A Type System for Object Models

Jonathan Edwards, Daniel Jackson and Emina Torlak
Computer Science & Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Cambridge, MA 02139
{jedwards, dnj, emina@mit.edu}

Abstract

A type system for object models is described that supports subtypes and allows overloading of relation names. No special features need be added to the modelling language; in particular, there are no casts, and the meaning of an object model can be understood without reference to types. Type errors are associated with expressions that are irrelevant, in the sense that they can be replaced by an empty relation without affecting the value of their enclosing formula. Relevance is computed with an abstract interpretation that is relatively insensitive to standard algebraic manipulations, so the user is not forced into particular syntactic forms.

Introduction

Object models describe state spaces in which the individual states are structured with sets and relations. They form the backbone of most object-oriented development approaches, and of efforts in 'model driven architecture'. Typically, an object model is presented as a diagram augmented with textual constraints. In UML, for example, the object model combines class diagrams with constraints in OCL [9]. Object models are used primarily for describing state invariants, but because they are essentially constraint languages, they can be used equally for describing operations in pre/post style.

Although a number of tools have been developed to support the construction and manipulation of object models, their type checkers have not been based on a coherent theory of types. Two important features of object models – subtyping and overloading – have not been well supported, and where they have, the resulting type system has usually been ad hoc and *syntactically fragile*: small refactorings render an expression ill-typed, forcing expressions to be written in particular ways, often with extensive annotation in the form of casts.

Existing approaches are typified by Z [12]and OCL [9]. Z has a classical set theoretic type system that supports neither subtyping nor overloading. OCL supports both, by adopting the type constructs of object-oriented programming languages. These constructs are not well suited to object models, however; they impose a considerable cost in complexity and annotation burden, and are syntactically fragile.

This paper presents a new type system for object models. It supports overloading and subtyping, and is syntactically robust, so that the full flexibility of the relational operators can be exploited. To reduce its cognitive burden on the user, the type system is designed so that it has no impact whatsoever on the semantics: models can be understood without any reference to types, even in the resolution of overloading. Type errors are never spurious, so the user is never in the position of rewriting a model just to please the type checker.

In this system, types are themselves expressions (of a simple form) that capture semantic approximations. There are two kinds of type. An expression's *bounding type* represents the set of values the expression may take. Its *relevance type* represents the subset of these values that can affect the meaning of the constraint in which the expression appears. If the relevance type is empty, it follows that the expression cannot take on a relevant value, and could have been replaced by a constant representing the empty relation. Such a case is regarded as a type error, since the expression is redundant, and it was likely not written with the expectation that it could have been trivially eliminated.

Resolution of overloading is a byproduct of this scheme. An overloaded relation name is desugared to the union of those relations that share the name (but which have distinct types). If exactly one of the relations in such a union is found to be relevant, the overloading is deemed to have been resolved successfully; otherwise an error is reported. This approach has the merit of giving a simple meaning to overloaded references that is independent of the type system, and resolves them as intuition expects. Moreover, since the resolution mechanism makes full use of the context in which an overloaded reference appears, it does not require that such references be used in a stylized fashion – always as navigations from an object of known type, for example.

The principle contributions of this paper are:

- A type system for object models that is efficient, fairly simple, and easy to implement, and which accommodates higher-arity relations;
- A notion of type error for object models, based on the idea of 'relevance', analogous to the idea of 'runtime error' in types for programming languages;
- A treatment of subtyping that exploits subtype information but requires no casts, and which requires no change to the underlying semantics of the language;
- A scheme for resolving overloading that exploits context more widely than existing approaches, and does not burden the semantics of the language with operational details or distinguish terms that should be equivalent; and
- A treatment of union types that allows more flexibility in modelling but which detects gratuitous, erroneous unions.

The type system is practical and effective. It has been implemented in the context of the Alloy Analyzer [1], and has been in use for 6 months. Its utility has been corroborated by optimizations it enables; a decomposition strategy called 'atomization' uses the type structure of expressions to rewrite them in a way that makes constraint solving more efficient [5].

The paper opens with an example to motivate the issues involved in typing object models (Section 1). It then proposes some criteria that

a type system for object models should satisfy (Section 2), and which are not satisfied by existing approaches (Section 7). The syntax and semantics of a core language are defined (Section 3), as a basis for the type system that follows (Section 4). To give a sense of how the system works in practice, typing issues that arise for some common idioms of object modelling are presented (Section 5). The paper closes with a discussion of some pragmatic issues in the implementation of the type system (Section 6) and concluding remarks.

1 Motivation

An object model describing a file system is shown in Figure 1. There are seven sets: Object, the set of all objects stored in the file system; Dir, the set of directory objects; Root, the set of root directories; File, the set of file objects; Link the set of all link objects; Block, the set of blocks that hold file data; and Name, the set of names objects may take.

The fat arrows represent subset relationships, and subsets of a common set are by default disjoint. An italicized set is *abstract*, meaning that it contains no elements beyond those belonging to its subsets. Thus every object is a file or a directory or a link, and root directories are directories.

The arrows with smaller, filled arrowheads represent relations. There are four relations: name, which maps objects to their names; to which maps links to the objects they are linked to; and two relations called contents, one that maps directories to their contents, and a second that maps files to their blocks. An object model would usually show multiplicities also – for example, that every object has one name – but these are not relevant to the concerns of this paper.

Some textual constraints augmenting the diagram are shown in Figure 2. These constraints are written in a core subset of the Alloy modelling language, whose syntax and semantics are given in Figures 4 and 5, and explained in Section 3. The full language provides features for organizing models and writing constraints more succinctly, but its details add nothing for the purposes of this paper.

The sample constraints of Figure 2 demonstrate forms that one might reasonably expect a type checker to accept:

- Application of a relation to a value belonging to its domain, as in o.name on line 2 (where o belongs to Object, and name is declared to map Object to Name);
- Comparison of values drawn from a common set, as in o.name = o.name′ on line 6 (where both expressions have values from Name);
- Application of a relation to a value drawn from a subset of its domain, as in Root.contents on line 21 (where contents is defined on Dir, of which Root is a subset);
- Comparison of values drawn from a set and its superset, as in d = Root on line 15 (where d is drawn from Dir, of which Root is a subset);
- Resolving of overloading according to context, as in d.contents on line 6 and f.contents on line 9 (where we expect the first to be resolved to the contents relation on Dir, and the second to the contents relation on File);
- Application of a relation to a value drawn from a superset of its domain, as in d.contents.to on line 24 (where d.contents is drawn from Object, but to is declared only on Link).

