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 subtyping, unions, and 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 mentioning types. A type error is associated with an expression that can be proved to be *irrelevant*, in the sense that it can be replaced by an empty set or relation without affecting the value of its enclosing constraint. Relevance is computed by a simple abstract interpretation.

Categories and Subject Descriptors: D.2.1 [Software Engineering]: Requirements/Specifications – *languages*; D.3.3 [Programming Languages]: Language constructs and features – *data types and structures.*

General terms: Design, languages, theory, verification.

Keywords: type systems, object models, specification languages, relational logic, vacuity detection, Z, OCL, Alloy.

This paper has been formatted to meet ACM requirements. A more attractive and legible version is available online at http://sdg.csail.mit.edu/publications.

INTRODUCTION

An object modelling notation is a simple kind of first-order specification language. 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.

OCL's approach to typing object models is to adopt the type constructs of object-oriented programming languages (such as Java), in particular the use of downcasts. This has the great advantage of familiarity to model writers, and reuse of well-studied notions. But there is reason to believe that object models might merit a type system of their own.

For a programming language, the purpose of a type system is to ensure that certain errors do not occur at runtime, in particular the application of an operation to values for which its behaviour is not defined. For a specification language, there is no runtime, so a different notion of error is required, and the use of runtime checks (such as casts) to compensate for approximation in the type analysis is of dubious benefit. Moreover, operators are usually designed to be total, so that their application is always defined.

In developing our type system, we have been motivated by the following criteria, which apply to specification languages in general, but have particular implications for object models:

· *Error detection*. The type checker should catch a wide class of errors that specifiers make in practice. This class should be well defined, so that users can understand the output of the type checker and its limitations. For object models, the hierarchical classification of objects is fundamental, and should be exploited by the type system (for example, with subtyping).

· *Low burden*. The type system should not complicate the syntax of the language, or require the pervasive use of annotations (such as casts). For object models, the same name is often used for different relations, so a treatment of overloading is needed.

· *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 algebraic rewrite. One should be able to rewrite the expression `s.p + t.p` as `(s + t).p`, for example. For object models, there is a basic symmetry to be respected. The direction of the relations is largely arbitrary, so reversing all relations, and replacing each occurrence of a relation name `r` by its transpose `~r` should have no effect on type checking.

· *Semantic independence*. Semantics 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.

Our type system satisfies an additional criterion:

· *Sound error reports*. Type checking produces no false alarms; an expression reported to be irrelevant is always irrelevant.

At the same time, our system makes no guarantees of completeness. Any expression flagged as an error is guaranteed to be vacuous, but it is possible to write an expression that is vacuous but not identified as such. This is one respect in which our type system differs markedly from traditional type systems.

In our system, types are themselves expressions (of a simple form) that capture semantic approximations. There are two kinds of type. An expression's *bounding type* approximates the value of the expression from above; it includes tuples that might belong to the relation denoted by the expression. Its *relevance type* approximates the portion of the value that is relevant in context; it includes tuples whose inclusion or exclusion can change the value of the enclosing formula. If the relevance type is empty, it follows that the expression is indistinguishable in context from 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 all relations sharing that name (but which are guaranteed by the syntax of declarations to have distinct types). If exactly one of the relations in such a union is found to be relevant, the overloading has 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 type system is effective in practice. It has been implemented in the context of the Alloy Analyzer [1], and has been in use for about 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 [4].

1 EXAMPLE

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

The arrows with fat, unfilled arrowheads represent subset relationships; 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 and that there is exactly one root directory – but these are not relevant to the concerns of this paper.

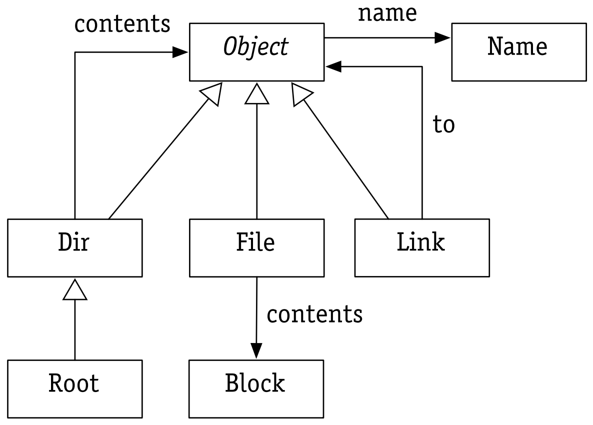

Figure 1: An object model for a file system

In the context of these declarations, we now consider a variety of candidate constraints, and explain how they are treated by our type system. They are written in a core subset of the Alloy modelling language, whose syntax and semantics are given in Figures 2 and 3, and explained in Section 2.

