Isomorph-free Model Enumeration:
A New Method for Checking Relational Specifications

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

So as fast as you can,
Think of something to do!
You will have to get rid of
Thing One and Thing Two!
—Dr. Seuss (1957)

Abstract

Software specifications often involve data structures with huge numbers of values, and consequently cannot be checked using standard state exploration or model checking techniques. Data structures can be expressed with binary relations, and operations over such structures can be expressed as formulae involving relational variables. Checking properties such as preservation of an invariant thus reduces to determining the validity of a formula, or, equivalently, finding a model (of the formula's negation).

A new method for finding relational models is presented. It exploits the permutation invariance of models—if two interpretations are isomorphic, then neither is a model or both are—by partitioning the space into equivalence classes of symmetrical interpretations. Representatives of these classes are constructed incrementally by using the symmetry of the partial interpretation to limit the enumeration of new relation values. The notion of symmetry depends on the type structure of the formula; by picking the weakest typing, larger equivalence classes (and thus fewer representatives) are obtained. A more refined notion of symmetry that exploits the meaning of the relational operators is also described.

The method typically leads to exponential reductions; in combination with other, simpler, reductions it makes automatic analysis of relational specifications possible for the first time.

1 Introduction

Relational Specifications

Many aspects of software are relational in nature. A relation can express containment—for example, of files in a directory; connectivity—between telephones in a network; naming—of distributed objects by global identifiers; ordering—of elements in a queue; and so on.

To investigate the design of a telephone switch, for example, we might model talking connections between phones as a binary relation

$$conns: Phone \leftrightarrow Phone$$

where $(p, q) \in conns$ when there is a connection from phone p to phone q. Making a call between phones *from* and *to* can then be modelled by a formula such as

$$to \notin ran\ conns \ \wedge \ conns' = conns \cup \{(from, to)\}$$

Here, *conns'* denotes the connections after execution of the operation. The expression *ran conns* denotes the range of the connection relation, namely the set of phones currently busy due to an incoming call. Note that the formula constrains both the state before execution of the operation (in the first conjunct), and the relationship between the pre- and post-states (in the second conjunct). It says that a call can be made only when the phone being called is not busy, and that the result is a new link between *from* and *to* in the set of connections.

Execution of the operation may be simulated by finding an assignment of values to the variables for which the formula is true. Simulation often reveals simple errors; this operation, for example, admits an execution in which *conns* is empty, and *from* and *to* are the same phone, *p* say, resulting in a connection from *p* to itself.

Another good way to find errors in an operation is to check that it preserves a desired invariant. Suppose, for example, that the connection relation is to have disjoint domain and range, so that, for billing purposes, no phone participates in both an incoming and an outgoing call at once. An invariant *I* is maintained by an operation *OP* when the formula

$$OP \wedge I \Rightarrow I'$$

is valid, where I' is the formula *I* with its variables primed. In our example we have

$$\begin{aligned}&(to \notin ran\ conns \ \wedge\ conns' = \ conns \cup \{(from, to)\}\\&\wedge dom\ conns \cap ran\ conns = \{\})\\&\Rightarrow dom\ conns' \cap ran\ conns' = \{\}\end{aligned}$$

To check the formula, we look this time for an assignment that makes the formula false; the execution just mentioned is such a case, since it gives *conns'* the value $\{(p, p)\}$.

This style of specification is central to model-based languages such as VDM [Jon86] and Z [Spi92, WD96, Jac97]. Specifications in these languages have been called ‘relational’ since the execution of an operation is modelled as a binary relation between pre- and post-states. Z is relational in an additional sense – and the one to which our title refers – in that the state itself is modelled with relations. This aspect of Z is shared with the style of specification known as ‘object modelling’, central to almost all of the object-oriented methods popular in commercial circles (eg, [SM88, R+91, BJR96]), in which the state space is described with an entity-relationship diagram [Che76], sometimes with additional textual constraints.

Analyzing Relational Specifications

This paper describes a technique for analyzing relational specifications. The underlying idea is very simple. Both simulation and checking amount to finding models of a relational formula, that is, assignments for which the formula is true. For simulation the formula is the description of the operation; for checking, the formula is the negation of an assertion about an operation. Models are found by a generate-and-test strategy: the formula is repeatedly evaluated for a series of assignments until one is found for which the formula is true.

The principal contribution of the paper is a reduction method based on symmetry. If an assignment can be shown to be isomorphic to one previously examined, it can be skipped; since the isomorphism classes are large, this prunes the search dramatically. The reduction is sound, so that if a model would have been found without the reduction, a model – but not necessarily the same one – will be found with the reduction.

The technique as a whole is not complete, since the search is only conducted over a finite universe of atoms. Completeness would rule out automation, since specification languages are invariably more expressive than the relational calculus, which is known to be undecidable [Sch79].

The technique has been implemented in a tool called Nitpick, whose specification language NP [JD96a] is roughly a subset of Z. NP may be viewed as a typed relational calculus [Tar41, SS93], with the addition of transitive closure and various syntactic structuring mechanisms. The example above is shown later (in Figure 10) as it would appear in NP. The differences are insignificant theoretically but vital in practice: the bundling of formulas into named 'schemas' such as *Switch* (for the state space) and *Call* (for the operation); explicit declaration of primitive types (here just *Phone*, the set of phones) and of variables (here just *conns* in the schema *Switch*); and an inclusion mechanism by which mention of a schema causes its declarations and formula to be included. Afficionados of Z will notice two respects in which NP differs: implicit declaration of primed variables representing the post state in schemas that model operations (marked by an argument list following the schema name), and treatment of claims as formulas too (but marked by the double colon to tell Nitpick to search for a model of the negation) rather than as meta-logical assertions in a proof system.

Nitpick's use is illustrated in Figure 1. The user provides a file containing specification schemas and claim schemas, along with a scope that assigns a size to each primitive type. A schema or claim is selected, and Nitpick then displays models as they are encountered during search, until some specified count (often 1) has been reached.

Experience with Nitpick/NP

Nitpick has been used in a variety of settings. In the Masters of Software Engineering program at Carnegie Mellon, students have used Nitpick on class projects to analyze several small but realistic designs. Nitpick has been applied to functional specifications (see [JD96b] for an application to the paragraph style mechanism of Microsoft Word); to checking existing proofs (finding a flaw in the proof of correctness of a handoff algorithm designed for the FAA's new air-traffic control system); to abstract designs (finding a bug in the first version of IPv6, a mobile host protocol due to become an Internet standard [Ng97]); and to software architectures (exposing cycles and races in the implicit invocation mechanisms used in many development environments).

Contrary to our initial expectations, we have found simulation to be invaluable. Implicit specifications, in which an operation's behaviour is defined as a conjunction of subformulae, are susceptible to over-constraint: it is easy mistakenly to define an operation with no executions at all. Simulation instantly reveals this problem.

Simulation does not require large models; there are usually quite enough complex cases to consider amongst the small models. Our hypothesis is that small models suffice for checking too. One might ask: how large a model must be considered to find any error? Since the language is undecidable, there can be no such bound. A more practical question is: how large a model is needed to find most errors occurring in a specification? This is an empirical question, and in our experience so far, the answer is often surprisingly small. In almost all of our experiments assigning a limit of 3 values to each type allows all the known flaws to be found.

Although Nitpick's suite of reductions allow many realistic specifications to be analyzed, the problem is still fundamentally intractable. Isomorph elimination alone is rarely sufficient to make checking feasible. With the addition of short circuiting [DJ96], however, we have been able to enumerate interpretations across a tree with 10^{20} leaves, obtaining reductions of 15 orders of magnitude. Isomorph elimination regularly contributes 6 orders of magnitude, and sometimes more. We have not had enough experience with term symmetry, a refinement described later in the paper, to determine its effectiveness; so far we have observed improvements from 20% to a factor of 10.

Increasing the size of the specification affects the performance in two ways. Keeping the set of variables fixed but making the formula more complex damages the performance of the reduction method described here, because the isomorphism classes are smaller, but it improves the performance of short circuiting.

On the other hand, adding new variables leads to greater reduction factors. At the same time, of course, the space grows exponentially. In the best case, an extra variable increases the search by a factor corresponding to the number of values it can take up to isomorphism. In practice, Nitpick is very sensitive to the addition of relation variables. Adding a scalar or a set of even a function, on the other hand, is often not a problem, nor is adding a relation that is tightly constrained (since short circuiting will cause many of its values to be ruled out).

2 Related Work

Our work appears to be the first attempt to analyze software specifications by semantic enumeration. Nitpick is currently the only tool that can analyze a relational specification fully automatically.

Theorem Provers

Small data structures have huge numbers of values. For this reason, enumerative analysis of software specifications has been regarded as infeasible. Research efforts have therefore focused mainly on syntactic analyses.

Recently, with the development of theorem provers such as PVS [OR+95] that incorporate powerful decision procedures, the analysis of realistic designs has become possible. A number of theorem provers have been developed specifically for Z [Jon92, SM96] or have been applied to it [BG94, ES94]; there are also specialized theorem provers for the relational calculus [BH94].

