Limes: An Aspect-Oriented Constraint Checking Language

Benjamin Mesing[1], Constantinos Constantinides[2], and Wolfgang Lohmann[1]

[1] University of Rostock, Germany
{benjamin.mesing, wolfgang.lohmann}@informatik.uni-rostock.de
[2] Concordia University, Montréal, Québec, Canada
cc@cs.concordia.ca

Abstract. In object-oriented software development, UML artefacts are used to illustrate and define the structure and the behaviour of the software system, while the semantics is usually described in a formal or informal specification language. The specification often consists of sets of constraints defined over the software components. When implementing the model, the specification is taken into consideration by the implementor. Since a significant proportion of the implementation consists of human-generated code, errors may be introduced in the implementation model. To detect these errors, the specified constraints need to be checked in the implementation. In this paper, we present *Limes*, an imperative constraint implementation language, which adopts aspect-oriented programming to describe constraint checking in a non-invasive way. *Limes* can be used at the design level, and can add constraint checking to the implementation.

1 Introduction

The Unified Modeling Language (UML) is a language for specifying and constructing the artefacts of software systems, and thereby allows to create models of systems. While UML models specify the structure and behaviour of systems, the semantics of the individual artefacts are usually captured in specifications expressed in a number of languages, most of them being declarative. Often specifications consist of sets of constraints specified over the artefacts. One specification language is the Object Constraint Language (OCL). Originally developed by IBM, the OCL is now part of the UML specification. In model driven engineering (MDE), UML models are often transformed directly into an implementation language. Since the transformation from the model to the implementation is not fully automated, some level of manual implementation is required. However, human-generated code might deviate from the specification, due to possible programming errors, or a misinterpretation or disregard of it. It is, therefore, desirable to be able to automatically check the implementation against the specification. However, an automatic translation of the specified constraints into executable code, and an instrumentation of this code into the target program is not a straightforward task. This is due to the large gap between the abstraction level of the specification language and that of the implementation

A. Rensink and J. Warmer (Eds.): ECMDA-FA 2006, LNCS 4066, pp. 299–315, 2006.
© Springer-Verlag Berlin Heidelberg 2006

language. While the former is normally a declarative language specificly designed for writing specifications, the latter is normally an imperative general-purpose language. Even though tools exist to instrument OCL constraints to the target program, these tools are language specific and they often make assumptions about implementation details.

In this paper we introduce *Limes*[1], a language which allows to imperatively specify how and when to check the constraints of a model in a platform independent way. It provides information to add constraint checking to the implementation of the model, and by adopting aspectual behaviour it is able to perform the checking in a non-invasive way, i.e. without the need to manipulate the implementation. *Limes* code does not rely on any implementation specific information, and can therefore be written while still being a the design level. The specification of *Limes* is confined within the domain of constraint checking. Alltogether, *Limes* allows to narrow the gap between specification and implementation and integrates well with MDE. *Limes* code provides all the information necessary to put the constraints into operation, and thus can be considered an implementation of the constraint checking.

The remainder of this paper is organised as follows: Section 2 provides a discussion on the fundamental concepts behind aspect-oriented programming. Section 3 forms the main part of the paper and provides an overview of the language, demonstrating its main features with examples. Section 4 outlines the architecture of our current prototypical *Limes* compiler. Section 5 discusses related work, followed by Sect. 6 discussing some general aspects of *Limes*. Section 7 concludes the paper and sketches some areas of future work.

2 Background: Aspect-Oriented Programming (AOP)

Despite the success of object-orientation in the effort to achieve separation of concerns, certain properties cannot be directly mapped in a one-to-one fashion from the problem domain to the solution space, and thus cannot be localised in single modular units, but their implementation cuts across other units. This crosscutting phenomenon manifests over the inheritance hierarchy. As a result, developers are faced with a number of problems including a low level of cohesion of modular units, strong coupling between modular units and difficult comprehensibility, resulting in programs that are more error prone. Crosscutting concerns include persistence, authentication, synchronisation and logging.

Aspect-Oriented Programming (AOP) [1,2] addresses those concerns by introducing the notion of an aspect definition, which is a modular unit that explicitly captures and encapsulates a crosscutting concern, and therefore "can not be cleanly encapsulated in a generalized procedure (i.e. object, method, procedure, API)" [1]. Even though AOP is neither limited to object-oriented programming nor to the imperative programming paradigm, we will restrict this discussion to the adoption of AOP in that context. There is currently a growing number of

[1] ['liːməs], named after the ancient Roman wall built to keep out "barbarians". *Limes* was designed to keep out bugs.

approaches and technologies to support AOP. Our work is based on the linguistic model introduced by the general-purpose aspect-oriented language AspectJ [3] which is perhaps the most notable technology today, with a collection of supporting tools and an active developer community.

