Boolean Compilation of Relational Specifications

Daniel Jackson
MIT Lab for Computer Science
545 Technology Square
Cambridge, Mass 02139, USA

dnj@lcs.mit.edu

submitted to CAV'98, category A (regular paper)

Abstract

Relational specifications, in which the states themselves are characterized by relations, have wide application in software design but, until now, limited support from automated tools. This paper describes a new method for finding errors in a relational specification. A property to be checked is cast as a relational formula, which, if the property holds, has no finite models. The user selects a 'scope' that bounds the size of candidate models. The relational formula is then translated into a boolean formula that has a model for every model of the relational formula within that scope.

The boolean formula is represented in conjunctive normal form (CNF), being both a good match for relational formulae and the required input format of the most efficient boolean solvers. Certain features of the boolean formula, however, cause an exponential blowup if a direct translation is attempted. But by using some simple strategies – case splitting, variable introduction, symmetry breaking and negation caching – it is possible to overcome these problems and make translation efficient. The resulting boolean formulae, although often huge, are easily handled by WalkSAT, a stochastic solver. For a variety of specifications we examined, compilation and solving together take less than 10 seconds.

1 Introduction

A relational specification is 'relational' in two senses. State transitions are characterized as a relation between the pre- and post-states of a single step; additionally, the states themselves are characterized by a collection of relations. This latter complexity has been one of the obstacles to applying model checking to the analysis of software.

Many languages have been developed for describing this kind of structure; in academia, formal specification languages such as Z [Spi92], and in industry, object constraint languages such as OCL [IBM97]. Tool support, however, has been confined to simple static checking or to interactive theorem proving.

Nitpick is a checker for relational specifications. Its language, NP [JD96b], is essentially the subset of Z corresponding to the relational calculus, and is thus undecidable [Sch79]. Nitpick cannot, therefore, decide whether a formula is valid; instead it attempts to refute it, by finding a counterexample – a model of the formula's negation. The search for counterexamples is conducted within a finite *scope* selected by the user. Our hypothesis, so far borne out by our experiences with Nitpick, is that small scopes suffice: that is, most errors in specifications can be demonstrated with small counterexamples.

The first version of Nitpick searched by enumerating relation values explicitly. A suite of reduction mechanisms, notably exploitation of symmetry in the search space [JDJ96, JJD97], made it possible to analyze a variety of small specifications. A draft standard for a mobile internet protocol, for example, was specified in NP; the subsequent Nitpick analysis revealed a serious flaw (that a routing loop might be set up, causing messages to circulate endlessly) [JNW97, Ng97].

In this paper, a new checking method based on boolean satisfaction is described. A claim – that some property of the specification holds – is negated, and translated from a relational formula into a boolean formula. A boolean solver is used to find a model of the boolean formula, which is then converted back into an assignment of relational values. The boolean formulae that are generated are usually huge – 1,000 variables and 20,000 clauses – but are easily within the grasp of stochastic solvers such as WalkSAT [SKC94]. This new method consistently outperforms our previous method, and appears to scale more readily to larger specifications and larger scopes.

The compiler generates formulae in conjunctive normal form (CNF). Fortuitously, CNF is not only well suited to representing relational formulae compactly, but is also the input format for the most efficient solvers. There are some features of relational formulae, however, that cause a blowup in the translation. How these are handled is the focus of this paper.

