Checking Relational Specifications With Binary Decision Diagrams

Craig A. Damon, Daniel Jackson and Somesh Jha
School of Computer Science
Carnegie Mellon University

I can't get no satisfaction
'Cause I try and I try and I try and I try
—Mick Jagger & Keith Richards, 1965

Abstract

Checking a specification in a language based on sets and relations (such as Z) can be reduced to the problem of finding satisfying assignments, or models, of a relational formula. A new method for finding models using ordered binary decision diagrams (BDDs) is presented that appears to scale better than existing methods.

Relational terms are replaced by matrices of boolean formulae. These formulae are then composed to give a boolean translation of the entire relational formula. Throughout, boolean formulae are represented with BDDs; from the resulting BDD, models are easily extracted.

The performance of the BDD method is compared to our previous method based instead on explicit enumeration. The new method performs as well or better on most of our examples, but can also handle specifications that, until now, we have been unable to analyze.

Authors' postal address: School of Computer Science, Carnegie Mellon University, 5000 Forbes Ave., Pittsburgh, PA 15213. Email: {cdamon, dnj, sjha}@cs.cmu.edu. Worldwide web: http://www.cs.cmu.edu/~nitpick. *This research was sponsored in part by a Research Initiation Award from the National Science Foundation (NSF), under grant CCR-9308726, by a grant from the TRW Corporation, and by the Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, and the Advanced Research Projects Agency (ARPA), under grant F33615-93-1-1330.*

Introduction

Many aspects of software are fundamentally relational, which is why a specification language like Z—essentially a syntax for the relational calculus—has many applications. Relational structures are not easy to analyze, however, and tool support has been slow in coming. Recent advances in hardware verification are hard to exploit in the context of software, since they assume that the complexity of a machine arises from the interaction of many simple parts, and not from data structures that are intractable even in isolation.

To express the kinds of properties commonly found in specifications, decidable calculi of relations—such as dynamic logic [Pra78]—appear to be insufficient. It seems, then, that there is no escape from the relational calculus, an undecidable language [TG87, Sch79], and that sound and complete analysis of arbitrary properties cannot be automated.

Automatic checkers are not precluded, however. A *checker* is a tool designed on the assumption that the property being checked does *not* hold. Unlike a *verifier*, a checker indicates unambiguously, with a counterexample, that the property fails to hold; it never casts doubt on the property when the tool's strategy is itself at fault. A checker, like a tester, provides no guarantees of correctness, but, by systematically examining huge numbers of cases, tends to expose flaws more reliably. Testing cannot show the absence of bugs, but equally can rarely show their presence. By exploiting the structure of the property being analyzed, checking can do better.

This paper presents a checking method for relational specifications. It guarantees to find a counterexample if one exists within the *scope* chosen by the user. A scope bounds the number of elements of the primitive types, and thus bounds the size of the relations that can be constructed, and the number of values of a relational variable. The method is based on boolean satisfaction and employs ordered binary decision diagrams [Bry86], a canonical representation of boolean formulae.

Binary decision diagrams have been used extensively in model checking, but never—as far as we know—in the checking of relational specifications. Although our method employs

some tricks (such as building closures by iterative doubling) well known in model checking research, the underlying technique is quite different: instead of a single diagram for the transition relation, we build arrays of diagrams representing relational expressions, and the solution is not obtained by fixpoint calculation.

The new checking method is incorporated in our Nitpick checker. Nitpick analyzes specifications in NP [JD96b], a first-order relational language that is roughly a subset of Z [Spi92]. Scalars, sets and binary relations are the only data types; there is the standard repertoire of operators (such as composition, image and restriction), and a much simplified schema calculus that distinguishes state and transition schemas, making it possible to provide some features lacking in Z (such as arguments to operations, and constant schema components).

In the next section, we explain how checking (or simulating) a specification can be viewed as a satisfaction problem for formulae involving set and relation variables. Then we show how this problem can be reduced to a satisfaction problem involving only boolean variables. A brief overview of BDDs is then given. We explain how our existing method works, and compare its performance to the new BDD method.

1 Checking is Satisfiability

In NP, specifications and properties are logical formulae. A claim that a specification has a given property is itself simply a formula; the claim holds if the formula is valid. This approach has many advantages. It gets rid of turnstiles and the complications of a meta language. It unifies the activities of checking and simulation; both become exercises in satisfiability. And it yields a simple and convenient notion of counterexample that nicely parallels the notion of execution.

To see how this works, consider a trivial specification of allocating a cell in a memory (Figure 1). The first line declares two primitive types representing addresses and contents. Then a *state schema* called *State* is declared. It bundles the declaration of 3 variables—a partial function associating contents with addresses, a fixed set of accessible addresses, and a set of used addresses—and, following the vertical bar, defines the used addresses to be those that have contents associated with them.

Then a *transition schema* called *Alloc* is declared. It models an operation over a state consisting of the 3 variables of the *State* schema. The argument a can be viewed as a parameter of the transition; it represents, in this case, an output: the fresh address generated by the operation.

Primed variables denote values after execution of the operation, so that by relating primed and unprimed variables, it defines a set of transitions. The $\lhd$ symbol is domain restriction: $S \lhd R$ is the relation R with all pairs whose first element is not in the set S removed.

The schema's constraints say that the address a is not currently used; that the set of used addresses is extended by a; and that the memory contents do not change at existing addresses. The mention of *State* before the bar not only incorporates the variable declarations of *State* but also its constraints, on both unprimed and primed copies of the variables.

$$[ADDR, CON]$$

$$\begin{aligned}
&State = [\\
&\quad mem: ADDR \to CON\\
&\quad const\ access: set\ ADDR\\
&\quad used: set\ ADDR\\
&| \quad used = dom\ mem\ ]
\end{aligned}$$

$$\begin{aligned}
&Alloc\ (a: ADDR) = [State \mid\\
&\quad a \notin used\\
&\quad used' = used \cup \{a\}\\
&\quad used \lhd mem' = used \lhd mem\ ]
\end{aligned}$$

$$Safe = [State \mid used \subseteq access]$$

$$AllocOK\ (a: ADDR) :: Safe \land Alloc(a) \Rightarrow Safe'$$

Figure 1: Specification of memory allocation

So *Alloc* is in fact short for the formula

$$\begin{aligned}
&used = dom\ mem \land used' = dom\ mem'\\
&\land access' = access \land a \notin used \land used' = used \cup \{a\}\\
&\land used \lhd mem' = used \lhd mem
\end{aligned}$$

with the first three subformulae coming from the instantiation of *State*.

