt benchmark has a similar specification to our GoF variant specification, requiring both a Decorator and ConcreteDecorator role, as well as Component and ConcreteComponent roles. The same inheritance and delegation relations between classes are required by P-MARt and Alas specifications. Tables D.6, D.7 and D.8 summarize the output of aggregating the PINOT and P-MARt benchmarks for each of the three included code bodies. It can be seen that our benchmark is in complete agreement with PINOT with regard to No sub variants, and with P-MARt with regard to GoF variants. PINOT identifies both GoF and No sub variants. The Decorator role in the GoF variant instance in JHotDraw is `DecoratorFigure`, for the instance in Swing it is `FlowView` and in JUnit it is `TestDecorator`.

The inclusion of three variants allows us to collect instances that occur in neither of the aggregated benchmarks, and, in fact, the Alas benchmark for the Decorator pattern is a superset of the PINOT and P-MARt benchmarks. We are able to distinguish instances uncovered by the PINOT benchmark into two variants: GoF and No sub.

D.4.1 Composite

Both the PINOT and P-MARt benchmarks address the Composite pattern. The PINOT specification of the Composite pattern is difficult to identify, as it is hardcoded in the tool.

variant	PINOT	P-MARt	Alas
GoF	0(0)	1(1)	1
No Sub	5(5)	0(0)	5
Forward if not null	0(0)	0(0)	1
Total	5	1	7

Table D.6: Instances of variants of the Decorator pattern in each benchmark from the JHotDraw code body

variant	P-MARt	Alas
GoF	1(1)	1
Total	1	1

Table D.7: Instances of variants of the Decorator pattern in each benchmark from the JUnit code body

variant	PINOT	Alas
GoF	1(1)	1
No Sub	18(18)	18
Forward if not null	0(0)	4
Total	19	23

Table D.8: Instances of variants of the Decorator pattern in each benchmark from the Swing code body

As we understand it, any class that has an array attribute of some superclass (including class `Object`) is an instance. As Java 1.4.2 (the language analyzed by PINOT and AVT) lacks generics, it is more difficult to identify the content type of a collection than an array. PINOT addresses this issue by searching for a call to a ‘put’ method on some collection attribute of a candidate Composite class that adds an instance of some superclass to the collection. A ‘put’ method is any method in a hard-coded list of collection mutation methods such as `Vector.add` or `Map.put`. This is a less strict specification than any of our variants, and by allowing the class `Object` to play the role of `Component`, and classes such as `String` and `Integer` to play the role of `Leaf`, it includes numerous instances that are unlikely to be intended by the original developer of the classes.

The P-MARt [Guéhéneuc and Antoniol, 2008] specification of the Composite pattern includes three classes: `Component`, `Composite` and `Leaf`, with `Composite` and `Leaf` both inheriting from `Component`. There is a composition relationship between `Composite` and `Component`. The composition relation in DeMIMA (the tool developed by the creators of the P-MARt benchmark) requires the ‘exclusivity’ and ‘lifetime’ properties. Exclusivity requires that ‘instances of the part might be instantiated before the whole is instantiated, but they must not belong to any other whole.’ The related lifetime property requires that the composed and composing objects are destroyed, or garbage collected in languages such as Java, at similar times. It is not clear how these properties are identified during manual code inspection.

Table D.9 summarizes the number of instances of Composite patterns in each of the aggregated benchmarks for each of the code bodies included in our evaluation. We include both pattern instances from the P-MARt benchmark. Neither the PINOT nor our Alas benchmarks are supersets of each other with regard to Composite pattern instances. Both PINOT and P-MARt report instances of the core and no remove variants, but do not make a distinction, as neither are concerned with the `remove` method role.

Table D.1 lists each instance by Composite class actor name and gives the variant specifications satisfied by each. `DefaultMutableTreeNode` declares a parent reference variable in its class definition, instead of inheriting the variable from the `Component` role, as in the `parentLinks` variant. A *parentLinks in Composite* variant could also be specified to handle this case.

	PINOT	P-MARt	Alas
JHotDraw	2(4)	1(1)	2
JUnit	-	1(1)	1
Swing	2(20)	-	4

Table D.9: Instances of the Composite pattern identified in both the Alas benchmark and aggregated benchmarks

	parentLinks	Unsafe	No operation	No remove
JHotDraw				
PertFigure	×	×	×	×
CompositeFigure	×	×	×	×
JUnit				
TestSuite	×	×	×	✓
Swing				
CompositeView	✓	✓	✓	✓
CompoundEdit	×	✓	×	✓
BranchElement	×	×	✓	✓
DefaultMutableTreeNode	×	✓	✓	×

Fig. D.1: Composite class actor of each identified instance, along with their variant classification

Appendix E

Generic behaviour specification

Generic behaviour arises when all valid implementations of a particular pattern or pattern variant include a behaviour, but some intrinsic detail of that behaviour can differ between implementations, making the creation of a single specification to represent these implementations difficult. The state of the art in DPSLs tends to overlook generic behaviour as unspecifiable or implementation-dependent, weakening the precision of the specification and increasing the likelihood of errors, usually false positives, during verification. Generic behaviour specification is the fourth and final major contribution of Alas. In this section, we discuss the features of Alas that address generic behaviour.

The actual guard conditions on conditional branchings in patterns implementations are implementation dependent and can thus rarely be specified exactly. In Alas, it is possible to specify generic guards, by indicating that a guard condition is (1) some function of the state of some object role or roles or (2) includes a particular conditional expression that may be combined with other implementation-specific conditional expression in a candidate implementation. In the former case, a comma-separated list of object roles is included in the guard condition belonging to an operand of a selection operator. In the latter case, a comma-separated list of conditions, suffixed with a `+` is placed in the guard. Both cases can also be combined in the same list, but it is a syntax error to include a `+`-suffixed and non-`+`-suffixed clause in the same list: a guard condition is either completely or generically specified. For example, the `alt` operator guarding the call to the `successor` role in the CoR specification of Figure 3.6 could be guarded with `successor` alone (instead of `successor != NULL`), to indicate that the behaviour is dependent on the state of the `successor`, but not necessarily its NULL-ness.