2 Example

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.

```
[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 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.

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.

Nitpick offers two functions: simulation and checking. 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.

3 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.

4 Scope

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 assignments involving at most 3 phones and 2 numbers, we would choose $S = \{Ph \rightarrow 3, Num \rightarrow 2\}$. 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 2 to 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 $k^2 v$ assignments. Increasing the scope from k to $k+1$ thus increases the space by 2^{2kv}. In practice, however, the time our method takes to find a model usually increases by a small constant factor as the scope is increased.

$[p \subseteq q] = \wedge_i \wedge_j [p]_{ij} \Rightarrow [q]_{ij}$	subset
$[p = q] = \wedge_i \wedge_j [p]_{ij} \Leftrightarrow [q]_{ij}$	equality
$[Id]_{ij} = (i = j)$	identity relation
$[0]_{ij} = false$	empty relation
$[Un]_{ij} = true$	universal relation
$[p \cup q]_{ij} = [p]_{ij} \vee [q]_{ij}$	union
$[p \cap q]_{ij} = [p]_{ij} \wedge [q]_{ij}$	intersection
$[p \setminus q]_{ij} = [p]_{ij} \wedge \neg[q]_{ij}$	difference
$[p ; q]_{ij} = \vee_k [p]_{ik} \wedge [q]_{kj}$	composition
$[p\sim]_{ij} = [p]_{ji}$	transpose
$[s <: p]_{ij} = [p]_{ij} \wedge [s]_{i0}$	domain restriction
$[dom\ p]_{i0} = \vee_k [p]_{ik}$	domain

Figure 2: Translation from relational to boolean form

5 Boolean Compilation

For a finite scope, a relational formula can be translated compositonally into an equivalent boolean formula. 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. 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. Note that the outermost boolean operator is conjunction: it is this property of relational formulae that makes CNF a good representation.

Sets and scalars are treated as degenerate relations. A set-valued term is translated into a vector of boolean formulae; 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 first translates the formula to be analyzed into an intermediate language with a more frugal repertoire of relational operators, to simplify the boolean translation. Rules for this intermediate language are shown in Figure 2. The language, and its relationship to the source language, is described in more detail in [Jac97].

6 Problems with Translation to CNF

In CNF, a boolean formula is represented as a set of clauses; a clause is a set of literals; a literal is a boolean variable or its negation. The formula is interpreted as a conjunction of the clauses; each clause is a disjunction of its literals.

Unforunately, direct translation of a relational formula into CNF is infeasible. To see why, consider how boolean operations are performed in CNF:

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

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

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

Conjunction is simply union of clause sets. Disjunction, however, requires taking the cross product; the size of a disjunction is, in the worst case, the product of the sizes of the original formulae. Negation is even worse: a clause is formed by taking a literal (and negating it) from each clause, in all possible ways. Negating a formula with n clauses of k literals each can lead to a formula of size k^n.

Disjunctions or negations arise at several places in translating a relational formula. They are, from most to least severe:

- *Top-level disjunction.* Disjunctions may appear explicitly in the relational formula for two reasons: because an operation is specified as a disjunct of cases, or because a claim asserts two constraints, and thus its negation becomes a disjunction that either constraint is violated.
- *Negative equalities.* The formula $p \neq q$ would be translated to $\vee_{ij} \; [p]_{ij} \neq [q]_{ij}$, 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.
- *Complex expressions.* 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 problematic in a similar way; 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 $[e]_{ij}$.

For each of these problems, Nitpick employs a mitigating strategy. These strategies are described in the following subsections.

6.1 Case Splitting

To handle top-level disjunction, we form a separate CNF formula for each disjunct and solve it separately. Typically no more than 4 such disjuncts arise in the checking of a single claim. Moreover, this case splitting corresponds naturally to the structure of the specification itself; knowing which case failed tells the specifier which of the branch of the operation is faulty, or which constraint of an invariant is violated.

6.2 Introduction of Temporary Variables

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 formulae:

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

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

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.

Transitive closure is handled as a byproduct of variable introduction. Rather than computing the series

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

we use the standard technique of iterative squaring [BC+92]:

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

For a scope of k, the series converges in log k steps. A closure expression is first unwound syntactically using this series; the resulting expression then becomes, like any other expression, subject to variable introduction. This treatment of closure does have one annoying consequence: translation becomes scope dependent at an earlier stage.

6.3 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 $[e]_{ij} \Rightarrow [p]_{ij}$ which involves a negation of $[e]_{ij}$. 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 $[e]_{ij}$ is smaller than $[e]_{ij}$ itself. Suppose e is a composition $p \,;\, q$ of variables. Then

$$[e]_{ij} = \vee_k \; p_{ik} \wedge q_{kj}$$

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

$$\neg[e]_{ij} = \wedge_k \; p_{ik} \vee q_{kj}$$

which gives k clauses of 2 literals each. Since CNF is not canonical, computing $[e]_{ij}$ 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 allowing 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.

Nitpick therefore employs the following strategy. When the element formulae for a composition are computed, their duals are also computed and cached. The representation of a formula thus includes two formulae: 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.