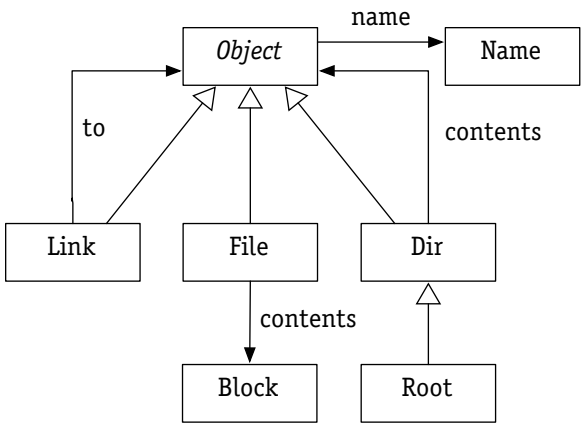

Figure 1: An object model for a file system

```
1   -- every object has one name
2   all o: Object | some n: Name | o.name = n
3
4   -- names are unique within a directory
5   all d: Dir | all o: d.contents |
6     all o': Object - o | not o.name = o'.name
7
8   -- every file has at least one block
9   all f: File | not f.contents = none
10
11  -- no directory contains itself, directly or indirectly
12  all d: Dir | not d in d.^contents
13
14  -- exactly one root directory
15  some d: Dir | d = Root
16
17  -- all objects reachable from the root
18  Object in Root.^contents
19
20  -- root directory contains only directories
21  Root.contents in Dir
22
23  -- no directory contains a link pointing to itself
24  all d: Dir | not d in d.contents.to
```

Figure 2: A sample of well-typed constraints

```
25  -- every block belongs to a file
26  all b: Block | some f: File | f in b.contents
27  all b: Block | some f: File | b in f.contents
28
29  -- directory does not share name with child
30  all d: Dir | not d in d.contents.name
31  all d: Dir | not d.name in d.contents.name
32
33  -- root directory has grandchildren
34  not Root.contents.contents = none
35  not (Root.contents & Dir).contents = none
```

Figure 3: A sample of ill-typed constraints. *Each constraint is followed by a corrected, well-typed version.*

Most of these are uncontroversial, perhaps with the exception of the last. This leniency is convenient in practice. More significantly, however, ruling it out would actually violate syntactic robustness, an essential property of the type system explained below (in Section 2).

Conversely, there are some constraints one would expect a type checker to reject. A sample is shown in Figure 3. For each case, the intended informal meaning of the constraint is given as a comment, followed by a version of the constraint that is wrong (and ill-typed) and a version that is correct (and well-typed). They demonstrate the following unacceptable forms:

- Application of a relation to a value that is disjoint from the relation's domain, as in `b.contents` on line 26 (where `b` belongs to `Block`, and `contents` maps either `Dir` or `File`, depending on how it is resolved);
- Comparison of two values that are known to be disjoint, as in `d in d.contents.name` on line 30 (where `d` belongs to `Dir` and `d.contents.name` belongs to `Name`);
- Ambiguous use of a relation name, as in `Root.contents.contents` on line 34 (where `Root.contents` might include members of both `File` and `Dir`, so that one cannot tell whether the second `contents` refers to the relation from `Dir` to `Object` or the relation from `File` to `Block`).

These examples of good and bad constraints are intended to suggest what a type checker might do; of course the system we describe below will accept the good ones and reject the bad ones. What is needed is a principle that will distinguish the good cases from the bad ones: in other words, a definition of 'type error' for object models.

2 Principles & Criteria

Here is the principle that underlies our type system:

- **Definition of Type Error**. An expression appearing in an object model is ill-typed if and only if it can be shown to be replaceable by the empty relation constant without affecting the meaning of the enclosing formula, using only the information present in the declarations of the sets and relations.

The motivating intuition is straightforward. If an expression is easily shown to be vacuous, why would the user have written it? In the course of designing, using, and refining our type system, however, we have come to recognize some criteria that characterize a type system for object models more broadly:

1. **Error detection**. The type checker should catch a wide class of errors that arise in practice. Since the type hierarchy is a central part of the object model, the information it conveys should be exploited by the type system.
2. **Low burden**. The type system should not complicate the syntax of the language, or require the pervasive use of annotations (such as casts). It should allow overloading, and resolve overloaded names automatically whenever possible.
3. **Syntactic robustness**. The type checker should be flexible in how constraints are written: it should not pass one expression but reject an equivalent expression obtained by a simple rewrite. In particular, the type system should respect the symmetry of object models, in which the direction of the relations is largely arbitrary (so that reversing all relations, and replacing each occurrence of a relation name `r` by its transpose `r'`, should leave every constraint invariant).
4. **Semantic independence**. The semantics of an object model should be independent of the results of a type analysis. For example, a reader should not need to understand a type-based mechanism for resolving overloading in order to determine the meaning of a constraint.

3 Language Syntax and Semantics

To explore fundamental questions about object modelling, we need to consider a core language – a tiny kernel smaller than a full-blown modelling language, but nevertheless containing all the elements relevant to our concerns. Such a language is defined in Figures 4 and 5.

The language is a standard first-order logic with relational operators, with one important difference. Relations may have arbitrary arity – that is, any number of columns greater than zero – and both the join (dot) and cross product (arrow) operators are generalized accordingly. Sets are represented as unary relations, and scalars are represented as singleton sets. This generalization is explained and justified elsewhere [6]; its benefits include a simpler semantics and a more succinct syntax. In the context of this paper, it allows us to set aside the orthogonal issue of partial function applications (since the expression `x.f` will simply yield an empty set when `x` takes a scalar value that is outside the domain of the relation `f`).

This semantics is very generous: it assigns a meaning to any constraint that is syntactically valid and respects variable scoping. A type system that identifies as errors terms that have no meaning will therefore not be useful. Note also that that compositions of partial relations are common; joining a relation to a scalar outside its domain evaluates cleanly to the empty set, and should not be considered an error.

4 Type System

Types assert semantic properties of expressions in a language – often that the value of an expression lies within a certain range of values. In our semantics, all expressions have relational values, so the meaning of a type will be a set of relations.

