1 Introduction

For functioning correctly, complex object-oriented software relies on application-specific *constraints* regarding the definition and use of program elements. Constraints are formalized programming rules that can be checked automatically. This thesis presents a framework for checking programmer-defined constraints.

Like strong typing in object-oriented programming languages, they either constrain how *program elements*, i.e., interfaces, classes, methods, and fields, have to be *defined* in a certain context, or they constrain how program elements have to be *used*. Unlike strong typing, which prevents errors on the level of the execution environment, constraints prevent errors on the semantic level of the program. This applies in particular to software that uses object-oriented frameworks [Lewis et al. 1995], which often make non-trivial assumptions on their correct adaptation and usage.

Semantic errors caused by violation of constraints may or may not manifest themselves as run-time errors on the level of the execution environment. In the case where constraint violations do not cause run-time errors, they lead to other undesirable effects, such as performance problems, semantic errors at the problem domain level, deadlocks, or maintainability problems due to violations of structural constraints. In the case where constraint violations do cause run-time errors, there is often no apparent connection to the place in the program which caused the constraint violation.

For example, a software system might be designed using a layered software architecture, in which each class belongs to a specific layer of the system. Using constraints, it is possible to allow method calls only between classes within one layer or from a higher layer to a lower layer. In most cases, a violation of this constraint, although creating undesired interactions that may lead to maintainability problems, will not manifest itself as a run-time error. However, if the initialization of the system proceeds from lower layers to higher layers, a constraint-violating call from a lower layer to a higher layer during the initialization phase might cause accesses to uninitialized data, which might lead to immediate run-time errors, or to run-time errors which occur at a later time.

It is important to note that constraints are created by *programmers* who want their classes to be used or extended only in a certain way. Thus, it is important that tools for checking constraints be useable both for advanced programmers who specify constraints and for everyday programmers whose code is checked against the constraints. Ideally, constraint specifications can serve as a means of formalizing assumptions underlying a certain software design and thus as a way of communicating these assumptions between different software developers.

We believe that it is important to study constraints in the context of a real language which is widely used. Therefore, the discussion in this thesis is based on the programming language

Java [Gosling et al. 1996], and we have not tried to make it language-independent.

1.1 CoffeeStrainer – a framework for checking constraints

This thesis presents a framework, called CoffeeStrainer, which allows to check programmer-defined constraints for Java. A previous version of CoffeeStrainer is described in [Bokowski 1999]. CoffeeStrainer constraints are unique in that they are modular, extensible and composable, and special support is provided for constraints on the usage of program elements. Additionally, CoffeeStrainer constraints can consist of static (compile-time) and dynamic (run-time) parts. CoffeeStrainer has been fully implemented. It supports separate checking of compilation units, and its performance in terms of static checking time is comparable to running a compiler.

Unlike previous work [Devanbu 1992, Chowdhury, Meyers 1993, Klarlund et al. 1996, Minsky 1996, Crew 1997] (see Chapter 7 for a detailed comparison), CoffeeStrainer takes a pragmatic approach and does not define a special-purpose constraint language. Instead, constraints are specified using Java, so that the programmer need not learn new syntax. This choice is based on the observation that many programmers like to stick to the syntax they have used for some time already, as demonstrated by the success of Java whose syntax is based on the widely-used programming language C++ [Stroustrup 1991].

Constraint code is embedded in Javadoc comments [Gosling et al. 1996]. Thus, constraint code and base-level code share the same structure, making it easy to find the rules that apply to a given part of the program, and allowing arbitrary compilers and tools to be applied to the source code that contains constraints. When defining a new rule, the programmer has access to a complete abstract syntax tree of the program that is to be checked.

Constraints refer to entities of the *abstract semantics graph* (ASG) [Devanbu et al. 1996] of a program, an abstract syntax tree [Aho et al. 1985] which is augmented with information gathered from name and type analysis. Thus, the primary focus of CoffeeStrainer is on *static constraints*, i.e. rules that can be automatically checked at compile-time. Tools or language support for checking *dynamic constraints*, first introduced under the name *assertions* [Floyd 1967], are well known and widely used [Meyer 1992, Meyer 1997]. CoffeeStrainer's contribution in this area is the integration of dynamic and static constraints in a single framework, which is unique in that it allows to insert run-time checks programmatically at arbitrary places of the program.

1.2 Contributions of this thesis

This thesis makes contributions in two different but related areas.

