Automating First-Order Relational Logic

Daniel Jackson
Laboratory for Computer Science
Massachusetts Institute of Technology
200 Technology Square
Cambridge, Mass 02139, USA
dnj@lcs.mit.edu

Abstract

An automatic analysis method for first-order logic with sets and relations is described. A first-order formula is translated to a quantifier-free boolean formula, which has a model when the original formula has a model within a given scope (that is, involving no more than some finite number of atoms). Because the satisfiable formulas that occur in practice tend to have small models, a small scope usually suffices and the analysis is efficient.

The paper presents a simple logic and gives a compositional translation scheme. It also reports briefly on experience using the Alloy Analyzer, a tool that implements the scheme.

Keywords

First-order logic; relational logic; Z specification; object models; automatic analysis; model finding; constraint solvers; SAT solvers.

1 Introduction

Relational logic adds to first-order logic the ability to combine predicates with special operators. For example, we can write the formula $\forall x,y.\ S(x) \wedge R(x,y) \Rightarrow T(y)$ as *S.R in T*, where *S.R* denotes the image of the set *S* under the relation *R*. The logic is more than a definitional extension of first-order logic, because it includes transitive closure.

In this paper, we present a fully automatic analysis for such a logic. Given a formula and a *scope*—a bound on the number of atoms in the universe—our analysis determines whether there exists a model of the formula (that is, an assignment of values to the sets and relations that makes the formula true) that uses no more atoms than the scope permits, and if so, returns it.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Foundations of Software Engineering San Diego, CA
© 2000 ACM 1-58113-205-0...$5.00

First-order logic is undecidable, so our analysis cannot be a decision procedure: if no model is found, the formula may still have a model in a larger scope. Nevertheless, the analysis is useful, since many formulas that have models have small ones.

The analysis problem, while made decidable by restriction to a finite universe, is still intractable asymptotically. In a scope of *k*, each relation increases the space of potential models by a factor of 2 to the k^2. Nevertheless, our analysis can handle a large space; in Section 4, we report on some case studies in which spaces of 2^{100} configurations were analyzed in seconds. And when a model exists, it is usually found rapidly, often within seconds, so that when the analysis takes a long time, one can reasonably bet that no model will be found.

The analysis was designed for object models, which lie at the heart of most object-oriented development methods, but until recently have had no support from tools. It has been implemented in the Alloy Analyzer [21], a tool that has been publicly available since September 1999. The logic described here is used as an intermediate language into which the source language, Alloy [16] is translated.

The analysis is used in two ways: to check consistency of a formula (by finding a model), and to check the validity of a theorem (by looking for a counterexample, namely a model of the theorem's negation). In the context of object modelling, consistency checking amounts to simulation—generating states and executions. Validity checking has a variety of forms: checking that one constraint follows from another, that one operation refines another, that an operation preserves an invariant, and so on.

Because of the logic's generality, however, it has a variety of other applications, such as: finding bugs in code; checking verification conditions in a specification tool; establishing consistency of requirements goals; analyzing architectural style descriptions; and generating snapshots from class diagrams.

To our knowledge, this paper presents the first practical algorithm for analyzing automatically the logic that underlies Z [34], OCL [39] and many other specification languages. Unlike our previous algorithm [15], which was limited to quantifier-free relational calculus, this algorithm handles a full logic with quantifiers, into which other languages can be easily translated.

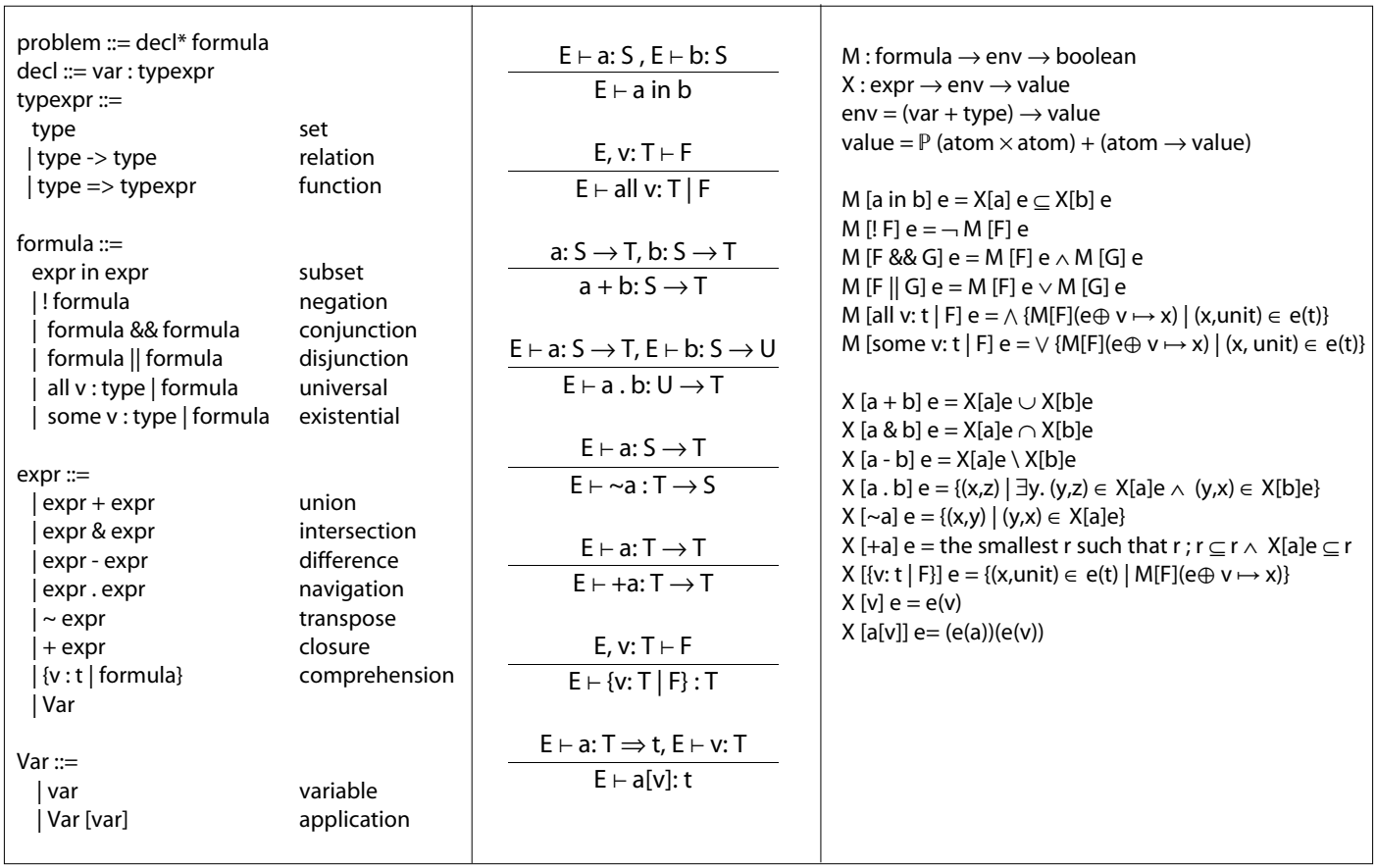

Figure 1: Syntax, type rules and semantics of the logic

Our paper is structured as follows. First, we present the logic, with its syntax, type system and formal semantics. The analysis itself is then explained. We report on some case study applications of the analysis, and give some performance results. The paper closes with a comparison to related work, and a brief discussion of other applications and future prospects.

