Boolean Compilation of Relational Specifications

Daniel Jackson
MIT Lab for Computer Science
January 1998

MIT-LCS-TR-735

Abstract

A new method for analyzing relational specifications is described. A property to be checked is cast as a relational formula, which, if the property holds, has no finite models. The relational formula is translated into a boolean formula that has a model for every model of the relational formula within some finite scope. Errors in specifications can usually be demonstrated with small counterexamples, so a small scope often suffices. The boolean formula is solved by an off-the-shelf satisfier.

The satisfier requires that the boolean formula be in conjunctive normal form (CNF). A naïve translation to CNF fails (by exhausting memory) for realistic specifications. This paper presents a preliminary design of a compiler that overcomes this problem, by exploiting typical features of the relational formulae that arise in practice. Initial experiments suggest that this method scales more readily than existing approaches and will be able to find more errors, in larger specifications.

Keywords: Software design analysis, formal specification, object model, Z, relational calculus, model finding, boolean satisfaction, WalkSAT, Nitpick.

1 Introduction

Two views are central in software design. The *event view* is concerned with which events may occur during execution of the system, and their ordering patterns. The *object configuration view* is concerned with which objects exist and their relationships to one another. Even though these views are not disjoint – as events occur, configurations change – it is often useful to examine them independently.

Most formal specification languages have been developed with primarily one of these views in mind: the event view for CSP, Statecharts and Lotos; the object configuration view for Z, VDM and Larch. Object oriented methods, such as OMT, Fusion and Syntropy, typically advocate the construction of a model for each view: the “dynamic model” for the event view (typically expressed as a Statechart) and the “static model” for the object configuration view (typically expressed as an entity-relationship diagram).

The analysis of event views is well supported by tools: in addition to simulators, there are now model checkers that can exhaustively analyze huge state spaces. Object configuration views, in contrast, are poorly supported. Existing automated tools offer only shallow syntactic checks of the description, and rarely expose problems in the software design itself.

This disparity is particularly remarkable given the centrality of object configurations. The event view is sometimes dominant (for example, in the design of protocols and process control systems), but usually plays a subsidiary role. Take a glance at an influential book on design patterns [G+95], and note the proportion of diagrams that describe object configurations.

Nitpick is an attempt to redress this imbalance. Given a constraint describing a set of object configurations, Nitpick can generate instances satisfying the constraint. It can check that declared constraints have intended consequences, by searching for instances that satisfy one constraint but not another. Nitpick also accepts descriptions of operations that describe transitions from one configuration to another, and can simulate these, and check a variety of their properties, such as whether an operation preserves a constraint.

The current implementation of Nitpick works by explicit enumeration. Using a variety of mechanisms to prune the search tree, it can analyze enormous configuration spaces. The tool has been used successfully to find flaws in a number of small designs, including a published draft of a mobile internet routing protocol [Ng97, JNW97]. Even small designs, however, tend to have huge configuration spaces. Indeed, this is perhaps why analysis of object configurations has not attracted the same attention as the analysis of state machines. A relation over a domain of 3 elements has 512 possible values; adding a single such relation to a specification thus increases the space by 3 orders of magnitude.

This paper describes a new method for analyzing descriptions of object configurations. Section 2 presents a toy example that conveys the flavour of the specification notation; Section 3 compares the notation to popular notations. Section 4 demonstrates a Nitpick analysis of the example.

Section 5 explains how simulation and checking can both be reduced to model finding. Section 6 articulates the hypothesis that most errors can be illustrated with small counterexamples.

Section 7 gives the intutions behind the new method. Sections 7 to 16 are the core of the paper; they present the structure of the compiler: the sequence of representations and their respective transformations.

Section 17 gives some performance results that compare the new method to our previous explicit method. Section 18 and 19 discuss related work and future plans.

2 Relational Notation: NP

Our notation, NP, is roughly a subset of Z [Spi92]. A reference manual [JD96b] describes the notation in detail. Here, we explain its basic elements with a toy example.

```
[Ph, Num]

Switch = [
  called : Ph <-> Num
  const net : Num -> Ph
  conns : Ph <-> Ph
  |
  conns = called ; net ]

Join (p : Ph ; n : Num) = [
  Switch
 |
  p in dom called
  n not in ran called
  called' = called U {p -> n} ]

JoinOK (p : Ph ; n : Num) :: [
   Switch | Join (p, n) and inj conns => inj conns' ]
```

Figure 1: NP specification of ESS

The Extremely Simple Switch (ESS) connects telephones by maintaining two relationships: *called*, which maps phones to numbers, and represents ongoing phone calls, and *net*, a constant relationship that maps numbers to the phones they name. When (*p1*, *n*) belongs to *called*, phone *p1* has an active connection to the number *n*; when (*n*, *p2*) belongs to *net*, *p2* is the phone specified by the number *n*. The composition of *called* and *net* is a relation *conns* that associates phones that are connected; it would include, in this case, the pair (*p1*, *p2*).

Part of a (faulty) specification for ESS is shown in Figure 1. The first line declares *Ph* and *Num* to be primitive types. The schema *Switch* that follows declares the three state components with an accompanying constraint that serves to define *conns* – redundant and introduced only for the convenience of later expressing certain properties – in terms of the other two.

The form of the arrow in a declaration indicates the kind of relation. Thus *called* is an arbitrary relation, allowing the modelling of conference calls in which a phone has called more than one number at a time, but *net* is a function.

Switch defines a set of configurations. Entity relationship diagrams can be used for the same purpose, but are not as expressive. The diagram of Figure 2 corresponds to the declarations of *Switch*, but does not express the constraint amongst the three relations.

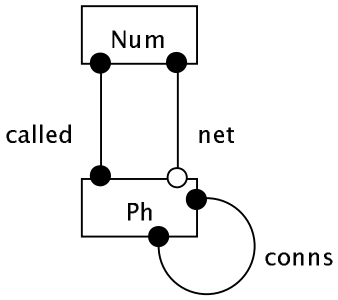

Figure 2: ER diagram corresponding to Switch schema of Figure 1

The schema *Join* specifies an operation in which a new party is added to a call. The parameter *p* is the phone at which the join is performed; it is constrained to be already making a call. The parameter *n* is the new number being called, and is constrained not to be already called. The effect of the operation is given by the last constraint. Here, the primed instance of the variable *called* denotes the value of the state component after the operation; it thus asserts that the operation adds the pair (*p*, *n*) to the *called* relation. The *net* relation, having been declared in the imported schema *Switch* to be constant, does not change; it is as if the assertion

$$net' = net$$

were included. Beyond this, there are no implicit frame conditions. In fact, the change in *called* will generally require a corresponding change in *conns*.

The third schema, *JoinOK*, is a *claim*. It is not part of the specification proper. Rather, it asserts a property of the specification: that if the relation *conns* is injective prior to execution of a join, it will be injective afterwards too. If *conns* is not injective, there is a call involving two phones acting as callers; for billing reasons, we prefer to avoid this situation, and construct all conference calls with a single caller.

3 Why Relational Specification?

This style of specification was pioneered by the developers of the Z specification language. For readers familiar with other notations, we shall explain briefly what features of the relational notation make it, in our opinion, a more suitable notation for describing object configuration views.

The principal difference between our language, NP, and Z is that NP is first-order: its data structures are simple relations, and it does not admit, for example, relations between relations. While Z is based on set theory, NP is based on the relational calculus. Z’s extra power makes it less tractable; NP gains much in tractability for a relatively small reduction in expressiveness.

The formal specification languages VDM and Larch have much in common with Z. Their data structures, however, are not based on set theory but rather on a collection of data types each with its own axiomatization. In a sense, NP is less abstract, since all data structures are encoded as graphs. But for describing configurations, graphs are so natural that bias is rare, and more abstract types are not needed.

The input languages of model checkers, such as SMV's assignment language, Murphi's Unity-like language and SPIN's Promela tend to have weak support for data structures, and do not provide primitives for composing relations and applying functions. They also tend not to support implicit specification fully, in which the result of an operation is defined by an assertion relating the variables of pre- and post-states, rather than by an explicit assignment.

Object-oriented methods, such as UML, OMT, Fusion and Syntropy, provide graphical notations for describing *object models*. These are extremely weak, and allow only the most elementary constraints between relations (such as that one is a subset of another) to be expressed. Various textual annotation languages have been designed, most recently UML's Object Constraint Language (OCL) [IBM97], which originated in the constraints of Syntropy. OCL is very similar to NP, although it does not have a formal semantics and does not appear to be able to express transitive closure. Its syntax looks very different from NP's however; it makes heavy use of scalar quantifiers in place of relational operators, and has no structuring mechanism akin to schemas.

In practice, object models are much less powerful than relational models, since they are usually interpreted in terms of implementation constructs. So *subset*, for example, is usually construed as *subclass*. Since standard object-oriented languages do not allow objects to migrate between subclasses, this means that the members of a subset cannot change. Unlike in NP, therefore, one could not introduce a set *Callers* that is a subset of *Ph*, corresponding to the phones making calls, since this set changes over time and thus cannot be a subclass.