AspectJ has influenced the design dimensions of several other general-purpose aspect-oriented languages, and provided the community with a common vocabulary based on its own linguistic constructs. In the AspectJ model, an aspect definition provides behaviour to be inserted over functional components. This behaviour is defined in method-like blocks called *advice* blocks. However, unlike a method, an advice block is never explicitly called. Instead, it is activated by an associated construct called a *pointcut* expression. A pointcut expression is a predicate over well-defined points in the execution of the program which are referred to as *join points*. When the program execution reaches a join point captured by a pointcut expression, the associated advice block is executed. Even though the specification and level of granularity of the join point model differ from one language to another, common join points in current language specifications include calls to methods and execution of methods. Most aspect-oriented languages provide a level of granularity which specifies exactly when an advice block should be executed, such as executing before, after, or instead of the code defined at the associated join point. As a result, with an aspect-oriented language we are able to make quantified statements such as "whenever there is a call to a particular method (or a group of methods), before running the code that should run, execute the code in a given advice block."

A program in any general-purpose aspect-oriented language is essentially two-dimensional: One dimension describes the functional components written as definitions of classes, while another dimension describes aspect definitions written in an *aspect language* [1]. Like a class definition, an aspect definition can also contain state and behaviour (variables and methods). Additionally it can contain pointcut expressions and advice blocks. Furthermore, much like functional components must be composed to perform a computation, functional components and aspects must also be composed. This composition is referred to as "weaving" and it is performed by a special tool called a *weaver*. The weaver evaluates the pointcut expressions and determines the join points where the code of the advice block is inserted. The weaving process (Fig. 1) may take place either statically or dynamically.

As an example, consider a system where all calls to any method of some target classes should be logged. The implementation of logging would be scattered over a number of modules. In this example, the method calls constitute the join points where logging behaviour must be executed. An aspect definition encapsulating the logging behaviour would contain an advice block to perform the logging—perhaps creating an entry in a log file—and bind the advice block to a pointcut expression defined as a disjunction over relevant method calls. Once any of the methods captured by the pointcut expression is called, the associated advice block executes. The join point model and the related pointcut expression mechanism of AspectJ is highly expressive, including support

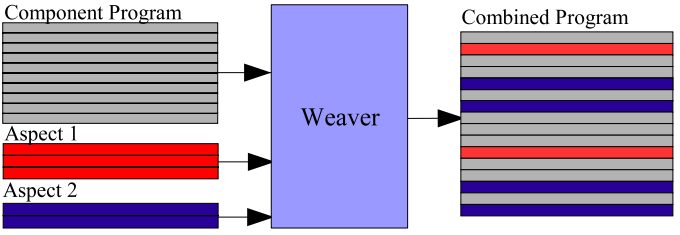

Fig. 1. The weaving process

for pattern expressions. In the example, pattern expressions would allow the definition of a logging aspect with such a complex behaviour as: "log all calls of set* methods in subclasses of a certain base class."

3 *Limes*: A Constraint Checking Language

Limes is a language to specify how and when to check constraints. It allows the description of constraint checking for a model, relying exclusively on structural information. Therefore *Limes* code can be written independently of the particular implementation of the model, and provides a platform independent model (PIM) for the constraint checking. In order to transform it to a platform specific model (PSM), we can follow either one of the following two approaches: First, constraints written in *Limes* can be transformed into the implementation language of the program to be checked, and second, these constraints can be transformed directly into executable code.

In the subsequent subsections we will first discuss the requirements for *Limes* and then provide an overview of the language, demonstrating its main features with examples. We will also discuss the problem of invariant checking and list conditions which an implementation of a system must fulfil in order to allow for the instrumentation of the model generated from its corresponding *Limes* definitions. The core specification of the *Limes* grammar is listed in the appendix of this paper.

3.1 Requirements for *Limes*

Our objective in building the requirements of *Limes* was to create a language which allows the specification of constraint checking for an implementation of a model without knowledge about implementation details. We aimed at the constraints defined by the types of assertions in the Design by Contract (DbC) principle [4]: preconditions, postconditions and invariants. Additionally, we placed the following requirements:

Describe constraints separately. Constraint checking should be described separately, without the need to modify the model or its implementation.

Free of side effect. The instrumentation of constraints should not affect the normal execution of the program, except for some unavoidable overhead in the execution speed of the program.

Platform independence. *Limes* code should not rely on any platform specific features.

Customisability. To allow the refinement of the constraint checking with implementation specific code, constraint checking specified in *Limes* should be modifiable in a non-invasive way.

Detailed context information. When detecting a violation of a constraint, as much information about the location of the error as possible should be made available.

Transformable. *Limes* should be easily transformable into different aspect-oriented languages. This allows to rapidly develop a transformation into various target languages, and thereby support a transformation to PSMs.

3.2 Features of *Limes*

Limes offers the following features:

Encapsulation of constraint checking. Constraint checking information is encapsulated in aspect definitions.