The set of all relations over a universe form a simple and well-understood algebraic structure. Recall that a relation is a set of tuples, and can thus be a subset of another relation. Relations can be partially ordered by subset, with a smallest and greatest relation (respectively the empty relation $\emptyset$ and the universal relation containing all possible tuples). Our type system will use this ordering to define the range of possible values for an expression: the *bounding type* of an expression will be a relation that acts as an upper bound on the values it may take. A second kind of type, the *relevance type*, will be a relation whose subrelations are those that may affect the value of the formula in which the expression appears.

Since the value of a type is a relation, it is natural to use our expression language itself as the type language. The sublanguage of expressions using just the union and product operators turns out to be adequate. A straightforward syntactic representation has some drawbacks, however, which leads us to represent types in a slightly different form – actually as relations, in which the atoms appearing in tuples are the basic types.

formula ::= elemFormula | compFormula | quantFormula
elemFormula ::= expr **in** expr | expr = expr
compFormula ::= **not** formula | formula logicop formula
logicop ::= **and** | **or** | =>
quantFormula ::= quantifier var **:** expr **|** formula
quantifier ::= **all** | **some**

expr ::= setName | relationName | var | **none** | expr binop expr
| unop expr
binop ::= - | + | & | . | ->
unop ::= ^ | ~

setName ::= identifier
relationName ::= identifier
var ::= identifier

Figure 4: Syntax of core language

M: Formula, Binding → Boolean
E: Expression, Binding → RelationValue

M[**not** f]b = ¬ M[f]b
M[f **and** g]b = M[f]b ∧ M[g]b
M[f **or** g]b = M[f]b ∨ M[g]b
M[f => g]b = M[f]b ⇒ M[g]b

M[**all** x: e | f]b = ∧{M[f] (b ⊕ x↦v) | v ⊆ E[e]b ∧ #v=1}
M[**some** x: e | f]b = ∨{M[f] (b ⊕ x↦v) | v ⊆ E[e]b ∧ #v=1}

M[p **in** q]b = E[p]b ⊆ E[q]b
M[p = q]b = E[p]b = E[q]b

E[**none**]b = ∅
E[p+q]b = E[p]b ∪ E[q]b
E[p&q]b = E[p]b ∩ E[q]b
E[p-q]b = E[p]b \ E[q]b
E[p.q]b =
{⟨p_1,…,p_{n-1}, q_2,…,q_m⟩ | ⟨p_1,…,p_n⟩∈E[p]b ∧⟨q_1,…,q_m⟩∈E[q]b∧ p_n=q_1}
E[p->q]b =
{⟨p_1,…,p_n, q_1,…,q_m⟩ | ⟨p_1,…,p_n⟩∈E[p]b ∧⟨q_1,…,q_m⟩∈E[q]b}
E[~p]b = {⟨p_m, … p_1⟩ | ⟨p_1, …, p_m⟩ ∈ E[p]b}
E[^p]b = {⟨x, y⟩ | ∃ p_1,…p_n | ⟨x, p_1⟩, ⟨p_1, p_2⟩, … ⟨p_n, y⟩ ∈ E[p]b}

variables: E[x]b = b(x)
sets: E[s]b = b(s)
relations: E[r]b = ∪ {b(r_i) | r_i has name r}

Figure 5: Semantics of core language

4.1 Base Types

We start by associating a *base type* with each of the sets declared in the object model, so that the set classification hierarchy becomes a type hierarchy. The base types in our example model are Object, Dir, Link, File, Block and Name.

Expression types are then written in disjunctive normal form as unions of products of base types. The expression to+name, for example, has the type

(Link -> Object) + (Object -> Name)

The presence of base types that have subtypes is troublesome: it means that the subtype ordering must be taken into account when composing types, and it allows the same type to be written in different ways. We therefore convert type expressions into a canonical form.

4.2 Canonical Form

The canonical form eliminates subtype comparisons from the type system, by eliminating all base types that have subtypes in favour of *atomic types*. The atomic types are a finer-grained set of types that partition the same universe of objects. They include all the base types which are not supertypes, and for each non-abstract supertype, a *remainder type* containing its 'direct instances' that belong to no subtype.

In the example model, the atomic types are Dir, Link, File, Block, Name. The base type Object is missing, because it is an abstract supertype, and thus is completely covered by its subtypes. Were it not declared to be abstract, there would be an additional atomic type representing the remainder set Object - (File+Dir+Link).

Since every base type is equal to a union of atomic types, we can rewrite every type expression in atomic form. The relation to, for example, which is declared to be from Link to Object is given the type

(Link -> Dir) + (Link -> Link) + (Link -> File)

Now because relational product is associative, and union is associative and commutative, we can represent a type as a set of tuples of atomic types, in this case:

{⟨Link, Dir⟩, ⟨Link, Link⟩, ⟨Link, File⟩}

This is the canonical form of a type: a *relation over the atomic types*. Unlike type expressions, this is a normal form: equal-valued types are equal. Because the atomic types are disjoint, subtype comparisons are eliminated from the type system in favor of exact matching.

This small shift of representation greatly simplifies the type system. Somewhat unconventionally, types become semantic objects rather than terms of a grammar. This turns out to be a benefit, as it permits the use of relational operators in the calculation of types.

A few subtleties are worth noting. The empty relation is a type; it will be the type of the empty relation constant none, and of ill-typed expressions. The type of a set will be a relation with one column, following the treatment of sets in our semantics. A type can have mixed arity – that is, it can contain tuples of different lengths – in order to represent ill-typed expressions, and expressions in which an overloaded relation name could be resolved to relations of different arity. Our semantics is carefully defined to admit such relations, although once successfully typed, a model will no longer require them.

4.3 Bounding Types

The computation of bounding types is formalized as an inference system in Figure 6. A typing judgment of the form

E ⊢ e: T

says that expression e has type T in the environment E. Environments bind quantified variables to their types; we assume that the types of the declared sets and relations are given by axioms (not shown here).