4 Nitpick in Action

Let's see what Nitpick can do with a specification. It basically offers two features: simulation and checking. Simulating a state schema produces instances of the state that satisfy the given constraints. Instructing Nitpick to simulate *Switch* in Figure 1, for example, will cause it to output the rather uninteresting case:

```
called: {}
net: {}
conns: {}
```

Simulating an operation schema produces sample transitions. For *Join*, Nitpick will produce

```
p: P2
n: N2
called: {P2->N1}
net: {}
conns: {}
called': {P2->N1,P2->N2}
conns': {}
```

exposing a simple flaw: we have allowed active calls to numbers that are not associated with phones.

Checking a claim also causes instances to be generated. The claim *JoinOK* is about an operation, and the instances will thus be transitions. These instances, however, are counterexamples: they correspond to cases that show the property to be invalid. For *JoinOK*, Nitpick will attempt to find a transition of *Join* from a state in which *conns* is injective to a state in which it is not, such as

```
p: P2
n: N2
called: {P1->N1,P2->N0}
net: {N1->P2,N2->P2}
conns: {P1->P2}
called': {P1->N1,P2->N0,P2->N2}
conns': {P1->P2,P2->P2}
```

This counterexample demonstrates (at least) two flaws with the specification. In the pre-state, *called* maps a phone to a number that is not in the domain of *net*; that is, there is an active call to a number not associated with a phone. The *net* function is not injective – there are two numbers mapped to the same phone; either this must be ruled out, or the definition of the operation must take it into account.

5 Specification Analysis as Model Finding

Both simulation and checking amount to the same problem: solving a relational formula. Solving a formula means finding an assignment to its variables that makes the formula true; such an assignment is a *model* of the formula. The formula is extracted from the specification by simply undoing the syntactic shorthands; from *JoinOK*, for example, Nitpick obtains the formula

```
conns = called ; net
and  conns’ = called’ ; net
and  p in dom called
and  n not in ran called
and  called’ = called U {p -> n}
and  inj conns
and  not inj conns’
```

A couple of features of this formula are worth noting. First, note that in translating a claim, the formula is negated: for a claim, an assignment for which the formula is false is required, namely a model of the formula's negation. Second, since a state invariant constrains the states both before and after an operation, the inclusion of the schema *Switch* brings two subformulae: one on the unprimed variables, and one on the primed variables. Third, observe the structure of the formula as a whole: a conjunction of (mostly positive) elementary subformulae. This is typical, and our method exploits it.

The free variables of such a formula may denote scalars, sets or relations. A set of type declarations, not shown above, is associated with each formula. For example, the variable *p* is constrained to be a scalar of type *Ph*. In constructing an assignment, therefore, we must first choose sets of values for the primitive types. Our types are uninterpreted, so we use symbolic names for the elements: *p1*, *p2*, *p3* for *Ph*, say. Given this set of 3 values for *Ph*, the variable *p* can take on 3 possible values. The variable *conns* can take on 512 values: there are 9 possible connections from a phone to a phone, each of which is present or absent.

6 The Small Scope Hypothesis

Every relational formula has infinitely many possible assignments. To make a finite search possible, we place an artificial bound by limiting the number of atoms in each type. A scope *S* is a mapping from the type names of a specification to natural numbers; $S(t)$ is interpreted as a bound on the number of atoms in the type *t*. We say than an assignment is within a scope, or that a scope admits an assignment, when the values of the variables in the assignment can be constructed using no more atoms than the scope allows. In our example, to consider only assignments involving at most 3 phones and 2 numbers, we would choose $S(Ph) = 3$ and $S(Num) = 2$. We shall say that a search has a scope of *S* when it considers only assignments within *S*. Sometimes we shall refer loosely to a "scope of 3", which simply means that $S(t) = 3$ for every type *t*.

The problem of finding models of relational formulas is highly intractable, since the number of assignments grows so rapidly with both the scope and the number of variables. For a scope of *k*, a relation can have k^2 edges and thus 2to the power k^2 values. The size of the space is the product of the number of values of each variable, so for *v* relational variables, there may be 2 to the power k^2v assignments. Increasing the scope from *k* to $k+1$ thus increases the space by 2^{2kv}.

It therefore seems very unlikely that a method based on search will be able to handle large scopes. This, perhaps, is why research on analysis of languages such as Z and VDM has focused almost exclusively on syntactic and not semantic methods.

The hypothesis underlying Nitpick is a controversial one. It is that, in practice, small scopes suffice. In other words, most errors can be demonstrated by counterexamples within a small scope. This is a purely empirical hypothesis, since the relevant distribution of errors cannot be described mathematically: it is determined by the specifications people write.

Our hope is that successful use of the Nitpick tool will justify the hypothesis. There is some evidence already for its plausibility. In our experience with Nitpick to date, we have not gained further information by increasing the scope beyond 6.

A similar notion of scope is implicit in the context of model checking of hardware. Although the individual state machines are usually finite, the design is frequently parameterized by the number of machines executing in parallel. This metric is analogous to scope; as the number of machines increases, the state space increases exponentially, and it is rarely possible to analyze a system involving more than a handful of machines. Fortunately, however, it seems that only small configurations are required to find errors. The celebrated analysis of the Futurebus+ cache protocol [C+95], which perhaps marked the turning point in model checking's industrial reputation, was performed for up to 8 processors and 3 buses. The reported flaws, however, could be demonstrated with counterexamples involving at most 3 processors and 2 buses.

7 Intuitions for a Boolean Method

For a finite scope, a relational formula can be translated into an equivalent boolean formula. The idea is simple. We represent relations as boolean matrices, each bit corresponding to a possible edge in the relation. A solution to the boolean formula is translated into a solution to the relational formula by constructing the relations whose edges are present or absent according to the values of the bits.

The translation can be performed compositionally. In general, a relational term is translated into a matrix of boolean formulae. For each relational variable, we introduce a matrix of boolean variables; each term is then translated by composing the translations of its subterms.

Suppose, for example, that we have translated a term *p* into a matrix [*p*], and a term *q* into a matrix [*q*]. The element in the ith row and the jth column of [*p*], which we write $[p]_{ij}$, is a boolean formula that is interpreted as being true when *p* maps the ith element of its domain type to the jth element if its range type. The translation of the term $p \cap q$ is given by

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

since an edge is in the intersection of two relations when it is in both relations. For each of the relational operators there is such a rule.

Elementary relational formulae are translated into boolean formulae. The rule

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

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*.

Sets and scalars are treated as degenerate relations. A set-valued term is translated into a vector of boolean formulas; the rules for set intersection and subset are just like the rules shown above, but with one index omitted. Scalars are represented as singleton sets; when a scalar variable is translated, a side-condition is created that asserts that exactly one of the elements of the set is present. This side condition is conjoined with the formula obtained from translation right at the end. Similar side conditions are generated to express properties of relations given in their declarations: that a relation is a function, or is total, etc.

Nitpick is thus, in essence, a compiler. Various syntax and type checks are applied to the specification. Given the user's selection of a schema to simulate or a claim to check, a relational formula is extracted. This formula is then compiled into a boolean formula, which is then presented to a boolean satisfier. By maintaining the appropriate associations between the relational variables and the boolean variables, the tool can then translate boolean assignments back into relational assignments.

Note that the major compilation step – obtaining the boolean formula – is scope dependent, since the dimensions of the boolean matrices are determined by the number of elements in the basic types. Each time the user changes the scope, the formula must be recompiled into a new boolean formula, which is then presented to the solver.

From a design point of view, it seems desirable to implement boolean formulae with an abstract data type. The translator would then call operations on the type to construct compound formulae rather than meddling directly with their syntax. The solver itself might be encapsulated within the abstract type, as an operation of type

```
BooleanFormula → BooleanAssignment.
```

To change the representation of formulae, or to choose a different solver, we would then need only to replace this abstract type. Using subclassing, and perhaps the Factory design pattern [G+95], we could even support these selections at runtime.

Were this scheme viable, our paper would be very short. Unfortunately, however, it turns out that the choice of solver is not a concern that can be easily separated from the rest of the design. Few solvers will accept a boolean formula with an arbitrary structure; the ones that appear to be most efficient for our problem require the formula to be in conjunctive normal form. A naïve translation produces such huge formulae that the translator seizes up on even tiny examples.