An execution of an operation is simply an assignment of values to variables that satisfies the formula. If we set the scope to 2 addresses and 2 contents, Nitpick gives names to the elements of the types

$$\begin{aligned}
&ADDR = \{a0, a1\}\\
&CON = \{c0, c1\}
\end{aligned}$$

and generates executions of *Alloc* such as:

$$\begin{aligned}
&mem = \{\}, mem' = \{a0 \mapsto c0\}\\
&mem = \{a0 \mapsto c0\}, mem' = \{a0 \mapsto c0, a1 \mapsto c0\}\\
&mem = \{a0 \mapsto c0\}, mem' = \{a0 \mapsto c0, a1 \mapsto c1\}
\end{aligned}$$

In fact, it also prints out values for *used*, *used'* and *a*, but not *access* or *access'* (which turn out to be irrelevant).

Safe is another state schema: it defines the states in which only accessible address are used. Finally, *AllocOK* is a *claim schema*. *Safe'* is short for a copy of the formula *Safe* with its variables primed; the claim thus asserts that if the invariant *Safe* holds before execution of *Alloc*, then it also holds after. Asked to check this claim, again for the same scope, Nitpick immediately generates the counterexample

$$access = \{\}, mem = \{\}, mem' = \{a1 \mapsto c0\}$$

If we correct the fault in *Alloc* (by requiring, with the addition of the assertion $a \in access'$, that only accessible addresses be chosen), no counterexamples are found. To increase our confidence, we extend the scope to 5 contents and 5 addresses. There are now about 9 billion executions of *Alloc* to be checked, which Nitpick covers in a few seconds.

Executions of a schema are thus assignments of values to the schema's variables for which the schema's formula is true. Counterexamples of a claim are assignments for which the claim's formula are false. So we see that executing implicit specifications and checking claims both reduce to the problem

of finding satisfying assignments, or models, of formulae: either to the formula as given (in the case of execution), or to its negation (in the case of checking).

In practice, of course, the two mechanisms are used differently; one counterexample dooms a claim and so we rarely examine more, whereas we frequently want to examine a number of executions. But in either case, the underlying challenge is the same: to generate satisfying assignments of a relational formula over small scopes as rapidly as possible.

2 Relations to Booleans

Essentially, our language consists of variable declarations and formulae, and some syntactic conveniences for naming and parameterizing formulae. A simplified syntax for declarations and formulae is shown in Figure 2.

Each formula is evaluated in the context of a set of declarations that gives a type to each variable; an assignment can only satisfy the formula if the values it associates with variables meet the type constraints too.

In the type expressions, *given* is the name of some given or primitive type; note there are no higher-order types (such as sets of relations). Functions and relations are not distinct types; as in Z, a declaration of a variable as a function is taken to be a declaration that its type is a relation, along with an additional constraint that its value is functional.

Our starting point for the boolean encoding is the familiar idea that values of variables can be built from bits. A relation is a bit matrix; a set is a bit array; and a scalar is a bit array with exactly one bit set. The dimensions of these structures depend on the scope.

Consider a scope Σ that limits the primitive type t to have at most $\Sigma(t)$ values. A relational variable

$$p: S \leftrightarrow T$$

will have values that are matrices of bits, with $\Sigma(S)$ rows and $\Sigma(T)$ columns, in which $p[ij]$, the bit in the ith row and the jth column, is set to 1 when the ith value of S is related to the jth value of T. Viewing the relation as a graph, each bit corresponds to a possible edge.

Variables are structured just like values, but with boolean variables in place of the bits. The variable p becomes a $\Sigma(S)\times\Sigma(T)$ matrix of variables p_{ij}.

To handle arbitrary terms we generalize further, replacing variables by boolean formulae. Let us write $[t]$ for the indexed set of boolean formulae corresponding to the term t. A relational term P with the same type as p is thus represented in general by an indexed set of formulae $[P]_{ij}$, with $i \in \{0,..., \Sigma(S)\}$ and $j \in \{0,..., \Sigma(T)\}$.

The formulae for a composite term are obtained from the formulae of its components according to the operator. For example, the term

$$P \cup Q$$

where P and Q are relational terms, is represented by the indexed set of formulae $[P \cup Q]_{ij}$ defined by

$$[P \cup Q]_{ij} = [P]_{ij} \vee [Q]_{ij}$$

Finally, elementary relational formulae are encoded by combining the constituent boolean formulae of their terms in the appropriate way. When P and Q are terms of relational type, for example, the formula

$$P \subseteq Q$$

is encoded as the conjunction

$$\bigwedge_{ij} (P_{ij} \Rightarrow Q_{ij})$$

Relational terms are thus translated into indexed sets of boolean formulae, and relational formulae into boolean formulae. Given the translations of the subformulae, the entire formula is then obtained simply by combining the subformulae with the logical operators.

The encoding of scalars is a more interesting problem. To match the encoding of sets and relations, it makes sense to represent a scalar term by an array of boolean formulae, with the interpretation that the ith bit is true when the scalar has the value of the ith element of the given type. Testing membership of a scalar in a set, for example, then reduces to checking that if the ith bit of the scalar is set to 1, the ith bit of the set is set to 1 also.