6.4 Symmetry Breaking

Negative equalities can 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. Often, however, a much more effective strategy can be applied.

We have shown previously [JDJ96, JJD97] that 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

$$F \wedge p \neq q$$

where F is an arbitrary subformula, and p and q are variables, in which for some particular values of i and j,

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

Applying the permutation that maps i and j both to 0, F must still hold, and we obtain a model in which

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

is true. We can therefore replace the disjunction

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

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

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

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, 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. Nitpick 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.

Permutation invariance holds only because the primitive types are uninterpreted. Even in the presence of interpreted types (such as integers), relational formulae 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.

Symmetry breaking can be applied both to negated equalities and negated inequalities. Negated equalities tend to arise only in a single context. The assertion that an operation leaves a state component unchanged has the structure

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

where *op* is the specification of the operation. Negated inequalities are more common; often, for example, an operation has a precondition that a scalar argument is not a member of some set. Since a set is translated into a one-dimensional boolean matrix, these inequalities are not expensive anyway.

7 Summary

Nitpick compiles a claim to be checked or an operation to be simulated in stages:

1. *Schema Expansio*n. Schema references are replaced by their declarations and assertions; implicit equalities for const variables are inserted; and so on. This results in a quantifier-free relational formula, along with a collection of type declarations.

2. *Case Splitting*. The relational formula is converted to dijsunctive normal form; each clause is subsequently treated as a separate problem. A clause consists of a set of elementary relational formulae, implicitly conjoined.
3. *Translation to Intermediate Language.* The elementary formulae are translated into a relational calculus with a more frugal repertoire of operators. Sets and scalars are treated as degenerate relations. Transitive closures are unwound syntactically.
4. *Temporary Variable Introduction.* Complex expressions are replaced by temporary variables, and appropriate equalities are added. Opportunities for symmetry breaking are identified.
5. *Translation to CNF*. Each elementary relational formula is translated into a boolean formula in CNF; the resulting formulae are conjoined together. Boolean formula are represented as tries, with negation caching.
6. *Solving*. The boolean formula is presented to a satisfaction algorithm. We have experimented with two: 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 currently is run as a standalone C program offline.

A technical report elaborates on details that were omitted from this paper for lack of space [Jac97]. In particular, it justifies the design of the intermediate language, and gives its semantics and typing rules.

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.

8 Performance Results

Some performance results for a variety of specfications are shown in Table 1. *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 [JD96a]. *Allocate* is a simplified fragment of a railway interlocking specification written in Z by Praxis UK PLC. *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 specifications are reproduced in full in the technical report [Jac97].

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 (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 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). The final column gives the time taken by the explicit version of Nitpick.

Timings were obtained on a modestly equipped machine; some insignificant platform discrepancies between the two versions of Nitpick are explained in the technical report [Jac97]. 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. 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).

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].

9 Related Work

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 excellently on some 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, 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.

None of the particular strategies Nitpick employs is especially novel, perhaps with the exception of negation caching. Symmetry breaking exploits the same properties of relational formulae that our previous method exploited [JDJ97, JJD97], albeit in a different manner. Variable introduction and case splitting are standard techniques. Our contribution is instead in recognizing the compatibility of CNF and relational formulae, and in demonstrating that simple translation strategies are enough to handle the formulae that arise in practice.

10 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.

11 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] Checking Relational Specifications with Binary Decision Diagrams. Craig A. Damon, Daniel Jackson and Somesh Jha. *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.

[IBM97] *Object Constraint Language Reference Manual*, September 1997. www.software.ibm.com/ad/ocl.

[Jac97] Daniel Jackson. *Boolean Compilation of Relational Specifications*. Technical Report MIT-LCS-TR-735. Lab for Computer Science, Massachusetts Institute of Technology, Cambridge, MA. December 1997. Available at www.pdos.lcs.mit.edu/~dnj/papers/tr735.ps.

[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.

[SLM92] Bart Selman, Hector Levesque and David Mitchell. A new method for solving hard satisfiability problems. *Proc. 10th National Conference on Artificial Intelligence.*

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

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

[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.

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 on a variety of specifications