· Each object has a name:

all o: Object | **some** n: Name | o.name = n

This can be typed with a system of exact matches. The relation `name` is applied to an object drawn from its domain, and there is no overloading. The equality is applied to expressions of the same type.

· Every block has a name:

all b: Block | **some** n: Name | b.name = n

There is a simple type error in the expression `b.name`, attributed to the disjointness of `Block`, the type of `b`, and `Object`, the domain of `name`. The expression can be replaced by the empty set `none`, and is therefore irrelevant.

· The root directory contains only directories:

Root.contents **in** Dir

The overloaded relation name `contents` is resolved to the relation on directories (rather than the relation on files), using the fact that `Root` is a subtype of `Dir`. The subset operator `in` compares sets of different types: `Object` on the left, and `Dir` on the right. Our type system permits this, since the sets are not disjoint, and the comparison therefore makes sense. Familiarity with subtype systems for programs might tempt us to insert a downcast (of the left-hand expression to `Dir`), but this would be circular, since the cast's assertion would merely restate the constraint.

· Every object has some contents:

all o: Object | **not** o.contents = **none**

This is rejected, because the reference to `contents` is ambiguous. If we intended the `contents` relation on directories, for example, we could have written

all o: Object | **not** (o & Dir).contents = **none**

disambiguating the relation by its domain, or

```
all o: Object | not o.contents & Object = none
```

disambiguating by its range. There is nothing special about these forms; disambiguation exploits the entire context, using all the type information available. The full language includes a domain restriction operator <: (with the same semantics as Z's), so that one can always write `s <: r` to refer unambiguously to the relation named `r` with domain `s`. Here, for example, we might have written one of these

```
all o: Object | not o.(Dir <: contents) = none
all o: Object | not o.(File <: contents) = none
```

depending on whether we intended the relation on directories or the relation on files. Of course neither of these is likely to be what we meant, since they imply in the first case that every object is a directory and in the second that every object is a file. We probably meant

```
all o: Object |
    not o.(Dir <: contents + File <: contents) = none
```

One might reason that the type system should have made this the default interpretation, and indeed that is exactly what the semantics prescribes: the formula is perfectly meaningful, although the type system rejects it. An alternative would be to report such ambiguities as warnings rather than errors. But we chose to flag them as errors, because this allows a simpler semantics for subsequent analysis (in particular, mixed-arity expressions can be ruled out), and because we believe that in most cases being forced to disambiguate clarifies the expression for the reader anyway. In this case, for example, it would be better to write two separate constraints:

```
all f: File | not f.contents = none
all d: Dir | not d.contents = none
```

Note that disambiguation requires no casts: we use only standard operators of the logic, which require no special interpretation in the type system.

· No file contains itself:

```
all f: File | not f in f.contents
```

The type system rejects this. The occurrence of `contents` must resolve to the relation on files, since otherwise the expression `f.contents` would always evaluate to the empty set. But this gives `f.contents` the type `Block`, making the left- and right-hand sides of the comparison disjoint. The expression `f.contents` is therefore irrelevant anyway, because it could be replaced by the empty set `none` without affecting the outcome of the comparison.

· No object contains itself:

```
all o: Object | not o in o.contents
```

This is accepted. If `contents` were resolved to the relation from files to blocks, the expression `o.contents` would be irrelevant, as in our last example. It is therefore resolved to the relation from objects to objects.