$$\frac{x : T \in E}{E \vdash x : T}$$

$$\frac{E \vdash p \,\&\, q : T}{E \vdash p = q : T}$$

$$\frac{E \vdash p \,\&\, q : T}{E \vdash p \;\mathbf{in}\; q : T}$$

$$\frac{E \vdash p : P,\ E \vdash q : Q}{E \vdash p \,\&\, q : P \,\&\, Q}$$

$$\frac{E \vdash p : P,\ E \vdash q : Q}{E \vdash p + q : P + Q}$$

$$\frac{E \vdash p : P,\ E \vdash p \,\&\, q : T}{E \vdash p - q : P}$$

$$\frac{E \vdash p : P,\ E \vdash q : Q}{E \vdash p \mathrel{->} q : P \mathrel{->} Q}$$

$$\frac{E \vdash p : P,\ E \vdash q : Q}{E \vdash p \,.\, q : P \,.\, Q}$$

$$\frac{E \vdash p : P}{E \vdash \hat{}p : \hat{}P}$$

$$\frac{E \vdash p : P}{E \vdash {\sim}p : {\sim}P}$$

$$\frac{E \vdash f,\ E \vdash g}{E \vdash f \;\mathbf{and}\; g}$$

$$\frac{E \vdash e : T,\ \ E, x : T \vdash f}{E \vdash \mathbf{all}\ x : e \mid f}$$

$$\frac{}{\vdash \mathbf{none} : \varnothing}$$

Figure 6: Inference Rules for Bounding Types

As an example, we will infer the bounding type of the intersection expression Object & Dir. The hypothesis of the intersection inference rule will include two judgments

$$\vdash \texttt{Object} : \{\langle \texttt{Dir}\rangle, \langle \texttt{Link}\rangle, \langle \texttt{File}\rangle\}$$
$$\vdash \texttt{Dir} : \{\langle \texttt{Dir}\rangle\}$$

Taking the intersection of the types as the rule requires, we obtain as the consequent

$$\vdash \texttt{Dir \& Object} : \{\langle \texttt{Dir}\rangle\}$$

Note how the type of an intersection expression is computed with an intersection of the types. In general, the effect of any operator can be approximated by its application in the abstract domain of types.

Some expressions, such as Dir.to, will have empty types. In that case, because the type bounds the value from above, the expression must have an empty value whatever the values of the declared sets and relations. We call this an *intersection error*. It can arise not only from intersections but also from joining mismatched relations. Of course the constant none will also be assigned the type $\varnothing$, but this will not be deemed to be an error.

$$\frac{\vdash f}{\vdash \bullet \downarrow f}$$

$$\frac{E \vdash C \downarrow p = q,\ E \vdash p : P,\ E \vdash q : Q}{E \vdash C[\bullet = q] \downarrow p :: P,\ E \vdash C[p = \bullet] \downarrow q :: Q}$$

$$\frac{E \vdash C \downarrow p \;\mathbf{in}\; q,\ E \vdash p : P,\ E \vdash q : Q}{E \vdash C[\bullet \;\mathbf{in}\; q] \downarrow p :: P,\ E \vdash C[p \;\mathbf{in}\; \bullet] \downarrow q :: P \,\&\, Q}$$

$$\frac{E \vdash C \downarrow p \,\&\, q :: T,\ E \vdash p : P,\ E \vdash q : Q}{E \vdash C[\bullet \,\&\, q] \downarrow p :: P \,\&\, T,\ E \vdash C[p \,\&\, \bullet] \downarrow q :: Q \,\&\, T}$$

$$\frac{E \vdash C \downarrow p + q :: T,\ E \vdash p : P,\ E \vdash q : Q}{E \vdash C[\bullet + q] \downarrow p :: P \,\&\, T,\ E \vdash C[p + \bullet] \downarrow q :: Q \,\&\, T}$$

$$\frac{E \vdash C \downarrow p - q :: T,\ E \vdash p : P,\ E \vdash q : Q}{E \vdash C[\bullet - q] \downarrow p :: T,\ E \vdash C[p - \bullet] \downarrow q :: Q \,\&\, T}$$

$$\frac{E \vdash C \downarrow p \mathrel{->} q :: T,\ E \vdash p : P,\ E \vdash q : Q}{E \vdash C[\bullet \mathrel{->} q] \downarrow p :: P',\ E \vdash C[p \mathrel{->} \bullet] \downarrow q :: Q'}$$

where

$$P' \equiv \{\langle P_1, \ldots P_n\rangle \in P \mid \exists \langle Q_1, \ldots Q_m\rangle \in Q \mid \langle P_1, \ldots P_n, Q_1, \ldots Q_m\rangle \in T\}$$
$$Q' \equiv \{\langle Q_1, \ldots Q_m\rangle \in Q \mid \exists \langle P_1, \ldots P_n\rangle \in P \mid \langle P_1, \ldots P_n, Q_1, \ldots Q_m\rangle \in T\}$$

$$\frac{E \vdash C \downarrow p \,.\, q :: T,\ E \vdash p : P,\ E \vdash q : Q}{E \vdash C[\bullet \,.\, q] \downarrow p :: P',\ E \vdash C[p \,.\, \bullet] \downarrow q :: Q'}$$

where

$$P' \equiv \{\langle P_1, \ldots P_n\rangle \in P \mid \exists \langle Q_1, \ldots Q_m\rangle \in Q \mid P_n = Q_1 \wedge \langle P_1, \ldots P_{n-1}, Q_2, \ldots Q_m\rangle \in T\}$$
$$Q' \equiv \{\langle Q_1, \ldots Q_m\rangle \in Q \mid \exists \langle P_1, \ldots P_n\rangle \in P \mid P_n = Q_1 \wedge \langle P_1, \ldots P_{n-1}, Q_2, \ldots Q_m\rangle \in T\}$$

$$\frac{E \vdash C \downarrow \hat{}p :: T,\ E \vdash p : P,}{E \vdash C[\hat{}\bullet] \downarrow p :: \ \{\langle P_1, P_2\rangle \in P \mid \ \exists \langle T_1, T_2\rangle \in T \mid \ *P(T_1, P_1) \wedge *P(P_2, T_2)\}}$$

where $*P(x,y) \equiv x = y \vee \langle x, y\rangle \in \hat{}P$

$$\frac{E \vdash C \downarrow {\sim}p :: T,}{E \vdash C[{\sim}\bullet] \downarrow p :: \{\langle P_1, \ldots P_n\rangle \mid \langle P_n, \ldots P_1\rangle \in T\}}$$

$$\frac{E \vdash C \downarrow f \;\mathbf{and}\; g,}{E \vdash C[\bullet \;\mathbf{and}\; g] \downarrow f\ ,\ E \vdash C[f \;\mathbf{and}\; \bullet] \downarrow g}$$

$$\frac{E \vdash C \downarrow \mathbf{all}\ v{:}e \mid f,\ \ E \vdash e : T}{E \vdash C[\mathbf{all}\ v{:}\bullet \mid f] \downarrow e :: T,\ \ E, v{:}T \vdash C[\mathbf{all}\ v{:}e \mid \bullet] \downarrow f}$$

Figure 7: Inference Rules for Relevance Types

4.4 Relevance Types

Bounding types alone give an unsatisfactory type system that is not syntactically robust; refactoring an expression can render it ill-typed. To overcome this, we compute a *relevance type* for each expression also. Aside from making the system robust, relevance types also provide a simple mechanism for resolving overloading.

