An Intermediate Design Language and its Analysis

Daniel Jackson
Laboratory for Computer Science
Massachusetts Institute of Technology
dnj@lcs.mit.edu

Abstract
A simple relational language is presented that has two desirable properties. First, it is sufficiently expressive to encode, fairly naturally, a variety of software design problems. Second, it is amenable to fully automatic analysis. This paper explains the language and its semantics, and describes a new analysis scheme (based on a stochastic boolean solver) that dramatically outperforms existing schemes.

1 Introduction

In the last few years, we have been investigating analysis techniques for software specifications to which state-machine-based analyses – such as model checking – do not apply. We developed a simple specification language, called NP [JD96a], that attempts to capture the essence of Z [Spi92] without many of its complexities, which can be entered and displayed in plain ASCII, and analyzed fully automatically. Our tool, called Nitpick [JD96b], takes NP specifications and can check the consistency of state and operation descriptions (a non-trivial task in the presence of conjunction), and more general properties such as invariant preservation and simple refinements.

Nitpick/NP has been applied in a variety of settings. It has exposed errors in a mobile internet protocol [JNW97, Ng97], in a simulation framework proposed as a standard by the Department of Defense, and in a variety of small case studies. It has been taught to students in the Masters of Software Engineering degree at Carnegie Mellon University, and used on class projects.

Recently, we have made a significant advance in the underlying checking technology. Using a new scheme based on boolean satisfaction, we are now able to analyze problems that are an order of magnitude larger. In comparison to the previous scheme in which the values of states were enumerated explicitly, this scheme might be termed 'symbolic'. It shares some basic features with symbolic model checking [BC+92] (but is largely unrelated in the details). The entire problem is solved 'at once', so that a solution that would be near the start of an explicit search takes no less time to find, and yet problems can be handled that are far too large for an explicit search.

The new scheme allows an improvement also in the architecture of the tool. To simplify the implementation, it helps to treat all datatypes – sets, scalars and relations – equivalently. But this made little sense for the explicit scheme, since the code for generating values of a scalar variable is very different from the code for generating values of a relation. In the symbolic scheme, however, sets and scalars can be treated as degenerate relations. Consequently, the front end of the tool encapsulates most of the complexity of the source language, and the analysis is performed on a much simpler intermediate language.

This paper presents the intermediate language and the new analysis scheme. By focusing on the intermediate language in its own right, rather than on the translation from the source language, we hope to make it easier for other tool builders to determine whether the analysis scheme might be useful to them. Aside from its use on Z-like specifications, the analysis might be applied to object models, or to architecture description languages.

In the rest of the paper, we explain first the language, its syntax, typing and semantics (section 2); illustrate the translation of source into intermediate form by showing some typical examples (section 3); explain how a variety of analysis problems can be reduced to model finding (section 4); present the analysis itself, essentially a compilation (section 5); and finally comment on the efficacy of the approach and its deficiencies.

2 The Intermediate Language

2.1 An Example

Let's start with a trivial illustration. The state of an employment database might be modelled by a relation *worksFor* from employees to companies, and *hasBoss* that associates employees with their bosses. First we declare these two state components:

$$worksFor : Employee \leftrightarrow Company$$

$$hasBoss : Employee \leftrightarrow Employee$$

Now we can write invariants in terms of these variables. To assert, for example, that every employee works for a company, we would write

$$dom\ worksFor = Un$$

SIGSOFT'98 11/98 Florida, USA

```
problem ::= typedecl* formula*

typedecl ::= var : type ↔ type

formula ::=
  expr = expr                 equality
  | expr ⊆ expr               subset
  | ¬ formula                 negation

expr ::=
  Id                          identity relation constant
  | Un                        universal relation constant
  | 0                         empty relation constant
  | expr ∪ expr               union
  | expr ∩ expr               intersection
  | expr \ expr               difference
  | expr ; expr               relational composition
  | expr ~                    transpose
  | expr +                    transitive closure
  | dom expr                  domain
  | expr <: expr              domain restriction
  | var                       variable
```

Figure 1: Syntax

where *Un* stands for the universal set or relation; in Z or NP, this would be written as *Employee* instead. A more tricky assertion is that each employee has at most one boss:

$hasBoss^{\sim} ; hasBoss \subseteq Id$

which would be appear in NP more succinctly as just *fun hasBoss* (asserting that *hasBoss* is a function), or implicitly as part of the declaration by writing the type as *Employee* → *Employee*. Finally, the assertion that a boss cannot work for a company unless each of her employees also works for that company would be written

$hasBoss ; worksFor \subseteq worksFor$

This would be no different in NP, since it lacks quantifiers, but might be written in Z as

$$\forall e : Employee, c: Company \cdot (\exists b : Employee \cdot (e, b) \in hasBoss \land (b, c) \in worksFor) \Rightarrow (e, c) \in worksFor$$

An example of an analysis problem is simply to check that a collection of such assertions forms a consistent invariant: that at least one satisfying state exists. If it finds such a state, the analysis will display it. For the assertions above, it might produce, for example:

$Employee = \{e1, e2, e3\}$
$Company = \{c1, c2, c3\}$
$worksFor = \{e1 \mapsto c1, e2 \mapsto c1, e3 \mapsto c2\}$
$hasBoss = \{e1 \mapsto e2\}$

2.2 Syntax and Semantics

The syntax of the intermediate language is shown in Figure 1. A problem to be analyzed is presented as a list of type declarations followed by a list of formulas, implicitly conjoined. Each formula is either an equation or inequation of two relational expressions, and may be negated.