The labour cost of interactive theorem proving is still high enough, however, to rule out its use in everyday software development. For safety critical systems or components, on the other hand, especially those involving subtle distributed algorithms, the cost appears to be justified.

Theorem provers do, of course, have the major advantage that, unlike Nitpick, they can be used to demonstrate that a correct specification has no errors (with respect to a given property). When the specification is faulty, however, they tend to perform poorly. It might be useful then, to incorporate Nitpick-like features in a theorem prover to spare the cost of attempting an impossible proof when a counterexample can be rapidly generated.

Using abstraction, it is possible to obtain assurance from an enumerative analysis. An infinite space of interpretations is reduced to a finite space in such a way that establishing a property on the finite space suffices. This approach is well developed in the context of model checking [CGL92, B+95, DGR96]. In our previous work, we have shown how similar ideas can be applied to relational specifications [Jac94], but it seems that, in practice, it is often hard (or impossible) to find appropriate abstractions.

Simulators

Existing simulation tools require the specification to be constructive; suitable subsets of Z [Val91] and VDM [LL91] have been defined. Closer to our approach, there have been attempts to analyze Z specifications with a Prolog interpreter [DKC89, WE92], but these do not appear to be practical yet.

Nitpick's 'derived variable' analysis [JD95] ensures that when a variable of the post-state of an operation is defined constructively, no search is actually performed; the equation is essentially converted into an assignment statement. Consequently, Nitpick will execute specifications as fast as any simulation tool. And due to short-circuiting [DJ96] and isomorph elimination, it can focus on executions that are likely to expose errors.

The equivalence classes into which our method partitions the formula's interpretations are 'revealing subdomains' in the jargon of testing theory [WO80]. Our method might have some application in testing of code also, although resource boundaries introduce discontinuities in behaviour where many bugs reside. Consequently, an enumeration that is confined to a small scope is unlikely to expose errors.

Model Finders

In his thesis on abstract relational algebras [Jip92], Jipsen gives an algorithm for

finding models of relational formulae. His purpose differs from ours; he uses the algorithm to prove theorems by demonstrating that the negation of an inference has no models.
It is not clear whether Jipsen's algorithm could be applied to specification checking. His language does not admit transitive closure, which appears in most of our specifications. Also, since the algorithm has only been executed by hand on small examples, it is hard to predict how an implemention would behave.

General-purpose tools have been developed for finding models of formulae in various languages. The satisfiability problem for boolean formulae has been especially well studied because of its wide applicability, and because, despite being NP-complete, it appears to be easy to solve in practice. For hard cases, local search techniques work surprisingly well [SLM92].

Symmetries involving permutation of boolean variables have been exploited in the analysis of first-order formulae [BS92, Cra92]. But as far as we know, ours is the only method to use symmetry in the assignable values themselves.

The FINDER tool [Sla94] uses backtracking search to find models of a logic with functions and equality. Its author graciously translated some of our examples into its input language, and was able to obtain very good results. In examining *Claim2* of the specification of Figure 12, for example, it finds 809 counterexamples per backtrack, suggesting that the search mechanism is extremely efficient. These results were obtained by translating the relational formula into the language of functions and equality, skolemizing to eliminate the implicit existential quantifiers, and then by adding extra constraints and directives to order the search and break symmetries.

Zhang has developed a scheme that exploits symmetry in this setting [Zha96]. His model finder, FALCON, uses a heuristic that orders the atoms of the universe as they are inserted in the table representing a function; as a result, many of the permutations of a function are not generated. This scheme has much in common with ours, but it does not exploit the typing of the formula, nor the symmetries of the functions themselves.

A serious obstacle in applying these techniques to relational specifications is transitive closure. For a given scope, a relational formula involving closures can be translated into a purely boolean formula, but the resulting formula can be immense: intuitively, there will be a disjunct for every path through a relation's graph, and so in the worst case, there will be exponentially many subformulae.

We have experimented [DJJ96] with a version of Nitpick based on Bryant's ordered binary decision diagrams (BBDs) [Bry92]. Very large boolean formulae can often be represented with small BDDs. Using iterative squaring, closures can be computed reasonably efficiently. For checking some specifications the boolean method works better, but usually isomorph elimination gives a reduction large enough to make the explicit method preferable. Unfortunately there seems to be no easy way to incorporate isomorph elimination into the BDD method; it appears that the boolean variable orderings that are good for the relational formulae (that is, give small BDD's) are not appropriate for the side-conditions that break symmetry.

Symmetry in Model Checking

Symmetry in the transition relation of a state machine has been exploited in checking methods for Petri nets [Sta91], for reachability analysis [ID93], and for temporal logic model checking [CFJ93, ES93]. A recent journal issue on the topic of symmetry in automatic verification [FMSD96] includes expanded versions of these papers [C+96, ES96, ID96] and others.

These methods exploit different subsets of the automorphisms of the transition relation. Ip and Dill's method [ID96] uses a special construct in the specification language, called a 'scalarset', to represent an uninterpreted type. A specification involving an array of processors, for example, might declare the process identifiers used to index the array as belonging to such a scalarset. During exploration of the state space, the checker not only avoids visiting states that have been previously visited, but also avoids those in the same orbit as – that is, equivalent under symmetry to – a visited state.

The methods of Clarke, Enders, Filkhorn and Jha [C+96], and of Emerson and Sistla [ES96], accommodate a more general class of automorphism groups, but require the user to specify the group. Moreover, these methods can, to some degree, take advantage of the structure of the property being checked.

All of these methods involve, explicitly or implicitly, an orbit test: given two states, one must determine whether they fall into the same orbit and can thus be treated as indistinguishable. The problem can be shown to be harder than graph isomorphism [Jha96], and in practice an approximation is used that sometimes distinguishes states in the same orbit (but never equates states in distinct orbits).

Although our method was inspired by [CFJ93], it actually has little in common with it or [ID93]. These methods are concerned with the equivalence of two states induced by the symmetries of the transition relation to which they belong; ours is concerned with the equivalence of two interpretations, and exploits symmetry during the construction of an interpretation. Our types, derived from the 'given types' of Z [Spi92], are all uninterpreted, and are thus 'scalarsets' in the jargon of [ID93].

Our method requires no orbit test; the only isomorph checking that occurs is limited to the internals of the graph generation subroutine. This is possible because our search is more structured, and the automorphisms of the partially constructed assignment are enough to determine which relation values should be subsequently generated.

Because the branching factor in our search tree is so large, our reductions are usually greater than those achieved by these other methods. In model checking, for example, a factor of 10 or 100 is typical, while we often see reductions of 10^5 or more.

An earlier version of our isomorph elimination method is described in [JJD96]. To avoid the generation of graphs under arbitrary colourings, it introduced explicit bijections called ‘wirings’ between the sides of one relation and another. This allowed us to generate an entire isomorph-free set of relations just once, and then vary the bijections according to the colouring context. The new method dispenses with the wirings, resulting in a simpler algorithm that is easier to justify and to implement. It also performs better, typically by a factor of 5–10, both because the old method generated some duplicate interpretations, and because it did not exploit term symmetry.

3 Overview of the Method

For both kinds of analysis that Nitpick performs, it is important that the models be generated roughly in order, so that models with smaller relations appear first. For simulation, this allows the user to observe when expected cases are missing; for checking, it helps by favoring simpler counterexamples.

One way to find models is simply to enumerate all assignments of values to variables. Unfortunately, the number of assignments grows exponentially with size. A relation on a set of k elements can have $k \times k$ edges, each of which is either present or absent, and thus has $2^{k \times k}$ possible values. For three relational variables over a set of 3 elements, there are about 10^8 assignments to enumerate. Moreover, the density of models amongst assignments is often very small, so one cannot assume that a model is likely to be found by a partial search.

This paper describes a method for enumerating assignments that allows a high proportion to be skipped. Along with other reduction mechanisms, it makes the search of huge assignment spaces possible. Employing the method described here, along with other reduction mechanisms, Nitpick can usually search spaces of 10^{12} assignments in less than a minute. The method is sound, so that a search within a given finite bound will always find a model if one exists.

The method is based on a simple observation. Many of the assignments that would be produced by a naive enumeration are isomorphic to one another: that is, one can be obtained from another by exchanging the labels of the relation’s nodes. Because the types in our specification language are uninterpreted, two isomorphic assignments must either both be models or not, so examining one suffices.

Since so many relations are isomorphic to one another, eliminating isomorphs has a dramatic effect. Table 1 compares, for small k, the number of $k \times k$ relations, and the number up to isomorphism. Generating an isomorph-free set of relations does not require expensive testing: the 6×6 relations, for example, can be generated in less than a minute.

Eliminating isomorphs is beneficial for models and non-models alike. Skipping non-models speeds up the search for models; this is the primary motivation. Skipping models is good for simulation, because it means that the user is not barraged with cases that are uninteresting relabellings of cases already seen.

Like our previous isomorph elimination method [JJD96], the method exploits the structure of the formula being checked as well as the structure of the enumerated relation values. The method described here, however, is more powerful; it can exploit symmetries not only in the values of relational variables but also in the values of terms, leading to larger reductions. It is also simpler.