Consider, for example, the expression (Dir+Link).to. This is well-typed: it has the bounding type Object. But written in the equivalent form Dir.to + Link.to, an intersection error would have resulted from the first subexpression. The problem can be seen in the original expression too: since the domain of to does not include Dir, we could just as well have written Link.to. The value of Dir is irrelevant, probably indicating that the expression is erroneous.

An expression's relevance type bounds the value it contributes to its context. If the relevance type is empty, the expression must be irrelevant, and a *relevance error* is reported. Note that this definition actually subsumes intersection errors, although it is helpful in practice to report the two differently.

The relevance type of an expression is relative to its context in a top-level formula; the same expression appearing in different contexts can have different relevance types. To capture this formally, we introduce a "hole" • into the language syntax, and define a context as a term containing at most one hole. For a given context C, the term C[t] will denote the term that results from filling the hole in C with the term t. Note that t itself can be a context, so that a hole in a deeply nested term can be represented.

Whereas bounding types are calculated bottom-up, relevance types are calculated top-down, using the relevance type of an expression to determine the relevance types of its subexpressions. Relevance types restrict the bounding type of an expression with information gleaned from its context. Thus an expression's relevance type is always a subset of its bounding type.

A formal inference system for relevance types is presented in Figure 7. A relevance type judgment

E ⊢ C ↓ e :: T

says that in the variable binding environment E, and in syntactic context C, the expression e has relevance type T.

To illustrate the rules, consider determining the relevance of Dir in the context of the formula:

(Dir+Link).to = **none**

The inference rule for the equality formula assigns a relevance type to the left-hand expression equal to its bounding type:

⊢ (• = **none**) ↓ (Dir+Link).to :: {(Object)}

The relevance rule for the dot operator now determines which component types of the arguments could contribute to the relevance type of the result. This winnows out the part of the type of Dir+Link that doesn't join up with to, resulting in the judgment:

⊢ (• . to = **none**) ↓ Dir+Link :: {(Object)}

The rule for union then intersects this type with the bounding types of the addends, with the result that Link is found to be irrelevant:

⊢ ((Dir + •) . to = **none**) ↓ Link :: ∅

4.5 Resolving Overloading

An overloaded relation name is taken to be short for the union of all relations that it might refer to (appropriately disambiguated, for example by qualifying their names with their declarations). Using relevance typing, we then check that exactly one of the relations is relevant. If none is relevant, the occurrence of the overloaded relation name is itself irrelevant, and a relevance error is reported. If more than one is relevant, an *ambiguity error* is reported.

For example, the relation name contents is overloaded by two relations of type Dir -> Object and File -> Block. In the expression Dir.contents, however, only the first will be relevant. The advantage of this technique of resolving overloadings is that it is semantic, not syntactic, and thus impervious to algebraic rewrites of an expression. Rewriting Dir.contents as ~(~contents.~Dir) does not affect the overloading, but would defeat a syntactic overloading mechanism based on leftward context.

When an overloaded name is resolved successfully, all but one of the referenced relations will be irrelevant, and thus replaceable by the empty relation. So the union for which the relation name stands will be equivalent to a disambiguated reference, and the semantics after resolution is thus identical to the semantics before. By treating overloading in this way, we have ensured that the particular mechanism for resolving overloads has no effect on the semantics of the language, which can be therefore understood without reference to its type system.

4.6 Arity

Our system resolves overloading between relations of different arity, exploiting arity when necessary. No extension to the rules is needed, so long as the types are allowed to take as values relations of mixed-arity – that is containing tuples of varying length. The semantics of Figure 5 was carefully designed to admit this. In particular, note that the transitive closure drops all non-binary tuples from the argument relations, since it might otherwise result in infinite relations.

4.7 Union Types

Support for ad hoc unions falls out naturally from our type system. An expression such as Dir+Link is legal, even though the subexpressions have disjoint types, so likewise an expression such as Dir+Name will be legal. The problem with ad hoc unions is that it is easy to include 'junk' inadvertently, but relevance typing will catch these problems. The occurrence of Name will be found to be irrelevant in the expression (Dir+Name).contents, for example, but relevant in the (implausible) expression

(Dir+Name).(contents+~name)

denoting the objects that are either contained in a directory or named by a name.

The resulting flexibility has real benefit. Unions generalize the type system into a lattice, instead of the tree structure of subtypes. This can allow new cross-cutting concepts to be modelled without refactoring the subtype hierarchy.

4.8 Soundness

It is straightforward to prove that an expression's bounding type really does bound its value – that is, the type system is *sound* – but space does not permit it. Surprisingly, perhaps, the system is also *complete*: the bound is exact, in the sense that if, for each occurrence of a variable, only its declaration is known (and not the particular variable appearing), there is some evaluation of the expression that can reach the bound.

Relevance typing is also sound, and we believe it to be complete, although we have not yet proved this result formally.

5 Idioms

To illustrate the application of the type system, we examine some characteristic constraints that arise in the use of some object modelling idioms. These idioms attempt to capture forms that recur in object models, and which experienced modellers recognize as standard structures (much like design patterns in programming). Here, we consider only a few that raise typing concerns; a larger repertoire will appear in a forthcoming book [7]. Our purpose is to demonstrate that the type system allows these common forms to be expressed in a natural way, and resolves overloading as expected.

5.1 Analogy

The generic form of the *Analogy* idiom and two instantiations are shown in Figure 8. Two sets and a relation (shown vertically) on the left form an 'analogy' to two sets and relation on the right, via mapping relations (shown horizontally). The overloading of relation names – in particular, using the same name for the relation and its analogue – is not only convenient, but captures the spirit of the idiom. The characteristic constraints capture the extent of the analogy, ranging from full isomorphism to weak forms of homomorphism such as

```
all a: A | a.r.x in a.x.r
```

which can be written in a pure relational form to highlight the commutativity

```
r.x in x.r
```

The first instantiation models a media asset management application, in which assets (such as photos and movies) have properties (such as aperture settings and durations). Each asset falls into an asset class with an associated set of property classes; the analogy constraint is that each asset has exactly one property for each of the associated property classes:

```
all a: Asset | all pc: a.class.properties | one p: Property |
p.class = pc
```

The second models a university structure and illustrates a different use of overloading. An analogy constraint here is that a department's head reports to the dean of the school that the department belongs to:

```
all d: Dept | d.head.reportsTo = d.belongsTo.dean
```

In all these cases, the overloading is resolved as expected (including, somewhat surprisingly, the formula r.x = x.r).