· No directory contains a link, directly or indirectly, that points back to itself:

```
all d: Dir | not d in d.^contents.to
```

Here, the relation `to` is applied to objects rather than links – namely the set `d.^contents` representing (using the transitive closure operator `^`) the descendants of `d`. Semantically, non-links contribute nothing when `to` is applied. Our type system permits the application because the type of `d.^contents` and the domain type of `to` overlap. One might expect a cast to be required here, but it would merely add clutter, since it would presumably be permitted under exactly the same circumstances that our type system permits the raw application.

· The contents of the root directory point to no objects:

```
Root.contents.to = none
```

This constraint typechecks just like the last example. Suppose, however, we miswrote it as

```
(Root + Root.contents).to = none
```

perhaps thinking that the root itself could be a link (which it cannot). The type system will identify the first occurrence of `Root` as vacuous; it can be replaced by the empty set `none` without affecting the value of the constraint as a whole. To catch errors such as this, a more sophisticated mechanism is needed than for the semantically equivalent constraint

```
Root.to + Root.contents.to = none
```

for which it is sufficient to detect the disjointness of `Root` from the domain of `to`. It may help to view our system as detecting 'disjointness at a distance'; the type checking algorithm involves a second top-down phase for this purpose.

2 LANGUAGE SYNTAX AND SEMANTICS

To explain the type system, we need a core object modelling language – smaller than a full-blown modelling language, but nevertheless containing all the elements relevant to our concerns. Such a language is defined in Figures 2 and 3. The logical operators have been restricted to the bare minimum; the quantifier `some` (used in one of the examples of Section 1) can be replaced by `all` and negation, for example.

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 [7]; 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 function `f`).

The declared relations and sets are represented in the grammar of Figure 2 by the terminal `rel`; the quantified variables by the terminal `var`. Quantification is first order, binding the variable to scalar values.

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 those terms that have no mean-

formula ::= elemFormula | compFormula | quantFormula
elemFormula ::= expr **in** expr | expr = expr
compFormula ::= **not** formula | formula **and** formula
quantFormula ::= **all** var **:** expr **|** formula

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

Figure 2: Syntax of core language

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

$M[\textbf{not}\ f]b = \neg M[f]b$
$M[f\ \textbf{and}\ g]b = M[f]b \land M[g]b$

$M[\textbf{all}\ x\text{: } e \mid f]b = \land\{M[f]\ (b \oplus x \mapsto v) \mid v \subseteq E[e]b \land \#v=1\}$

$M[p\ \textbf{in}\ q]b = E[p]b \subseteq E[q]b$
$M[p = q]b = E[p]b = E[q]b$

$E[\textbf{none}]b = \varnothing$
$E[p+q]b = E[p]b \cup E[q]b$
$E[p\,\&\,q]b = E[p]b \cap E[q]b$
$E[p-q]b = E[p]b \setminus E[q]b$
$E[p\,.\,q]b = \{\langle p_1,\ldots,p_{n-1}, q_2,\ldots,q_m\rangle \mid$
$\quad \langle p_1,\ldots,p_n\rangle \in E[p]b \land \langle q_1,\ldots,q_m\rangle \in E[q]b \land p_n = q_1\}$
$E[p\text{->}q]b = \{\langle p_1,\ldots,p_n, q_1,\ldots,q_m\rangle \mid$
$\quad \langle p_1,\ldots,p_n\rangle \in E[p]b \land \langle q_1,\ldots,q_m\rangle \in E[q]b\}$
$E[\sim p]b = \{\langle p_m, \ldots\ p_1\rangle \mid \langle p_1, \ldots, p_m\rangle \in E[p]b\}$
$E[\hat{}\,p]b = \{\langle x, y\rangle \mid$
$\quad \exists p_1,\ldots p_n \mid \langle x, p_1\rangle, \langle p_1, p_2\rangle, \ldots \langle p_n, y\rangle \in E[p]b\}$

variables: $E[x]b = b(x)$
relations: $E[r]b = \cup\ \{b(r_i) \mid r_i \text{ has name } r\}$

Figure 3: Semantics of core language

ing will therefore not be useful. Partial relations are common in models, and are often applied outside their domain (resulting in an empty set or relation). An expression that *might* evaluate to empty cannot reasonably be considered flawed, but our type system will reject an expression that *always* evaluates to empty.