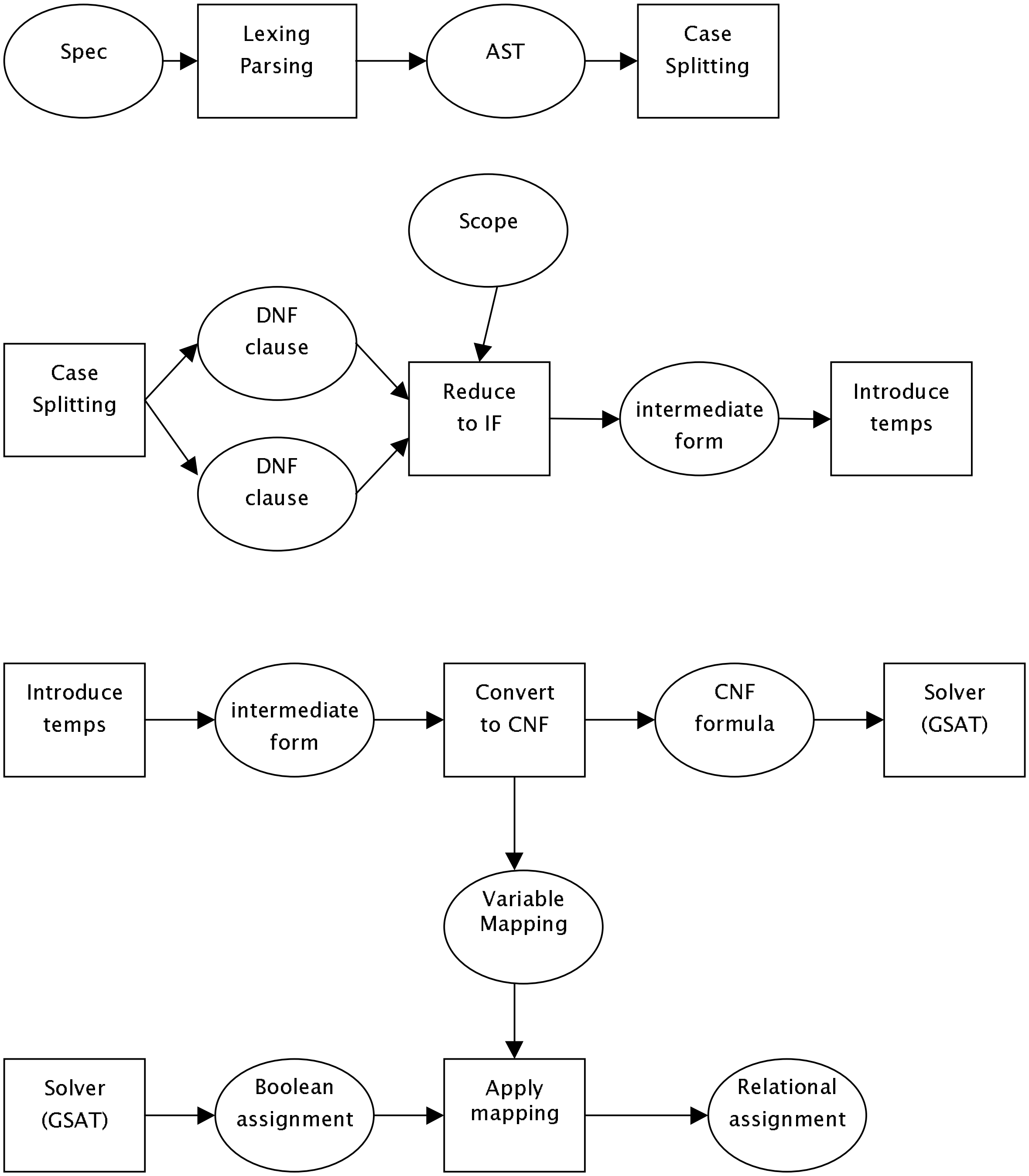

Figure 3: Architecture of Nitpick

There are three basic tricks Nitpick uses to reduce the size of the formula. The first is to decompose it into a disjunction of formulae, each itself in conjunctive normal form. Rather than attempting to construct a single huge formula, these formulae are solved independently; a solution to any one is a solution to the formula as a whole. The second is to introduce temporary variables; this is a tradeoff, since although it can reduce the formula size substantially, it can also make the formula harder to solve. The third is to find opportunities for decomposition of a formula into a disjunction in which several of the disjuncts can be discarded.

Each of these tricks exploits the structure of the relational formula. The decomposition into a disjunction follows natural cases in the specification itself. The introduction of temporaries is applied only when translating relational terms that tend to cause a blowup. Finally, the discarding of disjuncts relies on observing certain symmetries in the structure of the relational formula.

Loss of modularity is perhaps inevitable. After all, it would be foolish not to exploit the structure of the problem at hand to the greatest extent. A naïve translation yields a boolean formula in which this structure has been lost. It might, of course, be possible to recover it, but it makes more sense to exploit the structure where it is manifest: in the translation process itself.

8 Structure of the compiler

Nitpick is structured like a compiler (Figure 3). In the first phase, the specification is parsed and superficial syntactic checks are performed. Schema references are expanded, and for each schema or claim a relational formula is obtained, represented as an abstract syntax tree with symbol tables holding, for example, type information about variables. These formulas are from now on treated independently; a series of translations is applied to each.

In the second phase, each relational formula is converted to disjunctive normal form. This is a purely logical transformation that treats the elementary relational formulae as uninterpreted. The result of this phase is a set of sets of elementary relational formulae. From here on, each set is treated as an independent problem, from which a single boolean formula will be generated. These problems are translated and then solved in turn.

In the third phase, the relational subformulae themselves are compiled into a more primitive relational calculus with fewer operators. In the fourth phase, temporary variables are introduced and opportunities for symmetry breaking are identified. Finally, in the fifth phase, the boolean formula is generated, in conjunctive normal form. This formula is then presented to the solver. Either the solver fails to find a solution (usually but not necessarily because the formula is unsatisfiable), or it returns an assignment to the boolean variables that makes the boolean formula true. From this assignment, an assignment to the relational variables is easily constructed (using a mapping generated as a byproduct of the translation), and is displayed for the user.

Type checking is currently performed after the third phase, on the intermediate language. This has the advantage of greatly simplifying the type checker, but it does result in more cryptic error messages than would be obtained by analyzing the source code directly. In a production version of the tool, it would be better not to postpone type checking.

Subsequent sections elaborate on these phases in turn.

9 Relational Disjunctive Normal Form

The motivation for translating into DNF is two-fold: to decompose the checking problem, so that checking can be performed more efficiently, and to provide more useful feedback to the user.

A simplified version of the input language is given in Figure 4. We ignore the schema structuring, which affects only the design of the front end, and assume that a claim to be checked or schema to be simulated has been expanded into a relational formula. This formula language is essentially a first-order subset of Z. In the input language, the operators are written in ascii form; they appear in ascii here to make the distinction between the input language and the intermediate language (discussed later) clearer.

Disjunctive normal form (DNF), which we shall refer to as *relational* DNF (to avoid any confusion with the normal form we discuss later on boolean formulae) is a subset of the formula language, in which negation can only be applied to elementary formulae, and only formulae not containing disjunctions may be conjoined. In other words, the formula is represented as a disjunction of conjunctions of literals, where each literal is an elementary relational formula or its negation.

Putting an arbitrary formula in normal form can cause an exponential blowup. In practice, though, our specifications rarely generate formulae with more than a handful of disjuncts. Disjunction arises primarily from two idioms.

In operation specifications, disjunction arises from case splitting. Suppose for example, our *Join* operation (Figure 1) is to handle the case, not previously specified, in which the number being called has already been called. The operation might then be written:

```
Join2 (p : Ph ; n : Num) = [
  Switch
|
  p in dom called
  ((n not in ran called => called’ = called U {p -> n})
  and (n in ran called => called’ = called))
]
```

which says that if the number has already been called, the operation has no effect; otherwise the behaviour is as before. When asked to simulate this operation, Nitpick will convert the specification to DNF, obtaining two sets of elementary relational formulae, the first corresponding to one case:

```
formula ::=
  formula and formula
| formula or formula
| not formula
| formula => formula
| elemformula

elemformula ::=
expr <= expr                relation or set subset
expr < expr                 relation or set proper subset
| expr in expr              set membership
| expr = expr               relation, set or scalar equality

expr ::=
  expr U expr               union of sets or relations
| expr & expr               intersection of sets or relations
| expr \ expr               difference of sets or relations
| expr (+) expr             relational override
| expr ; expr               relational composition
| expr ~                    relational transpose
| expr . expr               relational image or function application
| expr +                    transitive closure
| expr *                    reflexive and transitive closure
| dom expr                  domain of a relation
| ran expr                  range of a relation
| expr <: expr              domain restriction
| expr :> expr              range restriction
| expr <; expr              domain subtraction
| expr ;> expr              range subtraction
| Un                        universal relation constant
| Id                        identity relation constant
| { }                       empty set or relation constant
| { expr, ..., expr }       set constructor
| { expr -> expr, ...}      relation constructor
```

Figure 4: A syntax of the formula sublanguage of NP

```
conns = called ; net
p in dom called
not n in ran called
called' = called U {p -> n}
```

and the second corresponding to the other case:

```
conns = called ; net
p in dom called
```

```
n in ran called
called' = called
```

These are subsequently treated as separate problems; the tools will attempt to find a model for the first, and then for the second. As we shall see later, avoiding top-level disjunction has a dramatic effect on performance. Moreover, the results obtained by this decomposition are exactly what the user would expect, since the two subproblems correspond to separate cases in the specification itself.

The second idiom from which disjunction arises occurs within a claim. The specifier might assert that an operation satisifes two properties. For example, we might want to say not only that *Join* preserves the invariant that *conns* is injective, but that it also preserves the invariant that no phone is both making and receiving a call at once. This latter invariant can be defined in its own schema

```
OneRole = [Switch | dom conns & ran conns = {}]
```

and we can then formulate the elaborated claim as

```
JoinOK1 (p : Ph ; n : Num) :: [
  Switch | Join (p, n) and inj conns and OneRole => inj conns' and OneRole']
```

This claim is, incidentally, weaker than the combination of the old claim (*JoinOK*) and a separate claim for maintenance of the new invariant, such as

```
JoinOK2 (p : Ph ; n : Num) :: [
  Switch | Join (p, n) and OneRole => OneRole']
```

since, by asserting that the conjunction of the invariants is maintained, its hypothesis includes both invariants; one invariant may thus contribute to the preservation of the other.