The main contributions are in the area of tools for checking programmer-defined constraints. CoffeeStrainer makes the following original contributions:

- *modular constraints*: Constraints are bound to classes, interfaces, or methods — there is no global set of checked constraints. Thus, compilation units can be checked independently, making the tool usable even for large systems. Moreover, constraints even from different sources, and referring to different parts of the program are combined in a natural way as the program is combined from those parts. Finally, by using inheritance, programmers can extend and refine constraints incrementally.

- *openness:* The system is implemented as an open object-oriented framework that executes programmer-defined constraint code at compile-time; it can be extended and modified by defining new object-oriented abstractions that are used by the constraint code.
- *usage constraints*: Like other tools, CoffeeStrainer supports constraints that refer to the *definition* of program elements. Unlike other tools, CoffeeStrainer also supports constraints that refer to the *usage* of program elements in other contexts, numerous examples of which have been found in existing software.
- *accessibility for practicians*: Unlike other approaches, CoffeeStrainer does not define a new constraint language; instead, it uses Java for constraints as well, considerably reducing the effort that would be required from a proficient Java programmer to start specifying constraints.

On top of that, this thesis contains additional contributions in the area of automatically checkable constraints:

- *study of constraints in existing software*: Often, constraints are already documented in existing software, but they cannot be checked because they are not formalized. We have examined a well-known set of classes, the Java standard library classes, for constraints that may be checked automatically (Chapter 2).
- *useful example constraints*: Throughout the thesis, a number of example constraints are presented in detail. The constraints are typical for modern object-oriented, framework-based software development, taken from actual experience in building and examining object-oriented software (see the list of example constraints on page 7).
- *confined types*: As an extended, non-trivial example, a system of constraints for strong encapsulation is presented. Confined types are useful for constructing secure software in the presence of dynamically-loaded, untrusted code by separating the types in a package into two different sets: While objects of ordinary, unconfined types form the public interface of the package, objects of confined types form the secure kernel of the package which can only be accessed from within the package (Chapter 5).

1.3 Thesis Structure

The remaining chapters of this thesis are organized as follows:

- Chapter 2 consists of a study of the Java standard classes, and presents several example constraints taken from the standard classes' documentation.

- Chapter 3 explains CoffeeStrainer, namely, how constraints can be specified and how they are checked at compile-time and at run-time.
- Chapter 4 discussed CoffeeStrainer's virtues and limitations .
- Chapter 5 gives an extended example for using CoffeeStrainer in the area of software-based security. This chapter is based on a paper co-authored by Jan Vitek [Bokowski, Vitek 1999].
- Chapter 6 contains information about the implementation of CoffeeStrainer.
- Chapter 7 compares CoffeeStrainer to related work.
- Chapter 8 draws conclusions and points out directions for future work.

1.4 Terminology

In this thesis, the terminology used for describing entities of the Java language is taken from the Java language specification [Gosling et al. 1996] wherever possible. For readers who are not familiar with the Java terminology, here are brief definitions of some key terms:

A Java *program* is made up of *compilation units* containing definitions of *classes* and/or *interfaces*. In Java, all variables and expressions are *statically typed*.

There are a number of *primitive types* — `boolean`, `byte`, `char`, `short`, `int`, `long`, `float`, and `double` — and *reference types*. Reference types are either *class types*, *interface types*, *array types*, or the special *null type*, the static type of the constant `null`.

There are rules for when one type is *assignable* to another type, i.e. when a value of a certain type may be assigned to a variable of another type. For primitive types, there are certain allowed *type conversions* that determine whether one primitive type is assignable to another primitive type. Primitive types are assignable only to other primitive types. Likewise, reference types are only assignable to other reference types. A reference type S is assignable to another reference type T iff S is equal to T or S is a *subtype* of T. The null type is a subtype of every reference type, and all reference types are subtypes of `java.lang.Object`. An array type AS is a subtype of another array type AT if the element type S of AS is a subtype of the element type T of AT; if either of the element types of AS or AT is a primitive type, AS and AT are not subtypes of each other. A class type is a subtype of its declared *superclass*; if a class does not declare a superclass, its superclass is `java.lang.Object`. An interface type is a subtype of `java.lang.Object` and of all declared *superinterfaces*. Subtyping is transitive, i.e., if S is a subtype of T and R is a subtype of S, then R is also a subtype of T. Sometimes, we will refer to the *supertypes* of a type S, the set of all types of which S is a subtype.