This core could equally well be used as a semantic basis for other object modelling languages (such as OCL, the constraint language of UML) even though their basic building blocks may appear different on the surface. Associations, association classes, attributes, compositions, and so on, are all easily modelled with relations.

3 TYPE SYSTEM

The two principles underlying our type system are:

Type Errors. An expression (other than the constant none) is ill-typed if and only if it can be shown to be irrelevant, in the sense that it is 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.

Resolution of Overloading. An occurrence of an overloaded relation name in an expression is semantically equivalent to the union of all relations with that name. The overloading is deemed acceptable if, when the occurrence is replaced by an explicit union expression, all but one of the relations is shown to be irrelevant.

Type checking is performed in two phases. Each phase computes a different kind of type. In the first, a *bounding type* is computed that is simply an approximation of the expression's value from above. In the second, a *relevance type* is computed; this approximates the set of tuples whose values might be relevant to the meaning of the formula as a whole. If an expression's relevance type is empty, then none of the tuples it might contain can contribute to the meaning of the formula, so the expression can be replaced by the empty relation and is irrelevant.

3.1 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, Link, File, Dir, Root, 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.

3.2 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 T, a *remainder type* $T containing its 'direct instances' that belong to no subtype.

In the example model, the atomic types are Link, File, $Dir, Root, Block, Name. The base type Object is missing, because it is an abstract supertype, and is thus completely covered by its subtypes. The atomic type $Dir is a remainder type, and represents the set Dir - Root.

