Implementing OCL Invariants with Constrained Data Types

Alexander Paar[1]

University of Pretoria, Pretoria 0002, South Africa

Abstract. The Object Constraint Language (OCL) facilitates textual specifications of constraints that apply to Unified Modeling Language (UML) models. In particular, OCL is used for the specification of invariants in UML class diagrams. This paper compares the implementation of OCL invariants in program code by means of aspect-oriented programming (AOP) on the one hand and by means of value space constrained data types on the other hand. By contradistinction of these two approaches it will be argued that the availability of constraint-based type derivation and value space-based subtyping in a statically typed programming language can provide for the enforcement of OCL invariants already at compile time as opposed to runtime.

1 Introduction

The Object Constraint Language (OCL) [9] has been included with the specification of the Unified Modeling Language (UML) [10,11] since UML version 1.1. OCL provides means for textually specifying constraints, which apply to model elements, that cannot otherwise be expressed by the diagrammatic notations of UML. On the other hand, OCL constraints always refer to a UML diagram. For example, class definitions of a UML class diagram complement the set of built-in OCL types. In fact, OCL is most frequently used to adorn UML class diagrams. OCL constraints can be attached to every UML model element. The OCL constraint set is subdivided into six broad categories, where each category of constraints is expressed using a similar syntax.

Invariants state that a condition must always be met by, for example, all instances of a class. *Pre-* and *postconditions* are restrictions that must be true before an operation is executed and after an operation has ended, respectively. Note that the OCL specification remains silent about what should happen if the precondition does not hold of the state before the execution of an operation (i.e. the caller is obliged to ensure that given preconditions hold). *Initial* and *derived values* specify the initial values of, for example, attributes and derivation rules for, for example, attributes, respectively. *Definitions* provide for the declaration of supplemental attributes and operations that are not part of the underlying UML model. *Body definitions* specify the bodies of query operations (i.e. operations that are assumed to terminate on every input). *Guards* are constraints that must be true before a state transition happens.

A very frequent constraint/model element combination is the use of OCL invariants to tag UML class attributes. Invariants are described using an expression that evaluates to true if the invariant is met. OCL constraints are always defined in a particular context that specifies the model element for which the OCL expression is defined. Invariants must be defined in a *classifier context*, where the augmented model element usually is a class or interface type. Invariants in a classifier context are denoted in the following form.

context [*c*:] *Type*
inv [*e*:] *expression*

OCL keywords are set in boldface. *Type* is the name of the contextual type of the given OCL *expression*, which has to be of type *Boolean*. OCL expressions are evaluated for instances of the contextual type. The optional parameters *c* and *e* can be used to define a variable of the given type and the given OCL expression, respectively. Alternatively to type variable *c*, the keyword *self* can be used to refer to the *contextual instance* (i.e. the instance for which the given expression is evaluated).

OCL is a typed language and allows for the use of five categories of types. *Model types* and *enumeration types* stem from user-defined type definitions in the underlying UML diagram. *Basic types* are OCL's built-in atomic types *Integer*, *Real*, *String*, and *Boolean*. The only generalization relation stipulated for basic types is *Integer* $<:$ *Real*. Model and basic types can be aggregated in OCL *collection types* (e.g., *Set(T)*). Note that there are no collections of collection types. The *special types* category comprises, among others, the supertype of all non-collection types *OclAny*.

All types are simply denoted by their names. The OCL specification defines a number of arithmetic and logical operations both on basic as well as collection types. For example, numerical types can be added and compared for equality (see [9] for the operations and well-formedness rules of OCL types). OCL expressions can be built-up from typed model elements and operations on them. OCL constraints apply OCL expressions in particular contexts for different purposes. Again, as indicated by the syntax description above, OCL language statements comprise 1) a context, which defines the scope to which a statement pertains, 2) a property that represents some characteristics of the context (e.g., a class attribute), 3) a well-formed OCL expression, and 4) possibly additional keywords such as *if*, *then*, *else* to denote conditional expressions.

In this work, only the frequently occurring case of unconditional constraints on class and instance attributes will be considered (i.e. invariant definitions in a classifier context).