2 The Logic

The logic is defined in Figure 1, with an abstract syntax (on the left), a type system (in the middle) and a semantics (on the right). Most of its features are standard, so we focus here on its novelties: the treatment of scalars as singleton sets, the encoding of sets as degenerate relations, and the dot operator used to form 'navigation expressions'. The motivation for the design of the logic is explained in detail in [15].

2.1 Syntax

The syntax is mostly identical to standard mathematical syntax, but we have chosen to use ASCII rather than typographic symbols for operators. This makes a stronger connection to our object modelling language, Alloy, which is pure ASCII for ease of use, and also helps us distinguish the operators of our syntax (such as &&) from the mathematical functions (such as ∧) used to define them.

The logic is strongly typed, and a formula is accompanied by declarations of the set and relation variables; we call the combination of a formula and its declarations a *problem*. Each declaration associates a type with a variable. There are three kinds of type:

- the *set* type *T*, denoting sets of atoms drawn from *T*;
- the *relation* type *S* -> *T*, denoting relations from *S* to *T*;
- the *function* type *T* => *t*, denoting functions from atoms of *T* to values of type *t*.

Types are constructed from basic types that denote disjoint sets of atoms. We use upper case names for basic types and lower case names for arbitrary types. So in the type *T* => *t*, the index type *T* must be a basic type but *t* may be a set type, relation type or another function type.

Functions correspond to predicates of arity greater than two. The predicate *Rides* (*r,j,h*) that holds when jockey *j* rides horse *h* in race *r*, for example, might be declared as a function

Rides : *Race* => *Jockey* -> *Horse*

and, for a given race *r*, the expression *Rides*[*r*] would then denote a relation mapping jockeys to their horses in that race. Functions retain the binary flavour of the logic: they fit naturally into

diagrams, lead to simpler expression syntax, and can accommodate multiplicity markings. In Alloy, the question marks in

Rides : *Race* => *Jockey?* -> *Horse?*

indicate that, in each race, a jockey rides at most one horse and vice versa. Also, by including functions in the logic, we are able to skolemize formulas (Section 3.1).

There are no scalar types. To declare a scalar variable, we declare it to be a set

$v : T$

and add a constraint that makes the set a singleton:

some x: $T \mid x = v$

This allows navigation expressions to be written uniformly, without the need to convert back and forth between scalars and sets, sidesteps the partial function problem, and simplifies the semantics (and its implementation) [16].

Formulas have a conventional syntax. There is only one elementary formula, stating that one expression is a subset of another; an equality of two expressions is short for a pair of inequalities, one in each direction. In quantified formulas, the variable is declared to have basic type, and is interpreted as being bound to singleton subsets of the type.

Expressions are formed using the standard set operators (union, intersection and difference), the unary relational operators (transpose and transitive closure), and the dot operator, used to form navigation expressions. The unary operators are prefixes, to make parsing easy.

Set comprehension has the standard form. Set and relation variables are expressions, but function variables, and functions in general, are not. Ensuring that functions can only be applied to variables guarantees that an expression involving a function is always well defined, since the function's argument will denote a singleton set.

2.2 Type System

We treat sets semantically as degenerate relations, viewing the set $\{e_1, e_2, \ldots\}$ as the relation $\{(e_1,unit), (e_2,unit), \ldots\}$ where *unit* is a special atom that is the sole member of a special type *Unit*. Unlike our treatment of scalars as singleton sets, this is purely a trick that makes the semantics more uniform, and it can be ignored by a user of the logic. The type of a variable declared as *v: T* is thus represented as $T \rightarrow Unit$, although we shall write this as T for short.

The typing rules determine which problems are well-formed. The judgment $E \vdash a : t$ says that in the type environment E, expression a has type t; the judgment $E \vdash F$ says that in environment E, the formula F is well-typed. We have omitted obvious rules (eg, for conjunction), and those that are identical to rules given (eg, for intersection).

A problem is type checked in an initial environment that binds each variable to the type as declared (with set types appropriately represented as relations to *Unit*). The environment is extended in the checking of quantified formulas and set comprehensions. For example, the rule for universal quantification says that the quantified formula is well-typed when its body is well-typed in the environment extended with the binding of the bound variable to its declared type.

The set operators can be applied to sets or relations; when + is applied to sets, for example, the type T will be *Unit*. Likewise, the dot operator can be applied to sets or relations, in any combination that the typing allows. Note that the typing rules make clear where sets alone are legal: for bound variables, and the arguments of function applications.

2.3 Semantics

The meaning of the logic is defined by a standard denotational semantics. There are two meaning functions: M, which interprets a formula as true or false, and X, which interprets an expression as a value. Values are either binary relations over atoms, or functions from atoms to values. Interpretation is always in the context of an environment that binds variables and basic types to values, so each meaning function takes both a syntactic object and an environment as arguments.

Each rule defines the meaning of an expression or formula in terms of its constituents. For example, the elementary formula *a in b* is true in the environment e when $X[a]e$, the relation denoted by a in e, is a subset of $X[b]e$, the relation denoted by b in e. The quantified formula *all* v: $t \mid F$ is true in e when F is true in every environment $e \oplus v \mapsto x$ obtained by adding to e a binding of v to x, where x is a member of the set denoted by the type t in e. The membership condition is written

$(x,unit) \in e(t)$

since the set $e(t)$ is, like all other sets, encoded as a relation. We assume that bound variables have been systematically renamed if necessary to avoid shadowing.

All operators have their standard interpretation, except the dot operator. When s is a set and r is a relation, $s.r$ denotes the image of s under r. Combining this with the treatment of scalars as singleton sets results in a uniform syntax for navigation expressions. For example, if p is a person, *p.mother* will denote p's mother; *p.parents* will denote the set of p's parents; *p.parents.brothers* will denote p's uncles; etc.