Genericity also applies in the specification of method calls. The Builder pattern ‘sep-

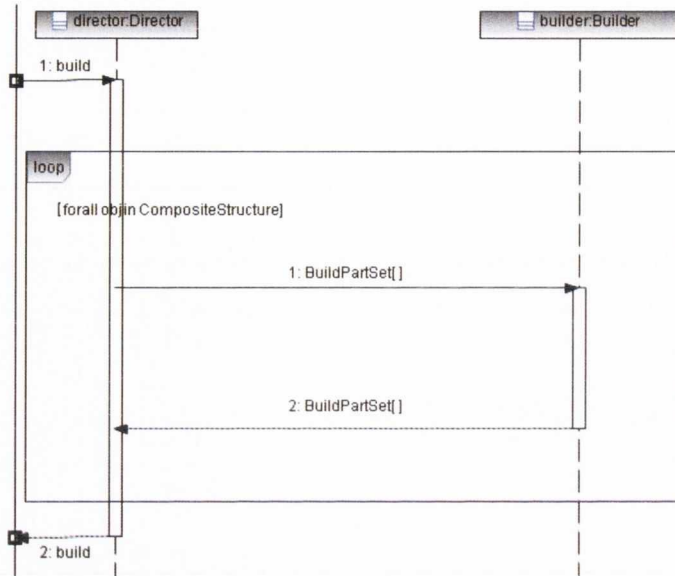


Fig. E.1: A generic specification of the Looping Director variant of the Builder pattern. No selector is provided for the method set: any arbitrary member of the `BuildPartSet` method set may be called on each iteration of the loop.

arates the construction of a complex object from its representation' by having a Director define the algorithm for the building process, but delegating the actual work to a Builder that knows the output representation. One variant discussed in the GoF catalogue ([Gamma et al., 1995, p.97]) involves a Director iterating over some input structure and calling different building methods on each iteration, depending on the type of the currently indexed object in the structure. We extend the concept of selectors for collection roles by applying them to method (and also class) set roles, allowing mutable bindings to method (and class) roles in a BD. Specifying a role with braces but no selector indicates that an arbitrary element from the set may be bound, but with a mutable binding (unlike UML). Our Alas specification of the Looping Director variant of the Builder pattern introduced above is shown in Figure E.1, where the building methods are represented with the `BuildPartSet` method set role.

Design patterns impose ordering constraints on sets of methods, though the number of methods is implementation-specific. In Alas, method sets are ordered (with the order defined at *binding time*, when the roles are initially bound to actors). Selectors applied to method sets may be integers, allowing the selection of a particular element of the set, but this does not solve the issue of unbounded-size method sets. We address this issue by allowing the selector to be a range, indicated by `[begin...end]` and by providing access to

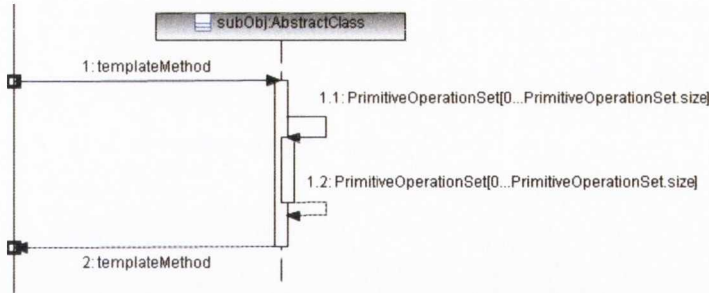


Fig. E.2: A generic specification of a Template Method. Each of the members of the `PrimitiveOperationSet` is called in sequence. The set may vary in size between valid implementations.

the size of the set: `<MethodSet>.size`. This mechanism is demonstrated in the context of the Template Method pattern in Figure E.2. A Template Method ‘define[s] the skeleton of an algorithm’ by calling all the members set of methods (`primitiveOperations`) in a fixed order, where the number and ordering of `primitiveOperations` will differ in each context. This specification will be expanded at binding time to a sequence of calls, beginning with the first element and continuing until and including the last element. A variant of the Builder pattern, where the Director contains a sequence of calls to individual `build` methods instead of a loop, is specified identically.

The final type of generic behaviour addressed is generic conditional branching. Some variants of GoF patterns relate the number of alternative control flows to the number of elements in a set of classes or methods. The Parameterized Factory variant of the Abstract Factory pattern, for example, involves a Factory Method that is capable of creating objects of multiple different types, choosing between the types using a parameter. Thus, it has a number of mutually-exclusive paths, each creating a new object, equal to the number of classes inheriting from the `AbstractProduct` class role (assuming that the Factory Method should be capable of producing each of the `ConcreteProducts`).

Alas provides the `repeat` keyword, that may be placed in the guard of an `alt` operand, with the meaning that the current operand may occur once or multiple times in a conforming implementation. A quantified statement, quantifying over some class or method set role, may follow the `repeat` keyword, indicating that a mutually-exclusive path must occur once for each member of the set. Any interaction involving the set role may have its binding influenced by a selector, as in the case of iteration over collection roles in Section 3.3.1.3. Figure E.3 specifies the Parameterized Factory variant behaviour described above. This

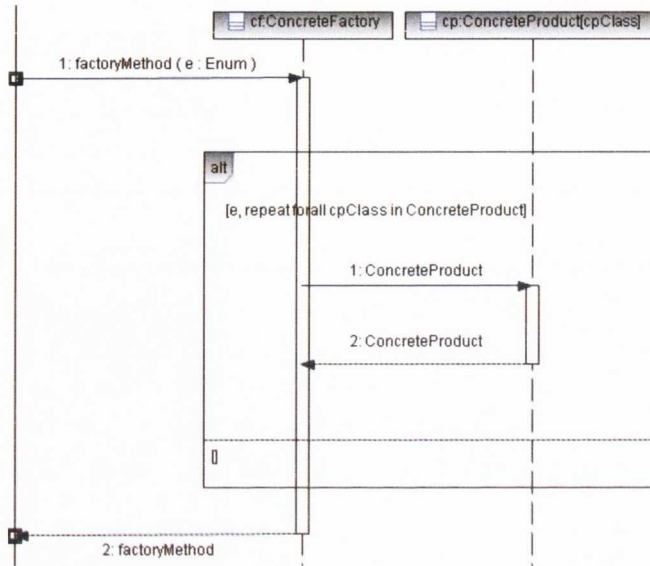


Fig. E.3: Generic specification of the ConcreteFactory’s `factoryMethod`. The specification states that for each class in the ConcreteProduct class set role, there exists an alternative path that creates an object of that class.

approach may also be used to describe the Looping Director variant, by specifying that an `alt` operand repeats once for every method in the `BuildPartSet` method set role. Note that the ConcreteProduct lifeline has a mutable-binding class set role and a fixed-binding object role. For a discussion of valid combinations of single roles, set roles and selectors, see Appendix A Section A.5.