5.2 Split Relation

The generic form of *Split Relation* is shown in Figure 9 in two forms (on the left), with two corresponding instantiations (on the right). In the first form, a relation r from A to B has the characteristic property that it can be split into two disjoint relations from A1 to B1 and from A2 to B2. The property can be written with quantifiers:

```
all a: A1 | a.r in B1
all a: A2 | a.r in B2
```

or more elegantly by applying the relation to whole sets:

```
A1.r in B1
A2.r in B2
```

An instantiation is shown in Figure 12. A library's catalog associates Id's with Holding's: ISBN's with Book's, and ReportNum's with TechReport's. The instantiated constraints are:

```
Book.id in ISBN
TechReport.id in ReportNum
```

Alternatively, the relation can be split into two distinct relations with the same name, as in Figure 11. The characteristic constraints are now embodied in the declarations, and expressions such as a1.r for are resolved as expected. An attempt to apply the id relation to a set s of objects that spans both subsets cannot now be written s.id, however, since the overloaded relation will not be resolvable. To overcome this, one can split the set into its subsets

```
(s & Book).id + (s & TechReport).id
```

or split the relation:

```
s.(Book<:id + TechReport<:id)
```

This expression uses the domain restriction operator <:, whose semantics is exactly as in Z. So long as a relation r is overloaded in such a way that no other relation shares its domain type (as the full Alloy language ensures), an occurrence of r that is intended to refer to the relation from a set S can always be disambiguated if the context is insufficient to do so by writing S<:r. It is a nice property of our resolution scheme that this will always work, and does not require a special casting operator.

This case might be regarded as disappointing, but of course there is no free lunch: one cannot report overloading ambiguities as errors and still expect such expressions as s.id to pass.

5.3 Composite

The generic form of *Composite* is shown in Figure 10: a Component is either a Leaf or a Composite, and a containment relation contents maps Composite to Component. Typically, there will also be relations (such as cp) corresponding to attributes of both Composite and Leaf objects, and relations (such as lp) corresponding to attributes of Leaf objects alone. The file system example of Section 1 is an instantiation of this idiom.

From a type perspective, the notable feature of this idiom is the ease with which sets of reachable objects can be obtained by closure, and accessed without casts. Typical constraints define the topology of the containment relation, for example that it be acyclic:

```
all c: Composite | not c in c.^contents
```

Common expressions collect all the components or leaves reachable from a given composite, and their attributes, such as

```
c.^contents & Leaf
(c.^contents & Leaf).cp
c.^contents.lp
```

In a language such as OCL, expressions such as these are more cumbersome (see Section 7). Z, on the other hand, would admit erroneous expressions such as

```
(c.^contents & Composite).lp
```

5.4 Singleton

The generic form of *Singleton* is shown in Figure 11: a set X has a subset S that is constrained to have one element:

```
some s: X | s = S
```

(Of course, in practice, the cardinality of the set S is indicated with a multiplicity marking and an explicit constraint is unnecessary.) The example of Section 1 includes an instance of *Singleton* in the relationship between Root and Dir.

From a type perspective, the significance of *Singleton* is that the cardinality of the subset should not affect its type. Note that the file system model will still type check if there is more than one root. Although standard mathematical practice distinguishes an element from the singleton set containing it, such a distinction is not needed to avoid ambiguity or paradox in object modelling languages (because they are first order). It is also not desirable. Type distinctions based on multiplicity – between singletons and general sets, and