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 -> Link) + (Link -> File)
    + (Link -> $Dir) + (Link -> Root)
```

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:

$\{\langle \text{Link, Link} \rangle, \langle \text{Link, File} \rangle, \langle \text{Link, \$Dir} \rangle, \langle \text{Link, Root} \rangle\}$

This is the canonical form of a type: a relation over atomic types. Each distinct type now has a unique representation, and, because the atomic types are disjoint, subtype comparisons are eliminated from the type system in favor of exact matching.

This small shift of representation is somewhat unconventional; types are usually represented as terms of a grammar rather than as semantic objects. But it greatly simplifies the type system, and permits the use of relational operators in the calculation of types (in both the formalism and the implementation).

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 some 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.

3.3 Informal Example

To convey the intuitions of the type system, let's walk through the type checking of one of the examples from Section 1, which will be formalized by the rules in subsequent sections.

Recall this constraint:

(Root + Root.contents).to = **none**

in which contents resolves to the relation on directories, and the first occurrence of Root is irrelevant. We start by elaborating the overloaded relation name, replacing it by a union of the two possible relations, subscripted with their domain types to distinguish them:

(Root + Root.(contents$_{\text{Dir}}$ + contents$_{\text{File}}$)).to = **none**

Now we infer bounding types for each expression. To save space, we'll shorten the names of the atomic types: D for \$Dir, F for File, L for Link, R for Root and B for Block. The relation and set names simply acquire the types of their declarations:

Root: $\{\langle \text{Root} \rangle\}$
contents$_{\text{Dir}}$: $\{\langle D, L\rangle, \langle D, F\rangle, \langle D, D\rangle, \langle D, R\rangle, \langle R, L\rangle, \langle R, F\rangle, \langle R, D\rangle, \langle R, R\rangle\}$
contents$_{\text{File}}$: $\{\langle F, B\rangle\}$
to: $\{\langle L, L\rangle, \langle L, F\rangle, \langle L, D\rangle, \langle L, R\rangle\}$
none: {}

The type of any expression (except for a set difference) is obtained by combining the types of the subexpressions exactly as the subexpressions themselves are combined: a union of types for union expressions, a join for join expressions, and so on. We thus obtain:

contents$_{\text{Dir}}$ + contents$_{\text{File}}$:
$\{\langle D, L\rangle, \langle D, F\rangle, \langle D, D\rangle, \langle D, R\rangle, \langle R, L\rangle, \langle R, F\rangle, \langle R, D\rangle, \langle R, R\rangle, \langle F, B\rangle\}$
Root.(contents$_{\text{Dir}}$ + contents$_{\text{File}}$) :
$\{\langle L\rangle, \langle F\rangle, \langle D\rangle, \langle R\rangle\}$
Root + Root.(contents$_{\text{Dir}}$ + contents$_{\text{File}}$) :
$\{\langle L\rangle, \langle F\rangle, \langle D\rangle, \langle R\rangle\}$
(Root + Root.(contents$_{\text{Dir}}$ + contents$_{\text{File}}$)).to :
$\{\langle L\rangle, \langle F\rangle, \langle D\rangle, \langle R\rangle\}$

Note how in the typing of the inner join, no tuples from contents$_{\text{File}}$ contributed. This will be made explicit in the relevance phase, and will be crucial to resolving the overloading. Similarly, in the typing of the outer join, the tuple contributed by Root did not contribute; this will cause the term to be identified as irrelevant.

The first phase is now complete; every expression has been given a bounding type. We now compute relevance types top-down. For an equality, the relevance type of each side is its bounding type: that is, any tuple that might appear in the relation is potentially relevant. To determine the relevance type of

Root + Root.(contents$_{\text{Dir}}$ + contents$_{\text{File}}$)

we examine its bounding type

$\{\langle L\rangle, \langle F\rangle, \langle D\rangle, \langle R\rangle\}$

(the canonical type of Object) and determine which tuples in this type contribute to the relevance type of its containing expression. Since to was typed as

$\{\langle L, L\rangle, \langle L, F\rangle, \langle L, D\rangle, \langle L, R\rangle\}$

we see that only $\langle L\rangle$ contributed. The relevance type of this expression is therefore just

$\{\langle L\rangle\}$

Now we push down into the union expression. A tuple will only be relevant in a subexpression if it also relevant for the union as a whole, so we intersect this relevance type with the bounding type of each constituent of the union. For Root, this yields the empty type immediately, and we have determined that this subexpression is irrelevant. For the two subexpressions of

Root.(contents$_{\text{Dir}}$ + contents$_{\text{File}}$)

we apply the rule for dot again, and obtain the relevance type $\{\langle R\rangle\}$ for Root, and $\{\langle R, L\rangle\}$ for contents$_{\text{Dir}}$ + contents$_{\text{File}}$. Applying the rule for union to this expression, we find that the relevance type for contents$_{\text{Dir}}$ is $\{\langle R, L\rangle\}$, but the relevance type for contents$_{\text{File}}$ is empty. The second possible resolution of contents is therefore irrelevant, and we have resolved it successfully to the first.

3.4 Bounding Types

The computation of bounding types is formalized as an inference system in Figure 4. A typing judgment of the form

$E \vdash e\colon T$

says that expression e has type T in the environment E, and the judgment

$$E \vdash f$$

says that formula f is well typed 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). Bounding types are inferred only for expressions; the rules for formulas are present only to propagate types of quantified variables from their declarations to their uses.

Typing always succeeds; the purpose of this phase is to approximate the value of each expression, not to find errors. Note how the type of each kind of expression is computed using the same relational operators as the expressions themselves.

The soundness claim for this system is simply that the types bound the expressions from above. Suppose a relation r is declared to have type decl(r), and an expression e containing no bound variables is determined to have bounding type btype(e) (represented as an expression). Then for any binding b that satisfies the declarations

$$\forall r: \mathrm{dom}(b) \mid b(r) \subseteq E[\mathrm{decl}(r)]b$$

the value of the expression is bounded by the value of its type:

$$E[e]b \subseteq E[\mathrm{btype}(e)]b$$