A scalar variable is a special instance of a term, so it will also be encoded as an array. We could simply introduce a boolean variable for each element of the array, but this raises a

decl ::=	
var type	
type ::=	
given	*scalar type*
set given	*set type*
given $\rightarrow$ *given*	*relation type, a partial function*
given $\leftrightarrow$ *given*	*relation type*
term ::=	
var	
{}	*empty set or relation*
L	*universal set or relation*
Id	*identity relation*
term $\cup$ *term*	*set or relation union*
term $\cap$ *term*	*set or relation intersection*
term $\setminus$ *term*	*set or relation difference*
term $\lhd$ *term*	*domain restriction*
{ *term* }	*set constructor*
dom term	*domain of relation*
term $^\sim$	*relational transpose*
term $+$	*transitive closure*
term ; *term*	*relational composition*
{ *term* $\mapsto$ *term* }	*relation constructor*
formula ::=	
term $\in$ *term*	*set membership*
term $=$ *term*	*equality of scalars, sets or relations*
term $\subseteq$ *term*	*subset or subrelation*
$\neg$ *formula*	
formula $\wedge$ *formula*	
formula $\vee$ *formula*	

Figure 2: Simplified syntax of formulae and declarations

scalar, $x : T$
  represent as array of formulae $[x]_i$ *with* $0 \le i < \Sigma(T)$
  over variables $\{x_0 \ldots x_r\}$ *where* $r = \lceil log\ \Sigma(T) \rceil - 1$
  $[x]_i$ *asserts that for the binary number* $b = x_0 x_1 \ldots x_r$
  $b = i$ (*when* $0 \le i < \Sigma(T) - 1$)
  $b \ge i$ (*when* $i = \Sigma(T) - 1$)

set, s : *set* T
  represent as array of variables s_i *with* $0 \le i < \Sigma(T)$

relation, $r : S \leftrightarrow T$
  represent as matrix of variables r_{ij}
  with $0 \le i < \Sigma(S)$ *and* $0 \le j < \Sigma(T)$

partial function, $f : S \to T$
  represent as matrix of formulae $[f]_{ij}$
  with $0 \le i < \Sigma(S)$ *and* $0 \le j < \Sigma(T)$
  using variables $\{f_{ik}\}$ *with* $0 \le k < r$ *where* $r = \lceil log\ (\Sigma(T)+1) \rceil$
  $[f]_{ij}$ *asserts that* j *in binary is* $f_{i0} f_{i1} \ldots f_r$

Figure 3: Encoding of variables

problem. For the array to represent a scalar faithfully, exactly one element of the array must be true. In other words, since scalars and sets share the same array structure, we must ensure that scalars correspond to singleton sets. Ensuring this is not difficult; we can just keep around a *restriction formula* that asserts this property and is conjoined to the formula at the end of the translation.

Although straightforward, this approach is wasteful. A scalar

x: T

has $\Sigma(T)$ possible values, and should only require $log\ \Sigma(T)$ boolean variables. The cost of solving boolean satisfaction by any method is often extremely sensitive to the number of variables, increasing exponentially in the worst case.

A far superior approach is to encode the scalar as a binary number. For $\Sigma(T) = 4$, for example, we introduce two boolean variables, x_0 and x_1, and we view x as the binary number $x_1 x_0$. The ith value of x is then the formula representing the binary encoding of i. It is this formula that is placed in the ith position of the array. So instead of the naive encoding of x which would give an array

$$\langle\ x_0, x_1, x_2, x_3\ \rangle$$

of four boolean variables and would require the restriction formula

$$\begin{array}{l} x_0 \vee x_1 \vee x_2 \vee x_3 \\ \wedge\ x_0 \Leftrightarrow \neg x_1 \wedge \neg x_2 \wedge \neg x_3 \\ \wedge\ x_1 \Leftrightarrow \neg x_0 \wedge \neg x_2 \wedge \neg x_3 \\ \wedge\ x_2 \Leftrightarrow \neg x_1 \wedge \neg x_0 \wedge \neg x_3 \\ \wedge\ x_3 \Leftrightarrow \neg x_1 \wedge \neg x_2 \wedge \neg x_0 \end{array}$$

we have instead the array

$$\langle\ \neg x_1 \wedge \neg x_0,\ \neg x_1 \wedge x_0,\ x_1 \wedge \neg x_0,\ x_1 \wedge x_0\ \rangle$$

which expresses exactly the possible values of x with only two boolean variables. When the scope is not a power of 2, there

Constants
  $[\{\}]_{ij} = 0$
  $[L]_{ij} = 1$
  $[Id]_{ij} = (i = j)$
Terms
  $[p \cup q]_{ij} = p_{ij} \vee q_{ij}$
  $[p \cap q]_{ij} = p_{ij} \wedge q_{ij}$
  $[p \setminus q]_{ij} = p_{ij} \wedge \neg q_{ij}$
  $[s \lhd p]_{ij} = [p]_{ij} \wedge [s]i$
  $[\{x\}]_i = [x]_i$
  $[dom\ p]_{ij} = \bigvee_j [p]_{ij}$
  $[p^\sim]_{ij} = [p]_{ji}$
  $[p\ ;\ q] = \bigvee_k . [p]_{ik} \wedge [q]kj$
  $[\{x \mapsto y\}] = [x]_i \wedge [y]_j$
Formulae
  $[e \in s] = \bigwedge_i [e]_i \Rightarrow [s]_i$
  $[p = q] = \bigwedge_{ij} [p]_{ij} \Leftrightarrow [q]_{ij}$
  $[p \subseteq q] = \bigwedge_{ij} [p]_{ij} \Rightarrow [q]_{ij}$
  $[\neg p] = \neg [p]$
  $[p \wedge q] = [p] \wedge [q]$
  $[p \vee q] = [p] \vee [q]$

Figure 4: Encoding of terms and formulae

are more binary numbers than values of the scalar, so we collapse all numbers above the last scalar index together. For example, if x has 3 values, its array encoding is

$$\langle\ \neg x_1 \wedge \neg x_0,\ \neg x_1 \wedge x_0,\ (x_1 \wedge \neg x_0) \vee (x_1 \wedge x_0)\ \rangle$$

in which the three values of the scalar correspond to the binary number $x_1 x_0$ being 0, 1, and, for the third, either 2 or 3.

Functions are encoded with the same scheme. We could represent functions as arbitrary boolean matrices, with a restriction formula constraining there to be at most a single 1 in each row. Instead, we view a function as an array of scalars, with the scalars encoded as described above; there are then fewer variables and no restriction formula. (Restriction formulae are, however, used elsewhere. NP declarations can be qualified by keywords: a relation can be constrained to be total or onto, for example, a function can be made bijective, and a set can be limited to at most one element.)

Figure 3 gives formal definitions of the encoding of variables and operators. For overloaded operators, the relational version is given; the set and scalar versions are almost identical.