Bibliography

VDMTools . The VDM++ to Java Code Generator ver.1.1. Technical report, CSK Corporation, 2010.

Alfred Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986. ISBN 0201101947.

Christopher Alexander. *The Timeless Way of Building*. Oxford University Press, 1979. ISBN 0195024028.

Robert Allen. *A Formal Approach to Software Architecture*. PhD thesis, Carnegie Mellon, School of Computer Science, January 1997. Issued as CMU Technical Report CMU-CS-97-144.

Awny Alnusair and Tian Zhao. Using Ontology Reasoning for Reverse Engineering Design Patterns. In Sudipto Ghosh, editor, *Models in Software Engineering*, volume 6002 of *Lecture Notes in Computer Science*, pages 344–358. Springer Berlin / Heidelberg, 2010. ISBN 978-3-642-12260-6. URL http://dx.doi.org/10.1007/978-3-642-12261-3_32. 10.1007/978-3-642-12261-3_32.

Scott W. Ambler. *The Elements of UML(TM) 2.0 Style*. Cambridge University Press, New York, NY, USA, 2005. ISBN 0521616786.

S. Ammour, Mikal Ziane, Xavier Blanc, and S. Chantit. A uml precise specification of design patterns using decoupling constraints. In *4th Workshop in Software Model Engineering (WiSME '05)*, 2005. INT LIP6 MoVe.

Tony Andrews, Shaz Qadeer, Sriram K. Rajamani, Jakob Rehof, and Yichen Xie. Zing: a model checker for concurrent software. In *Computer Aided Verification*, pages 484–487. Springer, 2004.

- G. Antonioli, G. Casazza, M. Di Penta, and R. Fiutem. Object-oriented design patterns recovery. *Journal of Systems and Software*, 59(2):181 – 196, 2001. ISSN 0164-1212. doi: 10.1016/S0164-1212(01)00061-9. URL <http://www.sciencedirect.com/science/article/pii/S0164121201000619>.
- Francesca Arcelli, Marco Zanoni, and Christian Tosi. A benchmark proposal for design pattern detection. In *2nd Workshop on FAMIX and Moose in Reengineering (FAMOOSr'08)*, 2008.
- AtlanMod. Atlantic Zoo, 2010. URL <http://www.emn.fr/z-info/atlanmod/index.php/Atlantic>. Last accessed: 11th March 2011.
- Thomas Baar. On the need of user-defined libraries in ocl. In *OCL and Textual Modelling workshop, MoDELS 2010*, 2010.
- Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. *The Spec# programming system: An overview*, pages 46–69. Springer Berlin / Heidelberg, 2005.
- Aline Lúcia Baroni, Yann-Gaël Guéhéneuc, and Hervé Albin-Amiot. Design patterns formalization. Technical report, École Nationale Supérieure des Techniques Industrielles et des Mines de Nantes, 2003.
- Ian Bayley and Hong Zhu. Formal specification of the variants and behavioural features of design patterns. *J. Syst. Softw.*, 83(2):209–221, 2010. ISSN 0164-1212. doi: <http://dx.doi.org/10.1016/j.jss.2009.09.039>.
- Kent Beck and Ward Cunningham. Using pattern languages for object-oriented programs. Technical report, Apple Computer Inc. and Tektronix Inc., 1987.
- Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, Peter W. O’Hearn, Thomas Wies, and Hongseok Yang. Shape Analysis for Composite Data Structures. In *Computer Aided Verification*, 2007.
- Federico Bergenti and Agostino Poggi. Improving UML designs using automatic design pattern detection. In *In Proc. 12th. International Conference on Software Engineering and Knowledge Engineering (SEKE 2000)*, pages 336–343, 2000.
- Dirk Beyer, Andreas Noack, and Claus Lewerentz. Simple and Efficient Relational Querying of Software Structures. In *Proceedings of the 10th Working Conference on Reverse Engi-*

neering, WCRE '03, pages 216–, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-2027-8. URL <http://dl.acm.org/citation.cfm?id=950792.951386>.

James M. Bieman, Greg Straw, Huxia Wang, P. Willard Munger, and Roger T. Alexander. Design patterns and change proneness: An examination of five evolving systems. In *METRICS '03: Proceedings of the 9th International Symposium on Software Metrics*, page 40, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-1987-3.

Kevin Bierhoff and Jonathan Aldrich. Permissions to specify the composite design pattern. In *Proceedings of the Specification and Verification of Component-Based Systems (SAVCBS) Workshop*, 2008.

Alex Blewitt, Alan Bundy, and Ian Stark. Automatic Verification of Design Patterns in Java. In *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated Software Engineering*, pages 224–232, New York, NY, USA, 2005. ACM. ISBN 1-59593-993-4. doi: <http://doi.acm.org/10.1145/1101908.1101943>.

Grady Booch. *Object-Oriented Analysis and Design with Applications*. Addison-Wesley, 1994. ISBN 0805353402.

Egon Börger and Wolfram Schulte. A Programmer Friendly Modular Definition of the Semantics of Java. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *Lecture Notes in Computer Science*, pages 541–541. Springer Berlin / Heidelberg, 1999. URL http://dx.doi.org/10.1007/3-540-48737-9_10. 10.1007/3-540-48737-9_10.

Cristiano Calcagno, Dino Distefano, Peter W. O’Hearn, and Hongseok Yang. Footprint Analysis: A Shape Analysis That Discovers Preconditions. In *Static Analysis, 14th International Symposium, SAS 2007, Kongens Lyngby, Denmark, August 22-24, 2007, Proceedings*, pages 402–418, 2007.

David R. Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of Pointers and Structures. In *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation, PLDI '90*, pages 296–310, New York, NY, USA, 1990. ACM. ISBN 0-89791-364-7. doi: 10.1145/93542.93585. URL <http://doi.acm.org/10.1145/93542.93585>.