Soundness is established by a standard induction over the rules using simple set theoretic reasoning. For example, the rule

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

says that if expressions p and q are bounded by the types P and Q respectively, then their union is bounded by P+Q. To establish its soundness, one need only show that

$$p \subseteq P \wedge q \subseteq Q$$
$$\Rightarrow p \cup q \subseteq P \cup Q$$

3.5 Relevance Types

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 type error is reported.

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, we introduce a "hole" (written •) 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 (which may itself be a context).

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.

Inference rules for relevance types are shown in Figure 5. A relevance type judgment

$$E \vdash C \downarrow e :: T$$

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

$$E \vdash C \downarrow f$$

says that the formula f is well typed in the variable binding environment E, and in syntactic context C.

Relevance typing, like bounding typing, never fails in the sense that an expression or formula cannot be typed at all. As for bounding typing, the rules for compound formulas and quantified formulas are present merely to propagate quantified variable declarations. The last rule is a technicality for starting the analysis; it says that top-level formulas (which have no context) are relevant. But the rules for the elementary formulas are more significant, since they act as the starting point for the relevance computation. Consider for example the rule for subset comparison:

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

This rule says that if expressions p and q have been bounded by the types P and Q respectively, then in the context of the formula p **in** q, the relevant portion of p is given by P, and the relevant portion of Q is given by P&Q. Note the asymmetry; tuples in p that cannot belong to q might be relevant, because if present they will cause the comparison to evaluate to false. But tuples in q that cannot belong to p are not relevant, since in general

$$p \subseteq q \Leftrightarrow p \subseteq q \cap p$$

The rules for the set operators are straightforward: the union rule, for example, says that the tuples relevant to a subexpression are those belonging to the bounding type that are relevant to the union as a whole.

The rules for dot and arrow are more complicated, since they require the computation of a quotient. In the rule for p.q, for example, the relevance type for p is obtained essentially by dividing the relevance type of p.q by the bounding type of q, finding all the tuples in the bounding type of p that contribute to the relevance type of p.q. In the definition of the quotients (below the inference rules for arrow and dot), the relational operators are used on tuples: thus a->b is simple concatenation of tuples a and b, and a.b is their join (namely concatenation with the matching element removed). The rule for transitive closure identifies the pairs of atomic types corresponding to edges that lie on a path that might contribute to the type of the closure.

A formalization of soundness for relevance types is left to future work. Informally, however, the principle is simple. If each expression e is given a relevance type rtype(e) within the context of a formula f, then the value of f in any binding that satisfies the declarations is invariant under a replacement of e by the intersection of e and rtype(e).

3.6 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 rela-

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

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

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

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

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

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

$$\frac{E \vdash p : P}{E \vdash \text{\^{}}p : \text{\^{}}P}$$

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

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

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

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

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

Figure 4: Inference Rules for Bounding Types

tion name is itself irrelevant, and a relevance error is reported. If more than one is relevant, an ambiguity error is reported.

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 mentioning types.

3.7 Arity

Our system resolves overloading between relations of different arity. 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 3 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.

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

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

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

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

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

$$\frac{E \vdash C \downarrow p \text{->} q :: T,\ E \vdash p : P,\ E \vdash q : Q}{E \vdash C[\bullet \text{->} q] \downarrow p :: P',\ E \vdash C[p \text{->} \bullet] \downarrow q :: Q'}$$
$$\text{where } P' \equiv \{a \in P \mid \exists\, b \in Q \mid a\text{->}b \in T\}$$
$$\text{and } Q' \equiv \{b \in Q \mid \exists\, a \in P \mid a\text{->}b \in T\}$$

$$\frac{E \vdash C \downarrow p . q :: T,\ E \vdash p : P,\ E \vdash q : Q}{E \vdash C[\bullet . q] \downarrow p :: P',\ E \vdash C[p . \bullet] \downarrow q :: Q'}$$
$$\text{where } P' \equiv \{a \in P \mid \exists\, b \in Q \mid a.b \in T\}$$
$$\text{and } Q' \equiv \{b \in Q \mid \exists\, a \in P \mid a.b \in T\}$$

$$\frac{E \vdash C \downarrow \text{\^{}}p :: T,\ E \vdash p : P}{E \vdash C[\text{\^{}}\bullet] \downarrow p :: P'}$$
$$\text{where } P' \equiv \{\langle P_1,P_2\rangle \in P \mid \exists \langle T_1,T_2\rangle \in T \mid {*P}(T_1,P_1) \wedge {*P}(P_2,T_2)\}$$
$$\text{and } {*P}(x,y) \equiv x=y \vee \langle x,y\rangle \in \text{\^{}}P$$

$$\frac{E \vdash C \downarrow \sim p :: T}{E \vdash C[\sim\bullet] \downarrow p :: \sim T}$$

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

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

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

Figure 5: Inference Rules for Relevance Types

3.8 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. Fortunately, 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 potentially 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 allows new cross-cutting concepts to be modelled without refactoring the subtype hierarchy. Unions also permit a simple treatment of null values when modelling code; the declaration