Transitive closure of a relational term is calculated by iterative squaring [BC+92]; instead of the straightforward iteration

$$R^+ = R \cup R^2 \cup R^3 \cup \ldots$$

we generate the series

$$\begin{array}{l} R_0 = R \\ R_{i+1} = R_i \cup {R_i}^2 \end{array}$$

in which, taking a graph view, R_i associates nodes that are connected by a path of at most 2^i edges. By reducing the number of steps to convergence (from the scope to its logarithm), and

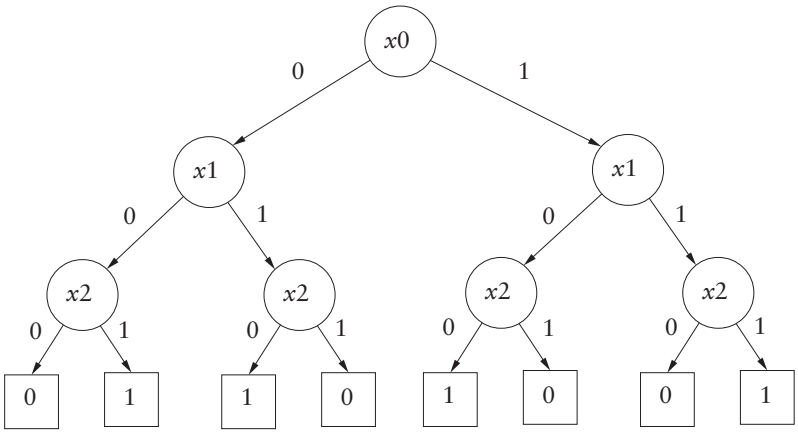

Figure 5: Binary decision tree

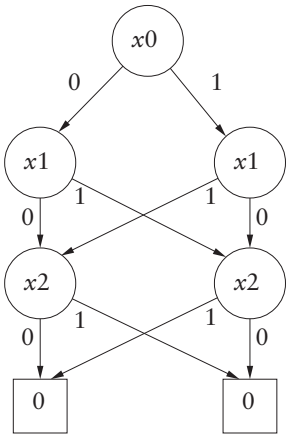

Figure 6: Binary decision diagram

thus the number of BDD operations performed, this method can reduce the cost of calculating closure significantly.

3 Binary Decision Diagrams

The efficiency of our method is due largely to a succinct representation of boolean formulae. To see how it works, consider first the binary tree (Figure 5) that encodes the formula

$$x_0 \oplus x_1 \oplus x_2$$

where $\oplus$ is exclusive or. To determine the value of the formula for a given assignment, start at the root of the tree and, at each decision node, take the branch corresponding to the value of the variable whose name labels the node. The leaf label then gives the value of the formula.

Viewed as a function, this formula computes the parity of the bits, so any path containing an odd number of 1's yields a 1 at the leaf. The subtrees reached by following the paths $\langle 0, 1\rangle$ and $\langle 1, 0\rangle$ are thus identical, as are the subtrees following s $\langle 0, 0\rangle$ and $\langle 1, 1\rangle$. These commonalities can be exploited by using a graph instead of a tree, with common subtrees shared (Figure 6). The number of nodes shrinks from 15 to 7. In general, the binary tree for the parity of n bits has $2^{n+1} - 1$ nodes, where the BDD for the same function has only $2n+1$ nodes. In some cases then, BDDs can be exponentially more succinct.

The BDD for a formula can be constructed compositionally from its subformulae. Let $|f|$ be the size of the BDD representing the formula f. Then the BDDs for $f \vee g$ and $f \wedge g$ can be computed from the BDDs for f and g in time $O(|f| \times |g|)$, the BDD for $\neg f$ in time $O(|f|)$ and the BDDs for $\exists x_i.f$ and $\forall x_i.f$ in time $O(|f|^2)$.

The ordering of the variables in two BDDs must be compatible if they are to be combined efficiently, so a single, total ordering on all variables is required. This gives BDDs a very desirable property: the representation is canonical, so if two formulae have the same BDD, they must represent the same boolean function, or, equivalently, have the same set of models.

The choice of variable ordering is critical. To see why, consider the formula

$$\wedge\, i \in \{1, n\}.\, x_i = x_i'$$

With the variable ordering

$$x_1 < x_1' < x_2 < x_2' < ... < x_n < x_n'$$

the BDD has size $3n+2$. If the unprimed variables all fall before the primed variables in the ordering, however, the BDD has a size greater than 2^n. Intuitively, the reason is that any two distinct assignments to the unprimed variables can be distinguished by some assignment to the primed variables. The subgraph representing the unprimed variables must therefore be a tree whose leaves list all possible assignments, and no sharing is possible until some primed variable has been included.

Unfortunately, there are functions whose BDDs are exponential irrespective of variable ordering, but luckily these are rarely encountered in practice.

We experimented with a variety of variable orderings for checking relational formulae. The naive ordering groups the boolean variables of each relation together. A better ordering interleaves rows: the boolean variables of the first row of the first relational variable, then the variables of the first row of the second relational variable, and so on. A third ordering we tried interleaves the variables of each relational variable symmetrically about the diagonal of the matrix. Surprisingly, the choice of ordering appears to make little difference; we settled on row interleaving because it performs marginally better than the naive ordering on average, and because the symmetrical interleaving makes little sense when scalars are encoded as binary numbers. We suspect, however, that there is a much better ordering heuristic that takes account of the grouping of relational variables into subformulae.

4 Explicit Nitpick

The BDD version of Nitpick is currently a prototype. A stable version based on explicit enumeration was implemented a year ago. It works by enumerating cases, each of which is an assignments of values to variables. Even for small scopes, the number of cases can be huge. A relation over 3 elements, for example, has 512 possible values, so for 3 relation variables, there are about 134 million cases to check.

To make checking faster, Nitpick employs 3 reduction mechanisms that trim the search space. All are sound: if there is a counterexample to a property within the given scope, then one will still be found (but not necessariy the same one).

Derived variable detection [JD95] is a static analysis that identifies constructive definitions in a specification, allowing the values of a variable to be obtained from other variables' values, rather than being enumerated independently.

Short circuiting [DJ96] exploits the fact that sometimes a partial assignment (that is, an assignment of values to a subset of the variables) determines whether the property holds, making enumeration of the remaining variables unnecessary. The order in which variables are enumerated determines how often this happens.

Isomorph elimination [JJD96] avoids the generation of cases that have the same shape as cases already examined (and are thus guaranteed not to be counterexamples if no counterexample has yet been seen). The isomorphism arises both from the structure of the underlying relation values and the type structure of the property being checked. All three mechanisms take advantage of the structure of the property being checked, and in combination produce dramatic reductions: often a factor of a million, and sometimes much more.