The rest of the paper is organized as follows. The next section gives a syntax for relational formulae and defines the basic notions. Section 5 justifies the choice of a syntax different (and much simpler) than our specification language NP. Section 6 gives a more formal overview of the reduction method. The remaining sections describe the method in detail, starting with the general principle of 'permutation invariance'. The method is shown to be sound, and an algorithm is given. A refinement of the method that exploits symmetry in the values of entire terms is also presented. The paper closes with some remarks about future work.

4 Basic Notions: Formulae, Models and Enumeration

A syntax for relational formulae is shown with its semantics and typing rules in Figure 2. A formula is accompanied by type declarations for each of its variables; there are no quantifiers, so all variables are free and can be declared together. The formula

$$p, q \colon \langle T, T \rangle$$
$$p \,;\, q = \emptyset$$

for example, has two free variables, p and q, declared to be homogeneous relations on elements of type T. The same formula could be typed differently

$$p \colon \langle S, T \rangle$$
$$q \colon \langle T, V \rangle$$
$$p \,;\, q = \emptyset$$

where p is now a relation from S to T, and q is a relation from T to V.

Types are unstructured: each is denoted by some arbitrary name. Moreover, it is crucial to our method that types are uninterpreted. A type name such as T denotes an arbitrary collection of values with no semantics beyond a notion of equality.

A relation is a directed, bipartite graph with a finite number of nodes and arcs. Arcs originate from nodes in the *left set* of the relation and terminate on nodes in the *right set*. The relation's domain and range are respectively subsets of the left and right, and consist of the nodes that have outgoing and incoming arcs.

Types add no expressive power. Having a type $\langle S, T \rangle$ means being a subrelation of the cross product $S \times T$, which is easily expressed as a formula. Types are used, as we shall see, to structure the search (and, as usual, to expose simple errors in the specification).

A formula evaluates to true or false under a given interpretation of its variables. There is a universe of atoms, U, from which relations are constructed. An interpretation $\langle \tau, A \rangle$ consists of a type assignment

$$\tau \colon Type \to \mathcal{P}\,(U)$$

that assigns a set of atoms $\tau\,[\![t]\!]$ to each type t, and a variable assignment

$$A \colon Var \ \to (U \leftrightarrow U)$$

that assigns a relation value $A[\![v]\!]$ to each variable v in the formula. The carrier sets of different types are disjoint

$$s \neq t \ \Rightarrow \ \tau\,[\![s]\!] \cap \tau\,[\![t]\!] \ = \varnothing$$

and the variable assignment must respect the typing, so that if $v \colon \langle s, t \rangle$,

$$A[\![v]\!] : \ \tau\,[\![s]\!] \leftrightarrow \tau\,[\![t]\!]$$

Under the semantics of the relational operators, an interpretation I gives a value $I\,[\![e]\!]$ to each relational term e, and thus a value $I\,[\![f]\!]$ to the formula f as a whole. Since there are no issues of variable scope, the semantics is trivial. Each operator o has a meaning O that is a function on relation values, so that the meaning of a term is obtained by applying the appropriate function to the meanings of its subterms.

If $I\,[\![f]\!]$ is true—that is the formula f evaluates to true under the interpretation I—then I is said to be a *model* of f. When we want to make the typing explicit, we shall write

$$I, Ty \vDash f$$

to say that I is a model of f under the typing Ty.

Taking the formula

$$p, q \colon \langle T, T \rangle$$
$$p \,;\, q = \varnothing$$

as an example, the models are the interpretation that assign values to p and q whose product is the empty relation, such as

$$\tau = \{T \mapsto \{0, 1, 2\}\}$$
$$A = \{p \mapsto \{(0, 0)\}, q \mapsto \{(1, 0)\}\}$$

In contrast, the interpretation

$$\tau = \{T \mapsto \{0, 1, 2\}\}$$
$$A = \{p \mapsto \{(0, 1)\}, q \mapsto \{(1, 0)\}\}$$

is not a model, because the product includes the arc (0, 0).

In our method, the search for models of a formula is bounded. The user declares a scope

$$\Sigma \colon Type \to \mathbb{N}$$

that associates a positive integer with each type. An interpretation $\langle \tau, A \rangle$ is said to belong to a scope Σ if for every type t

$$| \tau [\![t]\!] | = \Sigma[\![t]\!]$$

The scope induces a limit on the size of relation that can be assigned to a variable, and thus on the number of relations. Even for a small scope, the number of assignments can still be huge. For a scope of three, for example—that is, a scope for which $\Sigma[\![t]\!] = 3$ for all types t—a formula with 3 variables has 10^8 variable assignments.

Given a formula and a scope, the method generates a sequence of interpretations and displays those that are models. As we shall see, the actual values of atoms is immaterial, so, having fixed the scope, only the variable assignment A changes during enumeration.

An reasonable enumeration order produces smaller assignments first, where size is a global measure, such as the total number of arcs in the relation values. This is not easily implementable, so instead we guarantee a weaker, pointwise ordering. If two assignments A_1 and A_2 are generated that differ only in the values they assign to some variable v, then if A_1 comes first, the number of arcs in $A_1[\![v]\!]$ is no greater than the number of arcs in $A_2[\![v]\!]$.

Not all models within the scope are enumerated, but the method is sound: if a model exists within a scope, a model (but necessarily the same one) will be found. A simple isomorph elimination scheme might guarantee that if a model is not generated, then at least a permutation will be. Our method obtains better reductions, however, and does not make this guarantee.

5 Applicability of the method

The syntax used to illustrate the method (Figure 2) was chosen for its simplicity. Our notation, NP – see Figure 10 for an example – is more elaborate, primarily in three respects. First, its concrete syntax is richer. Using a form of Z's schema mechanism [Spi92] tailored to the description of abstract state machines as a structuring mechanism, it allows operation and state descriptions to be constructed incrementally, with commonalities factored out. It also distinguishes claims to be checked from the specification proper. Second, NP admits variables of scalar and set type. Third, a larger repertoire of relational operators is provided (along with additional operators to include sets and scalars).

A few points are in order to justify this simplification, and to explain why checking the full language is no harder. First, the concrete syntax is immaterial; the method operates purely at the semantic level, and the syntactic shorthands the NP notation provides are stripped away prior to checking.

More surprisingly, perhaps, the omission of scalars and sets is also insignificant. A formula involving scalars and sets can be translated into an equivalent formula involving only relations by the following scheme [SS93]. A subset of a set X is treated as a relation s on $X \times Y$ that pairs each element of the set with every element of Y, so that a relation s denotes a set when

$$s \,;\, Un = s$$

The domain and range of a relation r are then $r \,;\, Un$ and $r^{\sim}; Un$. A scalar is treated as a singleton set; a pair of scalars (x, y) becomes $x \,;\, y^{\sim}$.

In this paper, the method is cast in terms of purely relational formulae. It could be applied to formulae with sets and scalars by translating the formulae, and then translating the models back, but in practice the method is easily extended to work directly with sets and scalars.

The choice of relational operators is conventional; the language is essentially the relational calculus with the addition of transitive closure. In fact, however, the method is not sensitive to the operators chosen, but will work for any kind of relational formula whose operators satisfy a simple criterion.

Consider any permutation π of the universe U. Then π induces a permutation π' of the relation values defined by

$$\pi' R = \{(\pi u, \pi v) \mid (u, v) \in R\}$$

for each relation value R. If we view the meaning of each operator as a set of relation tuples, then π' induces a permutation π'' of the operators themselves

$$\pi'' O = \{(\pi' R_1, ..., \pi' R_n) \mid O\,(R_1, ..., R_{n-1}) = R_n\}$$

for each operator O. A *logical* operator is fixed under any permutation; that is, its characteristic set of tuples is mapped to itself. Equivalently, a logical operator commutes with permutation of the relation values. If O is a logical operator that includes $(R_1, ..., R_n)$ and it commutes with permutation, then

$$O\,(\pi' R_1, ..., \pi' R_{n-1}) = \pi' O\,(R_1, ..., R_{n-1}) = \pi' R_n$$

and thus O must also include $(\pi' R_1, ..., \pi' R_n)$ and be fixed under permutation. Take, for example, the composition operator; it is logical because

$$\begin{aligned}
&(\pi' p) \,;\, (\pi' q)\\
&= (\pi^{\sim} ; p \,;\, \pi) \,;\, (\pi^{\sim} ; q \,;\, \pi)\\
&= \pi^{\sim} ; p \;\,;\, q \,;\, \pi\\
&= \pi' \,(p \,;\, q)
\end{aligned}$$

Our method requires only that the operators be logical. Moreover, it can be shown that if an operator can be defined in terms of first-order predicate calculus, it must be logical; and if not, it seems unlikely that it could be included in a practical specification language anyway.

(The notion of logicality is due to Tarski [Tar41, Giv88]. His calculus is less expressive than first-order logic; it can be shown that although any first-order formula with at most three variables can be expressed, there are trivial formulae with four variables that are not expressible. To show that this result does not depend on a quirk of his formulation, Tarski proved that no addition of a finite number of logical operators increases the expressive power. Logicality, then, defined for Tarski the broadest notion of what constitutes a relational calculus. One might expect the lack of expressiveness with respect to first-order logic not to be of practical significance, since, if extra relational variables are admitted, projection can be expressed, and, by tupling, the restriction to three variables is overcome. But in practice the relational formulation often feels awkward. For software specifications, a more serious issue is the lack of transitive closure which we include explicitly because it cannot be encoded in the plain calculus).