Minimisation of side effects. Though imperative by nature, *Limes* prevents the introduction of any side effects other then changing terminating to non-terminating behaviour, and changes in exectution speed and ressource usage.

Semantic checking at Design Level. The semantic analysis of *Limes* code is based exclusively on structural model information and hence it can be performed with only a PIM available. It is not necessary to defer the semantic checking until the source code is generated.

Non-invasive customisability. It is possible to customise how and when to perform the constraint checking in the implementation of the main program using aspect inheritance.

Transformable. The specification of *Limes* was held as simple as possible. Additionally, we added only language features which can be mapped to existing aspect-oriented languages such as AspectC++ [5], AspectJ and Eos [6] (an aspect-oriented extension for C#). Therefore, transformation of *Limes* to common aspect-oriented languages should be straightforward.

3.3 An Overview of *Limes*

Limes is an imperative aspect-oriented language to implement constraint checking. It uses the notion of aspect definitions, pointcut expressions and advice blocks to encapsulate the checking of constraints. The constraint checking code relies on type information from the model which can be available in various forms, e.g. as a UML model, as the source of the implementation or as the byte code of the checked program.

In this subsection, we will provide an overview of *Limes*, illustrating its main features with examples. The examples will implement constraints for the class shown in Fig. 2.

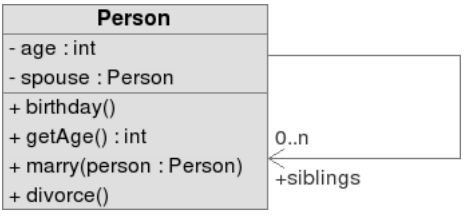

Fig. 2. Example class diagram

Aspects to Encapsulate Constraint Checking. *Limes* uses aspect definitions to encapsulate the checking of constraints. One aspect specifies the constraints for a single class. The name of the class to be checked is given in square brackets after the aspect name. Listing 1 shows the basic syntax of an aspect.

```
1 aspect PersonConstraints [Person] {
2   // constraint checking code here
3 }
```

Listing 1. Aspect stub

Pointcuts and Advices to Attach the Constraint Checking Code. A pointcut expression specifies the points in the execution where a constraint must be checked, and an advice block specifies the checking policy. The deployment of the pointcut and advice mechanism allows a non-invasive addition of the constraint checking to the corresponding class. Listing 2 shows the grammar rule specifying the syntax for the definition of an advice block. The syntax is specified in EBNF according to the rules provided in the appendix.

```
1 advice_def = advice_type ( typed_parameter_list ) [[
2                  pointcut_expr ]] advice_body.
3 advice_type = before | after | before after | around.
```

Listing 2. Grammar rule for an advice definition

The `typed_parameter_list` specifies the *pointcut signature*, a list of parameters available in the advice body. The parameters must be bound by (i.e. provided by) the pointcut expression. A pointcut expression is a predicate of atomic pointcut expressions. Among others, the following atomic pointcut expressions are available:

1. `execution(`*methodPattern*`)`: matches the execution of any method matching the specified method pattern.
2. `this(`*identifier*`)`: binds the current object (`this`) to the parameter of the pointcut signature named *identifier*.

3. `args(`*identifier1, identifier2,...*`)`: binds the parameters of the matched methods to the parameters named *identifier1, identifier2, ...* of the pointcut signature.

The body of the advice block contains the code performing the checking of constraints. An advice block can be executed `before`, `after`, `before` and `after`, or instead of (`around`) the code corresponding to the join point(s) specified in the pointcut expression. During the execution of an `around`-advice, the original behaviour defined by the join point can be invoked through a call to the special function `proceed()`. Listing 3 shows an advice definition implementing the checking of a precondition of the `divorce()` method. The pointcut expression given in line 2 together with the type of the advice (before) specify that the advice is executed before the execution of the `divorce()` method in class `Person`. Additionally, the current `Person` instance is bound to the parameter `self`. The meaning of the keyword `const` in line 1 is explained subsequently. To check the precondition, the advice calls the special function `precondition()` which tests the condition given as the first argument.

```
1  before(const Person self)
2    [[ this(self) && execution(void Person.divorce()) ]] :
3  { precondition(self.spouse != null , "divorce()"); }
```

Listing 3. Demonstrating precondition checking

Concepts to Prevent Side Effects. The specification of *Limes* provides various concepts to avoid the introduction of side effects through code written in *Limes* into the checked program.

To reduce the possibility of introducing an infinite loop, no conditional loop construct was added. The only type of loop in *Limes* is a `foreach` loop which allows the iteration over a collection with a fixed set of elements. However, it is still possible to create infinite loops by using recursive function calls.

Another provision is that every `around` advice must contain exactly one `proceed()` call which must not be conditional[2]. This ensures that the original behaviour defined by the join point is executed.