Isomorph elimination is particularly useful for simulation. In addition to reducing the search time, it filters out uninteresting variations on cases. For example, simulating the execution of *Alloc* (as described in Section 1) for 3 addresses and 3 contents yields only the 3 cases given, excluding 9 extra cases such as

$$mem = \{\}, mem' = \{a1 \mapsto c1\}$$

which is just a relabelling of the first case given:

$$mem = \{\}, mem' = \{a0 \mapsto c0\}$$

In our experience, many errors are found in surprisingly small scopes: often 3 or 4 elements of each type suffice. With such scopes, Nitpick can typically check within minutes (or even seconds) a specification with a handful of variables (not counting redundant variables, which although commonly used in Z specifications, add no cost to the checking, being eliminated by derived variable analysis).

Nitpick/NP has been applied to a number of small case studies. We have analyzed specifications of folder/aliasing structure in the Macintosh finder; a mobile IP protocol; plain old telephone service; and the inheritance of paragraph styles in Microsoft Word. We have taught NP to students taking the masters of software engineering degree at CMU, who have then used the notation and the Nitpick checker in a class project.

Our specification of the Macintosh finder, for example, has two function variables (one for the folder hierarchy and one for aliasing links), two set variables (files and folders), variables representing the drive and trash icons, and two additional redundant variables. For checking a property of the *move* operation for 4 objects (files or folders), there are about 1.6 billion cases to consider. The explicit version of Nitpick reduces this space to a space of 114,047 cases, which it exhausts in 12 seconds (on a Power Macintosh 7100).

5 Results

We ran 3 classes of examples to compare the new, BDD-based mechanism to the explicit one. The comparison is a little tricky. The explicit checker halts when it finds the first model (or counterexample), although it can be instructed to continue, displaying models as it finds them. The new method constructs a BDD for the formulae that contains all models; until this BDD is complete, no information is available. Extracting the first model from the BDD costs almost nothing, but extracting all models is often not feasible, since, lacking isomorph elimination, the BDD method generates many more models than the explicit method.

For the explicit method, we measured the time to the first model and the time to search the entire space; these coincide when there is no model. We also give the size of the space searched, and the size of the actual space (which, because of the reductions, is usually much larger).

For the new method, we measured the size of the final BDD, which is 1 when the formula or its negation is valid, and the size of the largest intermediate BDD. The time for constructing the final BDD is given, but not the time for extracting all models, which has little practical consequence.

Simple Theorems

First, we checked a collection of simple theorems (Figure 7, Table 1). For all of these, the new approach performed better, usually by at least an order of magnitude. The biggest differences involved simple formulae with 3 relational variables. Checking associativity of composition

$$p ; (q ; r) = (p ; q) ; r$$

took about a second rather than 50 hours (estimated from partial progress), and checking the Dedekind theorem

$$p ; q \cap r \subseteq (p \cap r ; q^{\sim}) ; (q \cap p^{\sim} ; r)$$

took 12 seconds rather than an estimated 30 hours. The explicit mechanism works badly on these examples because the spaces are huge (three 4×4 relations give 3×10^{14} cases); because such simple formulae lack structure that the reduction mechanisms can exploit; and because the formulae are valid, so the entire space must be searched.

Neither method can check the monotonicity of composition (*Mono*), which, unlike the other formulae given, requires 4 relational variables. The first theorem about the generators of a relation (*Gen*1) is actually invalid; its hypothesis should require r to be transitive. We noticed this error when we checked the formula for a scope of 4. It is an interesting example, first because it is rare for such an error not to be found on a smaller scope, and second because it shows, by comparison with the corrected theorem (*Gen2*), that the presence of many

$[S]$

$R = [p, q, r, s: S \leftrightarrow S; t, u: set\ S]$
$F = [f, g, h: S \to S; r, s: S \leftrightarrow S]$

$Connex :: [R \mid Un \subseteq (r \cup r^\sim) \Leftrightarrow (Un \setminus r) \subseteq r^\sim]$
$Assoc :: [R \mid p ; (q ; r) = (p ; q) ; r]$
$Tarski :: [R \mid r != \{\} \Rightarrow Un ; r ; Un = Un]$
$Schroder:: [R \mid p ; q \subseteq r \Leftrightarrow p^\sim ; (Un \setminus r) \subseteq (Un \setminus q)]$
$Dedekind :: [R \mid p ; q \cap r \subseteq (p \cap r ; q^\sim) ; (q \cap p^\sim ; r)]$
$Mono :: [R \mid p \subseteq q \wedge r \subseteq s \Rightarrow (p ; r) \subseteq (q ; s)]$
$Distrib :: [R \mid p ; (q \cup r) = p ; q \cup p ; r]$
$Gen1 :: [R \mid p^+ = r^+ \wedge p \subseteq r \Rightarrow r \cap (Un \setminus r ; r) \subseteq p]$
$Gen2 :: [R \mid p^+ = r^+ = r \wedge p \subseteq r \Rightarrow r \cap (Un \setminus r ; r) \subseteq p]$
$Clo1 :: [R \mid r^{*\sim} = r^{\sim *}]$
$Clo2 :: [R \mid (p \cup q)^* = (p^* ; q)^* ; p^*]$
$Clo3 :: [R \mid p^* ; q^* \subseteq (p \cup q)^*]$
$Fun1 :: [F \mid f ; g \cap h = (f \cap (h ; g^\sim)) ; g]$
$Fun2 :: [F \mid f \subseteq g \wedge g ; Un \subseteq f ; Un \Rightarrow f = g]$
$Fun3 :: [F \mid f ; (Un \setminus r) = f ; Un \cap (Un \setminus f ; r)]$

Figure 7: Some relational claims

counterexamples (reflected in the final BDD size) has little effect on the time taken.

Case Study Specifications

Second, we compared the two mechanisms on some case studies: *Phone*, a toy version of a specification of conference connections in a phone switch [JJD96]; *Finder*, an analysis of the move operation in the Macintosh finder [JD95]; and *Style*, an analysis of paragraph style inheritance hierarchy in Microsoft Word [JD96a].