6 Outline of the Method

Although our method relies on a tricky algorithm (to generate graphs), it is based on two simple observations.

The first is a direct consequence of the logicality of the operators. Since the universe is uninterpreted, relabelling the relation values of an interpretation can have no effect on whether it is a model or not. Put another way, any permutation π of the universe U, when extended over the interpretations I of a formula, fixes the subset $\mathcal{M}$ of models. Partition I so that two interpretations belong to the same class when there is a permutation that takes one to the other; these classes are called the *orbits* of the permutation group.

Each orbit contains either only models or only non-models. To find a model, there is no need to pick more than one interpretation from each orbit. Ideally, we would pick exactly one from each, but this is hard to do. Since the orbits can be very large, our method achieves a considerable speedup even though it may pick several interpretations from a single orbit.

The second observation exploits the type structure of the formula. A typing may give two relations common types when in fact they are independent. For example, in the formula

$$p = q \wedge r = s$$

p and q must share the same left and right types, as must r and s, but the types of p and r need not be related. Permuting the left sides of p and q together, but leaving r and s invariant, will clearly not affect whether an interpretation is a model. And yet, if the typing is needlessly strong—requiring, for example, that all four variables have the same type—this will not be expressible as a permutation of the universe. Our method therefore infers the most general typing of the formula, and uses the equivalence classes of its permutations. The interpretations under one typing can be put in one-to-one correspondence with the interpretations under another typing, but the weaker the typing, the more permutations there are, and therefore the fewer the equivalence classes.

Figure 3 summarizes these two observations. The small rings represent models; the dotted lines show how models under the declared typing (above) are matched with models under the more general inferred typing (below). The arrows between the models denote examples of permutations mapping one to another. The four models are divided into two orbits in the declared typing, but fall within a single orbit in the inferred typing because of the additional permutation π_c'.

In our method, the interpretations are constructed incrementally in a tree. The variables are ordered, and at each level of the tree, the values of one variable are enumerated. Each leaf of the tree corresponds to an interpretation. Figure 4 shows parts of such a tree. The leaf marked I gives the interpretation obtained by assigning to the variables the values of the shaded nodes: in this case the first variable R_1 has the value a, the second has the value b, and so on, with the last having the value g.

Isomorphic interpretations are not eliminated at the leaves—since that would still require construction of the entire tree—but by pruning during the enumeration. Rather than generating all values of a variable, we generate enough values to guarantee that at least one interpretation from each equivalence class will be present. The set of values generated as children of a node in the tree varies between nodes at the same level, because the notion of equivalence depends on the interpretation that has been constructed so far. As we shall see, given a partially constructed interpretation I, the method avoids generation of both v and πv if π is a symmetry of I. Figure 5 illustrates this. The internal node marked I corresponds to the partial interpretation; v_1 and v_2 are enumeration values such that

$$\pi\, v_1 = v_2$$

where π is a symmetry of I; that is, for every relation value v in I,

$$\pi\, v = v$$

In this situation, only one of v_1 and v_2 is required, so that if v_1 is included, v_2 can be safely skipped, and its entire subtree pruned away.

One could perhaps generate an isomorph-free set of interpretations, a full interpretation at a time. We use the incremental approach because it dovetails with another reduction scheme implemented in Nitpick. Often, it is possible to determine from the partially constructed interpretation that all or none of its completions will be models, so there is no need to continue the search. This *short circuiting* prunes the interpretation tree further, often by a factor even greater than that achieved by the isomorph elimination method [DJ96].

Sections 7 and 8 explain the two observations underlying the method. Section 9 justifies the incremental approach, and Section 10 explains the colouring scheme used to express the symmetries. Section 11 presents a refinement of the method. Algorithms for the basic and refined method are given in Sections 12 and 13.

7 Relabelling

Since the universe of atoms is uninterpreted, an injective relabelling of the relation values can have no effect on the meaning of the formula. This means that if an interpretation is a relabelling of one already examined, it can be safely ignored. We observed above, for example, that

$$\tau = \{T \mapsto \{0, 1, 2\}\}$$
$$A = \{p \mapsto \{(0, 1)\}, q \mapsto \{(1, 0)\}\}$$

is not a model of the formula

$$p, q\colon \langle T, T \rangle$$
$$p \mathbin{;} q = \varnothing$$

Likewise, permuting 0 and 1, the interpretation

$$\tau = \{T \mapsto \{0, 1, 2\}\}$$
$$A = \{p \mapsto \{(1, 0)\}, q \mapsto \{(0, 1)\}\}$$

is also not a model.

Consider any interpretation $I = \langle \tau, A \rangle$ of a formula f with types t_1, ..., t_n. Let $Sym(S)$ be the group of all permutations of the set S. A *relabelling* with respect to this interpretation is any permutation $\pi\colon U \to U$ of the universe that respects types:

$$\pi \in Sym\,(\tau\,[\![t_1]\!]) \times \ldots \times Sym\,(\tau\,[\![t_n]\!])$$

A relation value is relabelled by relabelling its left and right sides:

$$\pi\, R \mathrel{\widehat{=}} \pi^{\sim} \mathbin{;} R \mathbin{;} \pi$$

For convenience, we shall overload the names of permutations in this way, but the reader should remember that the permutation on the left of the definition acts on the entire relation, while the permutation appearing twice on the right acts on atoms. Similarly, we extend the relabelling over the assignment A

$$\pi\, A \mathrel{\widehat{=}} \{r \mapsto \pi^{\sim} \mathbin{;} R \mathbin{;} \pi \mid r \mapsto R \in A\}$$

and then over the entire interpretation

$$\pi\, I \mathrel{\widehat{=}} \langle \tau, \pi\, A \rangle$$

First, we show that relabelling commutes with the meaning function, so that it makes no difference at which point relabelling is applied during the evaluation of a term:

Lemma 1 (Relabelling commutes with interpretation) For any expression e, interpretation I and relabelling π,

$$\pi\,(I\,[\![e]\!]) = (\pi\, I)\,[\![e]\!]$$

Proof By structural induction. The base step, where e is a variable, is trivial. For the induction step, consider an expression $o(e_1, ..., e_n)$, and take O to be the meaning of the operator o (as defined in Figure 2). Since O is assumed to be logical,

$$\pi\, O(I\,[\![e_1]\!], ..., I\,[\![e_n]\!]) = O(\pi\, I\,[\![e_1]\!], ..., \pi\, I\,[\![e_n]\!])$$

and by hypothesis,

$$\pi\,(I\,[\![e_i]\!]) = (\pi\, I)\,[\![e_i]\!]$$

it follows that

$$\pi\,(I\,[\![o(e_1, ..., e_n)]\!]) = O((\pi\, I)\,[\![e_1]\!], ..., (\pi\, I)\,[\![e_n]\!]) = (\pi\, I)\,[\![o\,(e_1, ..., e_n)]\!] \qquad \square$$

Since consistent relabelling of two equal relations produces two other equal relations, the evaluation of the formula as a whole is unaffected:

Theorem 1 (Relabelling fixes the set of models) For any relabelling π of an interpretation I of a formula f,

$$(\pi\, I)\,[\![\, f\,]\!] = I\,[\![\, f\,]\!]$$

Proof Consider the meaning of a subformula under $\pi\, I$. By the definition of equality

$$(\pi\, I)\,[\![e_1 = e_2]\!] = ((\pi\, I)\,[\![e_1]\!] = (\pi\, I)\,[\![e_2]\!])$$

By lemma 1

$$((\pi\, I)\,[\![e_1]\!] = (\pi\, I)\,[\![e_2]\!]) \ = (\pi\,(I[\![e_1]\!]) = \pi\,(I[\![e_2]\!]))$$

and since π is bijective

$$(\pi\,(I[\![e_1]\!]) = \pi\,(I[\![e_2]\!])) = (I[\![e_1]\!] = I[\![e_2]\!])$$

and thus

$$(\pi\, I)\,[\![e_1 = e_2]\!] = (I[\![e_1]\!] = I[\![e_2]\!]) = I\,[\![e_1 = e_2]\!]$$

The extension to the full formula is a trivial induction. $\square$

An obvious corollary of the theorem is that the choice of the type assignment τ is not significant. We simply pick an arbitrary set of atoms for each type, whose cardinality matches the scope selected by the user.

8 Typing

The more relabellings there are, the fewer equivalence classes of interpretations there will be, and thus the fewer interpretations to be examined. A more general typing, which distinguishes two relations (or, more accurately, sides of relations) by giving them different types, when they were previously constrained to have the same type, will admit more relabellings, because the two types can be permuted independently.