Finally we added a *const* concept similar to the one supported by the C++ programming language [7]. This concept allows to define an object as being constant, i.e. denoting that its fields cannot be modified and none of its mutator methods (methods that modify its object) can be called. This relies on the model to provide the information which methods are mutators. In our implementation which utilises the UML model, every non-query method is assumed to be a mutator. Similarly, a constant object can only be given as an argument to a method, if the method declares that parameter not to be modified, which again relies on the model providing this information. Furthermore, whenever a call to

[2] A less restrictive condition would be that in every execution path `proceed()` must be called exactly once, but this is harder to check at compile time.

a method of a constant object returns an object, the returned object is assumed to be constant to prevent exposing internal objects. The `const` keyword can be added as a modifier to a variable definition, marking its referenced object to be constant. Advice parameters bound by pointcut expressions must be defined as `const`, since they expose objects of the main program. Consequently a constant variable can only be assigned to another constant variable.

Special Functions to Perform the Condition Checking. *Limes* offers a number of special functions to support the implementation of constraint checking. First, there are `precondition()`, `postcondition()`, and `invariant()`, each of them checking the corresponding constraint. As the first argument they require a boolean expression specifying the condition to be checked. Additional arguments can be given to specify context information about the constraint. To allow transformations to a target implementation language to implement different strategies in order to handle constraint violations, the semantics of those functions is only partially defined Only that the constraint is violated if the first argument evaluates to false, that those functions do not return a value, and the meaning of the additional arguments is defined by *Limes*. For example, while one transformation might raise an exception, another might choose to merely log the constraint violation. A possible Java implementation corresponding to the `precondition()` call given in Listing 3 line 3 is shown in Listing 4.

```
1 if (!(self.spouse != null))
2   System.err.println("Precondition " + expression.toString()+
3     " for "+class+"."+method+" violated");
```

Listing 4. Transformed precondition call

Other special functions available in *Limes* include `copy()` and `equals()`. Function `copy()` is defined to return a non-constant, deep copy of a given object. Function `equals()` performs a value comparison of its two arguments. Additionally there is an `==` operator, which checks whether its operands refer to the same object. Listing 5 shows an advice definition using `copy()` to save the old state of an object.

```
1 around(const Person self)
2   [[ this(self) && execution(void Person.birthday()) ]] :
3 {
4   Person old = copy(self);
5   proceed();
6   postcondition(old.getAge()+1==self.getAge(), "birthday()");
7 }
```

Listing 5. Advice definition utilising the `copy()` function

`foreach` to Deal with Collections. Associations with a multiplicity greater than one are common in software models and therefore a constraint language

must support this concept. For *Limes* a small hierarchy of `Collection` classes including `Collection`, `Bag`, `Set` and `Sequence` is defined. Those classes provide a small interface allowing the convenient implementation of the most common constraints. Additionally, we introduced a `foreach` loop which iterates over a `Collection` and allows to check more complex conditions. An advice definition implementing the invariant that each sibling's siblings must contain the `self`-object is shown in Listing 6. The information on whether a variable refers to a collection, must be provided by the model providing the type information. The UML model utilised by the current *Limes* implementation provides this information by explicitly illustrating collections through associations with a multiplicity greater than one.

```
1 before after(const Person self) [[ publicFunction(self) ]] : {
2   foreach(const Person current : self.siblings) {
3     invariant(current.contains(self));
4   }
5 }
```

Listing 6. Advice definition utilising `foreach`

Function and Pointcut Overriding to Support Customisability. *Limes* was designed to support the implementation of complex constraints. However, unanticipated needs occur in practice, which would usually lead to unconventional, error prone and unreadable workarounds (i.e. hacks). To reduce the need for those hacks, *Limes* supports aspect inheritance which enables the non-invasive customisation of constraint checking specified in *Limes*. If *Limes* proves not to be powerful enough to specify *how* to perform the constraint checking, virtual methods can be used to allow the target implementation language to refine the constraint checking code. If *Limes* proves not to be powerful enough to specify *when* to perform the constraint checking, pointcut definitions can be overridden in the target implementation. An example to illustrate this is given in Listing 7.

```
 1 aspect PersonConstraints [Person] {
 2   abstract void checkComplexPrecondition(const Person self);
 3   before(const Person self) [[ this(self) &&
 4     execution(void Person.marry(Person)) ]] :
 5   {
 6     checkComplexPrecondition(self);
 7   }
 8   abstract pointcut underageCheck(const Person self);
 9   before(const Person self) [[ underageCheck(self) ]] : {
10     invariant(self.age>=18 || self.spouse==null, "notUnderage");
11   }
12 }
```