The results are shown in Table 2. A scope of k for *Phone* means that k phones were considered, and for *Finder*, it means k objects (files or folders); a scope of $i \times j$ for *Style* means i paragraph styles and j formats. *Phone-B* and *Phone-C* refer to two different claims in the *Phone* specification (corresponding to the invariants *invB* and *invC* in [JJD96]).

On these examples, the mechanisms are comparable. The explicit mechanism is sometimes preferable because it finds the first counterexample in less time than it takes to construct the final BDD. *Phone-C* is checked faster than *Phone-B* by the explicit method because its type structure admits greater isomorphism reduction. This inherent symmetry may explain why the BDDs for *Phone-C* are smaller. It is puzzling that, despite this, *Phone-C*'s BDD takes longer to construct.

Surprisingly, the BDD mechanism does very poorly on the *Finder* example. We do not know why. We thought that perhaps the BDD mechanism would perform better if the claim being checked were valid and thus had no counterexamples, but checking such a claim did not support the hypothesis.

Throughout these examples, the binary number encoding of scalars (described in section 2) was applied. To test its efficacy, we ran the same examples with the naive encoding. For scopes of 3, the time to construct the final BDD increased by a factor of 10; for larger scopes, the examples could not even be run (for lack of memory).

Claim, $\Sigma(S)$		*BDD method*			*explicit method*		
		last size	*max size*	*time*	*time*	*#cases reduced*	*#cases full*
Connex	3	1	24	0s	0s	117	512
	4	1	57	0s	0.5s	4k	64k
Assoc	3	1	857	0s	38s	465k	1.3e8
	4	1	9k	1s	~50h	~6e9	2.8e14
Tarski	3	1	90	0s	0s	36	512
	4	1	300	0s	0s	317	64k
Schroder	3	1	7k	1s	120s	1.6e6	1.3e8
	4	?	?	?	~20h	~8e8	2.8e14
D'kind	3	1	3k	0.5s	145s	1.6e6	1.3e8
	4	1	130k	12s	~20h	~8e8	2.8e14
Mono	3	?	?	?	~3h	~2e8	6.9e10
	4	?	?	?	~30h	~2e9	1.8e19
Distrib	3	1	129	0s	140s	1.8e6	1.3e8
	4	1	400	0s	~10h	~4e8	2.8e14
Gen1	3	1	297	0.5s	6s	81k	262k
	4	5k	13k	6s	~30m	~2e7	4.3e9
Gen2	3	1	300	0s	3s	26k	262k
	4	5k	13k	6s	~1h	~5e7	4.3e9
Clo1	3	1	29	0s	0s	125	512
	4	1	150	0s	0.5s	4k	64k
Clo2	3	1	115	0.5s	7s	81k	262k
	4	1	800	11s	~2h	~1e8	4.3e9
Clo3	3	1	59	0s	6s	76k	262k
	4	1	600	4s	~1h	~5e7	4.3e9
Fun1	3	1	150	0s	0.5s	4k	262k
	4	1	550	0s	12s	140k	2.4e8
Fun2	3	1	27	0s	0s	195	4k
	4	1	64	0s	0.5s	2k	400k
Fun3	3	1	79	0s	0.5s	732	32k
	4	1	260	0s	3s	30k	4.1e7

Table 1: Performance results for claims of Figure 7
? indicates exhaustion of memory
~ marks an estimated figure

A Previously Uncheckable Specification

Third, we ran the new BDD mechanism on some examples that could barely be handled by the explicit mechanism. The specification given in the appendix is based on a Z specification of railway switching [Kin94] developed by Praxis, a British software company. Our aim was to capture some of the complexity of the original specification in a more succinct form, abstracting away many important concerns but exposing some fundamental issues. The explicit mechanism is barely able to check this specification. The state space is huge—ignoring derived variables, there are 7 sets, 2 functions and 2 relations—and the reduction mechanisms do not perform well. Although the counterexamples to the invalid claims were found very quickly, checking the valid claim (*AllocSafe*1) was barely feasible. The new method, however, handles the specification easily (Table 3).

Example, Σ		*BDD method*			*explicit method*			
		last size	*max size*	*time*	*time first*	*time all*	*#cases redcd*	*#cases full*
Finder	3	1	2k	0.5s	0.5s	0.5s	3k	2.6e6
	4	1k	43k	82s	1s	12s	114k	1.6e9
Phone-B	3	1k	1k	1.5s	0s	0.5s	2k	300k
	4	36k	34k	6.5s	0.5s	20s	150k	6.5E8
Phone-C	3	1k	1k	0.5s	0s	0.5	1k	300k
	4	28k	31k	19s	0.5s	4s	32k	6.5E8
Style	2×3	100	400	1.5s	1s	1.5s	7k	9.2E8
	3×3	150	700	1.5s	1s	2s	10k	2.9e10
	3×4	3k	5k	14s	8.5s	145s	1.1e6	1.7e14
	4×4	4k	8k	29s	11s	186s	1.5e6	6.1e15

Table 2: Performance results for case studies

Claim	*BDD method*			*explicit method*
	last size	*max size*	*time*	*time*
AllocSafe0	1	5k	4s	11h
AllocSafe1	4k	6k	2s	10s
AllocSafe2	4k	10k	2s	10m
AllocSafe3	5k	11k	2s	50s

Table 3: Performance results for reservation scheme. For all claims, $\Sigma(RESOURCE) = \Sigma(USER) = 3$, and the full space for explicit enumeration has 5E14 cases. Times given for the explicit method are for finding the first counterexample, or, if none, covering the space

6 Related Work

Many tools are available for analyzing interacting state machines: model checkers, reachability analyzers, simulators, and so on. Fewer tools have been developed for software specifications that involve relational structures. Most offer only type checking, syntax analysis, cross-referencing and pretty printing. Many incorporate some kind of simulation that, by restricting the language to an executable subset, allows operations to be tested on individual cases. Some provide theorem proving, either just for normalizing formulae, or as a full-fledged inference system. As far as we know, however, Nitpick is the only automatic checker for software specifications.

Running a simulator on a large set of test cases might seem to be like checking, but in practice it is very different. First, simulators tend not to provide case generation, so the user must construct the cases manually. Second, a simulator, unlike Nitpick, does not exploit the structure of the property being checked, and will therefore examine a vast number of cases needlessly. A reduction factor of a million, which is commonly achieved by the explicit version of Nitpick, means that Nitpick can effectively check a million times as many cases as a simulator (and in fact probably more, because the checking of individual cases is highly optimized in Nitpick). Third, simulators cannot handle implicit specifications, and can thus only check properties of operations and not of state invariants.