```
x: C + Null
```

says that `x` belongs either to the class `C` or is null. Without unions, a distinct null value is needed for each class.

3.9 Subtype Polymorphism

The accommodation of subtypes allows fragments of models to be used in more specific contexts than their definition might suggest. The most common use of this feature is to define functions and predicates over relations whose columns have the universal type. In the full language, the constant `univ` is the dual of `none`, and refers to the universal set. Its type is the union of all top-level types (analogous to `java.lang.Object` in Java). A function that defines the domain of a binary relation can thus be defined generically thus:

```
fun domain (r: univ -> univ): set univ { r.univ }
```

The Alloy library consists of a collection of modules containing such functions, defining common properties on orderings, sequences, graphs and trees. There is no performance cost to using such a function because it is automatically specialized by atomization [4]. There is an analysis cost, however. In comparison to parametric polymorphism (which is also available in Alloy), some type checking precision is lost: in this case, `domain(r)` will always have the type `univ`, irrespective of the type of `r`.

4 REALIZATION

The type system has been implemented in the latest version (3.0) of the Alloy Analyzer [1]. The implementation follows the formal rules closely, and was actually refactored several times as we discovered simpler and cleaner ways to formulate them. 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.
- When an empty bounding type is inferred for an expression (that is not the constant `none`), an error is reported immediately. The error messages thus distinguish a simpler class of 'disjointness errors' due, for example, to the intersection of disjoint expressions, or the application of a relation to a set that is disjoint from its domain.
- The formal system permits expressions of mixed arity. In practice, such expressions are not useful, and they complicate subsequent analysis. The implementation therefore rejects unions that result in mixed arity types after resolution of overloading.
- For the purpose of subsequent (deeper) analysis [6], 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.

5 RELATED WORK

The idea of approximating a variable by a set of possible values has been used before, most notably in Heinze's set-based analysis [5]. Our notion of relevance types seems to be novel. It has a similar spirit 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.

Most specification languages use simple type systems that require exact matches. The language of the PVS theorem prover has the most powerful subtypes [10], which may be expressed as arbitrary formulas. These are used primarily to ensure that partial functions are not applied outside their domain. Their expressiveness 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 – neither of which is an issue in our setting. To type 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.

5.1 Z

Z is not usually regarded as an object modelling language, although it could well be used as one. 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`$\leftrightarrow$`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 apply a function on links to 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.

Z's syntax is not well suited to models with elaborate classification hierarchies, because it permits only types and not arbitrary expressions on the right-hand side of declarations. So, for example, the relation `blocks` that maps files to their blocks would 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 overloading. Two components within the same 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.

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

5.2 Alloy 2.0

An earlier version of Alloy [7] 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, as in Z, an error such as `d.to` (where `d` belongs to `Dir` and `to` maps `Link` to `Object`) would not be caught. 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.

5.3 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. For such failure to make sense, it is necessary to interpret the language as operational rather than declarative. OCL is thus more like a side-effect-free programming language than a logic.

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 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`. 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, but the semantics of this situation is unclear.

The problems with OCL's type system are discussed by Schürr [11]. He also sketches a variety of solutions, some of which have features in common with our type system.

6 DISCUSSION & CONCLUSIONS

The type system we have described has resolved, in the context of the Alloy modelling language, a long-time dissatisfaction with previous efforts. It decouples semantics and types, and treats overloading systematically, and has thus simplified the language as a whole. In practice, it finds errors more effectively than our previous system, and its additional flexibility – particularly in supporting ad hoc unions and subtype polymorphism – has proven even more useful than we had anticipated. The type system is easy to implement, and its computational cost is trivial. It also provides a useful framework for subsequent optimizations [4].