Take, for example, the formula

$$p \,;q = \varnothing$$

with the typing

$$p, q \colon \langle T, T \rangle$$

and consider interpretations in which

$$\tau\,[\![T]\!] = \{a, b\}$$

The two variable assignments

$$\begin{aligned} A_1 &= \{p \to \{(a, a)\}, q \to \{(b, b)\}\} \\ A_2 &= \{p \to \{(b, b)\}, q \to \{(a, a)\}\} \end{aligned}$$

are equivalent, because of the relabelling (ab). The assignments

$$\begin{aligned} A_3 &= \{p \to \{(b, a)\}, q \to \{(b, a)\}\} \\ A_4 &= \{p \to \{(a, b)\}, q \to \{(a, b)\}\} \end{aligned}$$

are also equivalent to each other, but neither of A_1 or A_2 is equivalent to A_3 or A_4.

Now take the equally admissible typing

$$\begin{aligned} &p \colon \langle S, T \rangle \\ &q \colon \langle T, S \rangle \end{aligned}$$

with

$$\begin{aligned} \tau\,[\![T]\!] &= \{a, b\} \\ \tau\,[\![S]\!] &= \{c, d\} \end{aligned}$$

The four assignments

$$\begin{aligned} A_1 &= \{p \to \{(c, a)\}, q \to \{(b, c)\}\} \\ A_2 &= \{p \to \{(c, b)\}, q \to \{(a, c)\}\} \\ A_3 &= \{p \to \{(d, a)\}, q \to \{(b, d)\}\} \\ A_4 &= \{p \to \{(d, b)\}, q \to \{(a, d)\}\} \end{aligned}$$

are now all equivalent, since relabellings may be formed as combinations of the transpositions (ab) and (cd). Intuitively, whether the product of p and q is empty depends only on the coincidence of range elements of p with domain elements of q. We should be able to permute the left side of p and the right side of q independently of the 'inner' sides, and this is what the more general typing allows us to do. The most general typing

$p\colon \langle S, T\rangle$
$q\colon \langle T, V\rangle$

will admit 8 permutations; considering only the relations with at most one arc, this places the 16 possible assignments in 2 equivalence classes, so that there are only two cases to consider: the one in which the arcs do not meet in the middle (which is a model) and the one in which they do (which is not a model).

We pick the most general typing, since it leads to the greatest reduction in the search. To justify this, we must define what we mean by a typing, and show that the choice of typing does not prejudice the presence of models: if a model exists under one typing, it also exists under another.

To compare typings, it will be useful to have a notion of *untyped interpretation*, in which all variables have the same type $\langle u, u\rangle$ for some u. The choice of typing clearly affects the models of the formula, but it should not affect whether a model exists. So we define a typing to be *sound* for a given formula if it guarantees that whenever the formula has a model under that typing, it also has an untyped model, and vice versa.

Definition 1 (Type soundness). A typing Ty is *sound* for a formula F if

$$\exists I.\ I, Ty \models F \ \Leftrightarrow\ \exists I.\ I, Least \models F$$

where $Least$ is the least general typing: it assigns the type $\langle u, u\rangle$ to every variable, for some arbitrary type u. A type system is sound if it only produces sound typings for a formula.

It follows from this definition that, so long as the type system is sound, the presence of models is not sensitive to the typing: a formula has a model under one typing exactly when it has an untyped model, and thus exactly when it has a model under another sound typing.

Theorem 2 (Typing preserves modelhood) Given any sound typings Ty and Ty' of a formula F,

$$\exists I.\, I, Ty \models F \ \Leftrightarrow\ \exists I.\, I, Ty' \models F$$

To show that a type system is sound, it is sufficient to show that it is *consistent* in the following sense. With each type t we associate some arbitrary injection

$$\alpha_t\colon \tau\, [\![t]\!] \rightarrow U$$

that translates atoms in a typed interpretation into atoms in an untyped interpretation. For an expression $e\colon \langle s, t\rangle$, let $\alpha\, e$ be short for the expression in which the appropriate injections are applied to the left and right:

$$\alpha\, e \mathrel{\widehat{=}} \alpha_s{}^{\sim} ; e ; \alpha_t$$

Definition 2 (Type consistency) A type system is *consistent* if type translation commutes with every operator:

$$O\,(\,\alpha\, e_1, \ldots, \alpha\, e_n) \;=\; \alpha\, O\,(\, e_1, \ldots, e_n)$$

Type translation can be extended in the obvious way over an interpretation, and for a consistent type system, behaves like the permutations of the previous section. The proofs of the following two lemmas are simple variants of those of Lemma 1 and Theorem 1.

Lemma 2 (Translation commutes with interpretation) For any expression *e*, interpretation *I* and type translation scheme *α*,

$$\alpha\,(I\,[\![e]\!]) = (\alpha\, I)\,[\![e]\!]$$

so long as the type system is consistent.

Lemma 3 (Type translation preserves modelhood) For any type translation *α* of an interpretation *I* under a consistent typing *Ty* of a formula *f*,

$$I, Ty \vDash f \Leftrightarrow \pi\, I \vDash f$$

We can now show that consistency is enough to establish soundness:

Theorem 3 (Consistency implies soundness) Every consistent type system is sound.

Proof Given a consistently typed model $I = \langle \tau, A\rangle$ of some formula *f*, we show how to construct an untyped model $I' = \langle \tau', A'\rangle$. Define τ' so that it maps the arbitrary type *u* to a carrier set whose cardinality is that of the largest carrier set $\tau\,[\![t]\!]$. Now pick any set of injections

$$\alpha_t : \tau\,[\![t]\!] \rightarrow \tau'\,[\![u]\!]$$

Define A' so that

$$A'\,[\![r]\!] = \alpha\, A\,[\![r]\!]$$

for every variable *r*. By lemma 3, I' is a model.

To construct a typed model $I = \langle \tau, A\rangle$ from an untyped model $I' = \langle \tau', A'\rangle$, we define τ so that it maps each type *t* to a carrier set whose cardinality is that of $\tau\,[\![u]\!]$. Now pick any set of total injections

$$\alpha_t : \tau\,[\![t]\!] \rightarrow \tau'\,[\![u]\!]$$

and again require

$$A' [\![r]\!] = \alpha\, A\, [\![r]\!]$$

Since the α_t are bijections this time, this defines A uniquely. As before, the correspondence will ensure that I is a model when I' is. □

As an illustration, take the composition operator and its type rule (in Figure 2). Given variables

$$p\colon \langle s, t\rangle,\ q\colon \langle t, u\rangle$$

we have

$$\begin{aligned}&(\alpha\, p) \,;\, (\alpha\, q)\\ &= (\alpha_s{}^{\sim} ; p ; \alpha_t) ; (\alpha_t{}^{\sim} ; q ; \alpha_u)\\ &= (\alpha_s{}^{\sim} ; p ; q ; \alpha_u)\\ &= \alpha\, (p ; q)\end{aligned}$$

The typing rule guarantees that the inner injections match; the product $\alpha_t ; \alpha_t{}^{\sim}$ is the identity because α_t is total. To show that the type system is sound, a similar proof is required for each operator.

9 Enumeration Method

Theorems 1 and 2 form the basis of our method. The first justifies the skipping of isomorphic interpretations; the second allows the formula to be typed so that as many interpretations as possible are deemed isomorphic to one another. Theorem 3 is used to justify typing rules for new operators.

Given a formula *f*, the most general typing is inferred. Each type gets a carrier set containing the number of atoms specified by the scope. The assignment of values to variables is then explored by traversing a tree in which, at each level, the values of some variable are enumerated (Figure 4). A leaf of the tree corresponds to a path, each step of which assigns a value to a variable; the entire path thus defines an assignment. The formula is evaluated for each assignment, and if true, is printed out.

Traversal of the entire tree is not usually feasible, even for small scopes, but, exploiting Theorem 1, our method explores only a small part of it. Permutations induce an equivalence on assignments: if one can be permuted into another, either both or neither are models.

Our goal is to pick only one assignment from each such equivalence class. Since it is important not merely to filter assignments at the end, but to avoid generating them in the first place, we need a local criterion that can be applied during traversal of the enumeration tree.

At a given node in the tree, just prior to the enumeration of values for some variable r_i, a partial assignment of values to the variables $r_1, r_2, \ldots, r_{i-1}$ has been constructed. Take some value v for r_i and some permutation π. Must the permuted value πv also be considered? In general, the answer is yes, since, in combination with the partial assignment, the permuted value does not necessarily give a permutation of the extended assignment. However, if π is a symmetry of the partial assignment, applying it to the extended assignment affects only the new value, and so indeed the extended assignment will be a permutation.

Consider, for example, finding models of the formula

$$p\colon \langle S, T\rangle,\ q\colon \langle T, V\rangle$$
$$p \,;\, q = \varnothing$$

with

$$\tau = \{T \mapsto \{a, b\}, S \mapsto \{c, d\}, V \mapsto \{e, f\}\}$$

by enumerating first p and then q. Suppose the value of p just enumerated is $\{(a, c), (a, d)\}$. Then for the 4 values of q that have one arc

$$\{(c, e)\}, \{(d, e)\}, \{(c, f)\}, \{(d, f)\}$$

only one need be checked, since from any one the others can be obtained by applying the permutations (cd) and (ef), both being symmetries of p, the former because of the structure of p, and the latter because p does not involve the type V.