The use of aspect-oriented programming (AOP) [5,6] for enforcing OCL constraints at runtime will be compared with the implementation of OCL invariants by means of programming language data types. In particular, it will be shown how exemplary OCL invariants are translated into AspectJ [1] code by means of the Dresden OCL Toolkit [7]. As an alternative, the novel Zhi# programming language will be introduced, which provides programming language inherent

support for constraint-based type derivation and value space-based subtyping. It will be shown how the same OCL constraints can be simply translated into the use of data types that are derived from the OCL constraints. Thus, OCL invariants can be enforced by the programming language compiler as opposed to assertions at runtime.

2 Exemplary OCL Invariants

Ponder the following UML class *Person* with attributes *name*, *age*, and *eMail*. The value space of the OCL *Integer* type comprises the set of integer numbers; the OCL *String* type represents character chains of arbitrary length (i.e. both value spaces are unbounded). In practice, programming techniques such as *lazy evaluation* and *thunks* are required to construct infinite data structures. For the given *Person* class it is, however, a good idea anyhow to further constrain the set of admissible attribute values. Hence, the following UML model may be implemented in Java as shown below.

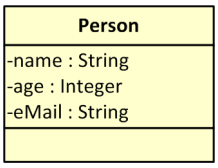

Fig. 1. Context class for attribute invariants

```
1 class Person {
2   String name;
3   int age;
4   String eMail;
5 }
```

The Java *String* API supports up to $2^{31} - 1$ (i.e. *Integer.MAX_VALUE*) characters. The Java *Integer* type implements signed 32-bit integer numbers. While the use of both Java data types does not impose any implementation issues anymore, it makes sense to even further restrict possible *name*, *age*, and *eMail* values of a *Person*. OCL constraints come to the rescue. The following invariant definitions restrict the age of a person to values *greater than or equal to zero* and *less than or equal to 110*; also, persons can only have names that are not longer than 40 characters.

context *Person*
  inv *age >= 0 && age <= 110*
  inv *name.size() <= 40*

At this point, it is important to understand that UML and OCL-based models only constitute a specification of the system that one wants to build. Especially, there are no prescriptions *how* to *implement* model elements and OCL expressions. In fact, the are a number of approaches how, for example, given OCL invariants can be enforced.

3 Implementing OCL Invariants

Stirewalt and Rugaber [16] employ C++ metaprogramming features. Specified OCL invariants are dynamically enforced by component wrappers, which are implemented as nested C++ template class instantiations. An alternative approach is provided by the Open C++ project [4]. The Open C++ meta-object protocol might be used to reprogram the compiler to guard modifications of class attributes. The GenVoca [2] tool, which is at a fundamental level similar to aspect-oriented programming, provides for the definition of higher level code constructs that can be used to mix program code with high-level design features such as invariants. Eventually, at the time of this writing, one of the most complete OCL compilers is the Dresden OCL Toolkit [7], where OCL constraints are translated into AspectJ-based constraint code.

3.1 Aspect-Oriented Programming

AspectJ [1] is an aspect-oriented extension of the Java programming language. It uses Java-like syntax (i.e. every valid Java program is a valid AspectJ program) to define special classes called *aspects*. In aspect-oriented programming (AOP) [5,6] aspects describe additional features or behaviors that are orthogonal to a number of core-level concerns of a program. Aspects support the encapsulation of concerns into separate, independent entities. Thus, AOP attempts to facilitate the composition of systems along a number of dimensions (in OOP, there is only one single such dimension). Additional behavior (called *advice*) that one wants to add to an existing program may, for example, be the checking of the values of instance variables in order to implement specified OCL invariants. Such checking may occur at certain points in the execution of a program. In AOP, a point in the control flow where the main program and an aspect meet is called *join point* (i.e. a join point exists there where Hoare logic places an assertion). A *pointcut* is an expression that determines whether a given join point matches. While an updated runtime environment could be conceived that understands additional AOP features, most AOP implementations such as AspectJ produce combination programs that are indistinguishable from non-AOP programs through a process called *weaving*. Early AspectJ implementations used source code-level weaving while nowadays class files can be combined, which frees developers from providing source code for all AOP-affected entities.

The modification of relevant program code that is described in pointcuts is accomplished automatically by AspectJ. It would, however, still be particularly tedious to manually devise join points and advices in order to implement

given OCL constraints. Again, tool support is available in form of, for example, the Dresden OCL Toolkit [7]. The Dresden OCL Toolkit is a modular toolset comprising libraries and stand alone tools that can be used to parse and type-check OCL constraints and to facilitate the instrumentation of Java programs for runtime verification of input OCL constraints. In particular, the Dresden OCL Toolkit can be used to automatically translate OCL constraints into AspectJ pointcuts and advices.