The result of DNF conversion of *JoinOK1* is two cases, one for violation of the first invariant:

```
conns = called ; net
p in dom called
not n in ran called
called' = called U {p -> n}
inj conns
dom conns & ran conns = {}
not inj conns'
```

and another for violation of the second:

```
conns = called ; net
p in dom called
not n in ran called
called' = called U {p -> n}
inj conns
dom conns & ran conns = {}
not dom conns & ran conns = {}
```

Again, checking these separately is a big performance win, and gives useful information to the user: namely which invariant is broken.

Elaborating both the claim and the operation, so that the claim is now

```
JoinOK3 (p : Ph ; n : Num) :: [
   Switch | Join2 (p, n) and inj conns and OneRole => inj conns' and OneRole']
```

would yield four analysis problems. When the checker finds a counterexample, it will indicate which branch of the operation violates the property and which property is violated.

Technically, the consequent of the implication of an apparently simple claim such as *JoinOK2* is actually a conjunction, since the schema reference *OneRole* is expanded into the conjunction

```
conns = called ; net
dom conns & ran conns = {}
```

A naïve translation would yield a case such as

```
conns = called ; net
p in dom called
not n in ran called
called' = called U {p -> n}
dom conns & ran conns = {}
not conns = called ; net
```

which is obviously not satisfiable. To avoid generating such spurious cases, the tool performs some basic simplifications during conversion to DNF, so that no conjunct contains both an elementary formula and its negation.

10 Intermediate Relational Language

The second major transformation is applied at the level of the relational operators, within the elementary subformulae. Its motivation is to simplify the code of the subsequent transformations. The result of this step, which can be viewed as an intermediate language, is a formula with the same structure, but with relational expressions expanded to compensate for a more frugal repertoire of relational operators.

The operators of this intermediate language are listed in Figure 5. They include the three set-theoretic operators (union, intersection and difference) that can be applied to sets and relations; the two quintessentially relational operators (composition and transpose); and three constants (the universal, identity, and empty relations). These, along with the equality operator for obtaining formulae from terms, together constitute a basic relational language equivalent to Tarski's relational calculus [Giv88, Tar41]. This language is as expressive as first-order predicate logic (so long as we permit definition of projection functions so that tuples can be constructed), and is thus undecidable [Sch79].

The language includes, additionally, the transitive closure operator, which extends its expressiveness, and is indispensable in practice. We also add two

formula ::=	
expr = expr	equality
| expr ⊆ expr	subset
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
| expr <: expr	domain restriction
| dom expr	domain

Figure 5: Intermediate Language

operators that add nothing in a formal sense, but which allow more efficient generation of boolean formulae. They are the domain operator, which takes a relation to the set of elements that it maps, and domain restriction, which given a set and a relation, produces the subrelation whose pairs have first elements in the set. We view these as the ‘bridging operators’ that connect relations and sets.

The language’s typing rules and semantics are given in Figures 6 and 7. *M* and *E* are the meaning functions for formulae and expressions respectively; *C* maps a type to its carrier set. Each operator is given a meaning in terms of naïve set theory. A relation is viewed as a set of pairs; the union of two relations thus becomes, for example, the union of the two sets of pairs. A set is not viewed directly as a set, but rather as a relation whose range type is the special type *Unit*, consisting of exactly one atom *unit*. To encode a set with this representation, we simply construct a relation that includes the pair (*e*, *unit*) for each element *e* of the original set.

The constants (universal, empty and identity) are to be considered as indexed sets of constants, each with a different type. So in an expression such as ($Un \cap p$), the appropriate instance of the constant will be chosen, whose type matches the type of the relation *p*.

We chose to represent sets as relations because it corresponds to the natural boolean representation of a set as a bit vector. The result is a cleaner backend, in which set-theoretic operators can be treated identically for the set and relation cases.

$$\frac{}{\mathsf{Id} : \mathsf{T} \leftrightarrow \mathsf{T}}$$

$$\frac{}{\mathsf{Un} : \mathsf{S} \leftrightarrow \mathsf{T}}$$

$$\frac{}{\mathsf{0} : \mathsf{S} \leftrightarrow \mathsf{T}}$$

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

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

$$\frac{\mathsf{p}: \mathsf{S} \leftrightarrow \mathsf{T},\ \mathsf{q}: \mathsf{S} \leftrightarrow \mathsf{T}}{\mathsf{p} \setminus \mathsf{q} : \mathsf{S} \leftrightarrow \mathsf{T}}$$

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

$$\frac{\mathsf{p}: \mathsf{S} \leftrightarrow \mathsf{T}}{\mathsf{p}\sim : \mathsf{T} \leftrightarrow \mathsf{S}}$$

$$\frac{\mathsf{p}: \mathsf{T} \leftrightarrow \mathsf{T}}{\mathsf{p}+ : \mathsf{T} \leftrightarrow \mathsf{T}}$$

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

$$\frac{\mathsf{p}: \mathsf{S} \leftrightarrow \mathsf{T}}{\mathsf{dom}\ \mathsf{p} : \mathsf{S} \leftrightarrow \mathsf{Unit}}$$

Figure 6: Typing rules for the intermediate language

A more elegant treatment of sets [SS93], which we have used to justify reductions in our explicit checker [JJD97], represents the set *s* as the relation corresponding to the cross product of *s* and the universal set. In this scheme, no additional operators are required: the domain of a relation, for example, is obtained by composing the relation with the universal relation. In practice, however, it has a major disadvantage. The source language expression

```
s <: r
```

where *s* is a set and *r* is a relation, would be translated into the intermediate language expression

```
S ∩ r
```

where *S* is the relation corresponding to ($s \times Un$). What is the type of *S* in this expression? Its domain type will be the type of the elements in the original set *s*. Its range type, however, must be the type of the range of *r*. Since the type of *s* in the source expression places no constraint on the range type of *r*, we must equally allow

```
S ∩ q
```

where *q* has a different range type from *r*. The relation *S* must therefore be regarded as polymorphic in its range type. This complicates the backend of the compiler far more than the addition of a few extra operators to the intermediate language.

Type declarations in Nitpick, as in Z, may involve implicit constraints. A relation may be declared to be a function, or to be total, for example. These constraints are translated into the operators of the intermediate language. For example, the assertion that the relation *p* is a function can be expressed as

```
p~ ; p ⊆ Id
```

Each such elementary formula extracted from a declaration is conjoined to every clause in the DNF representation, since type constraints apply in every case.

For example, the clause

```
conns = called ; net
conns' = called' ; net
p in dom called
n not in ran called
called' = called U {p -> n}
inj conns
not inj conns'
```

with type declarations

```
p: Ph; n: Num; called: Ph <-> Num; net: Num -> Ph; conns, conns': Ph <-> Ph
```

is translated into

```
conns = called ; net
conns' = called' ; net
p ∈ dom called
¬ n ∈ dom called~
```

```
called' = called ∪ (p <: (n <: Un)~)
conns ; conns~ ⊆ Id
¬ conns' ; conns'~ ⊆ Id
```

with type declarations

```
p: Ph <-> Unit; ; n: Num <-> Unit;
called: Ph <-> Num; net: Num <-> Ph; conns, conns': Ph <-> Ph
```

and additional type constraints

```
p ; p~ ⊆ Id       ¬ p = 0
n ; n~ ⊆ Id       ¬ n = 0
net~ ; net ⊆ Id
```

11 An Interlude: Boolean Conjunctive Normal Form

To motivate the last two stages of compilation, we must take a short digression to explain the normal form in which the boolean formula is finally cast, since its structure and properties motivate the design of these stages.

The boolean formula is represented in conjunctive normal form, that is, as a conjunction of disjunctions. A *formula* is a set of clauses; a *clause* is a set of literals; a *literal* is a boolean variable or its negation. Here are some examples of formulae and their representation in CNF:

$a \wedge b$	{{a}, {b}}
$a \vee b$	{{a, b}}
$a \Rightarrow b$	{{–a, b}}
$a \Leftrightarrow b$	{{–a, b}, {a, –b}}

In a compositional translation to conjunctive normal form, we will need to know, given two formulae *F* and *G* in CNF, how to create the CNF formula for $F \wedge G$, $F \vee G$, $\neg F$, etc. Conjunction is easy; the clauses of $F \wedge G$ are just the clauses of *F* and the clauses of *G*. Disjunction, on the other hand, is harder. From the identity

$$(a \wedge b) \vee (c \;\wedge d) = (a \vee c) \wedge (a \vee d) \wedge (b \vee c) \wedge (b \vee d)$$

we see that the clauses of $F \vee G$ are obtained by forming the cross-product of their individual clause sets. So although the size of $F \wedge G$ is at most the sum of the sizes of *F* and *G*, the size of $F \vee G$ may be as large as the product of their sizes.

