

A Framework for Checking Programmer-Defined Constraints on the Definition and Use of Program Elements in Java

Boris Bokowski

Doctoral Dissertation
2000

submitted at the
Department of Mathematics and Computer Science
Freie Universität Berlin

Reviewers:

Prof. Dr. Klaus-Peter Löhner, Freie Universität Berlin
Prof. Dr. Stefan Jähnichen, Technische Universität Berlin
Prof. Dr. Theo D'Hondt, Vrije Universiteit Brussel

Datum der Disputation: 29. Juni 2000

Boris Bokowski
Am Elfengrund 31A
64297 Darmstadt
bokowski@acm.org

Contents

1	Introduction	13
1.1	CoffeeStrainer – a framework for checking constraints	14
1.2	Contributions of this thesis	14
1.3	Thesis Structure	15
1.4	Terminology	16
1.5	Acknowledgements	17
2	Categories of Constraints	19
2.1	Classification dimensions	20
2.1.1	Static constraints vs. dynamic constraints	20
2.1.2	Type constraints vs. method constraints vs. field constraints	21
2.1.3	Definition constraints vs. usage constraints	21
2.1.4	Resulting categories	24
2.2	Static constraints	24
2.2.1	Type definition constraints	24
2.2.2	Method definition constraints	26
2.2.3	Type usage constraints	27
2.2.4	Method usage constraints	29
2.2.5	Field usage constraints	31
2.3	Dynamic constraints	31
2.3.1	Accidentally dynamic constraints	32
2.3.2	Inherently dynamic constraints	34
2.4	Alternative categorizations	36
2.4.1	Categorization based on degree of abstraction	37
2.4.2	Categorization based on programming language concepts	37

Contents

3	CoffeeStrainer Explained	39
3.1	CoffeeStrainer basics	39
3.1.1	The abstract semantics graph	39
3.1.2	A simple example	41
3.1.3	The structure of CoffeeStrainer constraints	45
3.2	Implementing static constraints	47
3.2.1	Implementing type definition constraints	47
3.2.2	Implementing method definition constraints	51
3.2.3	Implementing type usage constraints	54
3.2.4	Implementing method usage constraints	60
3.2.5	Implementing field usage constraints	61
3.3	Implementing dynamic constraints	62
3.3.1	Implementing Eiffel-style preconditions using tags	65
4	CoffeeStrainer Virtues and Limitations	69
4.1	Comprehensiveness	69
4.1.1	Complete abstract semantics graph	69
4.1.2	Based on source code	70
4.1.3	Limitations	70
4.2	Pragmatic choices	71
4.2.1	Integration with base-level code	71
4.2.2	No change to base-level language	72
4.2.3	Java as the constraint language	72
4.2.4	Predefined tree traversal	73
4.2.5	Separate checking of compilation units	73
4.2.6	Efficiency	73
4.2.7	Openness	74
4.2.8	Limitations	74
4.3	Elegance	74
4.3.1	Modularity	75
4.3.2	Usage constraints	76
4.3.3	Combining static and dynamic constraints	77
4.3.4	Limitations	80

5	Extended Example: Confined Types	81
5.1	Introduction	81
5.2	Security in programming languages	83
5.3	The class signing example	84
5.3.1	The security breach in detail	84
5.3.2	Class signing with confined types	85
5.4	Anonymous methods	87
5.5	Confined types	89
5.5.1	Confinement in declarations	91
5.5.2	Preventing widening	94
5.5.3	Preventing hidden widening	95
5.5.4	Preventing transfer from the inside	95
5.5.5	Preventing transfer from the outside	96
5.6	Example: public-key cryptography	97
5.7	Implementing confined types	101
5.8	Related approaches	106
5.8.1	Alias control	106
5.8.2	Security	107
5.9	Discussion	107
5.9.1	Confined types and genericity	107
5.9.2	Strong encapsulation	108
5.9.3	Optimization	108
5.10	Conclusion	109
6	Implementation of CoffeeStrainer	111
6.1	Barat – a front-end for Java	111
6.1.1	ASG Nodes	112
6.1.2	Elements of the abstract semantics graph	116
6.1.3	Retrieving ASG root objects	131
6.1.4	Visitors	133
6.1.5	Attributes	137
6.1.6	Implementation of Barat	139
6.2	From Barat to CoffeeStrainer	142
6.3	Performance Evaluation	143

Contents

7	Related Work	145
7.1	CCEL	150
7.2	Law-Governed Architecture	151
7.3	Category Description Language	153
7.4	GENOA	155
7.5	ASTLOG	162
7.6	Summary	167
7.7	Other related work	169
7.7.1	Other systems similar to CoffeeStrainer	170
7.7.2	Related work on constraints	171
7.7.3	Avoiding constraints	172
8	Conclusions	173
8.1	Summary	173
8.2	Constraints – a sign of bad design?	173
8.3	Directions for future work	175