$M : formula \to boolean$
$E : expr \to \mathbb{P}(atom \times atom)$
$C : type \to \mathbb{P}\, atom$
$A : var \to \mathbb{P}(atom \times atom)$

$M[p \subseteq q] = \forall a,b.\ (a,b) \in E[p] \Rightarrow (a,b) \in E[q]$
$M[p = q] = \forall a,b.\ (a,b) \in E[p] \Leftrightarrow (a,b) \in E[q]$
$E[Id : T \leftrightarrow T] = \{(a,a) \mid a \in C[T]\}$
$E[0] = \{\}$
$E[Un : S \leftrightarrow T] = \{(a,b) \mid a \in C[S] \land b \in C[T]\}$
$E[p \cup q] = \{(a,b) \mid (a,b) \in E[p] \lor (a,b) \in E[q]\}$
$E[p \cap q] = \{(a,b) \mid (a,b) \in E[p] \land (a,b) \in E[q]\}$
$E[p \setminus q] = \{(a,b) \mid (a,b) \in E[p] \land (a,b) \notin E[q]\}$
$E[p ; q] = \{(a,b) \mid \exists c.\ (a,c) \in E[p] \land (c,b) \in E[q]\}$
$E[p^{\sim}] = \{(a,b) \mid (b,a) \in E[p]\}$
$E[p+] =$ the smallest relation x such that $x ; x \subseteq x \land p \subseteq x$
$E[s <: p] = \{(a,b) \in E[p] \mid (a,unit) \in E[s]\}$
$E[dom\ p] = \{(a,unit) \mid \exists b.\ (a,b) \in E[p]\}$
$E[v] = A[v]$

Figure 2: Semantics

Relational expressions are built from a standard repertoire of operators that includes: three constants (the universal, identity, and empty relations); the three set-theoretic operators (union, intersection and difference); and the two quintessentially relational operators (composition and transpose).

Transitive closure is included too; this extends the language's power and is indispensable in practice. We also include two operators that add no power in a formal sense, but allow more efficient generation of boolean formulas. They are the domain operator, which takes a relation to the set of elements that it maps, and domain restriction, which takes a set and a relation, and produces the subrelation whose pairs have first elements in the set.

The semantics of the language is shown in Figure 2. A relation is modelled conventionally as a set of ordered pairs. A set is modelled as a degenerate kind of relation that maps each element of the set to the special value *unit*, the only member of the basic type *Unit*. This allows the set-theoretic operators to be treated uniformly for set and relation arguments; otherwise, we would have to regard them as overloaded, complicating both the language and its analysis.

There are separate semantic functions for formulas (M) and for expressions (E). The meaning assigned to a problem depends on an assignment A of values to variables. If problem P has the meaning *true* for a assignment A, then A is said to be a *model* of P.

The atoms from which the pairs of a relation are constructed are assumed to be drawn from some universe that is partitioned into disjoint, primitive types. The semantic function C maps each primitive type to a set of atoms. Since the atoms are uninterpreted, the choice of C is not significant although the size of $C(t)$ is: we call this the *scope*, since it bounds the assignments that can be constructed. $C(Unit)$ must contain exactly one atom.

The typing rules are shown in Figure 3. The judgment

$p: S \leftrightarrow T$

says that the relation p is constructed from pairs whose first elements are atoms drawn from the type S, and whose second elements are atoms drawn from the type T. All type variables represent uninterpreted, primitive types, so T cannot, for example, have values that are integers, or values that are themselves relations.

Note that the typing rules for constants have no hypotheses, and thus do not constrain the choice of primitive types. In general, these must be determined by type inference. In the formula

$p \cap Id = 0$

for example, which asserts that the relation p maps no element to itself, p will be required to be homogeneous, and the identity constant will be inferred to have the same type as p itself.

2.3 Expressive Power

Tarski developed a formalization of set theory that required no variables or quantifiers, based on a relational calculus comprising only the identity constant, union, complement, transpose, composition and equality [Tar41, TG87]. Tarski showed that his calculus is weaker than first order logic, but that it can express any first order formula containing no subformulas with more than 3 free variables.

Our language is essentially a version of Tarski's calculus with some convenient shorthands, plus transitive closure (which is not expressible in first order logic). Like the relational calculus [Sch79], therefore, it is undecidable, so properties cannot be checked fully automatically without sacrificing soundness or completeness.

The restriction to 3 variables does not seem to matter much in practice, but the lack of quantifiers is a nuisance, and often results in terse but mysterious relational expressions. Tarski gives an algorithm for eliminating quantifiers [Tar41], but it is intricate and tends to produce a much larger relational formula than one would write by hand.

2.4 Rationale for Language Design

Designing an intermediate design language is much like designing an instruction set. We took a RISC-like approach, starting with Tarski's calculus, and only adding crucial features. A feature was deemed crucial if it made the subsequent analysis either much simpler or much more efficient, or if it made the encoding of source formulas much more compact.