Negation is even worse; in the worst case it produces an exponential blowup. Applying de Morgan's laws to our sample expression, we get

$$\begin{aligned}&\neg\, ((a \vee b) \wedge (c \vee d))\\ &= \neg\, (a \vee b) \vee \neg\, (c \vee d)\\ &= (\neg a \wedge \neg b) \vee (\neg c \wedge \neg d)\end{aligned}$$

In general, the CNF of composite formulae is obtained according to these rules:

$$M\,[p \subseteq q] = \forall\ a,b.\ (a,b) \in E[p] \Rightarrow (a,b) \in E[q]$$

$$M\,[p = q] = \forall\ a,b.\ (a,b) \in E[p] \Leftrightarrow (a,b) \in E[q]$$

$$E\,[\mathrm{Id} : T \leftrightarrow T] = \{\ (a,a) \mid a \in C[T]\}$$

$$E\,[0] = \{\ \}$$

$$E\,[\mathrm{Un} : S \leftrightarrow T] = \{\ (a,b) \mid a \in C[S] \wedge\vee\ b \in C[T]\}$$

$$E\,[p \cup q] = \{\ (a,b) \mid (a,b) \in E[p] \vee (a,b) \in E[q]\}$$

$$E\,[p \cap q] = \{\ (a,b) \mid (a,b) \in E[p] \wedge (a,b) \in E[q]\}$$

$$E\,[p \setminus q] = \{\ (a,b) \mid (a,b) \in E[p] \wedge\ (a,b) \notin E[q]\}$$

$$E\,[p\ ;\ q] = \{\ (a,b) \mid \exists c.\ (a,c) \in E[p] \wedge\ (c,b) \in E[q]\}$$

$$E\,[p\sim] = \{\ (a,b) \mid (b,a) \in E[p]\}$$

$$E\,[p+] = \text{the smallest } x \text{ such that } x\ ;\ x \subseteq\ x \wedge\ p \subseteq x$$

$$E\,[s <: p] = \{\ (a,b) \in E[p] \mid (a,\mathrm{unit}) \in E[s]\}$$

$$E\,[\mathrm{dom}\ p] = \{\ (a,\mathrm{unit}) \mid \exists b.\ (a,b) \in E[p]\}$$

Figure 7: Semantics of intermediate language

$$[F \wedge G] = [F] \cup [G]$$

$$[F \vee G] = \{\ f \cup g \mid f \in [F] \wedge g \in\ [G]\}$$

$$[\neg F] = \{c \subseteq \mathrm{Vars}(F) \mid \forall\ f \in [F]\ .\ \exists!\ x.\ x \in\ f \wedge\ \neg x \in\ f\ \}$$

12 Translation to Boolean Conjunctive Normal Form

Understanding the penultimate step depends on understanding the final step, so we explain the latter first. The penultimate step involves no change of language: its input is a relational formula in DNF, and its output is another relational formula in DNF. In the final step, each DNF clause is translated into a boolean formula in CNF.

A relation value can be represented as a boolean matrix, with *true* in the ith row and the jth column exactly when the ith element of the domain type is related to the jth element of the range type. So a variable denoting a relation can be represented as a matrix of boolean variables, and, in general, a relational expression can be represented as a matrix of boolean formulae. It is easy to define a compositional translation that obtains the representation *X*[*e*] of a relational expression *e* from the representations of its parts. Relational composition, for example, corresponds to matrix product:

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

The full set of translation rules appears in Figure 8. No rule for transitive closure appears; as we shall see, the penultimate step (described in the next section) eliminates it.

For each elementary subformula, a single boolean formula is derived. Writing *B*[*F*] for the boolean formula derived from relational formula *F*, we have, for example,

$$B\,[p \subseteq q] = \wedge_i \wedge_j\, (X[p]_{ij} \Rightarrow X[q]_{ij})$$

which, in terms of graphs, simply says that a relation *p* is a subset of a relation *q* when the presence of any edge in *p* implies its presence in *q*. Note how equalities and inequalities translate smoothly into CNF: each element $X[p]_{ij}$ contributes a clause. The number of clauses thus rises linearly with the number of elements (and thus quadratically with the scope), although the clauses themselves grow more rapidly. This observation is a major motivation for the choice of CNF.

Since the relational formula has been converted to DNF, with each conjunct being regarded as a separate problem to solve, there are only two logical operators on relational formulae: negation and conjunction. Each is translated directly:

$$B[\neg F] = \neg B[F]$$
$$B[F \wedge G] = B[F] \wedge B[G]$$

Conjunction is well behaved: each new elementary relational formula adds a new set of clauses, so that the number of clauses in the final formula is linear in the size of the specification. Negation, as we have seen, however, is a major problem, and we shall explain below how its effects are minimized. Fortunately, the DNF clauses derived from practical problems tend to involve few negated formulae (rarely more than one).

We explained above (Section 10) how typing constraints are translated into elementary relational formulae, and subsequently have no special treatment. In our previous work [DJJ96], in which we used binary decision diagrams to solve the satisfaction problem, we took a different approach, in which the encoding of a variable exploits its type constraints.

Consider representing a total function to a set of 4 elements. Its value, seen as a bit matrix, must have exactly one bit true in each row. If, in the encoding of the variable, the row is represented as 4 separate boolean variables

$$< b_0, b_1, b_2, b_3 >$$

this constraint must be expressed as a side condition

$$\begin{array}{l} b_0 \vee b_1 \vee b_2 \vee b_3 \\ \wedge\ b_0 \Rightarrow\ \neg b_1 \wedge \neg b_2 \wedge \neg b_3 \\ \wedge\ b_1 \Rightarrow\ \neg b_0 \wedge \neg b_2 \wedge \neg b_3 \\ \wedge\ b_2 \Rightarrow\ \neg b_0 \wedge \neg b_1 \wedge \neg b_3 \\ \wedge\ b_3 \Rightarrow\ \neg b_0 \wedge \neg b_1 \wedge \neg b_2 \end{array}$$

$$B\,[p \subseteq q] = \wedge_i \wedge_j (X[p]_{ij} \Rightarrow X[q]_{ij}$$

$$B\,[p = q] = \wedge_i \wedge_j (X[p]_{ij} \Leftrightarrow X[q]_{ij}$$

$$X\,[Id]_{ij} = (i = j)$$

$$X\,[0]_{ij} = \mathrm{false}$$

$$X\,[Un]_{ij} = \mathrm{true}$$

$$X\,[p \cup q]_{ij} = X[p]_{ij} \vee X[q]_{ij}$$

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

$$X\,[p \setminus q]_{ij} = X[p]_{ij} \wedge \neg X[q]_{ij}$$

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

$$X\,[p\sim]_{ij} = X[p]_{ji}$$

$$X\,[s <: p]_{ij} = X[p]_{ij} \wedge X[s]_{i0}$$

$$X\,[\mathrm{dom}\ p]_{i0} = \vee_k X[p]_{ik}$$

Figure 8: Translation from relational to boolean form

Since each row in the matrix represents one of 4 possible values, only two boolean variables should be necessary. Viewing the pair of variables b_0b_1 as a binary number, we can instead represent the row like this

$$< \neg b_0 \wedge \neg b_1,\ \neg b_0 \wedge b_1,\ b_0 \wedge \neg b_1,\ b_0 \wedge b_1 >$$

in which the ith entry is the assertion that the binary number b_0b_1 denotes the integer i. This encoding reduces the number of boolean variables required from n to log n, and dispenses with side conditions. It applies widely, since not only functions but also scalars (which are treated as singleton sets) can be encoded in this manner.

In our BDD method, which was very sensitive to the number of variables, this gave a dramatic improvement in performance. But for the current method, these more sophisticated encodings perform considerably worse than simple encodings. In CNF, each entry in a row such as that shown above will consist of two clauses rather than one. This small difference is amplified by disjunction.

Consider, for example, translating the composition of two relational variables in a scope of k (that is, in which all domain and range types contain k elements). Each entry in the resulting matrix is the result of a disjunction of k terms. In the simple encoding, each term consists of two singleton clauses, so the entry can have 2^k clauses. In the more sophisticated encoding, each term consists of (2 log k) singleton clauses, so the entry is a factor of $(\log k)^k$ larger.

These calculations are borne out by our experiments. We implemented both encodings. The clever encoding not only brought no improvement, but in many cases caused the formula to outgrow available memory so rapidly that the formula could not be generated at all.

At a lower level, a crucial consideration is how the CNF formula is represented and whether simplifications are applied during translation. If a formula contains a clause and one of its subsets, the clause can be discarded; this is known as *subsumption* and follows from the fact that any model of F is also a model of $F \vee G$. Eliminating redundant clauses early on can have an enormous effect on the size of intermediate formulae.

Our prototype uses a trie-based representation developed by Zhang and Stickel [ZS94]. Using tries naturally eliminates some redundancy – since literals in clauses are lexically ordered, and prefixes are shared, it is easy to ensure that a clause never appears with one of its prefixes. The trie also supports a simple and efficient implementation of the Davis-Putnam satisfiability algorithm [DP60].