- John Cheesman and John Daniels. *UML components: a simple process for specifying component-based software*. Addison Wesley, 2001.
- Sigmund Cherem and Radu Rugina. A Practical Escape and Effect Analysis for Building Lightweight Method Summaries. In *Proceedings of the 16th international conference on Compiler construction, CC'07*, pages 172–186, Berlin, Heidelberg, 2007. Springer-Verlag. ISBN 978-3-540-71228-2. URL <http://dl.acm.org/citation.cfm?id=1759937.1759953>.
- Wei-Ngan Chin, Cristina David, Huu Hai Nguyen, and Shengchao Qin. Enhancing modular OO verification with separation logic. *SIGPLAN Not.*, 43(1):87–99, 2008. ISSN 0362-1340. doi: <http://doi.acm.org/10.1145/1328897.1328452>.
- E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, 1986. ISSN 0164-0925. doi: <http://doi.acm.org/10.1145/5397.5399>.
- Jame Coplien. C++ idioms. In *European Conference on Pattern Languages of Programs (EuroPLoP '98)*, 1998.
- Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252, New York, NY, USA, 1977. ACM. doi: <http://doi.acm.org/10.1145/512950.512973>.
- Werner Damm and David Harel. LSCs: Breathing Life into Message Sequence Charts. In *Formal Methods in System Design*, pages 293–312. Kluwer Academic Publishers, 1998.
- Christian Damus. Re: transitive closure, 2007. URL <http://dev.eclipse.org/newslists/news.eclipse.modeling.mdt.oc1/msg00538.html>.
<http://dev.eclipse.org/newslists/news.eclipse.modeling.mdt.oc1/msg00538.html>.
- A. De Lucia, V. Deufemia, C. Gravino, and M. Risi. Behavioral Pattern Identification through Visual Language Parsing and Code Instrumentation. In *Software Maintenance and Reengineering, 2009. CSMR '09. 13th European Conference on*, pages 99–108, march 2009. doi: 10.1109/CSMR.2009.29.

Birgit Demuth and Claas Wilke. Model and Object Verification by Using Dresden OCL. In *Proceedings of the Russian-German Workshop "Innovation Information Technologies: theory and practice"*, July 25-31, Ufa, Russia, 2009, 2009.

Dinakar Dhurjati, Manuvir Das, and Yue Yang. Path-Sensitive Dataflow Analysis with Iterative Refinement. In *SAS'06: The 13th International Static Analysis Symposium, Seoul, August 2006*, pages 425–442. Springer Berlin / Heidelberg, 2006. doi: 10.1007/11823230\27. URL http://dx.doi.org/10.1007/11823230_27.

Isil Dillig, Thomas Dillig, and Alex Aiken. Sound, Complete and Scalable Path-Sensitive Analysis. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation, PLDI '08*, pages 270–280, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-860-2. doi: 10.1145/1375581.1375615. URL <http://doi.acm.org/10.1145/1375581.1375615>.

Dino Distefano and Matthew J. Parkinson J. jStar: Towards Practical Verification for Java. In *OOPSLA '08: Proceedings of the 23rd ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 213–226, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-215-3. doi: <http://doi.acm.org/10.1145/1449764.1449782>.

Jing Dong, Paulo Alencar, and Donald Cowan. *Formal Specification and Verification of Design Patterns*, pages 94–108. IGI Publishing, 2007.

Jing Dong, Yajing Zhao, and Tu Peng. A review of design pattern mining techniques. *International Journal of Software Engineering and Knowledge Engineering (IJSEKE)*, 19(6):823–855, 2009.

E. Durr and J. van Katwijk. Vdm++, a formal specification language for object-oriented designs. In *CompEuro '92 . 'Computer Systems and Software Engineering', Proceedings.*, pages 214 –219, may 1992. doi: 10.1109/CMPEUR.1992.218511.

Eclipse. Eclipse IDE, 2012. URL <http://www.eclipse.org/>. Last accessed: 30th April 2012.

Amnon H Eden. Formal Specification of Object-Oriented Design. In *Proceedings of the International Conference on Multidisciplinary Design in Engineering*, 2001.

- Amnon H. Eden. Object-oriented modelling in lepus3 and class-z, 2008. URL <http://www.lepus.org.uk/ref/lepus3-tutorial.pdf>. <http://www.lepus.org.uk/ref/lepus3-tutorial.pdf> last accessed: 21st July 2008.
- Amnon H. Eden. Formal and precise software pattern representation languages, 2012. URL <http://www.eden-study.org/>. <http://www.eden-study.org> last accessed April 2012.
- Amnon H. Eden and Rick Kazman. Architecture, design, implementation. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 149–159, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-1877-X.
- E. Allen Emerson. Temporal and modal logic. *Handbook of theoretical computer science (vol. B): formal models and semantics*, pages 995–1072, 1990.
- Thomas Erl. *SOA Design Patterns*. Prentice Hall, 2008. ISBN 0136135161.
- David Evans. Static/dynamic analysis: Past, present and future. In *Verification Grand Challenge Workshop 2005, SRI Menlo Park*, 2005. www.cs.virginia.edu/~evans/talks/static-dynamic.ppt.
- Azadeh Farzan, Feng Chen, José Meseguer, and Grigore Roşu. Formal Analysis of Java Programs in JavaFAN. In *Proceedings of Computer-aided Verification (CAV'04)*, volume 3114 of *LNCS*, pages 501 – 505, 2004.
- R. Ferenc, A. Beszedes, L. Fulop, and J. Lele. Design pattern mining enhanced by machine learning. In *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on*, pages 295 – 304, sept. 2005. doi: 10.1109/ICSM.2005.40.
- Martin Fowler and Kendall Scott. *UML distilled (2nd ed.): a brief guide to the standard object modeling language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000. ISBN 0-201-65783-X.
- Robert B. France, Dae-Kyoo Kim, Sudipto Ghosh, and Eunjee Song. A UML-Based Pattern Specification Technique. *IEEE Transactions on Software Engineering*, 30(3):193–206, 2004. ISSN 0098-5589. doi: <http://dx.doi.org/10.1109/TSE.2004.1271174>.
- Robert B. France, Sudipto Ghosh, Trung T. Dinh-Trong, and Arnor Solberg. Model-driven development using UML 2.0: Promises and pitfalls. *IEEE Computer*, 39(2):59–66, 2006.