Tools that search for satisfying assignments have been developed for other purposes. The FINDER tool [Sla94], for example, uses backtracking to find models of a logic with functions and equality. Much of NP can be translated into FINDER's input language, and, with some tweaking of the input and the addition of some extra constraints and directives, can be analyzed with a performance that competes with Nitpick. Since transitive closure cannot be axiomatized in this logic, however, most of our examples cannot be translated at all.

Almost all our case studies seem to require transitive closure. Since the explicit boolean formula for a transitive closure grows exponentially with scope, it seems that translating NP checks directly into boolean formulae is not feasible, so there is no obvious application of boolean satisfiability algorithms. Overcoming this obstacle would be most worthwhile, since heuristic satisfiability algorithms based on local search (such as [SLM92]) might be capable of finding counterexamples to claims in much larger spaces.

Binary decision diagrams have many applications in hardware verification and synthesis; for a survey, see [Bry92]. The discovery that a relatively small BDD can implicitly characterize a huge set of states satisfying a temporal formula [BC+92] was perhaps the single advance that made model checking practical. The SMV model checker, based on this method, has recently been been applied in the analysis of software [AG93, WV95], but is limited to finite state machines over unstructured data. Many tools requiring simplification of large boolean formulae can benefit from BDDs; they have been used in the analysis of fault trees [CM93], for example, and are being incorporated into theorem provers such as PVS [OR+95].

Specialized tools have been developed for relations: Ralf, a theorem prover [BH94] and Relview, a tool for manipulating and displaying large relations [BS93]. Relview has been applied to the checking of Petri nets [BKU96]; unlike Nitpick, it is designed to handle a few huge relations, and not a huge number of small relations.

It is perhaps worth mentioning that although there has been much work on 'relational methods' and 'relational specifications', little of it is directly relevant to our project. Our specifications are relational in *two* senses: in the relation of states to one another and in the relations comprising the states themselves. Most work on relational methods is concerned only with relations in the first sense. For an excellent introduction to the theory of relations, with many applications, see [SS93].

7 Discussion and Future Work

The BDD-based method has many advantages over our previous method based on explicit enumeration. It works much faster on many examples, and handles examples that the explicit method cannot handle at all. It is also easier to implement, since BDDs can be treated, by and large, as an abstract data type. We have recently added quantification to NP; its implementation for the BDD method was much more straightforward.

On the other hand, the explicit method is better in some respects. Unlike the BDD method, the more counterexamples there are, the faster it works. Often only a tiny fraction of the space is searched before a counterexample is found; the BDD method, in contrast, gives no information until the entire BDD is constructed. Moreover, the cost of checking a case in the explicit method varies linearly with the length of the formula. In a typical claim, checking a single assignment involves executing hundreds of relational operations; adding a transitive closure will typically require only a few extra compositions to be performed. In the BDD method, however, a closure can increase the size of the intermediate BDD significantly.

Because an isomorph-free set of relations is cached before checking, the explicit method allocates no memory during checking. Knowing the structure of the search tree in advance is very useful: it allows progress to be estimated easily (though not always accurately), and should make parallelization straightforward.

We have experimented with various refinements of the BDD method. For example, we implemented a symmetry-breaking scheme based on some ideas taken from out previous work [JJD96]. The set of variables is partitioned according to the formula's most general typing. For each equivalence class (which corresponds to a type), the value of one variable can be fixed with respect to permutations. A scalar is fixed simply by setting its value; a set is fixed by always drawing the elements in a fixed order, so that, for example, a 2 element set must have the first two elements in the order. Relations are harder to fix; one can impose a restriction that has the effect of ordering the left or right side of the relation. In practice, however, fixing scalars and sets works best because variables can be eliminated, even though fixing a relation usually gives a greater reduction in the number of models. On the examples in the paper, this refinement reduces the time taken by factor of 10 on average.

We plan to port Nitpick to Unix. Our current implementation is for the Macintosh and suffers from its lack of memory, a severe handicap for any tool using BDDs. Increasing physical memory ten times should have a dramatic effect. With only 16MB of memory, we were unable to run the *Phone* example (see Table 2) on 5 phones of more; a handcoded version of the formula, run on a workstation with 256MB, easily scaled to 10 phones.

Paradoxically, an apparent advantage of the BDD method may turn out to be its downfall. Translating relational formulae into boolean formulae shields the checker from the complexity of the relational notation, resulting in a checking method that is, conceptually at least, much simpler than the explicit method. This simplicity, however, is gained at the expense of losing the relational structure. Consequently, refinements seem to be much harder to implement. Moreover, the effects of a refinement are much harder to predict, not only because of a loss of relational structure, but also because a BDD's size does not reflect the number of models it represents. Variable order is critical, but whether there exists a good ordering heuristic—based on the structure of the relational specification alone—remains to be seen.

Acknowledgments

The specification given in the appendix was developed jointly with Michael Jackson.

Appendix: Reservation System Specification

The specification of Figure 8 describes, rather abstractly, some of the key properties of a scheme for managing a pool of resources amongst several users. Many systems, including railway switching, include such a scheme.

The *Bookings* schema relates users and resources. Resources are partitioned into three sets: *free* resources have not been reserved by any user; *pending* resources have been reserved, but are not yet available for use; *granted* resources are available for use by the users who reserved them.

The *Resources* schema describes properties of the resources. Resources are partitioned into *open* and *closed* sets, to model the notion that a resource may be physically unavailable. Two relations are defined that characterize circumstances in which the simultaneous use of two resources may not be possible or desirable. *incons* associates resources that cannot be used together because of logical consistency, perhaps because the resources share some physical component. If paths in a railway are modelled as resources, for example, two paths that require a common switch to be set in different ways would be inconsistent. *overlap* associates resources that can but should not be used together, such as two paths that intersect or come dangerously close to one another. Both of these relations are antisymmetric and reflexive.

The *Users* schema describes the state of usage of the resources by users. In this simplified version, each resource is used by at most one user.

The *Allocate* schema describes an operation in which a resource r is allocated to a user u. The resource must have been reserved by that user, it must be both pending and open, and may not conflict with another reserved resource.