Classes and interfaces are *user-defined types*. They may contain *constructors* (classes only), *methods*, and *fields* (some restrictions apply). Constructors are invoked upon creation of an *object*, or *instance*, of a class. Like methods, they may have a number of *parameters*, each having a type and a name. Unlike constructors, methods do have a *name*. They also may have a *return type*. The types of the parameters make up the *signature* of a method or constructor. Defining methods with the same name but different signatures is called *overloading*. It is not allowed to define two methods with the same name and signature, but different return types. If a method is not declared `static`, it is called an *instance method*. When two instance methods with the same name and signature are defined in a type T and a subtype S of T, the method in S is said to *override* the method in T, and the return types of both methods must be the same. An instance method may be declared *abstract* or *concrete*. Concrete methods and static methods contain a *method body*, consisting of statements that make up the method's implementation. An abstract method has no method body. A class containing one or more abstract methods is an *abstract class*; interfaces must only contain abstract methods. *Concrete classes* can be instantiated (abstract classes and interfaces cannot). All abstract methods of supertypes of a concrete class C must be overridden by a concrete method in either C itself or in a superclass of C that is a subtype of the type that contained the abstract method. A field consists of a name, a type, and an optional *initializer*, which is an expression which will be evaluated to initialize the field's value at runtime.

A Java *program* consists of all classes and interfaces that are needed to execute a certain *main method*, which is a method defined as `public static void main(String[] args)`. Because Java allows dynamic loading of classes which may depend on user input, the term "program" is a run-time notion. However, because only compiled classes can be loaded dynamically, one might think of a program as the set of all classes and interfaces that have been compiled by the Java compiler. Thus, *compiling a program* means to compile all compilation units that contain classes or interfaces that might be executed. Likewise, *checking a program* means to check these compilation units.

A *named program element* – or *program element* for short – is either a class, an interface, a method, or a field.

1.5 Acknowledgements

First of all, I would like to thank my advisor Peter Löhr for his support and guidance. His views on object-oriented – and other – languages, language extensions and software architecture has certainly shaped my way of thinking. He shared many of his ideas with me and pointed out several interesting areas of research.

I would also like to thank the external thesis reviewers. Theo D'Hondt of Vrije Universiteit Brussels, together with his research group, played a very important role on my way to this thesis. His interest in my work showed me that after three years, finally, I had found a topic that was worth pursuing. The second external reviewer, Stefan Jähnichen of Technical University Berlin, supported me in many ways. I am very grateful for his help with finding a research position at GMD-FIRST in Berlin.

For three years, I received a grant from the graduate college "communication-based systems". I would like to thank Günter Hommel and all the other members of the graduate college for their valuable input but also for exerting the necessary pressure on me.

For several years, I was part of an excellent team from which I have learned a lot. Many thanks go to my co-authors and colleagues Enno Scholz, Gerald Brose, André Spiegel, and Markus Dahm for their support, for all the discussions at lunch-time, for valuable feedback and help, and for reading parts of this thesis in various stages. I would also like to thank Boris Groth for what I learned from him during my time at GMD FIRST. Last but not least, I would like to thank Lutz Kirchner, Christian Schuckmann, and Jan Schümmer for the learning-by-doing experience with building large and complex object-oriented frameworks and systems.

Special thanks go to my co-author Jan Vitek of University of Geneva (now Purdue University). It would have been much harder to finish my thesis without this collaboration, which was a main source of motivation for finishing my thesis.

I owe thanks to many researchers that I met at ECOOPs, OOPSLAs, and other opportunities. In particular, the many discussions with David Holmes, Doug Lea, James Noble, Jens Palsberg, John Potter, Patrick Steyart, Tom Mens, Urs Hölzle, Wolfgang De Meuter, and Yossi Gil helped me to find my place in the research community. Of course, it was great fun to meet them, too.

One colleague and friend, Enno Scholz, stands out from the others. He was available at any time, often enough just before submission deadlines. When there was little time, he proposed incremental improvements; when there was no time, he cheered me up; and when there was enough time, he pointed out where I was wrong or not good enough.

Many thanks go to my parents Barbara and Jürgen Bokowski, my family, and my friends for their kind interest and for not asking too often when I would finish my thesis.

Above all, I am indebted to my wife Petra. She profited the least from my ups and had to suffer the most from my downs. Thank you for your love.