- Lajos Jenó Fulop, Rudolf Ferenc, and Tibor Gyimóthy. Towards a Benchmark for Evaluating Design Pattern Miner Tools. In *Proceedings of the 2008 12th European Conference on Software Maintenance and Reengineering*, 2008.
- Erich Gamma and Kent Beck. *Contributing to Eclipse: Principles, Patterns, and Plugins*. Addison Wesley, 2003. ISBN 0321205758.
- Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
- David Garlan and Mary Shaw. An introduction to software architecture. In V. Ambriola and G. Tortora, editors, *Advances in Software Engineering and Knowledge Engineering*, pages 1–39, Singapore, 1994. World Scientific Publishing Company.
- Gonzalo Génova. What is a metamodel: the OMG’s metamodeling infrastructure. Technical report, Universidad Carlos III de Madrid, 2009.
- James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison-Wesley, 2005. ISBN 0201310082.
- David Grove, Greg DeFouw, Jeffrey Dean, and Craig Chambers. Call Graph Construction in Object-Oriented Languages. In *Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, OOPSLA '97*, pages 108–124, New York, NY, USA, 1997. ACM. ISBN 0-89791-908-4. doi: 10.1145/263698.264352. URL <http://doi.acm.org/10.1145/263698.264352>.
- Y.-G. Guéhéneuc. P-MARt: Pattern-like Micro-Architecture Repository, 2007. URL <http://www.ptidej.net/downloads/pmart/>. Last accessed: 10th April 2012.
- Y.-G. Guéhéneuc and G. Antoniol. DeMIMA: A Multilayered Approach for Design Pattern Identification. *Software Engineering, IEEE Transactions on*, 34(5):667–684, Sept.-Oct. 2008. ISSN 0098-5589. doi: 10.1109/TSE.2008.48.
- Richard Helm, Ian M. Holland, and Dipayan Gangopadhyay. Contracts: Specifying Behavioral Compositions in Object-Oriented Systems. *SIGPLAN Not.*, 25(10):169–180, 1990. ISSN 0362-1340. doi: <http://doi.acm.org/10.1145/97946.97967>.
- D. Heuzeroth, S. Mandel, and W. Lowe. Generating Design Pattern Detectors from Pattern Specifications. *Automated Software Engineering, 2003. Proceedings. 18th IEEE Interna-*

- tional Conference on*, pages 245–248, Oct. 2003. ISSN 1527-1366. doi: 10.1109/ASE.2003.1240313.
- Michael Hind. Pointer analysis: Haven't we solved this problem yet? In *PASTE'01*, pages 54–61. ACM Press, 2001.
- C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall International, 1985. ISBN 0131532715.
- Simon Hofer. Verification of design patterns. Master's thesis, Chair of Programming Methodology, Department of Computer Science, ETH Zurich, 2009.
- IBM. Jikes compiler, 2005. URL <http://jikes.sourceforge.net/>.
<http://jikes.sourceforge.net/>.
- Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11:256–290, April 2002. ISSN 1049-331X. doi: <http://doi.acm.org/10.1145/505145.505149>. URL <http://doi.acm.org/10.1145/505145.505149>.
- Neil D. Jones and Steven S. Muchnick. Flow Analysis and Optimization of LISP-like Structures. In *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '79, pages 244–256, New York, NY, USA, 1979. ACM. doi: 10.1145/567752.567776. URL <http://doi.acm.org/10.1145/567752.567776>.
- O. Kaczor, Y.-G. Guéhéneuc, and S. Hamel. Efficient Identification of Design Patterns with Bit-Vector Algorithm. In *Software Maintenance and Reengineering, 2006. CSMR 2006. Proceedings of the 10th European Conference on*, pages 10 pp. –184, march 2006. doi: 10.1109/CSMR.2006.25.
- Wolfram Kaiser. Become a programming Picasso with JHotDraw. JavaWorld, February 2001. URL <http://www.javaworld.com/javaworld/jw-02-2001/jw-0216-jhotdraw.html>.
<http://www.javaworld.com/javaworld/jw-02-2001/jw-0216-jhotdraw.html>.
- Dae-Kyoo Kim. *A META-MODELING APPROACH TO SPECIFYING PATTERNS*. PhD thesis, Colorado State University, Fort Collins, Colorado, 2004.
- Deokhwan Kim and Martin C. Rinard. Verification of semantic commutativity conditions and inverse operations on linked data structures. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI

- '11, pages 528–541, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0663-8. doi: 10.1145/1993498.1993561. URL <http://doi.acm.org/10.1145/1993498.1993561>.
- John Knudsen, Anders P. Ravn, and Arne Skou. Design verification patterns. In Cliff B. Jones, Zhiming Liu, and Jim Woodcock, editors, *Formal methods and hybrid real-time systems*, pages 399–413. Springer-Verlag, Berlin, Heidelberg, 2007. ISBN 3-540-75220-X, 978-3-540-75220-2. URL <http://dl.acm.org/citation.cfm?id=1793874.1793892>.
- C. Kramer and L. Prechelt. Design Recovery by Automated Search for Structural Design Patterns in Object-Oriented Software. In *Reverse Engineering, 1996., Proceedings of the Third Working Conference on*, pages 208–215, nov 1996. doi: 10.1109/WCRE.1996.558905.
- Neelakantan R. Krishnaswami, Jonathan Aldrich, Lars Birkedal, Kasper Svendsen, and Alexandre Buisse. Design patterns in separation logic. In *Proceedings of the 4th international workshop on Types in language design and implementation, TLDI '09*, pages 105–116, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-420-1. doi: 10.1145/1481861.1481874. URL <http://doi.acm.org/10.1145/1481861.1481874>.
- Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, 1994. ISSN 0164-0925. doi: <http://doi.acm.org/10.1145/177492.177726>.
- William Landi. Undecidability of Static Analysis. *ACM Lett. Program. Lang. Syst.*, 1(4): 323–337, 1992. ISSN 1057-4514. doi: <http://doi.acm.org/10.1145/161494.161501>.
- K Lano, J Bicarregui, and S Goldsack. Formalising Design Patterns. In *RBCS-FACS Northern Formal Methods Workshop*, 1996.
- Craig Larman. UML Interaction Diagrams: Basic Sequence Diagram Notation, 2005. URL <http://www.informit.com/articles/article.aspx?p=360441&seqNum=5>. Last accessed: 17th March 2011.
- Anthony Lauder and Stuart Kent. Precise Visual Specification of Design Patterns. In *ECOOP '98: Proceedings of the 12th European Conference on Object-Oriented Programming*, pages 114–134, London, UK, 1998. Springer-Verlag. ISBN 3-540-64737-6.
- Alain Le Guennec, Gerson Sunyé, and Jean-Marc Jézéquel. Precise Modeling of Design Patterns. In *In Proceedings of UML'00*, pages 482–496. Springer Verlag, 2000.