Lemma 4 Consider a partial assignment A that assigns relation values to variables $R_1, ..., R_{i-1}$, and consider extending it to B by assigning a value v to the variable r_i:

$$B = A \oplus \{r_i \mapsto v\}$$

Let G be the automorphism group of A: that is, the set of permutations over the universe U that leave all relation values of A fixed. Then for any $\pi \in G$,

$$\pi B = A \oplus \{r_i \mapsto \pi v\}$$

Proof π leaves $r_1, ..., r_{i-1}$ fixed, so $\pi A = A$, and thus

$$\pi (A \oplus \{r_i \mapsto v\}) = \pi A \oplus \{r_i \mapsto \pi v\} = A \oplus \{r_i \mapsto \pi v\}$$

as required. ◻

Our algorithm exploits this lemma by enumerating only a subset of the values of r_i, so that for every possible value v there is at least one value v' enumerated with $\pi v' = v$, for a symmetry π of the partial assignment.

Algorithm The assignments are constructed incrementally in a tree-like enumeration. Suppose that at some point the assignment A' has been generated, which assigns values to variables $r_1, ..., r_{i-1}$. Let V_i be the set of possible values of r_i. The algorithm extends A' by assigning r_i to each of the values in V_i', where $V_i' \subseteq V_i$, such that for every $v_i \in V_i$, there is an automorphism π of A' and a value generated $v_i' \in V_i'$ with $\pi\ v_i' = v_i$.

Theorem 4 For any full assignment A generated by a naive enumeration, the algorithm generates an assignment A' that is a permutation of A.

Proof By induction on the tree depth, with this hypothesis: for any assignment A_i of length i, an assignment A_i' is generated such that, for some permutation π

$$\pi A_i' = A_i$$

The base case is trivial: since the partial assignment is empty, its automorphism group is the full symmetry group $Sym(U)$. If v is not generated as a value for R_1, there is a permutation $\pi \in Sym(U)$ such that v' is generated, and $v' = \pi v$.

For the induction step, assume a value has just been assigned to r_{i-1}. We shall show that a permutation of

$$A \oplus \{r_i \mapsto v\}$$

is generated for any A and v. By hypothesis, there is a partial assignment A' that is generated such that

$$A = \pi A'$$

for some permutation π. We want to demonstrate that a v' is generated such that

$$A \oplus \{r_i \mapsto v\} = \delta\ (A' \oplus \{r_i \mapsto v'\})$$

for some permutation δ. Now for any value x of the variable r_i, the algorithm generates a v' such that, for some automorphism α of A',

$$x = \alpha\ v'$$

Pick $x = \pi^{\sim} v$, so that $v' = \alpha^{\sim}\pi^{\sim} v$. Then

$$\begin{aligned} & A' \oplus \{r_i \mapsto v'\} \\ & = A' \oplus \{r_i \mapsto \alpha^{\sim}\pi^{\sim} v\} \\ & = \alpha^{\sim}\alpha\ A' \oplus \{r_i \mapsto \alpha^{\sim}\pi^{\sim} v\} \end{aligned}$$

and

$$\begin{aligned} & \alpha^{\sim}\alpha\ A' \oplus \{r_i \mapsto \alpha^{\sim}\pi^{\sim} v\} \\ & = \alpha^{\sim}\ (A' \oplus \{r_i \mapsto \pi^{\sim} v\}) \\ & (\textit{since } \alpha \in \textit{Aut } A') \\ & = \alpha^{\sim}\ (\pi^{\sim} A \oplus \{r_i \mapsto \pi^{\sim} v\}) \\ & (\textit{since } A = \pi A') \\ & = \alpha^{\sim}\pi^{\sim}\ (A \oplus \{r_i \mapsto v\}) \end{aligned}$$

which completes the proof. □

10 Colouring Scheme

Our method exploits only a subset of the automorphisms; in practice, there is no simple way to express them all. An explicit representation of the automorphisms would not be appropriate for generating the new relation values – in the worst case, there is an exponential number – so instead they are represented implicitly by a colouring of the universe U. If two atoms a and b are given the same colour, there is an automorphism that maps one to the other while leaving all other atoms fixed: the transposition (ab), in other words, is an automorphism.

Constructing this colouring is made easier by the typing of the formula. Since distinct types have disjoint carrier sets, any automorphism of the partial assignment can be expressed as a product of permutations $\pi(t)$, where $\pi(t)$ acts only on the carrier set of the type t. The colourings of types can thus be regarded as independent.

For each relation, we consider permutations that act on the left or the right side. We colour the left nodes of each relation so that two nodes have the same colour if exchanging them leaves the relation invariant, and similarly for the right nodes. Note that composite automorphisms involving simultaneous permutations on both sides are not representable. For a relation whose left and right sides have the same type, the colouring requires the same permutation to be applied on both sides.

The colouring of the sides of the relations now induces a colouring of the universe. The types are considered one at a time. For each type, we colour its elements so that two have the same colour exactly when the nodes corresponding to those elements have the same colour in every relation side in which they appear. This colouring is not coarser than any of the colourings of the relations, so if two elements of a type have the same colour, their exchange leaves all relations invariant.

A set of values V' is now generated for the new relation that has the following property. For every relation value v in the full set V, a relation value v' is generated such

$$\pi\, v' = v$$

for some permutation π that respects the colouring of atoms.

Since π can be expressed as a product of transpositions, each of which is an automorphism of A, π itself is an automorphism of A too, and thus qualifies as a suitable permutation according to Theorem 4.

Examples

Consider again the formula

$$p\colon \langle S, T\rangle,\ q\colon \langle T, V\rangle$$
$$p \,;\, q = \emptyset$$

with

$$\tau = \{T \mapsto \{a, b\}, S \mapsto \{c, d\}, V \mapsto \{e, f\}\}$$

Figure 6 shows the possible sets of values that can be generated for a heterogeneous 2×2 relation: in the case of full symmetry (*A*), partial symmetry (*B*), and no symmetry (*C*). Each relation value is coloured according to the algorithm: two nodes (on a given side) are coloured the same when their transposition is an automorphism of the graph.

Assume that *p* is enumerated first. For *p*'s enumeration, the partial assignment is empty, and has the full symmetry group over the universe as its automorphisms. For each type, all elements are coloured the same, and 7 values of *p* are generated, as shown in contour *A* of Figure 6.

The enumeration of *q* now depends on the context: that is, the value assigned to *p*. When *p* has the value $\{(a, c), (a, d)\}$, which has the automorphism (*cd*), the elements of the types will again be coloured the same, except for those of *S*, which, because (*ab*) is no longer a symmetry, must have different colours. *S* is not, however, a type of *q*, and so the enumeration of *q* will produce 7 values (contour *A*). When *p* has the value $\{(a, c), (b, c)\}$, on the other hand, the elements of *T* will now be differently coloured, and 11 values for *q* will be produced (contour *B*).

When *p* has the value $\{(a, c), (b, d)\}$, the colouring of the types *S* and *T* will distinguish their elements because neither the transposition (*ab*) nor (*cd*) is itself an automorphism of *p*. Despite the automorphism (*ab*)(*cd*) that applies both simultaneously, the enumeration of q can only exploit the colouring, and although only the 7 values of contour *A* are needed, the 11 values of contour *B* will be generated.

Consider now a different formula:

$$p, q\colon \langle S, T\rangle$$
$$p \cap q = \varnothing$$

The intersection operator imposes a stronger type constraint, and there is thus less symmetry to exploit. As before, *p*'s enumeration can be restricted to 7 values. The enumeration of *q*, however, will be this minimal set only when *p* is fully symmetrical, namely being either the empty or the full relation. When *p* has the value $\{(a, c), (a, d)\}$, the symmetry of the right hand side will cause the elements of *T* to be uniformly coloured, and *q* will have 11 values. When *p* has the value $\{(a, c)\}$, with no symmetry, all 16 values of *q* (contour C) are required.

Consider a formula with a third variable such as

$$p\colon \langle S, T\rangle, q\colon \langle T, V\rangle, r\colon \langle S, V\rangle$$
$$(p \,;\, q) \cap r = \varnothing$$

and suppose the enumeration order is $\langle p, q, r\rangle$. The enumeration of *p* and *q* will proceed exactly as for the formula

$$p \,;\, q = \varnothing$$

The enumeration of r will now depend on the values of both p and q, since the colouring of S will come from the automorphisms of p and the colouring of V from those of q. When the partial assignment is

$$p \mapsto \{(a, c), (b, c)\}, q \mapsto \{(c, e)\}$$

for example, the elements of S will have the same colour, but the elements of T will have a different colour, and 11 values of r will be generated.

If the formula is instead

$$p, q, r: \langle S, T\rangle$$
$$p \cap q \cap r = \varnothing$$

say, the colourings of each type S and V will be obtained from both p and q. For the partial assignment

$$p \mapsto \{(a, c), (b, c)\}, q \mapsto \{(a, c)\}$$

the elements of S are coloured the same by p but differently by q, and must therefore, in the combination, be coloured differently. Both types will colour their elements differently, and 16 values of r will be generated.

Figure 7 shows the colourings of the types produced by various combinations of values of p and q for this formula.