In order to do so, MOF models (i.e. UML class definitions) can be imported from XMI files[1]. The Dresden OCL2 Toolkit for Eclipse features special tree views for browsing the class definitions of a UML diagram. UML class diagrams can then be complemented by constraint definitions from separate OCL files. The combined set of UML classes and OCL constraints is taken to generate AspectJ code that implements the specified OCL constraints for the given UML-derived Java classes. The Dresden OCL2 Toolkit will report errors if constraints are not applicable in the given context (e.g., because of unavailable type names). Eventually, a combined Java program can be compiled from user-defined Java source code and automatically generated AspectJ aspects.

The Dresden OCL Toolkit translates the OCL invariant introduced above, which restricts *age* values of *Person* objects, into the following AspectJ code (generated comments are omitted and fully qualified type names are abbreviated for the sake of conciseness). The invariant pertaining to *name* values of *Person*s would be translated likewise.

```
1  @Generated public privileged aspect InvAspect {
2    protected pointcut
3     allPersonConstructors(Person p):
4      execution(Person.new(..)) && this(p);
5    protected pointcut ageSetter(Person p) :
6      set(* Person.age) && this(p);
7    protected pointcut allSetters(Person p) :
8      ageSetter(p);
9    after(Person p) : allPersonConstructors(p) ||
10                     allSetters(p) {
11     if (!((p.age >= new Integer(0)) &&
12           (p.age <= new Integer(110)))) {
13       throw new RuntimeException("Error:
14        Constraint was violated!");
15     }
16   }
17 }
```

The *Generated* annotation marks Java source code that has been generated. The *privileged* AspectJ aspect *InvAspect* is granted access to private class members. At the execution join points defined by the first pointcut a program is

[1] Meta Object Facility (MOF) is a metadata architecture standardized by the Object Management Group (OMG). XML Metadata Interchange (XMI) is a supporting standard of MOF for exchanging metadata information in an XML-based format.

executing constructor calls of type *Person*. In the second pointcut, the *set* designator is used to describe all join points based on assignments to the *age* attribute in class *Person* in the primary code. The third pointcut collects the *age* attribute setters. The defined *after* advice executes after the execution of the given join points has completed (either with or without throwing an exception). The (configurable) advice implementation facilitates a runtime check that set values of the *age* attribute are within the borders defined by the initial OCL invariant.

Note that the second pointcut in the above listing covers *all* assignments to the *age* attribute (and not only the execution of dedicated setter methods). The defined advice is, however, only effective for instrumented class files that were available at compile time. Assignments made in additional client code are not (i.e. cannot) be checked.

The Dresden OCL Toolkit and AspectJ-based implementation of OCL constraints accomplishes a completely automatic and type-checked translation of given OCL constraints into a combined Java program. However, the presented approach suffers from the following obvious shortcomings.

- Four kinds of different input data are necessary (i.e. XMI, OCL, AspectJ, and Java files).
- A considerable tool suite and explicit user interaction are required.
- Assertions are executed at runtime. Hence, only runtime type checks are facilitated.
- Pointcuts are woven into client code, which must be available at compile time. No precautions are taken to check modifications by additional client code that may not adhere to stipulated OCL invariants.
- Constraints can only be formulated using OCL's limited type system. There is no means to state, for example, that a *Person* object can only have values for its *eMail* attribute that match a given regular expression.

3.2 The Zhi# Programming Language

Constrained Data Types in Zhi# The Zhi# programming language [12,13,15] is a proper superset of C# version 1.0 and is extensible with respect to external type systems. The Zhi# compiler framework[2] provides two extension points for semantic analysis and program transformation that can be implemented by compiler plug-ins to make external type definitions available in Zhi# program code. Thus, external type definitions can be seamlessly used with features of the C# programming language such as, for example, method overriding and user-defined operators. In particular, external type definitions can be used for the declaration of class and instance variables, which are subject to OCL invariants. In addition, Zhi# compiler plug-ins can contribute type inference for atomic data types.

In Zhi#, external types can be included using the keyword *import*, which works analogously for external types like the C# *using* keyword for .NET programming language type definitions. It permits the use of external types in a

[2] `http://zhisharp.sourceforge.net`

Zhi# namespace such that, one does not have to qualify the use of a type in that namespace. In Zhi# program text that follows an arbitrary number of *import* directives, external type and property references must be fully qualified using an alias that is bound to the namespace of the external type.