Listing 7. Customisable aspect

Here, the advice definition to check the precondition of the `marry()` method (lines 3–7) calls the abstract method `checkComplexPrecondition()`. This requires that the method is implemented in a derived aspect. In line 8 we define an abstract pointcut named `underageCheck`, which is used in the second advice definition (lines 9–11) to specify when to check the invariant `notUnderage`. Because the pointcut is abstract, it must be implemented in a derived aspect. The derived aspect can be defined in the implementation language used for the main program, and thereby use its full feature set for both the pointcut definition and the implementation of the abstract method. Listing 8 shows an AspectJ definition of an aspect that extends and customises the aspect `PersonConstraints` shown above. It is also possible to override method and pointcut definitions. The concept of overriding pointcut definitions is similar to the overriding of methods. Note that when overriding or implementing pointcut definitions, the pointcut signatures must match.

```
1 aspect PersonConstraintsRefine extends PersonConstraints {
2   void checkComplexPrecondition(Person self)
3   { /* an AspectJ precondition check */ }
4   pointcut underageCheck(Person self) : <an AspectJ pointcut expression>;
5 }
```

Listing 8. Customising an aspect in AspectJ

3.4 Invariant Checking with *Limes*

While specifying when to check pre- and postconditions is rather straightforward, specifying when to check invariants requires careful consideration. According to Bertrand Meyer, "[an] invariant must be satisfied after the creation of every instance of the class, (and) be preserved by every exported routine of the class (that is to say, every routine available to clients)"[4]. Most often this is interpreted as *an invariant must be satisfied after the constructor execution, and before and after the execution of every public method.* However, we believe that it is valuable to also be able to specify invariants which must hold true before and after the execution of protected methods. The example in Listing 9 shows the code for checking the constraint that a married person should not be underage.

```
1 aspect PersonConstraints [Person] {
2   pointcut protectedPublic(const Person self) [[
3     (execution(* Person.ctor(..)) || execution(protected+ * Person.*(..)))
4     && this(self) && !cflow(within(PersonConstraints)) ]];
5   before after(const Person self) [[ protectedPublic(self) ]] : {
6     invariant(self.getAge()>=18 || self.spouse==null, "notUnderage");
7   }
8 }
```

Listing 9. Demonstrating invariant checking

In lines 2–4 the pointcut `protectedPublic` is defined. It captures the execution of the constructor (`ctor`) and the execution of every method specified protected or less restrictive (i.e. protected and public). The `!cflow(within(PersonConstraints))` pointcut expression excludes every join point where the control flow is within the execution of any code in the `PersonConstraints`. It avoids invoking the advice when calling `getAge()` from the advice itself which would trigger an infinite loop. The `before after` advice in lines 5–7 executes before *and* after the join points captured by `protectedPublic`. This is equivalent to defining the same advice twice, once as a `before`, and once as an `after` advice. Note that even though this seems to execute the advice before a constructor call, *Limes* defines that `before after` does not execute advice blocks before constructor calls.

While it is easy to provide a pointcut expression to check invariants after the constructor execution and before and after the execution of every (public) method, this has two major drawbacks. First, it results in checking invariants unnecessarily frequently, and second it does not necessarily detect invariant violations at the points in the execution of the program where these violations take place. Consider, for example, an object **A** containing a reference to an object **B**. Now, if an invariant is specified for **A** which involves **B**, and **B** is changed outside of **A**, this constraint might be violated. The violation will go undetected until a (public) method of **A** is called. Using the pointcut language, it is possible to give a more sophisticated definition of when to check an invariant. Through abstract pointcuts it is also possible to delegate the definition of when to perform the checking to the implementation of the model. There the implementation language might provide a sophisticated join point model. Since a fine-grained specification of the condition under which invariants must be evaluated normally requires intimate knowledge of the implementation, we believe it would be justified to delegate this task to the implementor of the model in this case.

3.5 Target Language Requirements

When *Limes* should be transformed into the general-purpose implementation language used for the rest of the system, the target implementation must meet some requirements to allow for an easy transformation, the most notable of which are listed below:

1. The implementation must be written in an aspect-oriented language (or an aspect-oriented extension for the language must exist) which supports the notion of aspect definitions, pointcut expressions and advice blocks.
2. The implementation language must be able to express the pointcut expressions available in *Limes*.
3. The implementation must allow to perform a value comparison of objects.
4. The implementation must provide a consistent way to iterate over collections.
5. The implementation must provide a consistent way to create deep-copies of objects.

4 Implementation

We have implemented a parser and analyser for *Limes* in AspectJ. Together they create an abstract syntax tree (AST) decorated with type information. The AST can be used as the basis for further transformations, and is accessible through a Java API using the Visitor design pattern, but could also be serialised and read using another programming language. The parser is generated using the Java Compiler Compiler (JavaCC) [8] and creates the initial AST. The analyser decorates the AST with type information, accessing the type information of the model through a type information provider. Information providers based on different types of models can be implemented. We have only implemented one provider, based on the UML model. Other possible models include various internal models of CASE tools, but also the source code of a program or some kind of byte code. The support of different model types allows for an easy integration of *Limes* support into existing CASE tools, which often have their own internal model format. It also allows to add constraint checking to existing applications without any high-level models available.