By treating sets as degenerate relations, and by typing the dot operator loosely, we get as an added bonus that $q.{\sim}p$ is the composition of two relations p and q, and ${\sim}t.{\sim}s$ is the cross product of sets s and t. Alloy does not currently exploit this, and always uses the dot operator as relational image, but it costs nothing to make the logic more general. We can retrieve the simpler definition by noting that, in the semantic equation for $X[a.b]$, the variable z will be *unit* when a is a set, so the result will be a set also.

The meaning of a problem is the collection of well-formed environments in which its formula evaluates to true. An environment is well-formed if: (1) it assigns values to the variables and basic types appearing in the problem's

declarations, and (2) it is well-typed—namely that it assigns to each variable an appropriate value given the variable's type. For example, if a variable v has type $S \rightarrow T$ in an environment e, then $e(v)$, the value assigned to v in e, must be a relation from the set denoted by S to the set denoted by T.

The environments for which the formula is true are the *models* of the formula. To avoid that term's many overloadings, we often call them *instances* or *solutions* instead. If a formula has at least one model, it is said to be *consistent*; when every well-formed environment is a model, the formula is *valid*. The negation of a valid formula is inconsistent, so to check an assertion, we look for a model to its negation; if one is found, it is a *counterexample*.

Since the logic is undecidable, it is impossible to determine automatically whether a formula is valid or consistent. We therefore limit our analysis to a finite *scope* that bounds the sizes of the carrier sets of the basic types. We say that a model is *within a scope of k* if it assigns to each type a set consisting of no more than k elements. Clearly, if we succeed in finding a model to a formula, we have demonstrated that it is consistent. Failure to find a model within a given scope, however, does not prove that the formula is inconsistent (although in practice, for a large enough scope, it often strongly suggests it).

2.4 Example

As a trivial example, consider checking the theorem that for all relations r

```
all x: X | some y: Y | x.r = y
```

To check this, we would formulate its negation as a problem

```
r : X -> Y
! all x: X | some y: Y | x.r = y
```

whose models are those assignments in which r is not a total function. The analysis, as explained below, will actually generate a result such as

```
r = {X0, Y0), (X0, Y1)}
x = {X1}
```

that includes a value for the quantified variable x: this is a S*kolem constant* that acts as a witness to the invalidity of the theorem. Our analysis does not guarantee to give the smallest model; which model is generated depends on the SAT solver used. In most cases, however, the model is a small one, and in this case, our tool would make r empty.

3 Analysis

The analysis involves five steps:

1. Two simple manipulations—conversion to negation normal form and skolemization—are performed on the formula.
2. The formula is translated, for the chosen scope, into a boolean formula, along with a mapping between relational variables and the boolean variables used to encode them. This boolean formula is constructed so that it has a model exactly when the relational formula has a model in the given scope.
3. The boolean formula is converted to conjunctive normal form, the preferred input format of most SAT solvers.
4. The boolean formula is presented to the SAT solver.
5. If the solver finds a model, a model of the relational formula is then reconstructed from it using the mapping produced in Step 2.

We focus here on translation, Step 2. Steps 1 and 3 involve well-known manipulations and Step 5 is trivial. Step 4 is delegated to an off-the-shelf tool; because Step 2 generates a completely standard boolean formula, we can exploit advances in SAT technology without any change to our tool. Only Step 4 is computationally intensive, but its cost depends crucially on how the earlier steps are performed.

Much of the complexity of the translation arises from the elimination of quantifiers. Translating to QBF—quantified boolean formulas—would be much simpler, but would rule out the most powerful and highly tuned SAT solvers that are currently available.

3.1 Normalization of the Relational Formula

Before translating the relational formula, we convert it to negation normal form (NNF) and skolemize it. In NNF, only elementary formulas are negated. To convert to NNF, we simply push negations inwards using de Morgan's laws. The problem of Section 2.4

```
! all x: X | some y: Y | x.r = y
```

for example, becomes

```
some x: X | all y: Y | ! x.r = y
```

Skolemization eliminates existentially quantified variables. If a variable is existentially quantified in a formula that is enclosed by no universal quantifiers, it can be replaced by a scalar. Our formula is thus transformed to

```
all y: Y | ! x.r = y
```

with the addition of a free variable x: X and a constraint

```
some z: X | z = x
```

saying that x represents a scalar, resulting in the problem

```
r : X -> Y
x : X
all y: Y | ! x.r = y
some z: X | z = x
```

It might seem odd to replace one existential quantifier with another, but even in this trivial example it can be seen that the body of the added quantified formula is simpler than the body of the formula that was skolemized.

If a variable is existentially quantified in a formula that is enclosed by a universal quantifier, it is instead replaced by a function. For example,

```
all x: X | some y: Y | x.r = y
```

is converted to

MT : formula → booleanFormula tree
XT : expr → value tree
α tree = (var × (index → α tree)) + α
value = booleanFormulaMatrix + (index → value)

MT [a in b] = merge (MT[a], MT[b], λp,q. $\wedge_{ij}$ {$p_{ij} \Rightarrow q_{ij}$})
MT [! F] = map (MT[F], ¬)
MT [F && G] = merge (MT[F], MT[F], ∧)
MT [F || G] = merge (MT[F], MT[F], ∨)
MT [all v: t | F] = fold (MT[F], ∧)
MT [some v: t | F] = fold (MT[F], ∨)

XT [a + b] = merge (XT[a], XT[b], λp,q.μr. $r_{ij} = p_{ij} \vee q_{ij}$)
XT [a & b] = merge (XT[a], XT[b], λp,q.μr. $r_{ij} = p_{ij} \wedge q_{ij}$)
XT [a - b] = merge (XT[a], XT[b], λp,q.μr. $r_{ij} = p_{ij} \wedge \neg q_{ij}$)
XT [a . b] = merge (XT[a], XT[b], λp,q.μr. $r_{ij} = \exists k.\ p_{ik} \wedge q_{kj}$)
XT [~a] = map (XT[a], λp.(r | $r_{ij} = p_{ji}$))
XT [+a] = map (XT[a], closure)
XT [{v: t | F] = fold (MT[F], λF. μr. $r_{0i} = F(i)$)
XT [a[v]] = merge (XT[a], XT[v], λs,x. μs_i. x_{ii})
XT [v] = (v, λi.(μr. $r_{0j} = (i = j)$)) when v is quantified
XT [v] = create (v) otherwise