- Gary T. Leavens, Albert L Baker, and Clyde Ruby. Preliminary design of JML: a behavioral interface specification language for Java. *SIGSOFT Softw. Eng. Notes*, 31(3):1–38, 2006. ISSN 0163-5948. doi: <http://doi.acm.org/10.1145/1127878.1127884>.
- Gary T. Leavens, K. Rustan M. Leino, and Peter Müller. Specification and verification challenges for sequential object-oriented programs. *Form. Asp. Comput.*, 19(2):159–189, 2007. ISSN 0934-5043. doi: <http://dx.doi.org/10.1007/s00165-007-0026-7>.
- Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6):1811–1841, 1994. ISSN 0164-0925. doi: <http://doi.acm.org/10.1145/197320.197383>.
- Howard C. Lovatt, Anthony M. Sloane, and Dominic R. Verity. A pattern enforcing compiler (pec) for java: using the compiler. In *Proceedings of the 2nd Asia-Pacific conference on Conceptual modelling - Volume 43*, APCCM '05, pages 69–78, Darlinghurst, Australia, Australia, 2005. Australian Computer Society, Inc. ISBN 1-920-68225-2. URL <http://dl.acm.org/citation.cfm?id=1082276.1082285>.
- Mass Soldal Lund and Ketil Stølen. A Fully General Operational Semantics for UML 2.0 Sequence Diagrams with Potential and Mandatory Choice. In *FM 2006: Formal Methods*, 2006.
- Jeffrey K. H. Mak, Clifford S. T. Choy, and Daniel P. K. Lun. Precise Modeling of Design Patterns in UML. In *ICSE '04*, pages 252–261, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2163-0.
- David Mapelsden, John Hosking, and John Grundy. Design pattern modelling and instantiation using DPML. In *CRPIT '02: Proceedings of the Fortieth International Conference on Tools Pacific*, pages 3–11, Darlinghurst, Australia, Australia, 2002. Australian Computer Society, Inc. ISBN 0-909925-88-7.
- David Maplesden, John Hosking, and John Grundy. *A Visual Language for Design Pattern Modeling and Instantiation*, pages 20–43. IGI Publishing, 2007.
- Slaviša Marković and Thomas Baar. Refactoring OCL Annotated UML Class Diagrams. In *Model-Driven Engineering, Languages and Systems (MoDELS)*, pages 280–294, 2005.
- Bertrand Meyer. *Object-Oriented Software Construction, Second Edition*. Prentice Hall Professional Technical Reference, 1997. ISBN 0136291554.

Sun Microsystems. Java 2 Software Development Kit, 2010. URL <http://docs.oracle.com/javase/1.4.2/docs/>.

T. Mikkonen. Formalizing design patterns. *Software Engineering, 1998. Proceedings of the 1998 International Conference on*, pages 115–124, Apr 1998. ISSN 0270-5257. doi: 10.1109/ICSE.1998.671108.

Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized Object Sensitivity for Points-to Analysis for Java. *ACM Trans. Softw. Eng. Methodol.*, 14(1):1–41, January 2005. ISSN 1049-331X. doi: 10.1145/1044834.1044835. URL <http://doi.acm.org/10.1145/1044834.1044835>.

Peter Müller, Arnd Poetzsch-Heffter, and Gary T. Leavens. Modular invariants for layered object structures. *Sci. Comput. Program.*, 62(3):253–286, 2006. ISSN 0167-6423. doi: <http://dx.doi.org/10.1016/j.scico.2006.03.001>.

Janice Ka-Yee Ng. Identification of Behavioral and Creational Design Patterns through Dynamic Analysis. Technical report, Université de Montréal, 2008.

Janice Ka-Yee Ng, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. Identification of Behavioural and Creational Design Motifs through Dynamic Analysis. *J. Softw. Maint. Evol.*, 22:597–627, December 2010. ISSN 1532-060X. doi: <http://dx.doi.org/10.1002/smr.421>. URL <http://dx.doi.org/10.1002/smr.421>.

Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, 1999. ISBN 3540654100.

OMG. Object Constraint Language, Version 2.0, 2006. URL <http://www.omg.org/cgi-bin/doc?formal/2006-05-01>.

OMG. Unified Modeling Language: Superstructure <http://www.omg.org/docs/formal/09-02-02.pdf>, 2009.

OMG. Object Constraint Language, Version 2.2, 2010a. URL <http://www.omg.org/spec/OCL/2.2/>. <http://www.omg.org/spec/OCL/2.2/>.

OMG. Unified Modeling Language: Superstructure <http://www.omg.org/spec/uml/2.3/superstructure/pdf/>, 2010b.