A huge performance gain is obtained by *negation caching.* A direct implementation of the translation rules will sometimes cause the negation of a formula to be negated. Consider, for example, translating

$$t = p \,;\, q$$

where p and q are relational variables. Each element of the matrix representing the composition is a disjunction

$$\vee_k \; E[p]_{ik} \wedge E[q]_{kj}$$

which, in CNF, will require 2^k clauses of 2 literals each. Representing an equality formula causes negation of boolean formulae on both sides

$$\begin{aligned}&x = F\\&\equiv (x \Rightarrow F) \wedge (F \Rightarrow x)\\&\equiv (\neg x \vee F) \wedge (\neg F \vee x)\end{aligned}$$

So the negation of these elements will be required. Directly negating an element formula causes a second blowup, so a naive translation will result in a huge formula. But the negation of the element formula is its dual

$$\wedge_k \; E[p]_{ik} \vee E[q]_{kj}$$

and this formula has a small CNF representation: k clauses of 2 literals. When the element formula is computed, the compiler therefore computes its negation at the same time, and caches it; when the negation is later required, the cached formula is used, and no computation is performed. Additionally, whenever a formula’s negation is computed, the original formula is cached as its negation, so that no formula is negated twice.

13 Introduction of Temporary Variables

We now turn to the phase that precedes the final one. It involves a simple manipulation of the intermediate language formula, designed to minimize the insidious effects of disjunction when the boolean formula is subsequently generated.

This manipulation is nothing more than the introduction of fresh variables to replace relational subexpressions. Recall that the relational formula is in disjunctive normal form, each clause of which is treated separately. Suppose we have a formula that contains the subexpression *e*. Without changing the meaning of the clause, we can replace *e* by a fresh variable *v*, and add to the clause the equality $v = e$. By maintaining a set of replaced expressions, we avoid introducing unnecessary new variables; if *e* is to be replaced in another context, the same variable *v* is used. This brings the standard benefit of common subexpression elimination: each subexpression is only translated once. But its primary motivation is that introducing new variables can have a dramatic effect on the size of the final boolean formula.

Decisions about where to introduce variables are currently made according to some simple heuristics that seem to work well in practice. Although they seem plausible from a theoretical point of view, we have not shown them to be optimal, and it is likely they could be improved considerably.

A disjunction of *k* boolean terms each with *m* clauses can result in a formula of size m^k. Variables are introduced when either *m* or *k* is large. We shall consider the cases in order of increasing importance.

- The least dramatic case, but one that nevertheless merits variable introduction, arises when each element of the resulting matrix is obtained by combining, with disjunction, an element from each of two matrices. In translating the union expression $p \cup q$, we must therefore ensure that elements of the matrices resulting from translating *p* and *q* do not contain many clauses. Note that if *p* and *q* are relational variables, the elements of $X[p \cup q]$ will still only contain a single clause. Composition and intersection, on the other hand, always create elements with multiple clauses. So our heuristic is to replace any expression that appears in a union expression that is itself a composition or an intersection expression.
- A similar issue arises at the elementary formula level. Translating $p \subseteq q$ also involves an element-wise disjunction, so we replace either *p* or *q* with a variable when both are expressions involving composition or intersection.
- Translating relational composition gives rise to a disjunction of *k* terms, where *k* is the length of a row or column (ie, the scope). Each of these terms is itself the result of a conjunction, so composition is always expensive. All subexpressions therefore, unless trivial (a variable or the transpose of a variable), are replaced by variables when appearing in a composition expression.

The most dramatic effect of disjunction arises for negated formulae. Consider, for example, the elementary formula

$$p \neq 0$$

where *p* is some relational expression. This translates to a disjunction of k^2 terms for a scope of *k*. In practice, unless *p* is a variable, the translation is usually infeasible. So for any negated formula, we replace the expressions on both sides with variables. In this case, the result is a formula with just one clause, but for other negated formulae the result is usually still large. The translation of

$$\neg p \subseteq q$$

for example, has 2 to the power k^2 clauses when both *p* and *q* are variables. For any scope above 3, this is infeasible, so this case is given special treatment (see Section 15).

14 Transitive Closure

Translating transitive closure presents some special problems. The transitive closure of a relation *p*, *p*+, is the smallest relation *r* that includes *p*

$$p \subseteq r$$

and is transitive

$$r \,;\, r \subseteq r$$

and can be computed by the series

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

Since *p* must be homogeneous, we can view it as a graph with one node for each element of the domain type of *p*, and an edge from node *a* to node *b* whenever *p* relates *a* to *b*. Two nodes are associated by (*p* ; *p*) if there is a path of two edges from one to the other; by (*p* ; *p* ; *p*) if there is a path of three edges, and so on; and by the closure if there is a path of any length between them.

Computing the closure of a relation can thus be viewed as constructing paths in such a graph; in each step we compute a set of longer paths, and stop when we reach the fixpoint in which no path can be lengthened. How many steps might this take? If there are *n* nodes in the graph, a pair of nodes can either be connected with a path of length n or less, or not at all. It follows that a simple iterative computation will require at most *n* steps, each involving a union and on average *n/2* compositions.

A more efficient way to translate closure is by the series:

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

Since any path of length no greater than 2*k* can be decomposed into two paths each of length no greater than *k*, it follows by induction that the ith approximation in this series will associate nodes that are connected by a path of at most 2^i edges. In this case, however, we will reach convergence in log *n* steps. This technique is known in model checking [BC+92] as *iterative squaring*.

One way to encode closure is to apply the translation rules for composition and union on the fly, translating the approximations p_i. For exactly the reasons explained above (in Section 13), this will not generally be feasible: the unions lead to disjunctions of increasingly large formulae. So rather than using the series to evaluate the closure in boolean form, we use it to expand the closure expression syntactically. By performing this unwinding just prior to the stage in which variable introduction occurs, the effects of the disjunctions are mitigated as for any other complex expression.

Although this works well and is easy to implement, it does have one undesirable consequence. The compilation becomes scope dependent in an earlier phase. Without closure, only the final translation to a boolean formula depends on the scope; now variable introduction becomes dependent too. In practice this is a minor annoyance, since the bulk of the compilation time is in the final phase.

15 Symmetry Breaking

For relational formulae involving only the set operations (union, intersection and difference), the boolean formula grows only with k^2 for a scope of k. Composition, unfortunately, introduces a factor of 2^k. But far worse is the effect of certain negations: the formula

$$p \neq q$$

is equivalent to

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

which results in a disjunction of k^2 terms, resulting in a boolean formula of 2 to the k^2clauses. Generating such a formula is infeasible for $k > 4$.

Such formulae arise primarily in two places. Operations often have preconditions that assert that a scalar is not in a set; this becomes an inequality on relations, but of dimension $1 \times k$, so the blowup is no worse than for a composition. More seriously, it is common to assert that an operation leaves a state component unchanged. This gives a formula like

$$op \wedge p' \neq p$$

where *op* is the specification of the operation. A similar situation arises for a claim that an operation preserves an invariant that one relation is a subset of another.

To address this problem, we apply some ideas from our previous work on symmetry [JDJ96, JJD97]. As we have shown, the models of relational formulae 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. Consider now a model of the formula

$F \wedge p \neq q$

in which for some particular values of *i* and *j*,

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

Applying the permutation that maps *i* and *j* both to 0, we obtain a model in which

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

is true. We can therefore replace the disjunction

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

by the single term

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

which, if *p* and *q* are variables, reduces the boolean formula to 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, as we plan to explain in a forthcoming paper, more than one negated equality (or inequality) can be reduced in this way, so long as each reduction exploits permutation of a different type. The prototype tool ranks the negated subformulae according to their severity, and then allocates symmetry breaking reductions from the most to the least severe, making sure never to break symmetry on the same type twice. Fortunately, there is usually only one negated subformula on full relations, so the method works remarkably well.

Negative equalities can also be translated into disjunctions that are handled separately, in the same manner as top-level disjunctions. The formula

$p \neq q$

is equivalent to

$(\neg p \subseteq q) \vee (\neg q \subseteq p)$

The negated inequality

$\neg p \subseteq q$

can then be rewritten as

$p \cap (Un \setminus q) \neq \{\}$

which can be simplified with variable introduction to

$t = p \cap (Un \setminus q)$
$t \neq \{\}$

thus eliminating subsequent translation steps that would involve any more than elementwise operations on the boolean matrices. Problematic elementary formulae that cannot be handled by symmetry breaking (because all types have been ‘consumed’) might be handled in this way.

16 Solving the Boolean Formula

We have experimented with two solvers. We wrote a Davis-Putnam solver [DP60] using Zhang and Stickel's trie-based representation of formulae [ZS94] in Java as part of the prototype. We also wrote code to generate input files for Selman and Kautz's WalkSAT solver [SKC94], a descendant of GSAT [SLM92].

The two solvers are very different. WalkSAT, unlike Davis-Putnam, is incomplete: it may fail to find a satisfying assignment even though one exists. Also, because it starts from a random assignment, WalkSAT cannot be made to produce the simplest solutions first; in our Davis-Putnam implementation, by favoring false over true in the case splitting step, we are able to bias the solver towards small relations. On the other hand, WalkSAT can handle much larger problems. We found that Davis-Putnam often stops working when the number of boolean variables exceeds a few hundred. WalkSAT, on the other hand, is so fast that the time it takes to solve the formula is invariably dominated by the time the compiler took to construct it.

We have not found the incompleteness of WalkSAT to be a problem: we have yet to come across a problem that we believed had a solution but which WalkSAT failed to find.

Both methods have a desirable performance property. When there is a solution, it tends to be found very quickly; when there isn't, the solver can run for a very long time. The engine underlying our previous implementation of Nitpick [JJD97] did not have this property: in many cases, successful and unsuccessful searches took roughly the same amount of time (within a factor of 2 to 10). Of course we would like the solver to work fast whatever the outcome, but if forced to pick, a very uneven distribution biased towards yielding solutions is, in practice, much better.