merge : α tree, α tree, (α,$\alpha \rightarrow \beta$) → β tree
merge (x, y, o) = o(x, y)
merge ((u,t1), (u,t2), o) = (u, λi. merge(t1(i),t2(i),o))
merge ((u,t1), (v,t2), o) = (u, λi. merge(t1(i),(v,t2),o)) when u < v
merge ((u,t1), (v,t2), o) = (v, λi. merge((u,t1),t2(i),o)) when v < u
merge ((u,t), y, o) = (u, λi. merge(t(i),y ,o))
merge (x, (v,t), o) = (v, λi. merge(x,t(i),o))

map : α tree, ($\alpha \rightarrow \alpha$) → α tree
map (x, o) = o(x)
map ((u,t), o) = (u, λi.map (t(i),o))

fold : α tree, ((index→α)→ β) → β tree
fold ((u,t), o) = o(t) when t(i) elementary
fold ((u,t), o) = (u, λi. fold(t (i),o)) otherwise

create: var → value
create (v) = (r | r_{0i} = a fresh boolean variable F(v,i)) for v: S
create (v) = (r | r_{ij} = a fresh boolean variable F(v,i,j)) for v: S →T
create (v) = (r | r_i= create (v_i)) for v: S ⇒t

Figure 2: Translation rules and tree operations

all x: *X* | *x.r* = *y*[*x*]

by replacing *y* with the function

y: *X* => *Y*

and adding a constraint that each *y*[*x*] is a singleton. This scheme generalizes to an arbitrary number of universal quantifiers; we simply create a function indexed by as many types as necessary. Not all existential quantifiers are eliminated, however, since skolemization is not applied inside set comprehensions.

3.2 Overview of Translation

Given a relational formula, we can construct a boolean formula that has a model exactly when the original formula has a model in some given scope. Here's why. Once we have fixed the scope, a value of a relation from *S* to *T* can be represented as a bit matrix with a 1 in the *i*th row and *j*th column when the *i*th atom in *S* is related to the *j*th atom in *T*, and a 0 otherwise. The collection of possible values of a relation can thus be expressed by a matrix of boolean variables. Any constraint on a relation can be expressed as a formula in these boolean variables; and a relational formula as a whole can be similarly expressed by introducing boolean variables for each relational variable.

This was the analysis we presented in our previous work [15]. In this paper, we extend the scheme to include quantifiers. The idea is intuitively simple, but a little intricate in its details. One way to translate a universal formula would be to expand the body, by making a copy for each possible value of the quantified variable, and then conjoining these (or disjoining them, depending on the quantifier).

This approach is not compositional, though. Instead, for each formula, we generate a mapping from environments to boolean formulas; for each expression, we generate a mapping to matrices of boolean formulas. This mapping parameterizes the formula or matrix by the values of all the variables that will subsequently be bound.

Suppose we have a universal formula whose variable is *w*, and whose body mentions additionally the quantified variables *u* and *v*. The result of translating the body will be a mapping from environments that bind *u*, *v* and *w* to boolean formulas. To translate the formula as a whole, we form a new mapping whose environments bind only *u* and *v*, and which, for a given pair of values u_0 and v_0, yields the conjunction of the formulas that the previous mapping yielded for environments of the form $\{u \mapsto u_0, v \mapsto v_0, w \mapsto w_i\}$ for all values w_i of *w*.

This approach follows the semantics of the logic: from each syntactic object, we create a function from environments to meanings. In the semantics, however, the environment binds not only quantified variables but also set and relation variables. In our translation, the values of set and relation variables are encoded as matrices of boolean variables, and the environment binds only the quantified variables.

3.3 Tree Manipulations

Rather than treating the mappings abstractly, we show how they are represented and manipulated concretely. Figure 2 defines the translation scheme in terms of the translation functions (on the left) and some utility functions (on the right).

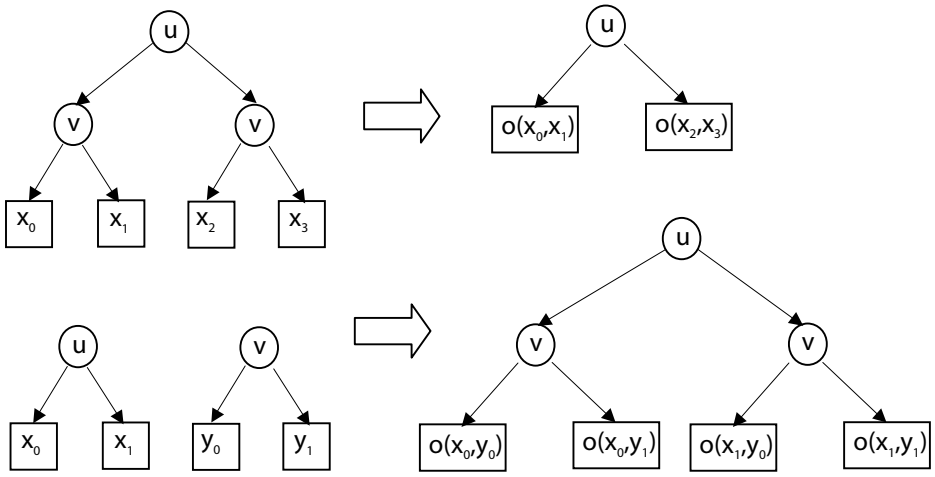

Figure 3: Tree operations: fold (above), merge (below)

There are two kinds of mapping, one for parameterizing formulas and one for parameterizing values (represented as indexed matrices of boolean formulas). These mappings are represented as trees, whose leaf nodes are the formulas or values; the tree manipulations are independent of the leaf type, and are thus described on a polymorphic tree. The internal nodes are labelled with variable names, and their outgoing edges are labelled with indices that correspond to the values of the variables.

For example, in a scope of 2, a relational formula with two free variables *u* and *v* would be represented as the tree shown in the top left-hand side of Figure 3. To find the formula for the case in which *u* takes on its first value, and *v* its second value, for example, we follow the first outgoing edge of *u* and the second outgoing edge of *v*, and reach the leaf formula x_1.

The translation rules involve applications of various tree operations defined in an ML-like notation on the right. These are: *merge*, which merges two trees by combining their leaves pairwise; *map*, which applies an operator to all the leaves; and *fold*, which collapses the lowest level of a tree by applying a function to all the leaves of each smallest subtree.

To translate a compound formula, we first translate its constituent subformulas, and then merge the resulting trees, combining the formulas at their leaves. If the two trees have the same variables and they appear from root to leaf in the same order, merging is easy: leaves aside, the trees are isomorphic, and we simply create a new tree with the same structure whose leaves are the pairwise combinations of the leaves in the original trees.