Up to now, two type system plug-ins are available that integrate ontological concept descriptions based on the Web Ontology Language (OWL) [8] and XML Schema Definition (XSD) [3] data types into the Zhi# programming language.

In Zhi# programs, types of different type systems can cooperatively be used in one single statement. The following code snippet exemplifies how a .NET *System.Int32* variable *age* can be assigned the XSD data type value of an OWL datatype property *hasAge* of an ontological individual ALICE. Further examples of use are available online[3].

```
1 import OWL chil = http://chil.server.de;
2 class C {
3   public static void Main() {
4     #chil#Person alice = new #chil#Person("#chil#ALICE");
5     int age = alice.#chil#hasAge;
6  }
7 }
```

Zhi#'s XSD plug-in is based on a formal type system for constrained atomic data types, which was devised as an extension of the simply typed lambda calculus with subtyping ($\lambda_{<:}$). In the λ_C-calculus, atomic data types can be derived through the application of value space constraints. Data types are inductively defined through their value spaces.

The type construction mechanism used in the λ_C-calculus subsumes the definition of constrained values spaces with OCL constraints based on OCL's built-in atomic data types (i.e. *Integer*, *Real*, *String*, and *Boolean*). Ponder the following subtyping rules of the λ_C-calculus (the λ_C-calculus is explained in detail in [14].) A type S is considered to be a subtype of T (denoted by $S <: T$) if the value space of S, denoted $\upsilon(S)$, is a subset of the value space of T as shown in Table 1. In particular, S and T must be derived from the same primitive base type since otherwise their value spaces would be disjoint. Note how the λ_C-calculus mimics the type construction semantics of XML Schema Definition. The rule S-VSPACE subsumes the width and depth subtyping rules S-WIDTH and S-DEPTH.

A type S is considered to be a subtype of U if both types are derived from the same base type T through the application of constraints and fewer constraints are defined for type U than for S. A *constraint* is an optional property that can be applied to an atomic data type to constrain its value space. The intuition that it is safe to add constraints to an atomic type is captured by the *width subtyping* rule S-WIDTH for constrained atomic types as shown in Table 2. Constraints that are defined for atomic types may vary as long as the value spaces of each corresponding constraint are in the subset relation (i.e. the constraints are in the sub-constraint relation denoted by $c <:: d$). The *depth subtyping* rule

[3] `http://sourceforge.net/p/zhisharp/wiki/Examples of Use/`

S-Depth for constrained atomic types as shown in Table 3 expresses this notion. The subtyping rule S-App as shown in Table 4 captures the notion that a constraint application always makes a type more specific, which is required for the soundness property of the λ_C-type system; a soundness proof was obtained in [14].

Table 1. S-VSpace

$$\frac{\upsilon(S) \subseteq \upsilon(T)}{S <: T}$$

Table 2. S-Width

$$\frac{S = \bigcap_{i \in 1..n+k}^{T} c_i \quad U = \bigcap_{i \in 1..n}^{T} c_i}{S <: U}$$

Table 3. S-Depth

$$\frac{\text{for each } i \;\; c_i <:: d_i \quad S = \bigcap_{i \in 1..n}^{T} c_i \quad U = \bigcap_{i \in 1..n}^{T} d_i}{S <: U}$$

Atomic types are derived through the application of value space constraints. Constraint applications on atomic types can be reduced to set operations on their value spaces. Let $T_n \equiv \bigcap_{i \in 1..n}^{T_0} c_i$ be an atomic type, which shall be derived from a given base type T_0 based on a number of constraints $c_1, .., c_n$. The effective value space $\upsilon(T_n)$ of the derived type T_n can be computed by reducing the sequence of constraint applications $T_0.c_1....c_n$ to the successive application of constraints c_k on type T_{k-1} for $k = 1, .., n$ such that, $T_k = T_{k-1}.c_k$. Accordingly, $\upsilon(T_k) = \upsilon(c_k\{\{TV \leftarrow T_{k-1}\}\})$. In words: The value space of type T_k is exactly the value space of constraint c_k with its type variable TV substituted by type T_{k-1} (i.e. constraint c_k applied on type T_{k-1}). A value space expression $\upsilon(c_k\{\{TV \leftarrow T_{k-2}.c_{k-1}\}\})$ may be rewritten as $\upsilon(c_k\{\{TV \leftarrow T_{k-2}\}\}) \cap \upsilon(c_{k-1}\{\{TV \leftarrow T_{k-2}\}\})$ as long as type definitions within a derivation chain are not subject to change.