- Matthew J. Parkinson. *Local reasoning for Java*. PhD thesis, University of Cambridge Computer Laboratory, 2005.
- Matthew J. Parkinson and Gavin M. Bierman. Separation logic, abstraction and inheritance. In *POPL '08: Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 75–86, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-689-9. doi: <http://doi.acm.org/10.1145/1328438.1328451>.
- Tu Peng, Jing Dong, and Yajing Zhao. Verifying behavioral correctness of design pattern implementation. In *SEKE*, pages 454–459, 2008.
- Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, 1992. URL citeseer.ist.psu.edu/perry92foundation.html.
- Niklas Pettersson. Design Pattern Detection Evaluation Suite (DPDES), 2010. URL <http://w3.msi.vxu.se/~npe/DPDES/>. Last accessed: 10th April 2012.
- Niklas Pettersson, Welf Lowe, and Joakim Nivre. Evaluation of Accuracy in Design Pattern Occurrence Detection. *IEEE Transactions on Software Engineering*, 99(RapidPosts), 2009. ISSN 0098-5589. doi: <http://doi.ieeecomputersociety.org/10.1109/TSE.2009.92>.
- Shaz Qadeer and Jakob Rehof. Context-bounded model checking of concurrent software. In *In TACAS*, pages 93–107. Springer, 2005.
- G. Ramalingam. The Undecidability of Aliasing. *ACM Transactions on Programming Languages and Systems*, 16(5):1467–1471, 1994. ISSN 0164-0925. doi: <http://doi.acm.org/10.1145/186025.186041>.
- G. Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM Trans. Program. Lang. Syst.*, 22(2):416–430, 2000. ISSN 0164-0925. doi: <http://doi.acm.org/10.1145/349214.349241>.
- Ghulam Rasool, Patrick Maeder, and Ilka Philippow. Evaluation of design pattern recovery tools. *Procedia Computer Science*, 3(0):813 – 819, 2011. ISSN 1877-0509. doi: 10.1016/j.procs.2010.12.134. URL <http://www.sciencedirect.com/science/article/pii/S1877050910005090>. World Conference on Information Technology.

Rational®. Rational Software Architect <http://www-306.ibm.com/software/awdtools/architect/swarchitect/>, 2007. URL <http://www-306.ibm.com/software/awdtools/architect/swarchitect/>.

Thomas Reps. Undecidability of context-sensitive data-independence analysis. *ACM Trans. Program. Lang. Syst.*, 22(1):162–186, 2000. ISSN 0164-0925. doi: <http://doi.acm.org/10.1145/345099.345137>.

John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science, LICS '02*, pages 55–74, Washington, DC, USA, 2002. IEEE Computer Society. ISBN 0-7695-1483-9. URL <http://dl.acm.org/citation.cfm?id=645683.664578>.

Noam Rinetzkyy, Mooly Sagiv, and Eran Yahav. Interprocedural Shape Analysis for Cutpoint-Free Programs. In *In SAS'05: Static Analysis Symposium, volume 3672 of LNCS*, pages 284–302. Springer, 2005.

Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. Liquid types. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation, PLDI '08*, pages 159–169, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-860-2. doi: 10.1145/1375581.1375602. URL <http://doi.acm.org/10.1145/1375581.1375602>.

Stan Rosenberg, Anindya Banerjee, and David A. Naumann. Local reasoning and dynamic framing for the composite pattern and its clients. In *Proceedings of the Third international conference on Verified software: theories, tools, experiments, VSTTE'10*, pages 183–198, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-15056-X, 978-3-642-15056-2. URL <http://dl.acm.org/citation.cfm?id=1884866.1884887>.

James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-oriented modeling and design*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1991. ISBN 0-13-629841-9.

James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language reference manual*. Addison-Wesley, 1999. ISBN 0-201-30998-X.

Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric Shape Analysis via 3-Valued Logic. *ACM Transactions on Programming Languages and Systems*, 24(3):217–298, 2002. ISSN 0164-0925. doi: <http://doi.acm.org/10.1145/514188.514190>.

- David A. Schmidt. Data Flow Analysis is Model Checking of Abstract Interpretations. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '98, pages 38–48, New York, NY, USA, 1998. ACM. ISBN 0-89791-979-3. doi: 10.1145/268946.268950. URL <http://doi.acm.org/10.1145/268946.268950>.
- Douglas C. Schmidt, Michael Stal, Hans Rohert, and Frank Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*. John Wiley and Sons, 2000.
- Bran Selic. The Pragmatics of Model-Driven Development. *IEEE Software*, 20(5):19–25, 2003. ISSN 0740-7459. doi: <http://doi.ieeecomputersociety.org/10.1109/MS.2003.1231146>.
- William R. Shadish, Thomas D. Cook, and Donald T. Campbell. *Experimental and Quasi-Experimental Designs for Generalized Causal Inference*. Houghton Mifflin, 2002. ISBN 0395615569.
- Shravan Shetty and Vinod Menezes. Verification of architectural rules and design patterns. Master's thesis, Department of Mathematics and Computer Science, Eindhoven University of Technology, 2011.
- Nija Shi. *Reverse Engineering of Design Patterns from Java Source Code*. PhD thesis, Computer Science, University of California, Davis, 2007a.
- Nija Shi. PINOT: Pattern INference and recOvery Tool, 2007b. URL <http://www.cs.ucdavis.edu/~shini/research/pinot/>. Last accessed: 7th March 2012.
- Nija Shi and Ronald A. Olsson. Reverse Engineering of Design Patterns from Java Source Code. In *ASE '06: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, pages 123–134, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2579-2. doi: <http://dx.doi.org/10.1109/ASE.2006.57>.
- Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. Pick Your Contexts Well: Understanding Object-Sensitivity. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '11, pages 17–30, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0490-0. doi: 10.1145/1926385.1926390. URL <http://doi.acm.org/10.1145/1926385.1926390>.

J.M. Smith and D. Stotts. SPQR: flexible automated design pattern extraction from source code. *Automated Software Engineering, 2003. Proceedings. 18th IEEE International Conference on*, pages 215–224, Oct. 2003. ISSN 1527-1366. doi: 10.1109/ASE.2003.1240309.

Bernhard Steffen. Data Flow Analysis as Model Checking. In *Proceedings of the International Conference on Theoretical Aspects of Computer Software, TACS '91*, pages 346–365, London, UK, UK, 1991. Springer-Verlag. ISBN 3-540-54415-1. URL <http://dl.acm.org/citation.cfm?id=645867.670930>.

Krzysztof Stencel and Patrycja Wegrzynowicz. Implementation Variants of the Singleton Design Pattern. In *OTM Workshops*, pages 396–406, 2008.

Toufik Taibi. *Design Pattern Formalization Techniques*. IGI Publishing, 2008. ISBN 9781599042190.

Toufik Taibi and Taieb Mkadmi. Generating Java Code from Design Patterns Formalized in BPSL. *Innovations in Information Technology, 2006*, pages 1–5, Nov. 2006. doi: 10.1109/INNOVATIONS.2006.301944.