Unfortunately, the trees are not generally isomorphic, since different subformulas mention different variables. We impose an ordering on the variables (by numbering them according to their quantification depth), and an invariant on the trees that the variables appear in this order. Now to merge two trees, we must essentially interpose an extra level in one tree whenever it omits a variable appearing in the other (see the lower part of Figure 3). The algorithm is given, on the right of Figure 2. The cases are to be interpreted sequentially, with the first one that matches being applied; *x* and *y* stand for values, *u* and *v* for variables, and *t*, *t1*, and *t2* for trees.

The *merge* function takes a different operator for translating different kinds of formula or expression. For example, when translating an elementary formula *a in b*, the operator is

$$\lambda p,q.\wedge_{ij}\{p_{ij}\Rightarrow q_{ij}\}$$

which takes two matrices of boolean formulas, and returns the formula that says that, for every *i* and *j*, the formula in the *i*th row and *j*th column of *p* implies the formula in the same position in *q*. This embodies the intuition that if the values of *p* and *q* are represented as bit matrices, then for the relation *q* to represent a superset of the relation *p*, it must have a 1 wherever *p* does. When there are no quantifiers, the trees are all degenerate, and *merge* reduces to the direct application of the operator—exactly as in our previous scheme [15].

The operator for union expressions

$$\lambda p,q.(\mu r.r_{ij}=p_{ij}\vee q_{ij})$$

uses the definition operator μ; the expression $\mu x.F$ denotes the value which when assigned to *x* makes the formula *F* true. So this operator says that the union of *p* and *q* is a matrix *r* such that the formula in the *i*th row and *j*th column of the result *r* is the disjunction of the corresponding formulas in the matrices of *p* and *q*. In other words, a pair belongs to the union of two relations if it belongs to either relation.

The other tree functions used in the translation are simpler than *merge*. Unary operators are translated with a *map* function that creates a new tree in which a corresponding unary translation operator has been applied to each leaf. The negation of a formula, for example, is obtained by negating the boolean formula at each leaf. The transpose of an expression is obtained

by taking the mirror image of the matrix at each leaf. The transitive closure is obtained by applying an operator that computes closure using iterative doubling, as explained in [15].

Quantified formulas are translated using *fold*. Its second argument is an operator that takes a tree of depth 1. Since the variables in the tree are ordered by quantification depth, the translation of the body of a quantified formula is sure to be a tree in which the quantified variable appears last, just above the leaves. To obtain the meaning of the formula as a whole, we therefore collapse the subtrees at the leaves, by disjunction or conjunction depending on the quantifier.

Set comprehensions are handled with *fold* too. Here, the operator creates a vector of boolean formulas, one for each leaf, in order, thus forming a set whose *i*th element is present when the *i*th value of the quantified variable makes the body formula true.

Finally, quantified variables are translated to trees of unit depth in which the *i*th subtree is the vector whose *j*th element is true when $i = j$ and false otherwise. Declared variables are translated into values: vectors and matrices of boolean variables for sets and relations respectively, and higher-dimensional structures for functions. In the definition of the function *create*, the indices range in the obvious way over the scope.

3.4 Example Translation

Our example formula

$$all\ y: Y \mid !\ x.r = y$$

would be translated, in a scope of 2, as follows. For x, we generate the vector $[x_0\ x_1]$ and for the relation r, the matrix $[r_{00}\ r_{01}\ ,\ r_{10}\ r_{11}]$, using 6 boolean variables in total. The variable y is represented as a tree whose root is labelled y, with branches to the two vectors [1 0] and [0 1] that correspond to y taking the first and second value of the type Y respectively.

Using the fourth M rule, the expression $x.r$ is translated to

$$[\ (x_0 \wedge r_{00}) \vee (x_1 \wedge r_{10}) \qquad (x_0 \wedge r_{01}) \vee (x_1 \wedge r_{11})\ ]$$

The formula $x.r = y$ gives a tree with y at the root, pointing to two formulas

$$((x_0 \wedge r_{00}) \vee (x_1 \wedge r_{10})) \wedge \neg ((x_0 \wedge r_{01}) \vee (x_1 \wedge r_{11}))$$
$$\neg ((x_0 \wedge r_{00}) \vee (x_1 \wedge r_{10})) \wedge ((x_0 \wedge r_{01}) \vee (x_1 \wedge r_{11}))$$

that are true when $x.r$ and y are equal for the first and second values of y respectively. We then map negation over the tree, which negates these two formulas, and then obtain the translation of the quantified formula by folding conjunction over the tree, obtaining

$$\neg (((x_0 \wedge r_{00}) \vee (x_1 \wedge r_{10})) \wedge \neg ((x_0 \wedge r_{01}) \vee (x_1 \wedge r_{11}))) \wedge$$
$$\neg (\neg ((x_0 \wedge r_{00}) \vee (x_1 \wedge r_{10})) \wedge ((x_0 \wedge r_{01}) \vee (x_1 \wedge r_{11})))$$

which is then presented to the SAT solver. Our implementation encapsulates boolean formulas in an abstract data type which allows it to simplify formulas during translation; the resulting formula would therefore be simpler than this.

3.5 Conversion to CNF , Solving and Mapping Back

The result of the translation step is a single boolean formula: that is, a formula over propositional variables with and $\neg$, $\wedge$ and $\vee$. No tree structure can remain because the only free variables in the relational formula are declared sets and relations, which, as explained above, are translated into boolean variables and do not appear as intermediate nodes in the tree.

This formula is converted to conjunctive normal form (CNF) before being handed to the solver. To avoid exponential blowup due to disjunctions, we introduce a temporary boolean variable for every subformula [29].

The solver, if it finds a solution, returns a model that assigns true or false to each boolean variable. From this assignment, we reconstruct a model of the relational formula as follows. If the scope is k, we create names *T0, T1, …* for each of the k atoms in each type T. For a relation $r : S \text{ -> } T$, we look up in the assignment for each $0 \le i, j < k$ the value of each boolean variable v_{ij} that was used to encode the relation r, and insert into r the pair (*Si, Tj*) if this value is true.

A solution to the boolean formula of Section 3.4, for example, has r_{00}, r_{01} and x_1 true, and all others false, which gives the result shown in Section 2.4.

4 Results

The analysis described here has been implemented in the Alloy Analyzer (AA)[21]. Alloy [16] is an attempt to combine the best features of Z [34] and the Object Constraint Language of UML [39] in a lightweight notation. It takes UML's emphasis on binary relations, and the expression of constraints with sets of objects formed by 'navigations', but with Z's much simpler semantics.