Domain and restriction. The domain and restriction operators are not strictly necessary, since the domain of relation p can be written as p ; *Un* and the domain restriction s <: p as (s ; *Un*) $\cap$ p. Type inference will ensure that, in the first case, the right-hand side of the universal constant will be *Unit*, so that the expression will have a set type, and that, in the second case, the universal constant appropriately lifts the set s to a relation whose type matches p's. A similar scheme is described in [SS93], but, lacking *Unit* to distinguish sets from relations, it encodes the domain restriction as $s \cap p$, and requires that sets such as s be polymorphic in their right-hand side – a major inconvenience. In short, the notion of *Unit* is essential, but the inclusion of the operators is debatable.

$$\frac{}{Id : T \leftrightarrow T} \qquad \frac{}{Un : S \leftrightarrow T} \qquad \frac{}{0 : S \leftrightarrow T}$$

$$\frac{p: S \leftrightarrow T,\ q: S \leftrightarrow T}{p \cup q : S \leftrightarrow T} \qquad \frac{p: S \leftrightarrow T,\ q: S \leftrightarrow T}{p \cap q : S \leftrightarrow T}$$

$$\frac{p: S \leftrightarrow T,\ q: S \leftrightarrow T}{p \setminus q : S \leftrightarrow T} \qquad \frac{p: S \leftrightarrow T,\ q: T \leftrightarrow V}{p ; q : S \leftrightarrow V}$$

$$\frac{p: S \leftrightarrow T}{p\sim : T \leftrightarrow S} \qquad \frac{p: T \leftrightarrow T}{p+ : T \leftrightarrow T}$$

$$\frac{p: S \leftrightarrow Unit,\ q: S \leftrightarrow T}{p <: q : S \leftrightarrow T} \qquad \frac{p: S \leftrightarrow T}{dom\ p : S \leftrightarrow Unit}$$

Figure 3: Typing

Subset and equality. Either of these could be dispensed with, but the cost is high. Subset formulas, as we shall see, are cheaper to analyze than equality formulas, so replacing subset by equality degrades performance. We also exploit equalities of the form $v = e$, where v is a variable, by replacing other occurrences of e with v (see section 5.4).

Union and intersection. Since our language includes complementation (via the difference operator and the universal relation), there is technically no need for both union and intersection, since one can always be converted to the other by DeMorgan's laws. In our analysis, however, intersection is cheap while union and complementation are expensive. Eliminating an operator destroys structure that the analysis would be forced to recover later.

Conjunction. Remarkably, Tarksi's calculus requires no logical operators, since the conjunction

$p \subseteq q \wedge v \subseteq w$

for example, can be written

$(p \setminus q) \cup (v \setminus w) = 0$

source language	translated form in intermediate language
ran p	dom p~
p is total	dom p = Un
p is a function	p~ ; p ⊆ Id
e represents a scalar	e ; e~ ⊆ Id ∧ ¬ e = 0
e ∈ s	e ⊆ s
(e, f) ∈ p	(f <: ((e <: Un)~))~ ⊆ p
image of s under p	dom ((s <: p)~)
p ⊕ q (relational override)	((Un \ dom q) <: p) ∪ q
p is acyclic	p+ ∩ Id = 0
s × t	(t <: ((s <: Un)~))~
∀a . (a,a) ∈ p	Id ⊆ p
∀a . (a,a) ∉ p	Id ∩ p = 0
∀a, b . a ∈ s ∧ b ∈ t ⇒ (a,b) ∈ p	(t <: ((s <: Un)~))~ ⊆ p
∀a, b, c . (a,b) ∈ p ∧ (b,c) ∈ q ⇒ (a,c) ∈ r	p ; q ⊆ r
∀a, b, c . (a,b) ∈ p ⇒ (b,c) ∈ q	p ; (Un \ q) = 0

Table 1: Sample translations

Omitting conjunction, however, would be disastrous, since the conjunctive structuring is essential for the translation into conjunctive normal form (see section 5.3).

3 Translating from Source to Intermediate

Table 1 shows some sample translations into the intermediate language. The cases in the upper half of the table show forms that occur in the NP source language, and which are translated by the Nitpick analyzer. The cases in the lower half cannot be expressed directly in NP.

The relation (p~; p) associates two elements of the range of p when they a common domain element is mapped to both. If this relation is a subset of the identity relation, any pair of range elements to which a domain element is mapped are equal. So p is a function exactly when

p~ ; p ⊆ Id

Recall that a set is represented as a relation whose right-hand side has the type *Unit*, denoting the primitive type containing exactly one element. Consequently, such a relation denotes a singleton set if its transpose is a function, and a scalar when, additionally, it is non-empty.

The language allows only binary relations, but relations of higher arity are easily simulated. A cartesian product X with components of types A and B is modelled with two projection functions

a : X → A
b : X → B

To ensure that a and b behave like projections, we assert that both are total functions, and that two X's mapped to the same A's by a and the same B's by b are equal:

```
dom a = Un                  // a is total
dom b = Un                  // b is total
a~ ; a ⊆ Id                 // a is a function
b~ ; b ⊆ Id                 // b is a function
(a ; a~) ∩ (b ; b~) ⊆ Id    // a and b uniquely determine X's
```

4 Reduction of Analysis to Model Finding

Given a problem represented as a collection of relational formulas, our analysis attempts to find a model: that is, an assignment of values to variables for which every formula is true. Many practical design questions can be cast as problems of this form:

Simulation. Consider an operation specified implicitly as a logical formula whose subformulas are relational. We put the logical formula in disjunctive normal form, obtaining a set of problems each corresponding to a 'path' through the operation. Suppose, for example, that our operation has the form

A(s) ⇒ C(s, s') ∧ B(s) ⇒ D(s, s')

where $A(s)$ and $B(s)$ are relational formulas whose relational variables include only components of the pre-state, and $C(s,s')$ and $D(s,s')$ are formulas including components of both pre- and post-states. Then, converting to disjunctive normal form, we obtain four problems

¬ A(s) ∧ ¬ B(s)
¬ A(s) ∧ D(s,s')
¬ B(s) ∧ C(s,s')
C(s,s') ∧ D(s,s')

corresponding to the cases in which the 'paths' executed are neither, D, C and both. A model in each case will be an execution of the relevant path.