As can be seen, the definition of the value space of a data type in the λ_C-calculus is equivalent to the definition of OCL invariants that include OCL's

Table 4. S-App

$$\frac{S <: T}{S.c <: T}$$

built-in atomic data types plus the available arithmetic operators on these types. The use of OCL's logical AND and OR operators corresponds to subsequent constraint applications on one λ_C-data type and the set union of two types' value spaces, respectively. For example, an integer data type *Age* with a value space that comprises integer numbers *greater than or equal to zero* and *less than or equal to 110*, can be constructed in the λ_C-calculus as follows, where P_{Integer} is the OCL *Integer* data type.

$Age \equiv P_{\text{Integer}}.c_{[\geq\ 0]}.c_{[\leq\ 110]}.$

Implementation of OCL Invariants The value space-based subtyping rules of the λ_C-calculus type system will certainly not suffice to implement the complete AspectJ join point model as outlined in Section 3.1. They are, however, a viable alternative for implementing OCL invariants that follow a common scheme: The value space of a variable is constrained by a conjunction of relational expressions. The left hand side of the relational expressions is the variable itself. The right hand sides comprise of constant arithmetic expressions (i.e. the values of the arithmetic expressions can be determined at design time). Invariant definitions of the form $var \prec expr\ (\&\&\ var \prec expr)*$ are thus used to substitute the declared type T_{var} of variable *var* with a constrained type T'_{var} whose value space is constrained in accordance with the ANDed relational expressions of the invariant definition: $T_{var} \mapsto T'_{var}$, where $T'_{var} \equiv T_{var}.c_{[\prec_1\ expr_1]}.\ldots.c_{[\prec_n\ expr_n]}.$

Assume there was an OCL plug-in for the Zhi# compiler framework (similarly to the existing XSD plug-in) that derives type definitions from OCL invariants on C#'s intrinsic data types. The initial *Person* class and OCL invariants of its attributes can then be rewritten as shown below to make use of XSD-like constrained data types. A type $Age \equiv P_{\text{Integer}}.c_{[\geq\ 0]}.c_{[\leq\ 110]}$ can be made available to declare the *age* attribute of type *Person*. In the same vein, a type $Name \equiv P_{\text{String}}.c_{[?<\ 40]}$, where the constraint $c_{?<}$ denotes the XSD constraining facet *xsd:maxLength*, can be used to represent strings with at most 40 characters. Zhi#'s XSD plug-in also provides for the XSD constraining facet *xsd:pattern* to match literals with regular expressions. In the following Zhi# code snippet, the OCL type system evidence in the *import* directive in line 1 indicates the import of OCL derived type definitions, which works analogously to XSD data types.

```
1 import OCL inv = namespace name;
2 class Person {
3   #inv#Name name;     // Name ≡ String with less than 40 characters
4   #inv#Age age;       // Age ≡ Integer ≥ 0 && ≤ 110
5   #inv#EMail eMail;   // EMail ≡ String matching an eMail regular expression
6 }
```

4 Conclusion & Outlook

The proposed implementation of OCL invariants with constrained atomic value types shows the following advantages over AOP-based approaches.

- OCL invariants can be substituted by context-free type definitions; the weaving of primary code and constraint definitions can be replaced by Zhi#'s source-to-source compilation to C# (i.e. several AspectJ dimensions can be coalesced into the core program code).
- In contrast to the Dresden OCL Toolkit, programmers are no longer obliged to provide MOF models and separate OCL constraint definitions in addition to the actual program code.
- No additional tool suite is entailed by the Zhi# solution. Only the Zhi# instead of a C# compiler is required.
- In contrast to aspect-oriented approaches to implement OCL invariants, Zhi#'s one single dimension of program code can be fully statically type-checked.
- Static type checks in a Zhi#program occur at all places in a program that match AspectJ pointcut definitions (i.e. the weaving of primary code and OCL-derived assertions is replaced by compile-time program analysis). Accordingly, constraint violations at runtime can be completely avoided.
- The Zhi# solution removes the irritating shortcoming of AspectJ when pointcuts are directly defined for assignments to class attributes. AspectJ advices that pertain to assignments to class attributes are interwoven with the client code and not with the restricted object. Consequently, OCL invariants can only be enforced for client code that is readily available at compile time while nothing can be done about conventional Java client code that is contributed later on. In Zhi#, runtime type checks are facilitated by the invariant object itself. Hence, Zhi# components can be used by arbitrary .NET assemblies, which lack Zhi#'s static typing of constrained data types, with invariant declarations still being effective.