4.1 Implementing the Analysis

The tool implements the logic presented here as an intermediate language for Alloy, although it does not offer functions in their full generality. In its current version, Alloy only allows functions from basic types to relations (which we call 'indexed relations'), and only outermost existentials are skolemized. Otherwise, the analysis is implemented as described here.

The tool is roughly 50,000 lines of code, of which 15,000 implement the front end (parsing, type inference, static semantic checks, schema calculus, translation to the intermediate language); 5,000 implement the translation that is the subject of this paper; 10,000 implement manipulations of the boolean formula (conversion to normal form, simplifications, conversion to various solver formats); 5,000 implement the user interface; and 15,000 implement a visualization mechanism. All the code is written in Java, except for the boolean formula manipulations, which are written in C.

4.2 Choice of SAT Solvers

The tool's backend wraps a collection of off-the-shelf SAT solvers [4,31,32,41]. The deterministic solvers SATO [41] and RelSAT [4] seem to work best. Early on, we had some success with

WalkSAT [31], a stochastic solver, but we assiduously avoided introducing too many temporary variables when converting to CNF [15]. This conversion step became a bottleneck, which was eliminated by more aggressive variable introduction. Unfortunately, the redundancy this adds to the formula foils stochastic solvers, so WalkSAT rarely works well.

We learnt an interesting lesson in our experiments with solvers. In our eagerness for platform-independence, we planned initially to implement our own solvers in Java. Our prototype tool included a trie-based implementation [42], in Java, of the same Davis-Putnam (DP) algorithm [9] that underlies many deterministic solvers. Because WalkSAT outperformed it so dramatically [15], and because our own Java implementation of WalkSAT came within a factor of 3 of the performance of the C implementation, we foolishly attributed the failure of our DP implementation to the Davis-Putnam method itself. Later, we discovered that a highly tuned implementation of Davis Putnam, such as SATO, performed orders of magnitude better. This experience made us appreciate the importance of a flexible backend, to which we could attach new solvers as they became available.

4.3 New Expressiveness

The most important consequence of this work has been the ability to add quantifiers to our language. Our previous analysis was limited to pure relational formulas with no quantifiers. In principle, the first-order properties that arise in software specifications can always be written without quantifiers [36]. To say "everybody likes a winner" we could write

Winner.~(Un \ likes)= {}

In this formula, the relational expression following the dot (the transpose of the complement of the *likes* relation) maps persons to persons who don't like them; the expression denotes persons who don't like some winner; and the formula as a whole says that the set of such persons is empty. Needless to say, this style of specification did not win many admirers, despite its terseness. We experimented with an algorithm of Tarski's [36] for performing the elimination automatically, but were not able to generate relational expressions of a reasonable size.

Now, with quantifiers, we can write instead

all p: Person | all w: Winner | w in p.likes

Our experience so far, in six months of using the language and its tool, suggests that quantifiers and navigation expressions make a big difference. While NP [14,17], the language of our Nitpick checker, was usable only by dedicated experts, we have found that students with only a modest background in discrete mathematics can pick up Alloy in a couple of days. (Gaining proficiency takes much longer, of course, but that has more to do with learning how to construct focused, abstract models than with details of any language.)

We have constructed and analyzed a variety of models in Alloy that would have been at the very least difficult to express in NP. Moreover, since the Alloy Analyzer (AA) is a far more powerful solver than Nitpick, we have been able to construct larger models. Whereas before we had to craft models carefully to make them analyzable, we no longer find it necessary to adjust our models, except to fix the (many) errors that the tool exposes. For a scope of 3, which is usually enough to catch most errors, Nitpick was limited to a state of about 5 relations; the new tool can handle 10 relations, and sometimes 20 or more, with ease. Examples include:

- *COM* [22]. We took the Z specification of Sullivan et al [35] and translated it into Alloy. The resulting model is about 150 lines long, and has 8 relations, 1 indexed relation, and 8 sets. Using AA, we were able to generate automatically the counterexamples that Sullivan and his colleagues had found by hand analysis.
- *Intentional Naming* [26]. Sarfraz Khurshid constructed a model of the design of a name server that allows services to be looked up by their properties [1]. The model is 130 lines long, and has 11 relations, 1 indexed relation (ie, function from a basic type to a relation), and 8 sets. A variety of problems were discovered with the design. AA takes no longer than 5 seconds to find any of the counterexamples.
- *UML Metamodel* [38]. We translated the entire core metamodel of UML from OCL [39], the constraint language of UML, into Alloy. The resulting model, which is about 400 lines long, is about half the size of the OCL version. It has 41 relations and 37 sets. We used AA to show that the metamodel is consistent, by generating a sample UML model that satisfies all the constraints (with additional constraints that rule out the trivial empty model). Finding this model takes 6 seconds.

The reader should bear in mind when considering the size of these models that a language like Alloy, NP or Z tends to be much more succinct than the languages used by model checkers. The input to a model checker such as SMV [6] or SPIN [13] is a rather low-level program. A model of Mobile IP that we built in SMV was 10 times longer than our NP version.

4.4 Performance

As an experiment, we analyzed two groups of models. The first group consists of the three models mentioned above, which demonstrate the capability of the new language and analysis. The second group consists of three models originally written in NP and analyzed with our previous SAT-based analysis [15]:

- *Finder*, a toy model of the Macintosh file system that uses transitive closure (which cannot be handled by tools that are restricted to first-order predicate logic);
- *Style*, a model of an aspect of the paragraph style mechanism of Microsoft Word that was developed as a class exercise [16];
- *Mobile IP*, a model that exposed a flaw in an internet protocol for forwarding messages to mobile hosts [19].

We translated these into Alloy using quantifiers, and analyzed them in AA to test whether quantifiers incur a significant cost.

The results are shown in Tables 1 and 2. For each example, and for a variety of scopes, we show the size of the space and some timings. The space is given as the number of bits used to

encode an assignment of values to sets and relations, so 100 bits corresponds to roughly 10^{30} configurations. These results are averaged over a few problems for each case study, some satisfiable and some unsatisfiable. The analyses in Table 1 involve only invariants, so the number of configurations is the number of states; the analyses in Table 2 involve executions, so a configuration is a pair (or for *Style*, a triple) of states. The number of boolean variables in the generated formula is often much larger because of variable introduction.

All timings are given in seconds, measured on a Pentium II with a 233MHz processor and 192MB of memory. Performance can often be significantly improved by selecting a different solver, or by tweaking solver parameters. All the measurements in the table, however, were taken using RelSAT with its default settings. Translation (which is included in the timings) takes less than 10 seconds in every case; the SAT solver is the bottleneck.