We have also implemented a *Limes* to AspectJ converter, which transforms the decorated AST into AspectJ. The implementation was mostly straightforward. The only major difficulty was the transformation of the atomic `return` pointcut expression available in *Limes* which exposes the return value of a method, since there is no such pointcut expression in AspectJ.

5 Related Work

Besides DbC, Unit testing [9] is another approach that aims at detecting implementation errors. It is used to test software artefacts by calling operations of them with a fixed set of input data, and checking assertions about the state after the operation. However, unit testing can only test with input data provided by the test case designer, who might not forsee all the input data possible during the execution of the program. On the other hand, constraint checking can check the constraints during the whole test cycle of the software. Nevertheless, unit testing provides a valuable addition to constraint checking, as it allows the specification of a fixed set of input data which is consequently tested. In fact, the combination of unit testing and automated constraint checking provides a powerful method for error detection [10].

Even though for many major programming languages without native DbC support frameworks and tools exist to add this missing feature, limited support exists for an automatic instrumentation of constraints specified for a model. Java is a notable exception here, where OCL is the main target for research dealing with constraint instrumentation. For example, the Dresden OCL Toolkit (DOT) [11,12] provides support for parsing and semantic checking of OCL expressions. Through a generator, the DOT is capable of generating Java code. In his work, Wiebicke [13] extends the DOT with the capability to instrument constraints to Java programs. To achieve this, the original source code is modified

and the constraint checking code is added. Wiebicke also lists a number of other tools dealing with the instrumentation of constraints. Since the DOT can also be utilised by other code generators, we feel it could also provide the basis for transforming OCL expressions to *Limes* code.

Another approach towards monitoring OCL constraints is proposed by Richters and Gogolla [14]. It is based on the USE tool [15] which allows to validate OCL constraints for an instance of a UML model. In order to test and validate an implementation, the authors use aspect orientation to detect changes in the implementation objects, and map those back to the modelling level, keeping an instance of the UML model synchronised with the implementation objects. They then validate the OCL constraints for the model instance. This approach requires a duplication of the application data and is implemented for the Java language.

Briand et al. [16] propose the adoption of grammar rules to transform OCL constraints into aspects, which instrument the constraint checking to the target program. Even though the approach is based on AspectJ and on some assumption about the implementation of the UML model, it could be adapted to transform OCL constraints to *Limes* code. In [17] the authors generalise their ideas and explain how AspectJ can be used to instrument constraints in general. Since the join point model used in *Limes* is similar to the one of AspectJ, this work could provide valuable guidelines for implementing constraint checking in *Limes*.

The issue when to check invariants and how to achieve this using AspectJ is discussed in [18]. The authors analyse OCL invariants and classify them according to the navigation paths. For each type, they provide a pattern to create aspects to check the invariants, focussing on specifying when to perform the check. The work provides guidelines for defining more sophisticated pointcut expressions for the check of invariants, then simply checking before and after every (public) method call. However, some of those patterns rely on per-instance aspects which are not available in *Limes*.

In [19] the authors facilitate AspectJ to implement internal and external operation contracts. Since the contracts are defined on the design level, *Limes* seems to be well suited to implement the operation contracts, and thereby enforcing operation semantics independently of the implementation language.

6 Discussion

There are two elements in *Limes* which will probably have a significant impact on the performance of the program where the constraints are checked. The first is the special function `copy()`, used to create a deep copy of an object. This can become very expensive for deep object hierarchies. Note that the problem of circular references must be addressed by *Limes* compilers implementing a deep copy mechanism. The second is the iteration over a collection, as this can lead to a large number of iterations. Particularly in combination with a frequent invariant checking (e.g. after every public method), this might significantly slow down the program. Even though this is not a problem for the program deployed

to the end user since the constraint checking can be disabled, it will become a problem if the program becomes so slow that it cannot be tested anymore.

The avoidance of introduction of side effects is a major concern for a constraint checking language. When using a fully compliant *Limes* compiler and having a model that truthfully provides information about its components, the only side effect that can be introduced through *Limes* code is changing terminating into non-terminating behaviour. However, in reality this might not always be given. We therefore discuss possible limitations that might lead to an introduction of side effects. First, there is the information about a method not modifying the object. For UML models this is defined by the modeller through the isQuery-attribute, but its enforcement is not supported in most implementation languages. Therefore, the implementation of a supposedly non-modifying method might not hold up to its promise and modify some properties of the object, allowing to introduce side effects through *Limes* code calling such a method. In the same way, the const-information about method parameters might not be honoured in the implementation. Another source for a potential introduction of side effects is the case where the implementation of the special function `copy()` does not create a full deep copy, but copies some parts flat.