Invariant preservation. The claim that an operation $OP(s,s')$ preserves an invariant *INV* gives rise to a problem of the form

OP(s,s') ∧ INV(s) ∧ ¬INV(s')

whose models are counterexamples: executions of the operation from states that satisfy the invariant to states that do not. If the invariant involves a conjunction or the operation involves a disjunction, a set of problems must be generated. Which problem has a model will then indicate which path of the operation violates which conjunct of the invariant.

Refinement. Given a concrete operation $OPc(sc,sc')$, an abstract operation $OPa(sa,sa')$, and an abstraction function from concrete states sc to abstract states sa, represented as a formula $A(sc,sa)$, the claim that *OPc* refines *OPa* gives rise to the problem

$OPc(sc,sc') \wedge A(sc,sa) \wedge A(sc',sa') \wedge \neg OPa(sa,sa')$

whose solutions are transitions of *OPc* that violate the behaviour described in *OPa*. Again, if there are disjunctions in *OPc* or conjunctions in *OPa*, there will be a set of problems rather than a single problem.

5 Analysis Scheme

Since the language is undecidable, there is no algorithm that can determine whether a model exists. So instead, we conduct a search for a model within finite bounds. The user specifies a scope

$scope: Type \rightarrow Nat$

that associates a natural number with each primitive type. A model will only be considered if it can be constructed with no more atoms in each primitive type than the scope permits. Usually, the same bound is assigned to all types, and we often talk informally of a 'scope of 3' to mean that each type has at most 3 atoms. In practice, the scope required to find a design flaw is often surprisingly small. In almost all cases we have studied, a scope that assigns no more than 3 atoms to each type suffices; in a few cases, a scope of 5 or 6 is necessary.

The idea behind the analysis is simple. For a finite scope, the possible values of a relation can be described as bit matrices, and the set of all possible values of a relation can be described by a matrix of boolean variables. Viewing a relation as a graph, each boolean variable is associated with a potential edge, and its truth value determines whether the edge is present or missing from the relation. Any property of relations can then be expressed using boolean variables alone, and a relational formula may be translated to a purely boolean one.

Having obtained the boolean formula, we apply an off-the-shelf boolean solver to find a model. As a byproduct of translation, we generate a mapping between relational and boolean variables. This mapping is applied in reverse to the boolean solution to reconstitute the relational values.

Since the formula may be enormous, backtracking solvers seem to be inadequate, and we have resorted to a hill-climbing solver. The solver is incomplete; it is possible get stuck at a local minimum, and thus fail to find a model within a given scope, even when one exists. In our experiments, this has been very rare; in only one case have we had to run the solver for more than 5 minutes to find a model.

Oddly, the bottleneck in our scheme appears to be not in solving the boolean formula but in generating it. Our initial attempts to generate a boolean formula exhausted memory on even toy problems. Most of our effort has been focused on techniques for manipulating the relational formula to minimize the blowup on translation. We have put less effort into techniques to minimize the boolean formula during its construction; these would likely bring further improvements in performance.

5.1 Sample Translation

To see how this works, consider translating the formula

$p, q: T \leftrightarrow T$

$p \neq \{\} \wedge q \neq \{\} \wedge p ; q = \{\}$

for a scope of 2. Start by introducing a variable $p[i,j]$ to represent the edge from the ith atom in the domain of p to the jth atom in the range. The first subformula says that p has at least one edge; it becomes the boolean formula

$p\,[0,0] \vee p\,[0,1] \vee p\,[1,0] \vee p\,[1,1]$

Likewise, the second subformula becomes

$q\,[0,0] \vee q\,[0,1] \vee q\,[1,0] \vee q\,[1,1]$

The third subformula asserts that there is no atom with an incoming edge in p and outgoing edge in q. As a boolean formula, this assertion becomes

$\neg (p\,[0,0] \wedge q\,[0,0])$
$\neg (p\,[0,0] \wedge q\,[0,1])$
$\neg (p\,[0,1] \wedge q\,[1,0])$
$\neg (p\,[0,1] \wedge q\,[1,1])$
$\neg (p\,[1,0] \wedge q\,[0,0])$
$\neg (p\,[1,0] \wedge q\,[0,1])$
$\neg (p\,[1,1] \wedge q\,[1,0])$
$\neg (p\,[1,1] \wedge q\,[1,1])$

The problem is now the conjunction of these 10 formulas. One solution is

p[0,0] true, q[1,1] true, all other p[i,j] and q[i,j] false

If T is represented by the atoms $\{t0,t1\}$, this corresponds to the relational model

$p = \{t0 \mapsto t0\}$
$q = \{t1 \mapsto t1\}$

In fact, the translation is not performed exactly like this. Rather than translating the third subformula in one fell swoop, we construct a translation of the product $p ; q$, as a matrix of boolean formulas. The formula in the top left-hand position of this matrix, for example, is

$(p\,[0,0] \wedge q\,[0,1]) \vee (p\,[0,1] \wedge q\,[1,1])$

Asserting that the product is empty now amounts to asserting the falsehood of each such boolean formula. This compositional approach, described in the next section, is clean but not viable if applied naively.

5.2 Compositional Translation

The translation from relational to boolean form can be implemented compositionally, as shown in Figure 4. Each relational variable becomes a matrix of boolean variables; each expression becomes a matrix of boolean formulas; and each relational formula becomes a boolean formula.

The dimensions of the matrix created for a variable are determined by the scope. For a variable $v : S \leftrightarrow T$, the matrix has $scope(S)$ rows and $scope(T)$ columns.