List of example constraints

All fields should be declared <code>private</code> (implemented by <code>AllFieldsArePrivate</code>).....	41
Superclass needs no-arg constructor (occurs in <code>java.io.Serializable</code>).....	49
Overriding method should call <code>super</code> (occurs in <code>java.awt.Container</code>).....	52
Do not synchronize on <code>Writer</code> objects (occurs in <code>java.io.Writer</code>).....	59
Method should not be called by application code (occurs in <code>java.awt.Component</code>).....	60
Field accesses should be in <code>synchronized</code> method (occurs in <code>java.beans.beancontext.BeanContextSupport</code>).....	62
Caller needs valid certificate (implemented by <code>SecureObject</code>).....	63
Method preconditions (implemented by <code>ProgrammingByContract</code>).....	68
Do not compare object references using <code>==</code> (implemented by <code>IdentityComparisonDisallowed</code>).....	77
Null value is invalid for this type (implemented by <code>NullValueInvalid</code>).....	78
Do not transfer references to confined objects out of confining package (implemented by <code>ConfinedType</code>).....	102
Persistency framework (implemented by <code>Storable</code>).....	149

Contents

List of Figures

2.1	Constraint categories	24
2.2	Static type definition constraints per package	25
2.3	Static method definition constraints per package	27
2.4	Static type usage constraints per package	28
2.5	Static method usage constraints per package	30
2.6	Genericity constraints per package	33
2.7	Aliasing constraints per package	33
2.8	Sequencing constraints per package	34
2.9	Other dynamic constraints per package	36
3.1	An example abstract semantics graph (ASG).	40
3.2	Generated constraint class for <code>AllFieldsArePrivate</code>	42
3.3	Student example	43
3.4	Definition constraint methods called for the example ASG	48
3.5	Names and signatures of definition constraint methods (Part I)	50
3.6	Names and signatures of definition constraint methods (Part II)	51
3.7	The ASG of class <code>Container</code> and a subclass	53
3.8	Constraint class generated for a method constraint	55
3.9	Usage constraint methods called for the example ASG	56
3.10	Names and signatures of usage constraint methods	58
3.11	Names and signatures of method usage constraint methods	60
3.12	Names and signatures of field usage constraint methods	61
3.13	Simple implementation of preconditions	67
3.14	Complete implementation of preconditions	68
5.1	Signatures without confined types	84
5.2	An ad-hoc fix of the security problem	85

List of Figures

5.3	Signatures with confined types	86
5.4	Examples for anonymous methods	87
5.5	Anonymous methods in existing code.	89
5.6	References between objects in inside and outside packages.	90
5.7	Transferring references, package <code>inside</code>	92
5.8	Transferring references, package <code>outside</code>	93
5.9	Inheritance and usage relationships between package <code>rsa</code> and package <code>secure</code>	98
5.10	Package containing RSA algorithm	99
5.11	Confining a type in a different package	100
5.12	Implementation of <code>ConfinedType</code> (Part I – implementing C1, C2, C6)	102
5.13	Implementation of <code>ConfinedType</code> (Part II – implementing C3)	103
5.14	Implementation of <code>ConfinedType</code> (Part III - implementing C4, C5, C7, C8)	104
5.15	Implementation of constraints for anonymous methods	105
6.1	The interface <code>Node</code>	113
6.2	Interface of class <code>barat.Barat</code>	132
6.3	Example for visitor pattern	133
6.4	Screendump of graphical user interface	143
6.5	Comparing the performance of <code>CoffeeStrainer</code> and Sun’s <code>JAVAC</code>	144
7.1	The example framework	147
7.2	Example of using the persistency framework	148
7.3	Implementation of constraints P1-P4 using <code>CoffeeStrainer</code>	149
7.4	Implementation of P1 using <code>CCEL</code>	150
7.5	Top-level <code>GENOA</code> procedure	156
7.6	Checking P1 with <code>GENOA</code> : procedure <code>HasPublicEmptyNoArgCtor</code>	157
7.7	Checking P2 with <code>GENOA</code> : procedure <code>DontCallNotifyRetrieved</code>	159
7.8	Checking P4 with <code>GENOA</code> : <code>WrapWriteAccessInTryFinally</code>	161
7.9	Helper functions for implementing P4	162
7.10	Implementing P1, P2, and P4 as a top-level <code>ASTLOG</code> predicate	163
7.11	Implementing P1 with <code>ASTLOG</code>	164
7.12	<code>ASTLOG</code> utility predicates	165
7.13	Implementing P2 with <code>ASTLOG</code>	165

List of Figures

7.14	Implementation of P4 with ASTLOG	166
7.15	Lines of code needed for implementing the example constraints	169
7.16	Summarized comparison of systems for static constraint checking	170

List of Figures