17 Performance results

Some performance results for a variety of specfications are shown in Table 1; the specifications are reproduced, mainly in full, in the Appendix. The first two examples are toy benchmarks; *Phone* is the specification of Figure 1 (but for a different claim, *invB_preserved*) and *Finder* is a specification of the directory structure of the Macintosh Finder.

Style is a specification of the paragraph style hierarchy of Microsoft Word, described in detail in [JD96a].

Allocate is a simplified fragment of a railway interlocking specification written in Z by Praxis UK PLC. The performance figures are for the claim *AllocSafe2.*

Mobile IP is a specification of a draft version of a mobile internet protocol for IPv6, written by an undergraduate at Carnegie Mellon [JNW97, Ng97]. The performance figures are for the claim *loc_update_OK*, which exposed a flaw in the protocol.

Table 1 is to be interpreted as follows. The column marked *Cases* gives the number of clauses in the DNF representation of the claim; *Formulae* gives the maximum number of relational formulae in each clause. In the analysis of *Style*, for which there were 12 clauses, a model was always found in the first clause; the other columns refer to this clause alone.

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 formulae were solved using WalkSAT, except when DP appears in parentheses after the solving time; for these, the built in Davis-Putnam solver was sufficient (ie, found a solution in less than 30s). All timings were obtained using Sun-Soft's just-in-time compiler, running under Windows NT on a Pentium 133MHz processor with 64MB of memory. All times are wall clock times; no attempt was made to get precise numbers for times less than 1 second.

The final column gives the time taken by the explicit version of Nitpick, coded in C and running on a Macintosh with a 66MHz PowerPC 601 processor and 24MB of RAM. Since this machine is perhaps 3 times slower than the Windows machine, the solving times for the new method should be multiplied by 3. (The translation times do not need the same adjustment, because of a compensating discrepancy between Java and C).

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. This scope could not be checked with the new method, because the prototype currently allows only scope settings that give every type the same bound.

As can be seen, the new method outperforms the old method in almost all cases. It seems likely that it will be able to handle much larger models. 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).

The explicit method is barely able to handle the *Mobile IP* example, and cannot handle *Allocate* at all. *Allocate* is particularly well suited to the new method, because it involves several relations but few compositions. *Finder*, on the other hand, is biased towards the explicit checker; its relations are constrained to be functions, and transitive closure (which is no more expensive for the explicit checker than any other operator) appears several times in the formula. It is nevertheless surprising that WalkSAT takes so long to solve the scope 6 instance of *Finder* – it required about 760 thousand flips of boolean variables, distributed over 8 separate attempts from different, randomly generated starting assignments.

Tables 2 and 3 show the effects of negation caching and symmetry breaking. In both, the italicized columns represent measurements taken without the optimization. Negation caching has a huge effect, increasing with the scope; in many cases, it reduces the final formula size by an order of magnitude. The scope 4 instance of *Style* is worth noting: the translation time, but not the formula size is dramatically reduced. This is probably because negation caching saves unnecessary computation, and reduces the size of many formulae generated in the course of the translation.

Symmetry has a negligible effect on all examples except for *Style*, the only example whose formula involves negated equalities of relation-valued variables. The italicized numbers in this table were obtained with symmetry breaking off, but negation caching on.Without symmetry, the formula for a scope of 5 cannot even be generated.

Example	Cases	Formulae	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 1: Performance compared to explicit method

Example	Scope	Vars	*Clauses*	Clauses	*Trans*	Trans	*Solve*	Solve
Phone	3	66	*601*	313	*1s*	0s	*0s (DP)*	0s (DP)
	4	112	*2808*	856	*0s*	0s	*0s (DP)*	0s (DP)
	5	170	*12489*	2189	*31s*	0s	*1s (DP)*	0s (DP)
	6	240	*52923*	5413	*6m,12s*	1s	*5s (DP)*	2s (DP)
Finder	3	273	*1793*	1370	*0s*	0s	*0s*	1s (DP)
	4	464	*6935*	3500+	*2s*	0s	*0s*	0s
	5	705	*27831*	8483+	*27s*	0s	*5s*	3s
	6	996	*112635*	20866	*9m,32s*	9s	*?*	2m,33s
Style	3	408	*2908*	2234	*0s*	0s	*6s (DP)*	6s (DP)
	4	704	*57377*	53495	*1m,51s*	17s *	*0s*	0s
	5	1080	*??*	??	*??*	??	*??*	??
Mobile IP	3	438	*3330*	2439	*1s*	0s	*0s*	0s
	4	760	*13242*	6608	*17s*	3s	*0s*	0s
	5	1170	*52826*	16691	*2m,56s*	9s	*0s*	0s

Table 2: Effect of negation caching

Example	Scope	Vars	*Clauses*	Clauses	*Trans*	Trans	*Solve*	Solve
Style	3	408	*2234*	1864	*0s*	0s	*2s (DP)*	6s (DP)
	4	704	*53495*	4385	*17s*	2s	*0s*	0s
	5	1080	*??*	9977	*??*	6s	*??*	5s

Table 3: Effect of symmetry breaking

18 Related Work

Our previous method worked by explicit enumeration of relation values, with two principal mechanisms to prune the search: *short circuiting*, in which a partial assignment could be rejected by determining that any extension to a full assignment would not yield a model [DJ96], and *isomorph elimination*, which exploited symmetries in the search space to avoid the generation of a high proportion of the relation values [JDJ96, JJD97].

On most of our examples, the new boolean method performs better. In particular, the explicit checker does badly when there are several variables representing relations that are not constrained to be functions, and for which there are few mutual constraints; in this case, short circuiting fares badly, and the search is often intractable. The explicit checker always prefers functions to relations; the boolean checker, in contrast, prefers relations since there are then no side conditions. On the other hand, increasing the complexity of the formula itself tends to improve the performance of the explicit checker, but it has a detrimental effect on the boolean checker. The explicit checker can take advantage of equalities by eliminating variables; the boolean checker, in contrast, introduces variables to avoid complex expressions and reduce translation time, and pays the price in solving time. For this reason, the explicit checker might be better as a simulator: if operations are expressed mostly constructively (with post-state variables equated to expressions involving pre-state variables), it can explore the execution of a sequence of operations with little extra cost.

In his PhD thesis, Craig Damon is investigating *bounded generation*, a new pruning mechanism for the explicit checker that seems, from initial experiments, to be promising. It remains to be seen how it will perform in comparison to the boolean checker.

We have experimented before with a boolean checker, representing boolean formulae not in CNF but with ordered binary decision diagrams (BDDs) [DJJ96]. Although that method performed very well on some small examples, it did not appear to scale. Unlike CNF, a BDD is canonical, so translation cannot be separated from solving; if the formula has no models, it must be the formula *false*. Translation into BDDs can take vast amounts of memory, and offers no discount for finding only some models, since the final BDD represents all models. The BDD-based checker also suffered from the well known unpredictability of BDDs; a small change in the variable ordering might have a dramatic effect, and problems that looked alike were often not equally hard to solve. The canonicity of BDDs is essential in the context of model checking, because it allows detection of fixed points. But for our problem, canonicity is unnecessary and its cost is not warranted.

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, where the problem of finding a plan that satisfies a set of constraints is reduced to finding a model of a boolean formula [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 compilation process rather different from those employed in planning.

19 Future Work

19.1 Algorithms and Data Structures

The trie representation of CNF is reasonably compact – typically a third smaller than a representation without sharing of subclauses – and, because of the ordering of variables, efficiently manipulated. The trie structure eliminates some redundancy, but not all: a clause may appear with a subclause if the subclause is not a prefix. We performed some experiments to determine what proportion of clauses held in the trie are redundant; it appears to be around 30% for large tries. It might be possible to eliminate more redundancy without too much additional computation, perhaps by using a different data structure such as a suffix tree.

19.2 Language Extensions

Several features would add nothing in expressive power but would make NP an easier language to use.

Quantifiers over scalar variables would relieve the burden of encoding all constraints with relational operators alone. One common idiom that calls for quantifiers (or perhaps a dual of the composition operator) arises from an assertion that would be written with quantifiers thus:

```
forall x, y, z. (x -> y) in p and (y -> z) in q => (x -> z) in r
```

To encode this in relational operators, we rewrite with negations

```
not exist x, y, z. (x -> y) in p and (y -> z) in q and not (x -> z) in r
```

and then translate negation using complementation:

```
p ; q & (Un \ r) = {}
```

The purely relational version is often – as here – terse, elegant and obscure.

An *abstraction mechanism for generic functions and constraints* would simplify specifications by factoring out complex idioms. Tree structure, for example, arises frequently; rather than axiomatizing each tree structure anew, the specifier might write *Tree*(*p*) to assert that a relation *p* mapping nodes to their parents describes a tree, with the predicate *Tree* previously defined:

```
Tree (p) = (Fun(p) and Acyclic(p) and One(Roots(p)))
Acylic (p) = (p+ & Id = {})
Roots (p) = (ran p \ dom p)
One (s) = (Fun(Un :> s))
Fun (p) = (p~ ; p ⊆ Id)
```

Ternary relations are not common, but are clumsy to encode as binary relations and should be supported directly. A ternary relation r on $A \times B \times C$ is probably best treated curried, so that $r.a$ denotes a relation on $B \times C$.