Suppose we have translated an expression p into a matrix $X[p]$, and a term q into a matrix $X[q]$. The element in the ith row and the jth column of $X[p]$, which we write $X[p]_{ij}$, is a boolean formula that is interpreted as being true when p maps the ith element of its left-hand type to the jth element of its right-hand type. The translation of the term $p \cap q$ is given by

$X[p \cap q]_{ij} = X[p]_{ij} \wedge X[q]_{ij}$

since an edge is in the intersection of two relations when it is in both relations. Other operators are treated similarly.

Elementary relational formulas are translated into boolean formulas. The rule

$B[p \subseteq q] = \wedge_{ij} X[p]_{ij} \Rightarrow X[q]_{ij}$

for example, states that a relation p is a subset of a relation q when there is an edge in q corresponding to every edge in p.

$B : formula \to booleanFormula$
$X : expr \to booleanFormulaMatrix$

$B\,[\neg f] = \neg\, B\,[f]$
$B\,[p \subseteq q] = \wedge_i \wedge_j (X[p]_{ij} \Rightarrow X[q]_{ij}$
$B\,[p = q] = \wedge_i \wedge_j (X[p]_{ij} \Leftrightarrow X[q]_{ij}$
$X\,[Id]_{ij} = (i = j)$
$X\,[0]_{ij} = false$
$X\,[Un]_{ij} = true$
$X\,[p \cup q]_{ij} = X[p]_{ij} \vee X[q]_{ij}$
$X\,[p \cap q]_{ij} = X[p]_{ij} \wedge X[q]_{ij}$
$X\,[p \setminus q]_{ij} = X[p]_{ij} \wedge \neg X[q]_{ij}$
$X\,[p ; q]_{ij} = \vee_k X[p]_{ik} \wedge X[q]_{kj}$
$X\,[p\sim]_{ij} = X[p]_{ji}$
$X\,[s <: p]_{ij} = X[p]_{ij} \wedge X[s]_{i0}$
$X\,[dom\ p]_{i0} = \vee_k X[p]_{ik}$
$X\,[v] = a\ matrix\ of\ fresh\ boolean\ vars$

Figure 4: Translation from relational to boolean form

The boolean formula for a problem is then just the conjunction of the boolean formulas for the collection of relational formulas.

There is no translation rule for closure; instead, closure is replaced by union and composition. The closure of p is given by the series

$$p \cup (p ; p) \cup (p ; p ; p) \cup \ldots$$

which, for a scope of k, converges in at most k steps. To see why, note that the closure essentially computes whether there is a path between a given pair of nodes in the graph representing p; if there is a path, there is one with at most k edges. A more efficient translation uses the technique known as iterative squaring [BC+92]:

$$p_0 = p$$
$$p_{i+1} = (p_i ; p_i) \cup p_i$$

For a scope of k, this series converges in log k steps. The unwinding is purely syntactic; it simply generates from a closure expression an expression involving union and composition that is equivalent for the given scope. This expression cannot usually be translated directly, but will be reduced by the method described in section 5.4 below.

5.3 CNF Representation

The boolean formulas are represented in conjunctive normal form (CNF), that is, as a conjunction of disjunctions of literals. Two factors motivate the choice of CNF. First, most off-the-shelf boolean solvers require their input in CNF. Second, CNF is well suited to the typical structure of relational problems: note how, in Figure 4, the translation of an elementary relational formula *conjoins* the element formulas (of which there are k^2 for a scope of k), and the translation of a problem conjoins the translations of its constituent formulas.

Unfortunately, direct translation of a relational formula into CNF is infeasible. To see why, consider how boolean operations are performed in CNF. Viewing a CNF formula as a set of clauses, each being a set of literals, we can define the logical operators for CNF as follows:

$$CNF\,[F \wedge G] = CNF\,[F] \cup CNF\,[G]$$
$$CNF\,[F \vee G] = \{\, f \cup g \mid f \in CNF\,[F] \wedge g \in CNF\,[G]\}$$
$$CNF\,[\neg F] = \{c \mid Literals(c) \subseteq NegLiterals(F) \wedge\ \forall f \in CNF\,[F] \,.\, \exists!\, x.\, x \in f \wedge \neg x \in c\ \}$$

Conjunction is simply union of clause sets. Disjunction, though, is cross product; the size of a disjunction is, in the worst case, the product of the sizes of the original formulas. The negation of a formula F is the set of all clauses that can be obtained by taking one literal from each clause of F and negating it. In the worst case, the size of the negation is exponential in the number of clauses of the original formula. (To exclude bogus literals from each clause c, its literals are constrained to be drawn from the set of negations of literals already appearing in F).

Because of their heavy cost, disjunction and negation of complex formulas must be avoided. Translating a problem as given is usually hopeless; even small problems easily exhaust memory. To minimize CNF explosion, the problem is manipulated prior to translation in two ways: introduction of temporary variables and symmetry breaking. When a negation must be performed, the cost can often be reduced by a simple technique we call 'negation caching'.

5.4 Temporary Variable Introduction

Relational union translates directly to disjunction. These disjunctions, however, are elementwise, and only become intolerable when attempting to translate a union of complex relational expressions. Relational compositions are worse, because they involve k disjunctions for a scope of k for each element of the result. Equalities are also problematic; the formula $p = e$, where p is a variable and e is a relational expression, creates terms of the form $[e]_{ij} \Rightarrow [p]_{ij}$ which involves a negation of $[e]_{ij}$.

As a concrete example of these problems, consider following the rules of Figure 4 to translate the expression $(p ; q) ; r$ for a scope of 3, where p, q and r are variables. Each element in the translation of $(p ; q)$ is a disjunction of 3 formulas, each of which contains 2 clauses of one literal each; in CNF, this element therefore has $2^3 = 8$ clauses. In the matrix representing the entire expression, each element will be a disjunction of 3 formulas, each with 9 clauses, which gives a formula of $9^3 = 729$ clauses. Now suppose that this expression sits on the right-hand side of an equality: $v = p ; q ; r$. Translating the equality will require negating each element of the translation of the expression, resulting in a formula of perhaps 2^{729} clauses.