One might argue about the use of a high-level, imperative constraint checking language. After all, there exist formal specification languages like OCL or Z and general-purpose aspect-oriented languages capable of implementing constraints. Especially so, since techniques exist for the automatic conversion from OCL to AspectJ. However, we believe that it would be valuable to have a platform independent language specifically designed for the specification of constraint checking. First, there are persons more comfortable with using imperative languages which might find writing *Limes* code easier than writing a declarative specification. Second, the conversion from a declarative to an imperative language requires a complex transformation which must be written once for each specification language to each implementation language. Here, *Limes* could serve as an intermediate language, first converting the specification language into *Limes*, and then converting *Limes* into the implementation language. For each specification language, this requires the complex transformation from the declarative to an imperative language to be implemented only once, leaving only the easier transformations from *Limes* to the target implementation language to be done multiple times. Compared to implementing the constraints manually in the target implementation language, *Limes* provides the advantage of being platform independent and being specifically designed for this task.

7 Conclusion and Future Work

In this paper we provided an overview of *Limes*. By demonstrating its main features with examples, we discussed how *Limes* can be utilised to implement DbC constraints. Conceptually, *Limes* is located between an expression based specification language like OCL, and a general-purpose implementation language. It

can help bridging the semantic gap between the high level expression and the low level implementation language.

Even though we have tested *Limes* in a small scale, a test in a real system including measurements of the runtime performance needs to be done in order to see how well *Limes* can scale up. Furthermore, until now we have only implemented a translator of *Limes* to AspectJ. To prove that our approach is not limited to that particular language, we plan to implement translations to AspectC++ and Eos. The current specification of *Limes* provides the basic features necessary to implement constraint checking. However, there are a number of improvements to be done. For example, an atomic pointcut expression to match query methods could be used to avoid invariant checking for methods which cannot modify the state of the object. In case of well known error conditions, e.g. signalled by exceptions, it might be reasonable to allow a violation of certain constraints. Hence, allowing to restrict constraint checking to take place only if methods exit normally is another desirable feature. Also, we plan to provide access for the advice definitions to context information about the current join point. This can provide valuable information, like the called method, in case of a constraint violation, and thereby support the *detailed context information* requirement listed in Sect. 3.1. Currently this information must be explicitly handed in the form of an argument to the checking functions.

References

1. Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-oriented programming. In Akşit, M., Matsuoka, S., eds.: Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP). Volume 1241., Springer-Verlag (1997) 220–242
2. Elrad, T., Filman, R.E., Bader, A.: Aspect-oriented programming: Introduction. Communications of the ACM **44**(10) (2001) 29–32
3. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of AspectJ. Lecture Notes in Computer Science **2072** (2001) 327–355
4. Meyer, B.: Applying "Design by Contract". Computer **25**(10) (1992) 40–51
5. Spinczyk, O., Gal, A., Schröder-Preikschat, W.: AspectC++: An aspect-oriented extension to the C++ programming language. In: CRPITS '02: Proceedings of the Fortieth International Conference on Tools Pacific, Australian Computer Society, Inc. (2002) 53–60
6. Rajan, H., Sullivan, K.: Eos: instance-level aspects for integrated system design. SIGSOFT Softw. Eng. Notes **28**(5) (2003) 297–306
7. Stroustrup, B.: The C++ Programming Language, Third Edition. Addison-Wesley Longman Publishing Co., Inc. (1997)
8. JavaCC: JavaCC home page. `https://javacc.dev.java.net/` (2006)
9. Beck, K., Gamma, E.: Test infected: Programmers love writing tests. Java Report **3**(7) (1998)
10. Cheon, Y., Leavens, G.T.: A simple and practical approach to unit testing: The JML and JUnit way. In: ECOOP '02: Proceedings of the 16th European Conference on Object-Oriented Programming, Springer-Verlag (2002) 231–255

11. Finger, F.: Design and implementation of a modular OCL compiler. Diploma thesis, TU-Dresden (2000)
12. TU-Dresden: Dresden OCL toolkit. `http://dresden-ocl.sourceforge.net/` (2006)
13. Wiebicke, R.: Utility support for checking OCL business rules in Java programs. Diploma thesis, TU-Dresden (2000)
14. Richters, M., Gogolla, M.: Aspect-oriented monitoring of UML and OCL constraints. In: Proceedings of the 4th AOSD Modeling With UML Workshop. (2003)
15. Richters, M.: The USE tool: A UML-based specification environment. `http://www.db.informatik.uni-bremen.de/projects/USE/` (2006)
16. Briand, L.C., Dzidek, W.J., Labiche, Y.: Using aspect-oriented programming to instrument OCL contracts in Java. Technical Report SCE-04-03, Software Quality Laboratory, Carleton University (2004)
17. Briand, L.C., Dzidek, W.J., Labiche, Y.: Instrumenting contracts with aspect-oriented programming to increase observability and support debugging. In: Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05), IEEE Computer Society (2005) 687–690
18. van der Straeten, R., Casanova, M.: Stirred but not shaken: Applying constraints in object-oriented systems. In: Proceedings of the NetObjectDays2001. (2001)
19. Constantinides, C., Skotiniotis, T.: The provision of contracts to enforce system semantics throughout software development. In: Proceedings of the Eighth IASTED International Conference on Software Engineering and Applications. (2004)
20. International Organization for Standardization: ISO/IEC 14977:1996: Information technology — Syntactic metalanguage — Extended BNF. International Organization for Standardization (1996)