11 Term Symmetry

The generation of relations can actually exploit more permutations than the automorphisms of the partial assignment. A permutation may not leave the values of all the *variables* in the assignment fixed, but it may nevertheless leave the values of all computed *terms* fixed. These *term symmetries* can be viewed as 'semantic automorphisms', since they depend on the meaning of the relational operators.

Suppose, in the analysis of a formula f, we have constructed a partial assignment whose domain is the set of *assigned variables*. Now each term or subformula in f mentions a set of variables. Call the terms and subformulae that mention only assigned variables *closed terms* and *closed subformulae.* If a term or subformula is closed, its value is defined; if not, it contains 'holes' and the assignment must be extended to bind further variables before it has a value. Call the assigned variables that appear only in closed subformulae *formula-closed variables*, and those that appear only in closed terms, but are not formula-closed, *term-closed variables*.

Take a permutation that leaves fixed the values of all variables except for some formula-closed variable. Since uniformly permuting all the variables in a subformula cannot affect its value (by Theorem 1), this permutation maps models to models, and only one of the interpretation and its permutation are needed. To account for such permutations in the colouring scheme, we simply ignore the colouring of formula-closed variables in computing the colouring of the universe. The result is a coarser colouring, and thus a reduced enumeration.

A similar argument can be made for term-closed variables. A permutation that leaves the value of a term fixed can be regarded as an automorphism even if it alters the values of the term's variables. To account for these permutations, we ignore the colouring of term-closed variables, but instead we colour the value of the term itself, and incorporate its colouring in computing the colouring of the universe.

This refinement can never increase the number of assignments considered. It follows directly from the logicality of an operator $\circ$ that

$$Aut\,(p \circ q) \;\supseteq\; Aut\,(p) \cap Aut\,(q)$$

so that any automorphisms that can be exploited using the variable values alone can also be exploited using the value of the term. This does not mean, of course, that using the richer notion of automorphism will always decrease the time taken to find a model, since, as we shall see, it complicates the algorithm (albeit by a constant factor).

Examples

Consider again the formula

$$p\colon \langle S, T\rangle, q\colon \langle T, V\rangle, r\colon \langle S, V\rangle$$
$$(p \,;\, q) \cap r = \varnothing$$

with

$$\tau = \{T \mapsto \{a, b\}, S \mapsto \{c, d\}, V \mapsto \{e, f\}\}$$

and assume the enumeration order $\langle p, q, r\rangle$. For the partial assignment

$$p \mapsto \{(a, c), (b, c)\}, q \mapsto \{(c, e), (c, f), (d, e)\}$$

the basic method would colour the elements of S the same but V differently because (ef) is not an automorphism of q. However, both p and q are term-closed when r is enumerated, and the value of the term $(p \,;\, q)$ corresponding to this assignment is the full relation. Accounting for this allows r's enumeration to be reduced from 11 to 7 values.

12 Basic Algorithm

The user enters a formula with type declarations, and selects a scope bounding the size of each of the declared types. The first phase of the analysis is 'static':

1. The formula is checked for syntax and type errors.
2. An appropriate variable ordering is selected; currently we use an ordering heuristic designed for short-circuiting [DJ96].
3. The most general typing is computed by type inference.

Assume that we now have an array *var* of variables, so that *var* [1] is the first variable and *var* [*n*] is the last; an array *left* of left-side types and *right* of right-side types, so that (*left* [*i*], *right* [*i*]) is the type of the *i*th variable; an array *sig* representing the scope, indexed by types, so that *sig* [*t*] is the number of atoms in type *t*; and an array *tau* indexed by types, so that *tau* [*t*] is the set of atoms for type *t*, having the cardinality *sig* [*t*].

Since type inference associates new types with variables, the scope *sig* cannot be exactly as defined by the user. It is constructed, however, to have the same effect: if the declared types are *left*0 [*i*] and *right*0 [*i*] and the declared scope is *sig*0 [*t*], then the scope satisfies

$$\begin{aligned} &sig\,[left\,[i]] = sig0\,[left0\,[i]] \\ &sig\,[right\,[i]] = sig0\,[right0\,[i]] \end{aligned}$$

so that the size of the *i*th variable is constrained identically.

The second phase of the analysis is 'dynamic':

4. An array *eq* of equivalence relations is allocated; *eq*[*t*] will represent the colouring of the type *t*. Initially, all equivalence classes are set to be as coarse as possible, equating all elements.

5. The assignments are then searched by the recursive function *enumerate* (Figure 8), which displays every assignment *A* that is a model of the formula *F*. The assignment is represented here simply as a list of pairs associating the variable and its value. The initial call to *enumerate* takes an empty assignment list, the array of coarsest equivalences and the index of the first variable.

The core of the algorithm is the call to the iterator *gen*. Given sets of atoms *tl* and *tr* for the left and right sides of the relation to be generated, along with equivalence relations *el* and *er* representing their colourings, the call

$$gen\,(tl, tr, el, er)$$

yields a sequence of triples

$$(x, el', er')$$

where *x* is a relation from the set *tl* to the set *tr*. Suppose *x* has the left colouring *exl*, so that two atoms *a* and *b* in *tl* are related by the equivalence *exl* just when the transposition (*ab*) leaves *x* invariant, and a right colouring *exr* defined similarly. Then *el'* (*er'*) is the coarsest equivalence that is no coarser than *el* and *exl* (*er* and *exr*). The iterator thus yields a new equivalence for each type that accounts for the structure of the new relation value. These then replace the old equivalence classes in the array *eq*.

The iterator yields a set of relation values that may exclude isomorphs. For each possible relation *v* from *tl* to *tr*, at least one relation *v'* is generated such

$$v = \pi\, v'$$

where π is a colour-preserving permutation, that is, a permutation of the nodes of the relation such that left nodes are never mapped to right nodes, and vice versa, and each node is mapped to a node of the same colour (according to the equivalences *el* and *er*).

A minimal *gen* iterator that never yields a superfluous value can be constructed using the techniques described in [McK96]. Generating graphs up to isomorphism is a surprisingly difficult problem; in our implementation, it occupies a significant part of the code.

13 Refined Algorithm

Term symmetry (Section 11) complicates the algorithm. Since the colouring of a variable's value becomes irrelevant when the variable is closed, it is not possible to maintain the colourings of the types alone. Instead, the colourings of the relation values are retained so that the type colouring can be recomputed.

For each term in the formula, a fresh variable is added; when the term has a value, it will be assigned to this *term variable*. The ordering of enumerated variables now induces a mapping from each variable to the set of term variables that are defined once the enumerated variable has been assigned a value.

A second mapping can be defined that associates with each enumerated variable a set of relevant variables that are to be included in determining the colouring of its types. This set includes, in general, both enumerated variables and term variables. When a variable is formula-closed, it will no longer appear in the relevant set at all; when a variable is term-closed, it will not appear, but the appropriate term variable will appear instead.

Term symmetry can only be effective with a good variable ordering. The smaller the set of relevant variables, the coarser the colouring equivalence. A simple but effective ordering attempts to close as many terms as possible at a given stage. Fortunately, the ordering selected for short-circuiting [DJ96] also appears to be reasonable.

The refined enumerate function, *enumerateR*, is shown in Figure 9. Note that the colourings of individual variables are passed around rather than the colourings of types, *leq* [i] and *req* [i] giving the coluring of the left and right sides of the ith variable respectively.

The array *var* of variables must now include the term variables (following the enumerated variables, which are still indexed from 1 to n), and the type arrays *left* and *right* must now include their types too. Additionally, three new arrays must be constructed during static analysis: *termvars*, whose ith element is the set of term variables defined once the $i-1$th variable has been enumerated; *term*, whose ith element is the term corresponding to the term variable *var* [i]; and *relvars*, whose ith element is the set of variables that are relevant for the colouring of the ith variable.

Four new functions are required: *join*, which given two equivalence classes returns the coarsest equivalence no coarser than either; *evalterm*, which returns the value of a term under a given assignment of values to variables; and *colour*, which returns the left and right colourings of a relation value.

The iterator *genR* is essentially the same as *gen*, but is not required to return colourings.

14 Example

To see how the method is applied in practice, consider a specification of call connection in a telephone switch (Figure 10). Although this example is clearly a toy, a realistic specification of telephone switching can be constructed in a similar style [MZ94]. A more interesting application of Nitpick is described in [JD96b].

The first line declares a type *Phone* representing an abstract set of telephones. The system state is declared in the schema *Switch*; it consists of a single relation variable *conns* whose interpretation is that $(p, q) \in conns$ when a call from p to q is active. Members of the domain of the relation (such as p) are making calls; members of the range (such as q) are receiving calls.

The call operation, given as a second schema *Call*, takes two arguments: *from*, the telephone from which the connection is requested, and *to*, the target of the request. The mention of *Switch* before the vertical bar includes two copies of the variables of the *Switch* schema: unprimed, to represent the pre-state (before execution of the operation), and primed, to represent the post-state (after execution). The formula, following the bar, relates the pre- and post-states. In this case, the operation is not total: a new call is constructed and added to the set of connections only if the called party *to* is not already receiving a call.

Two invariants are then declared. First, although *conns* is not a function (because of conference calls), we might reasonably expect it to be injective, so that there is most one phone calling a given phone (and thus a single party to bill for each call). Second, no party should both be making and receiving a call at once: each phone plays a single role. These invariants are packaged as schemas and given the names *OneCaller* and *OneRole*.