Toufik Taibi and David Chek Ling Ngo. Formal Specification of Design Patterns - A Balanced Approach. *Journal of Object Technology*, 2(4):127–140, 2003.

Toufik Taibi, Angel Herranz, and Juan Jose Moreno-Navarro. Stepwise refinement validation of design patterns formalized in tla+ using the tlc model checker. *Journal of Object Technology*, 8(2):137–161, March 2009. ISSN 1660-1769. doi: 10.5381/jot.2009.8.2.a3. URL http://www.jot.fm/contents/issue_2009_03/article3.html.

H. Theiling and C. Ferdinand. Combining Abstract Interpretation and ILP for Microarchitecture Modelling and Program Path Analysis. In *Real-Time Systems Symposium, 1998. Proceedings., The 19th IEEE*, pages 144–153, dec 1998. doi: 10.1109/REAL.1998.739739.

N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S.T. Halkidis. Design pattern detection using similarity scoring. *Software Engineering, IEEE Transactions on*, 32(11):896–909, nov. 2006. ISSN 0098-5589. doi: 10.1109/TSE.2006.112.

Nikos Tsantalis. Design Pattern Detection using Similarity Scoring, 2009. URL <http://java.uom.gr/~nikos/pattern-detection.html>. Last accessed: 8th April 2012.

- Jilles van Gorp and Jan Bosch. Design erosion: problems and causes. *Journal of Systems and Software*, 61(2):105 – 119, 2002. ISSN 0164-1212. doi: DOI:10.1016/S0164-1212(01)00152-2. URL <http://www.sciencedirect.com/science/article/B6V0N-44X09DV-1/2/5aeefd30957f7ffba93ce7d2426b2e2a>.
- Mandana Vaziri and Daniel Jackson. Some Shortcomings of OCL, the Object Constraint Language of UML. In *Proceedings of the Technology of Object-Oriented Languages and Systems (TOOLS 34'00)*, TOOLS '00, pages 555–, Washington, DC, USA, 2000. IEEE Computer Society. ISBN 0-7695-0774-3. URL <http://portal.acm.org/citation.cfm?id=832261.833293>.
- VDMTools. The Java to VDM++ User Manual ver.1.0. Technical report, CSK Corporation, 2010.
- Bernhard Volz and Stefan Jablonski. Towards an open meta modeling environment. In *Proceedings of the 10th Workshop on Domain-Specific Modeling, DSM '10*, pages 17:1–17:6, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0549-5. doi: 10.1145/2060329.2060366. URL <http://doi.acm.org/10.1145/2060329.2060366>.
- Markus von Detten. Towards systematic, comprehensive trace generation for behavioral pattern detection through symbolic execution. In *Proceedings of the 10th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools, PASTE '11*, pages 17–20, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0849-6. doi: 10.1145/2024569.2024573. URL <http://doi.acm.org/10.1145/2024569.2024573>.
- Patrycja Wegrzynowicz and Krzysztof Stencel. Towards a Comprehensive Test Suite for Detectors of Design Patterns. In *ASE '09: Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 103–110, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3891-4. doi: <http://dx.doi.org/10.1109/ASE.2009.85>.
- Lothar Wendehals. Specifying Patterns for Dynamic Pattern Instance Recognition with UML 2.0 Sequence Diagrams. In *Proceedings of the 6th Workshop Software Reengineering (WSR)*, volume 24/2, pages 63–64. Softwaretechnik-Trends, 2004.
- Lothar Wendehals and Alessandro Orso. Recognizing Behavioral Patterns at Runtime using Finite Automata. In *WODA '06: Proceedings of the 2006 international workshop*

on *Dynamic systems analysis*, pages 33–40, New York, NY, USA, 2006. ACM. ISBN 1-59593-400-6. doi: <http://doi.acm.org/10.1145/1138912.1138920>.

Jim Woodcock and Jim Davies. *Using Z: specification, refinement, and proof*. Prentice-Hall, 1996. ISBN 0139484728.

Zhi-Xiang Zhang, Qing-Hua Li, and Ke-Rong Ben. A New Method for Design Pattern Mining. In *Machine Learning and Cybernetics, 2004. Proceedings of 2004 International Conference on*, volume 3, pages 1755 – 1759 vol.3, aug. 2004. doi: 10.1109/ICMLC.2004.1382059.

Hong Zhu, I. Bayley, Lijun Shan, and R. Amphlett. Tool support for design pattern recognition at model level. In *Computer Software and Applications Conference, 2009. COMPSAC '09. 33rd Annual IEEE International*, volume 1, pages 228 –233, july 2009. doi: 10.1109/COMPSAC.2009.37.

Appendix F

List of Acronyms

- AF: Abstract Factory
- AVT: Alas Verification Tool
- AST: Abstract Syntax Tree
- BC: Badly configured
- BD: Behaviour Diagram
- BPSL: Balanced Pattern Specification Language (DPSL) [Taibi and Ngo, 2003]
- CFG: Control-Flow Graph
- CoR: Chain of Responsibility
- DA: Dynamic Analysis
- dComp: Destruction-only composition copystate
- DEEBEE: Design Pattern Evaluation Benchmark Environment (Benchmark) [Fulop et al., 2008]
- DFA: Data-Flow Analysis
- DPML: Design Pattern Modelling Language [Maplesden et al., 2007]
- DPSL: Design Pattern Specification Language
- DPVT: Design Pattern Verification Tool

- GEBNF: Graphic Extension of BNF (DPSL) [Bayley and Zhu, 2010]
- GoF: Gang of Four
- iComp: Initialization-only composition copystate
- OCL: Object Constraint Language [OMG, 2010a]
- OC/VDM++: Object Calculus/Vienna Development Method++ (DPSL) [Lano et al., 1996]
- OOML: Object-Oriented Modelling Language
- OOPL: Object-Oriented Programming Language
- PINOT: Pattern Inference and Recovery Tool (DPVT) [Shi and Olsson, 2006]
- P-MARt: Pattern-like Micro-Architecture Repository (Benchmark) [Guéhéneuc, 2007]
- SA: Static Analysis
- SD: Structure Diagram
- SanD: Static and Dynamic Specification Language (DPSL) [Heuzeroth et al., 2003]
- UC: Unknown configuration
- UML: Unified Modeling Language [OMG, 2009]
- WC: Well configured