For the first group, two kinds of analysis were performed (Table 1). The column marked *Instance* gives the time taken to find a solution when one exists: for the UML example, this instance demonstrates consistency of the constraint, and for all others it represents a counterexample to a theorem (or for Intentional Naming, counterexamples to several theorems). The column marked *Exhaust* gives the time taken to exhaust the space when no solutions exist. The UML example includes constraints that require at least one of each model element; this rules out a solution in a scope of 2. For all other cases, the analyses in this column involve checking a variety of valid theorems. Because no theorems were checked for the UML metamodel, there are no values for exhausting the state space for scopes above 2.

As can be seen, the new method can handle spaces of 100 bits (10^{30} configurations) with ease. When the solver has to exhaust the space, the timings, and their variances, increase dramatically with scope. In fact, almost all the problems for which an instance exists have an instance in a small scope (2 for *COM*, 2 and 3 for *Intentional Naming*, and 3 for *UML*). The timings for larger scopes are thus a bit specious, but they do suggest that it is better to start with a small scope and increase it gradually.

For the second group (Table 2), we considered only cases in which a solution is found, since our previous analysis used a stochastic solver that ran forever when no solution existed. The column *Old* gives the timings from our old paper. These were run on a machine that is about two thirds of the speed of the machine on which our new experiments were performed, so these timings could be reduced. The new method sometimes performs worse than the old method, probably because the relational expressions give a tighter encoding of the problem. But it scales better: because of a translation bottleneck (overcome by our use of standard methods [28]) the previous method could not handle specs of the size of those in Table 1, even if they were written without quantifiers.

Example	*Scope*	*Space*	*Instance*	*Exhaust*
COM	2	60 bits	3s	1–4s
	3	132	11s	37–218s
	4	240	71s	240–?s
Intentional Naming	2	64	1s	1s
	3	141	3–31s	19–59s
	4	256	18–346s	??
UML Metamodel	2	228	n/a	6s
	3	465	11s	n/a
	4	784	17s	n/a

Table 1: Results for new models

Example	*Scope*	*Space*	*Old* [16]	*New*
Finder	5	160 bits	3s	9s
	6	216	162s	13s
Style	3	90	1s	3s
	4	156	2s	3s
	5	250	11s	6s
Mobile IP	3	175	0s	1s
	4	280	3s	6s
	5	600	8s	29s

Table 2: Results for old models

5 Related Work

Unlike our previous analysis [15], the analysis described here can handle quantifiers, and can handle larger specifications. Both analyses dramatically outperform an earlier analysis, based on explicit enumeration of set and relation values [18], on which our Nitpick checker was based. Recently, Craig Damon has improved this earlier analysis with more powerful pruning schemes, but since he has not yet incorporated quantifiers, it is not possible to compare to our new analysis.

As far as we know, there are no other analyses for a first-order logic that handle quantifiers and transitive closure. A variety of model finders have been developed for group-theoretic investigations [eg, 33]; these work on a logic of uninterpreted functions, and do not handle relations or closure. Several animators for Z have been developed using Prolog as an underlying engine [11,12, 24,43], but these cannot handle large spaces.

Most other tools for relational notations are less automatic. Theorem provers (such as Z/Eves [7] and PVS [28]) can—unlike the Alloy Analyzer—prove theorems, but do not generate counterexamples, and need help with lemmas and proof strategy. Execution engines (such as the IFAD tool [2] for VDM) limit the notation to an executable subset and make the user provide test cases.

Model checkers are designed to handle the complexity of interleaving, and not complexity in the state structure itself. Their input languages do not offer relations as types, and require relational operators to be specified algorithmically at a low level. Explicit model checkers (such as SPIN [13]) do not permit declarative specification, in which invariants and operations are given by conjunction of constraints.

This and our previous analysis [15] might be described as 'symbolic', because, as in symbolic model checking [6], there is no explicit representation of individual states. Our notion of scope is implicit in most applications of model checking, since the model itself usually assumes some fixed number of processors, cache lines, etc.

Boolean satisfaction has been used before in planning [10,25] (which is essentially reachability analysis) and more recently in linear temporal logic model checking [6], but the encodings are rather different from that described here.

6 Future Prospects

Relational logic has many applications. Our analysis might be useful in a variety of tools, beyond the Alloy Analyzer:

- A CASE tool (such as Rose [30]) might use our analysis to generate object diagrams from class diagrams;
- An architectural style tool might use our analysis to check the consistency of style constraints (expressed, for example in AML [40] or Darwin [27]) and generate sample architectures;
- A tool for developing requirements (such as KAOS [37]) might use our analysis to check the consistency of goals;
- A refinement tool (such as the B tool [3]) might use our analysis as a verification condition tester, to find counterexamples to proof obligations before attempting a proof.

We have recently developed a strategy for translating code into this logic. Using our analysis, we are able to check a variety of code properties, such as absence of executions that dereference null pointers or create undesirable sharings, and conformance to user-defined specifications [23].

Acknowledgments

Ian Schechter and Ilya Shlyakhter contributed to the implementation. This research was funded by the National Science Foundation (under grant CCR-9523972), by the MIT Center for Innovation in Product Development (under NSF Cooperative Agreement Number EEC-9529140), and by an endowment from Douglas T. Ross.

The Alloy Analyzer may be freely downloaded for a variety of platforms from http://sdg.lcs.mit.edu/alloy.

References

[1] William Adjie-Winoto, Elliot Schwartz, Hari Balakrishnan and Jeremy Lilley. The design and implementation of an intentional naming system. *Proceedings of the 17th ACM Symposium on Operating Systems Principles* (SOSP '99), Kiawah Island, South Carolina, December 1999.

[2] Sten Agerhold and Peter Gorm Larsen. The IFAD VDM Tools: Lightweight Formal Methods. *FM-Trends 1998*: 326-329.

[3] *The B-Tool.* B-Core(UK) Ltd, Harwell, Oxfordshire, England. http://www.b-core.com/btool.html.

[4] R.J. Bayardo Jr. and R. C. Schrag. Using CSP look-back techniques to solve real world SAT instances. *Proc. of the 14th National Conf. on Artificial Intelligence*, 203–208, 1997.

[5] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic Model Checking without BDDs. *Tools and Algorithms for the Analysis and Construction of Systems* (TACAS'99), LNCS 1579, Springer-Verlag, 1999.

[6] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, Vol. 98, No. 2, pp.142–170, June 1992.

[7] Dan Craigen, Irwin Meisels and Mark Saaltink. Analysing Z Specifications with Z/EVES. *Industrial-Strength Formal Methods in Practice*, eds. J.P. Bowen and M.G. Hinchey, September 1999.

[8] Craig A. Damon. *Selective Enumeration*. PhD Thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, August 2000.