To avoid these problems, complex expressions are broken up by the introduction of temporary variables. The formula

$$p = (q \cap r) \cup (s \cap t)$$

for example, is broken into 3 formulas:

$$p = temp_0 \cup temp_1$$
$$temp_0 = (q \cap r)$$
$$temp_1 = (s \cap t)$$

The introduction of temporaries has the additional benefit of common subexpression elimination: multiple occurrences of a single expression are replaced by the same temporary variable.

Currently, we use some simple heuristics to determine when to introduce variables: for all subexpressions of a relational composition, unless trivial (a variable or its transpose); for all subexpressions of a union that involve composition or intersection;

for one side of an equality, and the left-hand side of an inequality, when both sides are compound expressions; for both sides of a negated equality or inequality. We have not shown these to be optimal, but they seem to work well in practice.

5.5 Symmetry Breaking

We have shown previously [JDJ96, JJD98] that the models of relational formulas are *permutation invariant*: given a model of a formula, the assignment that results from permuting the atoms of the underlying universe will also be a model. For example, the formula discussed above

$p, q : T \leftrightarrow T$

$p \neq \{\} \wedge q \neq \{\} \wedge p ; q = \{\}$

with model

$p = \{t0 \mapsto t0\}$

$q = \{t1 \mapsto t1\}$

must also have the model

$p = \{t1 \mapsto t1\}$

$q = \{t0 \mapsto t0\}$

since the atoms *t0* and *t1* are uninterpreted, and exchanging them is insignificant.

Consider now a model of the formula

$p : S \leftrightarrow T$

$F \wedge (p \neq 0)$

where F is an arbitrary subformula, and p and q are variables. The models of F form a set that can be partitioned into equivalence classes by such permutations. Suppose S is represented by the set of atoms {*s0*,*s1*}, and T by the set {*t0*,*t1*}. Then the models of $(p \neq 0)$ are all the non-empty assignments to p, such as

$p = \{s0 \mapsto t0, s1 \mapsto t0\}$

Clearly, for any edge we might pick, there is an assignment to p including that edge that is a model of $(p \neq 0)$. But more interestingly, there must also be a model of the formula as a whole in which p contains that edge. Any model of the formula must have p contain some edge, and by permuting the atoms of S and T, we can effectively choose the edge.

This observation can be exploited in the translation. Rather than translating $(p \neq 0)$ as the disjunction

$\vee_{ij} X[p]_{ij}$

we can break the symmetry, and pick an element of the relation to be true; picking the top left-hand element gives

$X[p]_{00}$

for example.

For this case, the effect is not dramatic: $(p \neq 0)$ can be translated anyway into a single clause. But a negated equality $\neg (p = q)$ would be translated to $\vee_{ij} X[p]_{ij} \neq X[q]_{ij}$, which for a scope of k is a disjunction of k^2 terms. Even when p and q are variables, constructing this formula is infeasible for a scope of 5 or more. We therefore break symmetry on these troublesome formulas, in this case allowing us to replace the disjunction

$\vee_{ij} \neg X[p]_{ij} \Leftrightarrow X[q]_{ij}$

whose size varies, in the worst case, with the exponent of the square of the scope, by the single term

$\neg X[p]_{00} \Leftrightarrow X[q]_{00}$

which gives a formula of a single clause!

This argument has some subtleties. If p is a homogeneous relation, its indices cannot be permuted independently. Consequently, we have to include a diagonal and an off-diagonal element. In general, more than one negated equality (or inequality) can be reduced in this way, so long as each reduction exploits permutation of a different type. We therefore rank the negated subformulas according to their severity, and then allocate symmetry breaking reductions from the most to the least severe, making sure never to break symmetry on the same type twice.

Permutation invariance holds only because the primitive types are uninterpreted. Even in the presence of interpreted types (such as integers), relational formulas are still permutation invariant over the uninterpreted types. So symmetry breaking can still be used in the presence of interpreted types, so long as it is used only to justify permutations of uninterpreted atoms.

5.6 Negation Caching

As noted above, translating an equality $p = e$, where p is a variable and e is a relational expression, creates terms of the form $X[e]_{ij} \Rightarrow X[p]_{ij}$ which involves a negation of $X[e]_{ij}$. Introduction of temporary variables creates a large number of these equalities. Worse, the expression e is often complex – precisely why it was replaced by a temporary in the first place.

Often, however, the negation of $X[e]_{ij}$ is smaller than $X[e]_{ij}$ itself. Suppose e is a composition $p ; q$ of variables. Then

$X[e]_{ij} = \vee_k X[p]_{ik} \wedge X[q]_{kj}$

which gives, for a scope of k, 2^k clauses of k literals each, whereas

$\neg[e]_{ij} = \wedge_k X[p]_{ik} \vee X[q]_{kj}$

which gives k clauses of 2 literals each. Since CNF is not canonical, computing $X[e]_{ij}$ and then negating it will not result in this small formula, but on the contrary produces an even larger formula.This effect is compounded by redundancy in the CNF representation. A clause is redundant if one of its subsets also appears as a clause. By using a trie representation of CNF [ZS94], we eliminate some of these redundant clauses – namely those that are lexical prefixes of clauses already present – but not all of them.