For these reasons, we believe that our type system provides a more attractive solution to the problem of typing object models than the adoption of the standard type system designed for programming languages, which seems to offer fewer benefits at the cost of greater complexity. Several tools to support OCL are currently under development – IBM, for example, has plans to include an OCL constraint engine in Rational Rose – and the choice of a type system for UML/OCL will become an important practical question.

Despite its benefits, the system we have described in this paper has some significant defects:

· The flattening of the hierarchy into atomic types violates modularity. If a truly modular analysis is required, in which separate modules are to be analyzed independently, and the subtype ordering may be split across modules, the analysis could be reformulated using subtype comparisons.

· Although reports of irrelevance are never false alarms, reports of ambiguity may be. For example, the formula

  all o: Object | **not** (o **in** Dir **and** o.contents = **none**)

  is rejected because the occurrence of contents is deemed to be ambiguous. With a full knowledge of the semantics, however, one can see that there is in fact no ambiguity. If contents refers to the relation on files, then either the first conjunct is false (when o is a file), or both are true (when o is a directory), so the second conjunct is irrelevant.

· A systematic treatment of irrelevance yields error reports that can be confusing at first sight. For example, the pair of formulas

  all o: Object | **not** o **in** o.contents
  all o: Object | **not** o.contents **in** o

  (the first of which is discussed in Section 1) are treated differently. In the first, the overloaded mention of contents is resolved successfully, but in the second, it is deemed ambiguous. This is technically correct; interpreting contents as the relation on files does lead to a vacuity in the first case (and can therefore be ruled out), but not in the second. Despite the semantic asymmetry of the operator, a user might be surprised by the asymmetry in type checking. The inference rules can be adjusted to eliminate this problem, by inferring a relevance type of P & Q for both sides, albeit at the expense of generating errors that are now technically spurious.

  There is another case in which a spurious report is perhaps desirable. In a comparison by equality of disjoint sets, such as

  Name = Dir

  there is actually no irrelevance, since the formula will be true when both sets are empty. Nevertheless, it seems reasonable to flag this as an error (and this is indeed what our implementation does), since it could have been written less obscurely as a pair of independent formulas

  Name = **none and** Dir = **none**

· There is one idiom we have come across that might benefit from special syntactic support for types. In the current system, one cannot write

  e **in** T => ... e ...

  and expect the occurrence of e on the right to be treated as having type T, despite the 'test' on the left, because the implication operator is logical, and the situation is therefore no different from the example above in which the type system fails to exploit semantic information that is clear to the specifier.

· Type systems for programs have typically play two important roles. On the one hand, they allow a class of errors to be rapidly detected. On the other hand, they allow a class of basic properties to be inferred. Our type system provides the first, but does not really provide the second. Even the simple formula

  all d: Dir | d.contents **in** Object

  cannot be proven by our type system, because the bounding types are not lower bounds, and therefore the typing of the right-hand side as Object might represent a set smaller than the set Object itself. It would be nice if one could extract from a successful run of the type checker some properties that might be of interest to the specifier, but it seems that this would require a different kind of type system.

Our type system might have application to programs. One possibility is that the notion of relevance might be used to perform a kind of property-specific slicing. Another possibility, which we are actively investigating, is that the type system might be a basis for a relational programming language that combines the features of an object-oriented language and a relational database.

ACKNOWLEDGMENTS

Manu Sridharan designed and implemented a 'subscoping' scheme for Alloy that included some of the subtyping structure described in this paper. Mandana Vaziri suggested the key idea that an overloaded relation be viewed semantically as the union of its resolutions. The paper benefited greatly from comments on earlier drafts by Tony Hoare, Derek Rayside and Robert Seater.

This material is based upon work supported by the National Science Foundation under Grant No. 0086154 (Design Conformant Software) and Grant No. 0325283 (Safety Mechanisms for Medical Software), and by the NASA High Dependability Computing Program under cooperative agreement NCC-2-1298. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NASA or the National Science Foundation.

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

[5] Nevin Heinze. Set-based analysis of ML programs. *Proceedings of the ACM Conference on Lisp and Functional Programming*, pp. 306–317.

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

[7] Daniel Jackson, Ilya Shlyakhter and Manu Sridharan. A Micromodularity Mechanism. *Proc. ACM SIGSOFT Conf. Foundations of Software Engineering/European Software Engineering Conference*, Vienna, September 2001.

[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.