A Grammar

In this appendix we provide the grammar for *Limes*. The grammar is described in EBNF, as defined by the ISO 14977 standard [20], with the following two deviations from the standard: First a sequence of terminals and non-terminals is not separated by commas, and second we use a "+" to denote that the proceeding group must be repeated one or more times. Besides, we mark terminals by setting them in **bold** font or by underlining them.

```
unit = [package] {aspect}.
package = package identifier {. identifier}.
aspect = aspect identifier [ type ] { aspect_body }.
aspect_body = {variable_def | method_def | pointcut_def | advice_def}.

full_qualified_name = identifier {. identifier}.
type = full_qualified_name.
simple_method_call = identifier ( argument_list ).
argument_list = [expr {, expr}].
nested_identifier = {(identifier | simple_method_call) .} identifier.
identifier = letter alphanum*.
alphanum = letter | digit.
letter = a | b | .. | z | A | B | .. | Z | _.
digit = 0 | 1 | .. | 9.

method_def = abstract_method_def | concrete_method_def.
abstract_method_def = abstract  method_signature ;.
concrete_method_def = method_signature block.
method_signature = ([const] type | void) identifier ( typed_parameter_list ) [const].
```

```
variable_def = variable_decl [ = expr() ] ;.
variable_decl = [const] type identifier.
typed_parameter_list = [variable_decl {, variable_decl}].
untyped_parameter_list = [identifier {, identifier}].

advice_def = advice_type ( typed_parameter_list ) [[ pointcut_expr ]] advice_body.
advice_type = before | after | before after | around.
advice_body = block.

pointcut_def = abstract_pointcut_def | concrete_pointcut_def.
abstract_pointcut_def = abstract pointcut pointcut_signature ;.
concrete_pointcut_def = pointcut pointcut_signature [[ pointcut_expr ]] [;].
pointcut_signature = identifier ( typed_parameter_list );

expr = unary_expr {binop unary_expr}.
unary_expr = simple_expr | ( expr )  | unop unary_expr.
simple_expr = method_call | nested_identifier | real_literal | integer_literal |
              string_literal | bool_literal | null.
method_call = {(identifier | simple_method_call) .} simple_method_call.
binop =  logical_binop | arithmetical_binop.
unop = logical_unop | arithmetical_unop.
arithmetical_unop = ++ | -- | -.
arithmetical_binop = + | - | * | / | < | > | <= | >= | == | !=.
logical_binop = && | ||.
logical_unop = !.
bool_literal = true | false.
real_literal = digit+ . digit+.
integer_literal = digit+.
string_literal = " any_char_not_quote* ".

pointcut_expr = unary_pc_expr {logical_binop unary_pc_expr}.
unary_pc_expr = simple_pointcut_expr | ( pointcut_expr ) | logical_unop unary_pc_expr.
simple_pc_expr = call_pc | execution_pc | within_pc | cflow_pc | target_pc |
                 this_pc | args_pc | return_pc | pointcut_reference.
call_pc      = call      ( method_pattern ).
execution_pc = execution ( method_pattern ).
within_pc    = within    ( type_pattern ).
cflow_pc     = cflow     ( pointcut_expr ).
target_pc    = target    ( full_qualified_name ).
this_pc      = this      ( full_qualified_name ).
result_pc    = result    ( full_qualified_name ).
args_pc      = args      ( full_qualified_name {, full_qualified_name} ).
pointcut_reference = identifier ( untyped_parameter_list ).
method_pattern = [access_pattern] (type_pattern | void) [type_pattern .]
                 id_pattern (signature_pattern) [const].
id_pattern = wildcard_literal.
type_pattern = wildcard_literal {. wildcard_literal} [+].
signature_pattern = [type_pattern {, type_pattern} [, ..] | ..].
access_pattern = [!] access_modifier [+].
access_modifier = public | protected | package | private.
wildcard_identifier = * {alphanum+ [*]} | letter {alphanum} {* alphanum+} [*].

block = { command_sequence }.
command_sequence = (block | try_block | statement)+.
try_block = try block (catch ( variable_decl ) block)+.
statement = foreach_stmt | if_stmt | expr_stmt | assign_stmt | return_stmt |
            loop_control_stmt | skip_stmt.
foreach_statement = foreach ( variable_decl : expr ) block.
if_else_statement = if ( expr ) block [else block].
expr_stmt = expr ;.
assign_stmt = (variable_decl | nested_identifier) = expr ;.
return_stmt = return [expr] ;.
loop_control_stmt = (break | continue) ;.
skip_stmt = skip ;.
```