We therefore employ the following strategy. When the element formulas for a composition are computed, their duals are also computed and cached. The representation of a formula thus includes two formulas: the formula itself, and optionally its negation. When the negation of a formula is required, the cached formula, if present, is used directly; if absent, the negation is computed and cached. This simple idea has a very dramatic effect on performance.

5.7 Solving

We have experimented with two boolean solvers: a trie-based version [ZS94] of Davis-Putnam [DP60], which we implemented as part of Nitpick in Java, and WalkSAT [SKC94], a stochastic solver developed at Bell Labs, which is run as a native method in C.

WalkSAT works by local search, and is thus incomplete: it may fail to find a model of the boolean formula when one exists. This has rarely been a problem in practice; almost always, a model, if one exists, is found on the first try. Davis-Putnam works only on small problems; when there are more than a few hundred boolean variables, it does not terminate in a reasonable time. It is, however, deterministic and thus complete: it will eventually find a model if one exists. We have also experi-

mented briefly with GRASP [SS96], a more modern deterministic algorithm, but it seems unable to match the performance of WalkSAT on practical examples.

WalkSAT uses an intriguingly simple strategy. Its metric is the number of clauses in the CNF formula that are satisfied; at each step, it attempts to change the assignment by flipping the value of a boolean variable so as to increase the metric. The basic algorithm is as follows:

```
for i = 1 to MAX_TRIES {
   T = randomly generated assignment
      for j = 1 to MAX_FLIPS
         if T satisfies formula return T
         v = the variable whose flipping gives largest increase
                                                in #clauses satisfied
         T = T with v flipped
return "no assignment found"
```

WalkSAT adds an element of random walk to the search. At each step, it tosses a coin: with probability p, it follows the standard scheme; with probability $(1-p)$, it picks a variable from an unsatisfied clause and flips it.

Some performance results for a variety of analyses are shown in Table 2. *Phone* and *Finder* are toy specifications. *Style* is a specification of the paragraph style hierarchy of Microsoft Word, constructed as a class project in the Masters of Software Engineering degree at Carnegie Mellon; it is described in detail in [JD96b]. *Allocate* is a simplified fragment of a railway interlocking design written in Z by Praxis UK PLC. *Mobile IP* is a description of a draft version of a mobile internet protocol for IPv6, written by an undergraduate at Carnegie Mellon [JNW97, Ng97]. The descriptions are reproduced in full in the technical report [Jac97]; they are all in NP, Nitpick's source language, and were translated automatically into the intermediate language. For each example, a representative claim was analyzed; in all cases, the claim has a counterexample in the smallest scope.

The column marked *Problems* gives the number of problems generated by the claim; *Formulas* gives the maximum number of relational formulas in each problem (after variable introduction). The columns marked *Vars* and *Clauses* give the number of boolean variables and clauses in the emitted formula; *Translate* and *Solve* give the times to perform the translation and solve the boolean formula. All formulas were solved using WalkSAT, except when DP appears in parentheses after the solving time; for these, the Davis-Putnam solver was sufficient (ie, found a solution in less than 30s). The final column gives the time taken by the previous, explicit version of Nitpick (as described in [JDJ96, JDJ98]).

The formulas generated are large, but not necessarily hard to solve. Formulas can be classified according to the average number of literals per clause, and the ratio of the number of variables to the number of clauses. It seems that hardest classes of formulas are those for which the probability of satisfiability is about 50%; for clauses of 3 literals, this corresponds to a ratio of 4.3 clauses/variable [MSL92]. Our formulas often have 10 or more literals/clause; we have not yet studied whether they fall into the hard or easy category.

Timings were obtained on a modestly equipped machine; the two versions of Nitpick actually run on different platforms, but are roughly comparable. The entry ?? indicates that a model was not found in a reasonable time; we set a bound of one hour. The explicit checker did find a model for *Mobile IP*, but for a smaller scope in which different bounds were associated with different types.

As can be seen, the new method outperforms the old method in almost all cases. (One reviewer of this paper claimed that the old method is 'clearly hopeless'. Remarkably, it isn't: not because it scales well – which it does not – but because counterexamples can often be found in tiny scopes. In the Mobile IP case study, for example, one message and two hosts was sufficient to expose a serious flaw.) Moreover, the new method is much simpler to implement; our entire prototype is only about 6000 lines of Java code, of which almost half is concerned with front-end functionality (parsing and static checks).

Negation caching has a dramatic effect on all the examples, often reducing the formula size, and translation time, by an order of magnitude. Symmetry has a marked effect only on examples involving negated equalities. On *Style*, for example, it reduces the number of clauses from 53,495 to 4,385 for a scope of 4, and translation time from 17s to 2s. Detailed performance figures appear in the technical report [Jac97].

6 Conclusions

We have experimented before with a boolean analysis, representing boolean formulas not in CNF but with ordered binary decision diagrams (BDDs) [DJJ96]. Although that method performed excellently on small examples, it did not scale well. BDDs, unlike CNF, are a canonical representation. In model checking, this is essential – it allows fixpoints to be detected – but in our setting there is little benefit, and the extra cost is unwarranted.

The remarkable success of the stochastic solver GSAT [SLM92] and its descendants has made boolean translation attractive in other domains too. The closest application to ours is in planning [KS96, EMW97]. The planning problem is technically closer to the model checking problem than to our problem: namely finding a sequence of transitions in a state machine that leads to a state satisfying a given property (in planning, the goal, and in model checking, the negation of the invariant). The focus on data structures rather than on transition sequences makes our scheme rather different.