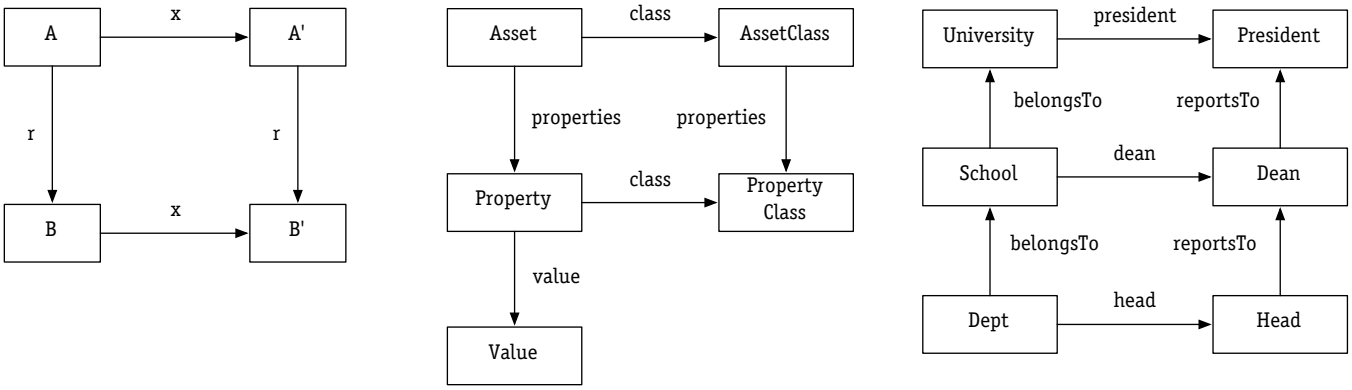

Figure 8: Generic form of *Analogy* with two instantiations

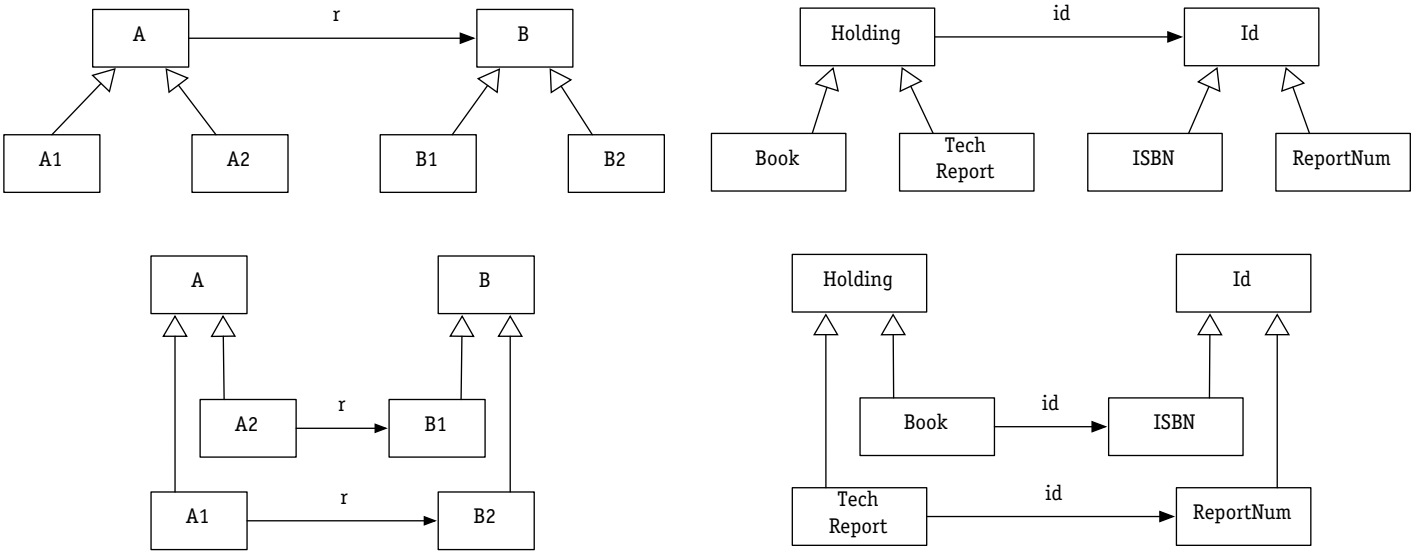

Figure 9: Two generic forms of *Split Relation* with instantiations

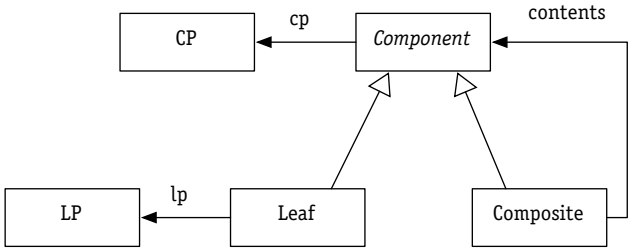

Figure 10: Generic form of *Composite*

between functions and relations – spoil the uniformity of the navigation syntax. (In OCL, for example, the target multiplicity of a relation determines whether an out-of-domain navigation results in undefined or an empty set.)

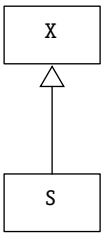

Figure 11: Generic form of *Singleton*

6 Realization

The type system has been implemented in the latest version of the Alloy Analyzer, which has been used within our group internally for about 6 months, and is due for release shortly. The implementation follows the formal system closely, and was actually refactored several times as we discovered simpler and cleaner ways to formulate it. It does differ from the formal system, however, in some important respects:

- All bounding and relevance computations are performed on base rather than atomic types. Since error messages should use base types and not atomic types in order to be more easily understood, the implementation must either convert to base types for error reporting, or work with base types and perform subtype comparisons. We chose the latter approach, because its performance is better (the base type representation being more compact), even though type checking is computationally trivial anyway.
- For the purpose of deeper analyses [4], it turns out to be inconvenient to have an empty relation constant that is polymorphic in arity. The constant none is thus treated as unary in Alloy – in other words, as an empty *set* – and an empty binary relation is written none -> none. To catch arity errors involving none, the type system must represent such an expression with a type from which its arity can be obtained. The empty type conveys no arity information, so none is assigned instead the special base type None, distinct from the empty type.

7 Related Work

7.1 Z

Z is not really an object modelling language, although its clean set theoretic basis has made it a common foundation for formalization of object models. There are two versions of Z known as 'Spivey Z' [12] and 'Standard Z' [14], but the differences need not concern us here.

A Z specification opens with the declaration of some *given types* that partition the universe of atoms. Compound types are formed from these given types, and from other compound types, using type constructors: A↔B, for example, gives the type of all relations from A to B. A *schema* is a named set (perhaps empty) of variable declarations, and has an associated *schema type*. One schema can extend another, and inherit its declarations.

Type checking requires exact matches. The basic relational and set theoretic operators are polymorphic, with a type inferred according to context. There is no subtyping. Distinct schemas have distinct types, even if one includes the other, so schema inclusion cannot be used to structure a type hierarchy.

Because the type system does not exploit subtyping, errors in which disjoint subsets are confused are not detected. For example, an expression such as to d that attempts to obtain the objects linked to by a directory d would be well-typed, since Link and Dir would be subsets of a given type Object, and would thus be indistinguishable to the type checker.

Transcribing an object model into Z is tedious, because Z's declaration syntax only permits types and not arbitrary expressions on the right hand side. So, for example, the relation blocks that maps files to their blocks would have to be declared as

blocks: Object ↔ Block

and then constrained explicitly

dom blocks ⊆ File

since File, not being a type, cannot appear in the declaration. Similarly, all the information in the hierarchy of sets must be written out explicitly as constraints.

The same name may be used for components in different schemas, but since extending a schema results in a new schema of an incomparable type, this cannot be exploited to support object model overloading. Components within a schema cannot be given the same name. Transcribing an object model into Z would thus require disambiguating overloaded relation names.

Free types allow disjoint unions to be represented, but their use is usually not recommended, because the projection operators must be applied pointwise, and cannot be applied to sets.

7.2 Alloy 2.0

An earlier version of Alloy [6] used the type system of Z, but inferred the types of relations from their declarations. This allowed the subtype hierarchy to be expressed in the declaration syntax, so that block, for example, would be declared from File to Block, but inferred to have the type Object -> Block. The subtypes were not exploited in type checking, however, so an error such as d.to (where d belongs to Dir and to maps Link to Object) would not caught, and relation names could not be shared amongst subsets of a common superset. Resolution of overloading was syntactically fragile, being handled with an ad hoc mechanism that relied on the leftward context of an application (as in OCL).

It was dissatisfaction with this type system that led us to develop the new one. The lack of true subtyping meant that type checking could not exploit information evident even to novice users. Subsets did not have their own namespaces, so a relation name could not be overloaded except across top-level sets. Worst of all, the special treatment of top-level sets and the rather ad hoc resolution mechanism made the type system hard to explain. The new system is more uniform, simpler and more powerful.