The 4 claims assert that this operation preserved various invariants: that there are no conflicts amongst reserved resources (*AllocSafe0*); that resources that conflict with open resources are not granted for use (*AllocSafe1*); that there are no inconsistencies amongst resources in current use (*AllocSafe2*); and that there are no inconsistencies or overlaps amongst resources in current use (*AllocSafe3*).

None of the claims except *AllocSafe0* hold. The performance of the checker on these claims is given in Table 3 in the body of the paper.

```
[User, Resource]

Bookings = [
        reservedBy: Resource → User
        pending, granted, free: RN part Resource
        reserved: set Resource
|       reserved = pending ∪ granted
        reserved = dom reservedBy ]

antirefsym (r: Resource ↔ Resource) =
        [ | r~ = r and r ∩ Id = {}]

Resources = [
        open, closed: RS part Resource
        const overlap, incons, excludes: Resource ↔ Resource
        excluded: set Resource
|       antirefsym (overlap)
        antirefsym (incons)
        excludes = incons ∪ overlap
        incons.open ⊆ closed
        excluded = excludes.open ]

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 ∪ {r}
        reservedBy' = reservedBy
        usedBy' = usedBy ∪ {r ↦ u} ]

AllocSafe0 (r: Resource; u: User) :: [
        Resources Users Bookings
|       Allocate (r) ∧ excludes.reserved ∩ reserved = {}
        ⇒ excludes.reserved' ∩ reserved' = {} ]

AllocSafe1 (r: Resource; u: User) :: [
        Resources Users Bookings
|       Allocate (r) ∧ excluded ∩ granted = {}
        ⇒ excluded' ∩ granted' = {} ]

AllocSafe2 (r: Resource; u: User) :: [
        Resources Users Bookings
|       Allocate (r) ∧ incons.used ∩ used = {}
        ⇒ incons.used' ∩ used' = {} ]

AllocSafe3 (r: Resource; u: User) :: [
        Resources Users Bookings
|       Allocate (r) ∧ (incons ∪ overlap).used ∩ used = {}
        ⇒ (incons ∪ overlap).used' ∩ used' = {} ]
```

Figure 8: Specification of reservation scheme

References

[AG93] J.M. Atlee and J.D. Gannon. State-based model checking of event-driven systems requirements. *IEEE Transactions on Software Engineering*, January 1993.

[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.

[BH94] Rudolf Berghammer and Claudia Hattensperger. *Computer-Aided Manipulation of Relational Expressions and Formulae Using RALF*. Technical Report, Institut fur Informatik und Praktische Mathematik, Christian-Albrechts Universitat Zu Kiel, Kiel, Germany, 1994.

[BKU96] Rudolf Berghammer, Burhard von Karger and Christiane Ulke. Relation-algebraic analysis of Petri nets using RELVIEW. *Second International Workshop on Tools and Algorithms for the Construction and Analysis of Software (TACAS '96)*, Passau, Germany, March 1996, Lecture Notes in Computer Science 1055, Springer, eds. T. Margaria and Bernhard Steffen, pp. 49–69.

[Bry86] R.E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. on Computers*, C-35(8), 1986.

[Bry92] R. E. Bryant. Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams. *ACM Computing Surveys*, vol. 24, no. 3, pp. 293–318, 1992.

[BS93] R. Berghammer and G. Schmidt. RELVIEW: a computer system for the manipulation of relations. *Proc. 3rd Conference on Algebraic Methodology and Software Technology (AMAST 93)*, University of Twente, Netherlands, June 1993. Workshops in Computing, Springer, pp. 405–406.

[CM93] O. Coudert and J.C. Madre. Fault tree analysis: 10^{20} prime implicants and beyond. *Proc. Annual Symposium on Reliability*. Atlanta, NC, January 1993.

[DJ96] Craig A.Damon and Daniel Jackson. Efficient Search as a Means of Executing Specifications. *Proc. Second International Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '96),* Passau, Germany, March 1996, Lecture Notes in Computer Science 1055, Springer, eds. T. Margaria and Bernhard Steffen, pp. 70–86.

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

[JD96a] D. Jackson and C. A. Damon. Elements of Style: Analyzing a Software Design Feature with a Counterexample Detector. *Proc. International Symposium on Software Testing and Analysis*, San Diego, CA, January 1996.

[JD96b] Daniel Jackson and Craig A. Damon. *Nitpick: A Checker for Software Specifications (Reference Manual)*. Technical Report CMU-CS-96-109, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, January 1996.

[JJD96] Daniel Jackson, Somesh Jha and Craig A. Damon. Faster Checking of Software Specifications by Eliminating Isomorphs. *Proc. ACM Symposium on Principles of Programming Languages*, St. Petersburg Beach, FL, January 1996.

[Kin94] Trevor King et al. *Logical elements of interlocking systems. Part A: Formal specification of conceptual foundation of British Rail route signalling.* Draft for academic use. Ref: S.P0339.41.1. Network South East, UK, November 1994.

[OR+95] Sam Owre, John Rushby, Natarajan Shankar and Friedrich von Henke. Formal Verification for Fault-Tolerant Architectures: Prolegomena to the Design of PVS. *IEEE Transactions on Software Engineering*, 21(2), February 1995, pp. 107–125.

[Pra78] V.R. Pratt. A practical decision method for propositional dynamic logic. *Proc. 10th Annual ACM Symposium on Theory of Computing*, pp. 326–337, San Diego, CA, May 1978.

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

[Sla94] John K. Slaney. FINDER: Finite Domain Enumerator, System Description. *Proc. 12th International Conference on Automated Deduction*, Lecture Notes in Artificial Intelligence series, Springer Verlag, Berlin, 1994, pp. 798–801.

[SLM92] Bart Selman, Hector Levesque and David Mitchell. A new method for solving hard satisfiability problems. *Proc. 10th National Conference on Artificial Intelligence (AAAI-92)*, San Jose, CA, July 1992, pp. 440–446.

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

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

[TG87] Alfred Tarski and Steven Givant. *A formalization of set theory without variables*. Colloquium Publications, vol. 41, American Mathematical Society. Providence, Rhode Island, 1987.

[WV95] Jeannette Wing and Mandana Vaziri-Farahani. Model checking software systems: a case study. *Proc. 3rd ACM SIGSOFT Conference on Foundations of Software Engineering,* Washington, DC, August 1995.