Example	Problems	Formulas	Scope	Vars	Clauses	Translate	Solve	Explicit
Phone	1	10	3	66	307	0s	0s (DP)	0s
			4	112	842	0s	0s (DP)	0.5s
			5	170	2159	0s	0s (DP)	35s
			6	240	5413	1s	2s (DP)	5m
Finder	1	47	3	273	1364	0s	1s (DP)	0.5s
			4	464	3500	0s	0s	1s
			5	705	8483	0s	3s	11s
			6	996	20779	9s	2m,33s	2m,12s
Style	12	71	3	408	1864	1s	6s (DP)	1s
			4	704	4385	2s	0s	11s
			5	1080	9977	6s	5s	11m,45s
Allocate	1	41	3	192	522	0s	1s (DP)	10s
			4	316	931	0s	0s	??
			5	470	1473	0s	0s	??
			10	1690	6693	6s	0s	??
Mobile IP	1	68	3	438	2436	0s	0s	??
			4	760	6548	3s	0s	??
			5	1170	16536	8s	0s	??

Table 2: Performance of new method versus old method

The scheme described in this paper is novel in two respects. The novelty of the intermediate language is not in its operators, but in its typing. By representing sets as relations with *Unit* as their right-hand type, we are able to treat sets and relations uniformly, making the language and analyzer simpler, without incurring any overhead in the boolean translation. *Unit* always has one atom, so a set is translated, as expected, into a $k \times 1$ matrix – namely a vector.

The novelty of the translation is in preempting the blowup by manipulating the relational formula prior to translation. The treatment of closure is particularly simple and effective (although it does make the manipulation of the relational formula dependent on the scope). No doubt further improvements could be obtained by optimizing the manipulations of the boolean formulas themselves – how they are represented and combined.

The notion of scope is perhaps unremarkable, but seems to be the lynchpin to making progress in automating the analysis of software specifications. Nevertheless, the incompleteness resulting from an ad hoc bound on the search, compounded by the incompleteness of the solver, is of course undesirable. Sometimes the user will conclude that a property holds only because the search was conducted in too small a scope, or was not given enough time to find a counterexample.

We are investigating ways to mitigate this problem. The analyzer might, for example, provide an estimate of the probability that a model exists but has not been found; from this, the user could determine for how long to run the solver. In a safety-critical application, it would be appropriate to construct an abstraction that establishes that a search within a given scope suffices to show correctness for all scopes, but for mainstream software development, the effort is not worthwhile.

Researchers in formal methods have devoted almost all their attention to analysis schemes that gain completeness at the expense of automation. It seems to us that fast and fully automatic analyses whose results are correct with high probability are more likely to appeal to practitioners, and offer exciting new research challenges.

Acknowledgments

We are very grateful to Greg Nelson, who suggested dusting off our BDD-based checker and trying again with Davis Putnam; to Mihai Badoiu, Somesh Jha, Craig Damon and Jeannette Wing for helpful discussions; and to the National Science Foundation for sponsoring this work in part (under grant CCR-9523972).

References

[BC+92] 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.

[DJJ96] Craig A. Damon, Daniel Jackson and Somesh Jha. Checking Relational Specifications with Binary Decision Diagrams. *Proc. 4th ACM SIGSOFT Conf. on Foundations of Software Engineering*, San Francisco, CA, October 1996, pp.70–80.

[DP60] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM*, Vol. 7, pp. 202–215, 1960.

[EMW97] 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.

[Jac97] Daniel Jackson. *Boolean Compilation of Relational Specifications*. Technical Report MIT-LCS-TR-

735. Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA. December 1997. Available at .

[JD96a] Daniel Jackson and Craig A. Damon. *Nitpick Reference Manual.* CMU-CS-96-109. School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, January 1996.

[JD96b] 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.

[JDJ96] Daniel Jackson, Craig A. Damon and Somesh Jha. Faster Checking of Software Specifications. *Proc. ACM Conf. on Principles of Programming Languages*, St. Petersburg Beach, FL, January 1996, pp. 79–90.

[JJD98] 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.

[JNW97] Daniel Jackson, Yuchung Ng and Jeannette Wing. A Nitpick Analysis of IPv6. Submitted to *Formal Aspects of Computing.*

[KS96] 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.

[MSL92] David Mitchell, Bart Selman and Hector Levesque. Hard and easy distributions of SAT problems. *Proc. 10th National Conference on Artificial Intelligence (AAAI-92),* San Jose, CA, July 1992, pp. 459–465.

[Ng97] Yu-Chung Ng. *A Nitpick Specification of IPv6*. Senior Honor's Thesis, Computer Science Department, Carnegie Mellon University, May 1997.

[Sch79] Wolfgang Schoenfeld. An undecidability result for relational algebras. *Journal of Symbolic Logic*, 44(1), March 1979.

[SKC94] Bart Selman, Henry Kautz and Bram Cohen. Noise strategies for improving local search. *Proc. AAAI-94*, pp. 337–343, 1994.

[SLM92] Bart Selman, Hector Levesque and David Mitchell. A new method for solving hard satisfiability problems. *Proc. 10th National Conference on Artificial Intelligence (AAAI-92),* San Jose, CA, July 1992.

[Spi92] J. Michael Spivey. *The Z Notation: A Reference Manual*. Second ed, Prentice Hall, 1992.

[SS93] Gunther Schmidt and Thomas Stroehlein. *Relations and Graphs*. EATCS Monographs in Theoretical Computer Science, Springer-Verlag, 1993.

[SS96] 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.

[Tar41] Alfred Tarski. On the calculus of relations. *Journal of Symbolic Logic*, 6(1941), pp. 73–89.

[TG87] Alfred Tarski and Steven Givant. *A Formalization of Set Theory Without Variables*. American Mathematical Society, Colloquium Publications, Volume 41, 1987.

[ZS94] 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.