7.3 OCL

OCL [9, 13] is the constraint language of UML. It adopts the type constructs of object-oriented programming languages, in particular the use of downcasts. Downcasts cannot be checked statically, so the semantics assigns a special 'undefined' value to an expression in which a downcast fails.

All of the fundamental operators, including the basic set operators, are defined in a library of datatypes. These datatypes are parametrically polymorphic, and are instantiated implicitly according to context. The standard co- and contravariance rules govern the types of arguments. This results in a type system that is complicated and not always intuitive.

Consider, for example, writing in OCL the Alloy expression

```
d.contents.to
```

representing the set of objects pointed to by links in the directory d. In Alloy, the second relation application is well-typed because Object, the type of d.contents, intersects with Link, the left-hand type of to. OCL would require the type of d.contents to a be a subtype of the left-hand type of to. To fix this, we might try to extract out the set of links using intersection:

```
d.contents->intersect(Link).to
```

This will not work, however. The intersection operator is given the inferred type Object -> Object based on the type of d.contents; the Link argument is accepted by covariance, but the resulting type is still Object. For this reason, it is necessary to use a special type testing operator oclIsTypeOf, but since this can only be applied pointwise, we need to introduce a quantifier to apply it element by element, and then downcast it with the special casting operator oclAsType:

```
d.contents->select (oclIsTypeOf (Link))
    ->collect (oclAsType(Link).to)
```

Overloading is permitted, and is resolved by leftward context in a navigation expression. This is syntactically fragile; it means, for example, that associations cannot be navigated backwards, and distributivity cannot in general be exploited. A subtype may 'redefine' an association belonging to a supertype. In the *Split Relation* idiom (Figure 9, for example, there may be 3 relations with the same name, from `A` to `B`, `A1` to `B1`, and `A2` to `B2`. The semantics of this situation is unclear.

7.4 Other Approaches

Our notion of relevance is similar, in a very different context, and using different methods, to the notion of 'vacuity' in model checking (see, for example, [2]), in which a temporal logic property is examined to see if it has a subformula that does not influence the satisfaction of the property as a whole.

Object Z [3] is a variant of Z that supports inheritance (through schema inclusion). It does not support subtyping or overloading, however.

Schuerr notes some of the same problems in OCL's type system as we do [11]. He also sketches a variety of solutions, some of which have features in common with ours, but none of which is worked out in full.

Predicate subtypes [10] are used in the PVS theorem prover largely to ensure that partial functions are not applied outside their domain. Subtypes are characterized by arbitrary formulas. This gives great expressive power, but renders type checking undecidable.

Lamport and Paulson discuss whether specifications should be typed [8]. For them, a 'specification language' is a formalism for theorem proving. Most of their concerns arise from the application of partial functions outside their domain, and from higher-order constructs, and therefore do not apply to object modelling. To type set unions, they advocate the use of disjoint sum types. In relational notations, disjoint sums are not convenient, since the type constructors do not meld well with relational operators. This is why 'free types' are rarely used in Z. In our approach, the type of a union of values is simply a union of types.

In the conclusion of their abstract, having weighed the merits of typed and untyped languages, Lamport and Paulson suggest: 'It may be possible to have the best of both worlds by adding typing annotations to an untyped specification language'. This, we believe, is what our type system achieves, albeit in a first-order setting that is simpler than the higher-order setting they had in mind.

8 Conclusions

We have proposed a type system for object models that seems to work well in practice. It finds errors one would expect it to find, and requires no additional annotations. Overloading is resolved with the entire available context, allowing the full power of the relational operators to be exploited.

A pleasing consequence of the uniformity of this system is that, despite some extra work required in implementation, it is conceptually simpler than our previous system. The top-level sets in the classification hierarchy do not have special status, and overloading requires no special rules. By a cleaner separation of types and values, an untyped semantics has been regained, so that the language offers the simplicity of a traditional logic, but the error-detection of a typed language.

References

[1] *The Alloy Modelling Language and Analyzer*. Papers and tool available at: http://alloy.mit.edu. Software Design Group, Computer Science and Artificial Intelligence Laboratory, MIT, Cambridge, MA.

[2] Roy Armoni, Limor Fix, Alon Flaisher, Orna Grumberg, Nir Piterman, Andreas Tiemeyer and Moshe Y. Vardi. Enhanced Vacuity Detection in Linear Temporal Logic. *Computer-Aided Verification* (CAV 2003), Boulder, Colorado, July 2003.

[3] R. Duke, G. Rose, and G. Smith. *Object-Z: A Specification Language Advocated for the Description of Standards*. Technical Report 94-45, Software Verification Research Centre, School of Information Technology, The University of Queensland, December 1994.

[4] Daniel Jackson. Automating First-Order Relational Logic. *Proc. ACM SIGSOFT Conf. Foundations of Software Engineering*, November 2000.

[5] Jonathan Edwards, Daniel Jackson, Emina Torlak and Vincent Yeung. Subtypes for Constraint Decomposition. *International Symposium on Software Testing and Analysis*, Boston, MA, July 2004 (to appear).

[6] Daniel Jackson, Ilya Shlyakhter and Manu Sridharan. A Micromodularity Mechanism. *Proc. ACM SIGSOFT Conf. Foundations of Software Engineering/European Software Engineering Conference* (FSE/ESEC '01), Vienna, September 2001.

[7] Daniel Jackson. *Analyzable Models for Software Design*. In preparation.

[8] Leslie Lamport and Lawrence C. Paulson. Should your specification language be typed? *ACM Transactions on Programming Languages and Systems*, 21 (3): 502-526, 1999.

[9] *Response to the UML 2.0 OCL RFP*. Submitters: Boldsoft, Iona, Rational Software Corporation, and Adaptive Ltd. OMG Document ad/2003-01-07. Available at: http://www.klasse.nl/ocl/.

[10] John Rushby. Subtypes for Specification. *IEEE Transactions on Software Engineering*, Volume 24, Number 9. September 1998, pp. 709-–720.

[11] Andy Schürr. New Type Checking Rules for OCL Expressions. *Proc. Workshop Modellierung 2001*, Bad Lippspringe, Germany, March 2001, pp. 91-100.

[12] J. Michael Spivey. *The Z Notation: A Reference Manual*. Second edition, Prentice Hall, 1992.

[13] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison Wesley, 1999.

[14] *Information technology – Z formal specification notation – Syntax, type system and semantics*. ISO Standard ISO/IEC 13568:2002.