[9] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM*, Vol. 7, pp. 202–215, 1960.

[10] Michael D. Ernst, Todd D. Millstein and Daniel S. Weld. Automatic SAT-Compilation of Planning Problems. *Proc. 15th International Joint Conference on Artificial Intelligence* (IJCAI-97), Nagoya, Aichi, Japan, August 1997, pp. 1169–1176.

[11] Daniel Hazel, Paul Strooper and Owen Traynor. Possum: An Animator for the SUM Specification Language. *Proceedings Asia-Pacific Software Engineering Conference and International Computer Science Conference*, pages 42-51, IEEE Computer Society, December 1997.

[12] M.A. Hewitt, C.M. O'Halloran and C.T. Sennett. Experiences with PiZA, an animator for Z. *10th International Conference of Z Users* (ZUM'97), Reading, England, April 1997.

[13] Gerard J. Holzmann. The Model Checker Spin. *IEEE Transactions on Software Engineering, Special issue on Formal Methods in Software Practice*, Volume 23, Number 5, May 1997, 279-295.

[14] Daniel Jackson. Nitpick: A Checkable Specification language. *Proc. First ACM SIGSOFT Workshop on Formal Methods in Software Practice,* San Diego, CA, January 1996, pp. 60–69.

[15] Daniel Jackson. An Intermediate Design Language and its Analysis. *Proc. ACM Conference on Foundations of Software Engineering*, Florida, November 1998.

[16] Daniel Jackson. *Alloy: A Lightweight Object Modelling Notation*. Technical Report 797, MIT Laboratory for Computer Science, Cambridge, MA, February 2000. Available at: http://sdg.lcs.mit.edu/~dnj/publications.

[17] Daniel Jackson and Craig A. Damon. *Nitpick Reference Manual.* CMU-CS-96-109. School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, January 1996.

[18] Daniel Jackson and Craig A. Damon. Elements of Style: Analyzing a Software Design Feature with a Counterexample Detector. *IEEE Transactions on Software Engineering*, Vol. 22, No. 7, July 1996, pp. 484–495.

[19] Daniel Jackson, Somesh Jha and Craig A. Damon. Isomorph-free Model Enumeration: A New Method for Checking Relational Specifications. *ACM Transactions on Programming Languages and Systems*, Vol. 20, No. 2, March 1998, pp. 302–343.

[20] Daniel Jackson, Yuchung Ng and Jeannette Wing. A Nitpick Analysis of IPv6. To appear, *Formal Aspects of Computing.*

[21] Daniel Jackson, Ian Schechter and Ilya Shlyakhter. Alcoa: the Alloy Constraint Analyzer. Proc. International Conference on Software Engineering, Limerick, Ireland, June 2000.

[22] Daniel Jackson and Kevin Sullivan. COM Revisited: Tool-Assisted Modelling and Analysis of Software Structures. *Proc. Foundations of Software Engineering* (FSE 2000), San Diego, CA, November 2000.

[23] Daniel Jackson & Mandana Vaziri. Finding Bugs with a Constraint Solver. *Proc. International Conference on Software Testing and Analysis* (ISSTA 2000), Portland, OR, August 2000.

[24] R. D. Knott and P. J. Krause. The Implementation of Z Specifications using Program Transformation Systems: The SuZan Project. *The Unified Computation Laboratory*, IMA Conference Series No 35 (Editors: C Rattray, R G Clark), Clarendon Press, Oxford, 1992, pgs 207-220.

[25] Henry Kautz and Bart Selman. Pushing the envelope: planning, propositional logic, and stochastic search. *Proc. 5th National Conference on Artificial Intelligence*, 1996, pp. 1194–1201.

[26] Sarfraz Khurshid and Daniel Jackson. Exploring the Design of an Intentional Naming System with an Automatic Constraint Analyzer. *Proc. Automated Software Engineering*, Grenoble, France, September 2000.

[27] J. Magee, N. Dulay, S. Eisenbach and J. Kramer. Specifying Distributed Software Architectures. *Proceedings of 5th European Software Engineering Conference* (ESEC 95), Sitges, Spain, September 1995

[28] Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107-125, February 1995.

[29] D. Plaisted and S. Greenbaum. A structure-preserving clause form translation. *Journal of Symbolic Computation*, 2, 293–304, 1986.

[30] *Rose Visual Modeling Tool*. Rational Software Corporation, Cupertino, California. Inc.

[31] Bart Selman, Henry Kautz and Bram Cohen. Noise strategies for improving local search. *Proc. AAAI-94*, pp. 337–343, 1994.

[32] J.P.M. Silva and K.A. Sakallah. Grasp – A New Search Algorithm for Satisfiability. *IEEE International Conference on Computer Aided Design*, San Jose, CA, November 1996, pp. 220–227.

[33] John Slaney. Finder: Finite domain enumerator, system description. Proc. 12th International Conference on Automated Deduction. Lecture Notes in Artifical Intelligence. Springer-Verlag, Berlin, 798–801.

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

[35] K.J. Sullivan, J. Socha and M. Marchukov. Using Formal Methods to Reason about Architectural Standards. *Proceedings of the International Conference on Software Engineering* (ICSE97), Boston, Massachusetts, May 1997.

[36] Alfred Tarski and Steven Givant. *A Formalization of Set Theory Without Variables*. American Mathematical Society, Colloquium Publications, Volume 41, 1987.

[37] Axel van Lamsweerde, Robert Darimont and Emmanuel Letier. Managing Conflicts in Goal-Driven Requirements Engineering. *IEEE Transactions on Software Engineering*, Vol. 24, No. 11, November 1998.

[38] Mandana Vaziri and Daniel Jackson. *Some Shortcomings of OCL, the Object Constraint Language of UML*. A response to Object Management Group RFI on UML. December 1999. Available at: http://sdg.lcs.mit.edu/~dnj/publications.

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

[40] David S. Wile. AML: An Architecture Meta-Language. *Automated Software Engineering*, *14th IEEE International Conference*, Cocoa Beach ,Florida, USA, October 1999.

[41] Hantao Zhang. SATO: An Efficient Propositional Prover. *Proc. of International Conference on Automated Deduction* (CADE-97).

[42] Hantao Zhang and Mark E. Stickel. *Implementing the Davis-Putnam Algorithm by Tries*. Technical Report 94-12, Artificial Intelligence Center, SRI International, Menlo Park, CA. December 1994.

[43] Jia Xiaoping. An Approach to Animating Z Specifications. *Proceedings of the 19th Annual IEEE International Computer Software and Application Conference* (COMPSAC'95). August 1995, Dallas, TX. pp. 108–113.