Integers would add expressive power. Subtraction, addition and comparison alone would allow a sequence of elements of type T to be modelled more naturally as a function from a prefix of the naturals to T, where currently we are forced to represent the sequence more abstractly as an ordering of buckets that contain values. Indexing of sequences would simply be an instance of function application; concatenation would be provided explicitly. How to handle the semantics of integer addition over a finite scope is not clear.

20 Acknowledgments

This work benefited in its early stages from discussion with Somesh Jha and Craig Damon. Aaron Greenhouse implemented the trie representation of CNF, the Davis Putnam solver and a preliminary version of the prototype's parser. Thanks also to Greg Nelson, who persuaded me to 'dust off' the boolean approach and give it another try. This research was funded in part by NSF grant CCR-9523972.

21 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.

[C+95] Edmund M. Clarke, Orna Grumberg, Hiromi Hiraishi, Somesh Jha, David E. Long, Kenneth L. McMillan, Linda A. Ness. Verification of the Futurebus+ cache coherence protocol. *Formal Methods in System Design*, 6, 217–232, 1995.

[DJ96] C. Damon and D. Jackson. Efficient Search as a Means of Executing Specifications. *Proc. Tools for Construction and Analysis of Software*, Passau, Germany, March 1996, pp. 70–86.

[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.

[G+95] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[Giv88] Steven Givant. Tarski's development of logic and mathematics based on the calculus of relations. *Colloquia Mathematica Janos Bolyai 54*, Algebraic Logic, Budapest, Hungary, 1988.

[IBM97] Object Constraint Language. www.software.ibm.com/ad/ocl.

[JD95] *Semi-executable Specifications.* Daniel Jackson and Craig A. Damon, CMU-CS-95-216, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, November 1995.

[JD96a] Elements of Style: Analyzing a Software Design Feature with a Counterexample Detector. Daniel Jackson and Craig A. Damon. *IEEE Transactions on Software Engineering*, Vol. 22, No. 7, July 1996, pp. 484–495.

[JD96b] Daniel Jackson and Craig A. Damon. *Nitpick Reference Manual.* CMU-CS-96-109. School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, January 1996.

[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.

[JJD97] Daniel Jackson, Somesh Jha and Craig A. Damon. Isomorph-free Model Enumeration: A New Method for Checking Relational Specifications. To appear, *ACM Transactions on Programming Languages and Systems.*

[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.

[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.*

[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.

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

[ZS94] Hantao Zhang and Mark E. Stickel. *Implementing the Davis-Putnam Algorithm b yTries.* Technical Report 94-12, Artificial Intelligence Center, SRI International, Menlo Park, CA. December 1994.

Appendix: Benchmark Specifications

Phone

```
[Ph, Num]

Switch = [
   Called: Ph <-> Num
   const Net: Num -> Ph
   Conns: Ph <-> Ph
|
   Conns = Called ; Net
]

 Join  (p: Ph; n: Num) =  [
   Switch
|
   p  in  dom (Called)
   not (n in ran Called)
   Called' = Called U {p -> n}
]

invB =  [Switch | (dom Conns) & (ran Conns)  = {}]
invC = [Switch | fun (Conns~)]

InvB_preserved (p: Ph; n: Num)  :: [Switch | Join(p,n) and invB => invB']
InvC_preserved (p: Ph; n: Num)  :: [Switch | (Join(p,n) and invC) =>invC']
```

Finder

```
[OBJ]

Finder = [
  const drive, trash: OBJ
  const files, folders: set OBJ
  dir, links: OBJ -> OBJ
  trashed, aliases: set OBJ
 |
  {drive, trash} <= folders \ dom dir
  ran dir <= folders
  trashed = dir~+.{trash}
  not drive in trashed U {trash}
  aliases <= files
  aliases = dom links
  links+ & Id = {}
  files & folders = {}
  files U folders = OBJ
]

Move (x, to: OBJ) = [
  Finder
|
  x not in dir*.{to}
  dir' = dir (+) {x -> ((links* ;> aliases).to)}
  links' = links
]

TrashingWorks (x, to: OBJ) :: [Finder |
  Move (x, to)
  and to in trashed U {trash}
  => x in trashed'
]
```

Style

```
[style, format]

Tree = [
  based : style -> style
  const normal : set style
  |
  normal & dom based = {}
  ran based \ normal <= dom based
  based+ & Id = {}
  ]

Sheet = [
  Tree
  delta, assoc : style -> format
  |
  normal <: assoc = normal <: delta
  normal <; assoc = normal <; ((based ; assoc) (+) delta)
  ]

ChgParent (s, from, to : style) = [
  Sheet
  |
  s in dom based and from = based.s
  based' = based (+) {s -> to}
  assoc' = assoc
  {s} <; delta = {s} <; delta'
  assoc.to = assoc.from => delta' = delta
  ]

Xi ()= [const Sheet]

Claim (s, from, to : style) ::
   [Style | ChgParent (s, from, to) ; ChgParent (s, to, from) => Xi()]
```

Allocate

```
[USER, RESOURCE]

Bookings = [
  reservedBy : RESOURCE -> USER
  pending, granted, free : set RESOURCE
  reserved : set RESOURCE
|
  reserved = dom reservedBy
  pending & granted & free = {}
  pending & free = {}
  granted & free = {}
  pending U granted U free = RESOURCE
]

Resources = [
  open, closed : set RESOURCE
  const overlap, incons, excludes : RESOURCE <-> RESOURCE
  excluded : set RESOURCE
|
  overlap~ = overlap and overlap & Id = {}
  incons~ = incons and incons & Id = {}
  excludes = incons U overlap
  incons.open <= closed
  excluded = excludes.open
  open & closed = {}
  open U closed = RESOURCE
]

Users = [
  usedBy : RESOURCE -> USER
  used : set RESOURCE
|
  used = dom usedBy
]

Allocate (r : RESOURCE; u : USER) = [
   Resources Users Bookings
|
  r in (pending & open) \ (excludes.reserved)
  {r -> u} <= reservedBy
  granted' = granted U {r}
  reservedBy' = reservedBy
  usedBy' = usedBy U  {r -> u}
]

AllocSafe0  (r : RESOURCE; u : USER) :: [
   Resources Users Bookings
|
  Allocate (r, u) and (excludes.reserved & reserved = {})
=> excludes.reserved' & reserved' = {}
]

AllocSafe1  (r : RESOURCE; u : USER) :: [
   Resources Users Bookings
|
 Allocate (r, u) and (excluded & granted= {})
=>excluded' & granted' = {}
]

AllocSafe2  (r : RESOURCE; u : USER) :: [
  Resources Users Bookings
|
 Allocate (r, u) and (incons.used & used = {})
=> (incons.used' & used' = {})
]

AllocSafe3  (r : RESOURCE; u : USER) :: [
  Resources Users Bookings
|
 Allocate (r, u) and (incons U overlap).used & used = {}
=> (incons U overlap).used' & used' = {}
]
```

Mobile IP

```
[HOST, MSG, TS]

net = [
  router: HOST
  cached: set HOST
  subh: set HOST
  clock: TS
  caches: HOST -> HOST
  cache_exp_time: HOST -> TS
  updates: set MSG
  to, from, where: MSG -> HOST
  send_time, exp_time: MSG -> TS
  const precedes: TS -> TS
  const before: TS <-> TS
|
  dom to = updates
  dom from = updates
  dom where = updates
  dom send_time = updates
  dom exp_time = updates
  dom cache_exp_time = dom caches
  exp_time <= send_time; precedes+
  caches & Id = {}
  (from; from~) & (to; to~) & (send_time; send_time~) <= Id
  (from~; to) & Id = {}

  /* axiomatize the time ordering as a total order */
  /* for now, make do with weaker constraint */
  before = precedes+
  before & Id = {}
]

mh_arrive (h:HOST; m:MSG; t:TS) = [net |
   not router = h
   router' = h
   not m in updates
   t in (before.{clock})
   clock' in (before.{clock})
   subh <= cached
   cached' = subh
   cache_exp_time' = (cached' <: cache_exp_time) :> (before.{clock'})
   caches' = dom(cache_exp_time') <: caches
   updates' = updates U {m}
   send_time' = send_time U {m -> clock}
   exp_time' = exp_time U {m -> t}
   to' = to U {m -> router}
   from' = from U {m -> h}
   where' = where U {m -> h}
]

update_arrival (m:MSG) = [net |
   clock' in before.{clock}
   subh <= cached
   cached' = {to.m} U subh
   cache_exp_time' = (cached' <: (cache_exp_time (+) {to.m -> exp_time.m})) :>
(before.{clock'})
   caches' = dom (cache_exp_time') <: (caches (+) {to.m -> where.m})
   router' = router
   updates' = updates
   to' = to
   from' = from
   where' = where
   send_time' = send_time
   exp_time' = exp_time
]

acyclic_caches = [net | caches+ & Id = {}]

host_move_OK (h:HOST; m:MSG; t:TS) ::
         [net | acyclic_caches and mh_arrive (h, m, t) => acyclic_caches']

loc_update_OK (m:MSG) ::
         [net | acyclic_caches and update_arrival (m) => acyclic_caches']
```