Implementing an OCL plug-in for the Zhi# compiler framework appears feasible due to the similarity of OCL invariant definitions to the XSD type system. OCL's four basic types correspond to XSD's built-in *xs#integer*, *xs#decimal*, *xs#boolean*, and *xs#string* types. The OCL subtyping relationship *Integer* $<:$ *Real* is matched in the existing XSD compiler plug-in by, for example, implemented covariant coercions of *xs#integer* data types to floating point numbers. Furthermore, an OCL plug-in for the Zhi# compiler framework may offer supplemental operators on basic OCL types. For example, plain OCL lacks means to restrict *eMail* values of the exemplary *Person* class to strings that match a given regular expression.

The proposed solution is readily available for OCL invariants that can be mapped to XSD data types. It is conceivable to develop an OCL plug-in for the Zhi# programming language to fully cover OCL's type system. This plug-in may even be extended with sensible data types and operators to increase the expressiveness of invariants that can be defined for UML class attributes.

References

1. AspectJ, December 2008. www.eclipse.org/aspectj/.
2. D. Batory, V. Singhal, J. Thomas, S. Dasari, B. Geraci, and M. Sirkin. The GenVoca model of software-system generators. *IEEE Software*, 11(5):89–94, September 1994.
3. P. V. Biron and A. Malhotra. XML Schema Part 2: Datatypes Second Edition. Technical report, World Wide Web Consortium (W3C), October 2004.
4. S. Chiba. Macro processing in object-oriented languages. In *28th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS)*, page 113, Washington, DC, USA, November 1998. IEEE Computer Society.
5. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Aksit and S. Matsuoka, editors, *11th European Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer Verlag, June 1997.
6. R. Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning Publications, July 2003.
7. S. Loecher and S. Ocke. A metamodel-based OCL-compiler for UML and MOF. *Electronic Notes in Theoretical Computer Science*, 102:43–61, November 2004.
8. D. L. McGuinness and F. van Harmelen. OWL Web Ontology Language Overview. Technical report, World Wide Web Consortium (W3C), February 2004.
9. Object Management Group. Object Constraint Language, version 2.0. Technical report, Object Management Group, May 2006.
10. Object Management Group. Unified Modeling Language, infrastructure. Technical report, Object Management Group, February 2009.
11. Object Management Group. Unified Modeling Language, superstructure. Technical report, Object Management Group, February 2009.
12. A. Paar. Zhi# – programming language inherent support for ontologies. In J.-M. Favre, D. Gasevic, R. Lämmel, and A. Winter, editors, *ateM '07: Proceedings of the 4th International Workshop on Software Language Engineering*, number 4/2007 in Mainzer Informatik-Berichte, pages 165–181, Mainz, Germany, October 2007. Johannes Gutenberg Universität Mainz. Nashville, TN, USA.
13. A. Paar. *Zhi# – Programming Language Inherent Support for Ontologies*. PhD thesis, Universität Karlsruhe (TH), Am Fasanengarten 5, 76137 Karlsruhe, Germany, July 2009.
14. A. Paar and S. Gruner. Static typing with value space-based subtyping. In *6th International Conference on Evalutation of Novel Approaches to Software Engineering (ENASE)*, June 2011. submitted for publication, paper draft online at http://www.alexpaar.de/publications/drafts/PaarG11.pdf.
15. A. Paar and D. Vrandečić. Zhi# – OWL aware compilation. In *8th Extended Semantic Web Conference (ESWC)*, May 2011. accepted for publication, paper draft online at http://www.alexpaar.de/publications/drafts/PaarV11.pdf.
16. K. Stirewalt and S. Rugaber. Automated invariant maintenance via OCL compilation. In L. C. Briand and C. Williams, editors, *8th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, volume 3713 of *Lecture Notes in Computer Science*, pages 616–632, Berlin / Heidelberg, October 2005. Springer Verlag.