Finally, the last two schemas are claims (distinguished by the double colon) that the two invariants are preserved by the operation. The schema names on the right-hand side stand for their respective formulae. When primed, the name of an invariant schema stands for the same formula but with its variables primed. The first claim can thus be read "if a *Call* operation is executed and the invariant *OneCaller* holds prior to execution, then it also holds after execution". Note that when the precondition of the operation does not hold, the hypothesis of the implication is false and the claim is vacuously true. As expected, therefore, the claim says nothing about invalid invocations.

Nitpick will generate instances of the first four schemas. If a scope of 2 is chosen, it will generate the state of Figure 11a as an instance of *OneRole*, for example, and the transition of Figure 11b as an instance of *Call* . The first claim is valid, so Nitpick will find no counterexamples in any scope.

The second claim is not valid, because the specification should have precluded not only *to* being called but also *to* being a caller (and *from* being called); Nitpick finds the counterexample illustrated in Figure 11c.

Figure 12 shows an elaborated version of the specification in which telephones and their directory numbers are distinguished. In this case, *Claim1* is also invalid, because a single phone may have more than one number (that is, *Net* is not injective).

Table 2 shows the performance of the Nitpick checker on the claims of the simpler specification of Figure 10. The reduction factor, shown in parentheses, increases exponentially, in the best case to a factor of almost 5 orders of magnitude. Although both claims contain the same variables, Claim1 shows a greater reduction because it has a weaker typing. Nitpick infers that the left and right sides of *Conns* are not necessarily the same type, and splits the type into two independent types corresponding to the callers and receivers of calls:

> *Conns*, *Conns'*: *Callers* $\leftrightarrow$ *Receivers*
> *from*: *Callers*
> *to*: *Receivers*

Since the formulae of this example are so simple, no further reduction due to term symmetry is obtained.

Table 3 shows the performance for the elaborated example. For each scope, the first line gives the reduction without term symmetry switched on, and the second line includes it. A number of features are worth noting. Again, *Claim1* is checked faster than *Claim2* because of its type structure. The reduction factors are larger than for the simpler example, because there are more variables. Term symmetry, surprisingly, has a larger effect on the smaller scopes. There are very few opportunities for term symmetry in such a small example, and they all involve scalars, which contribute significantly to the symmetry of a relation only when the relation is very small.

In both tables, the figures give the size of the entire search. When the claim is not valid, the checker may be halted at the first counterexample, which may – and often does – come early in the search.

15 Future Work

Three areas of future work are easily identified. First is the question of variable ordering. Its importance is clear for term symmetry, since a poor ordering can prevent the closure of terms, but it matters in the simpler context too.

Second is the question of representing automorphisms. Our colouring scheme has the merit of being straightforward and easy to implement, but it accounts only for automorphisms that can be expressed as products of transpositions that are themselves automorphisms. We have recently developed a scheme that allows a larger class of automorphisms to be exploited without a big increase in cost.

Third is the question of the underlying generation algorithm. Our current implementation of the iterator (described in Section 11) is not minimal: it overgenerates by about 10%, thus inflating the size of the search. In our previous version of Nitpick, we used a minimal iterator provided by McKay (based on the ideas of [McK81, McK94, McK96]). We hope eventually either to adapt this to the new setting, or improve our own algorithm.

There are many other open questions. For example, in practice a different enumeration order might be preferable, such as one that lists pathological cases first, so that empty relations are followed immediately by full ones. A random tester based on our method could also be built, using a generator that selects graphs uniformly across isomorphism classes.

Acknowledgments

Many thanks to Brendan McKay for providing us with code for isomorph-free graph generation; to John Slaney for running his FINDER tool on some of our examples; to Jeannette Wing for her feedback on this work and for piloting an early application of Nitpick; to the CMU Masters of Software Engineering classes of 1996 and 1997 for using Nitpick in their class projects and independent studies; and to the three anonymous reviewers whose suggestions have made a rather long paper a little more digestible.

This research was sponsored in part by a Research Initiation Award from the National Science Foundation, 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, under grant F33615-93-1-1330.

References

[B+95] S. Bensalem, A. Bouajjani, C. Loiseaux and J. Sifakis. Property preserving simulations. *Formal Methods in System Design*, Volume 6, No. 1, January 1995.

[BG94] J. Bowen and M.J.C. Gordon. Z and HOL. *Z User Workshop*, Cambridge, England, 1994, Springer-Verlag Workshops in Computing, pp. 141–167.

[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.

[BJR96] Grady Booch, Ivar Jacobson, James Rumbaugh. *The Unified Modelling*

Language for Object-Oriented Development. Documentation set, version 0.9, Rational Software Corporation, Santa Clare, CA (http://www.rational.com).

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

[BS92] Belaid Benhamou and Lakhdar Sais. Theoretical study of symmetries in propositional calculus and applications. *Automated Deduction (CADE-11): Proc. 11th International Conference on Automated Deduction*, Saratoga Springs, NY, June 1992. *Lecture Notes in Artificial Intelligence*, Vol. 607, Springer-Verlag, Berlin, 1992.

[C+96] E.M. Clarke, R. Enders, T. Filkorn and S. Jha. Exploiting symmetry in temporal logic model checking. in [FMSD96].

[CFJ93] E.M. Clarke, T. Filkorn and S. Jha. Exploiting symmetry in temporal logic model checking. *Fifth International Conference on Computer-Aided Verification*, June 1993.

[CGL92] E.M. Clarke, O. Grumberg and D.E. Long. Model Checking and Abstraction. *Proc. ACM Symposium of Principles of Programming Languages*, January 1992.

[Che76] Peter Chen. The entity-relationship model—toward a unified view of data. *ACM Transactions on Database Systems*, vol. 1, no. 1, pp. 9-36, 1976.

[Cra92] James Crawford. A theoretical analysis of reasoning by symmetry in first-order logic (extended abstract). AAAI-92 Workshop on Tractable Reasoning, 1992, pp. 17–22.

[DGR96] D. Dams, O. Grumberg and R. Gerth. Abstract Interpretation of Reactive Systems. *ACM Transactions on Programming Languages and Systems*, to appear.

[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.

[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.

[DKC89] A.J.J. Dick, P.J. Krause and J. Cozens. Computer-aided transformation of Z into Prolog. *Z User Workshop*, Oxford, 1989, J.E. Nicholls, ed.,*Workshops in Computing*, Springer Verlag, 1990, pp. 71–85.

[ES93] E. Allen Emerson and A. Prasad Sistla. Symmetry and Model Checking. *Fifth International Conference on Computer-Aided Verification*, June 1993.

[ES94] Marcin Engel and Jens Ulrik Skakkebaek. *Applying PVS to Z*. Technical Report ID/DTU ME 3/1, ProCos Project, Department of Computer Science, Technical University of Denmark, Lyngby, Denmark.

[ES96] E. Allen Emerson and A. Prasad Sistla. Symmetry and model checking. in [FMSD96].

[FMSD96] Formal Methods in System Design, Volume 9, Numbers 1/2, Kluwer Academic Publishers, August 1996.

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

[ID93] C. Ip and D. Dill. Better verification through symmetry. *Proc. 11th International Symposium on Computer Hardware Description Languages and their Applications*, April 1993.

[ID96] C. Norris Ip and David L. Dill. Better verification through symmetry. in [FMSD96].

[Jac94] Daniel Jackson. Abstract model checking of infinite specifications. *Proc. Formal Methods Europe* , Barcelona, Spain, October 1994.

[Jac96] Daniel Jackson. Nitpick: A Checkable Specification Language. *Proc.Workshop on Formal Methods in Software Practice*, San Diego, CA, January 1996.

[Jac97] Jonathan Jacky. *The Way of Z: Practical Programming with Formal Methods*. Cambridge University Press, 1997.

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

[JD96a] 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.

[JD96b] Daniel Jackson and Craig A. Damon. Elements of Style: Analyzing a Software Design Feature with a Counterexample Detector. *IEEE Transactions on Software Engineering*, July 1996, Vol. 22, No. 7, pp. 484–495.

[Jha96] Somesh Jha. *Symmetry and induction in model checking*. Doctoral thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, December 1996.

[Jip92] Peter Jipsen. Computer-aided investigations of Relation Algebras. PhD thesis, Dept. of Mathematics, Vanderbuilt University, Nashville, Tennessee, May 1992.

[JJD96] Daniel Jackson, Somesh Jha and Craig A. Damon. Faster Checking of Software Specifications by Eliminating Isomorphs. *Proc. ACM Symp.*

on Principles of Programming Languages, St. Petersburg Beach, FL, January 1996.

[Jon86] Cliff B. Jones, *Systematic Software Development Using VDM*, Prentice Hall International, 1986.

[Jon92] R.B. Jones. ICL ProofPower. *British Computer Society Formal Aspects of Computer Science*, Series 3, 1(1), 1992, pp. 10–13.

[LL91] Peter Gorm Larsen and Poul Bogh Lassen. An executable subset of Meta-IV with loose specification. In S. Prehn, W.J. Toetenel (eds.), *VDM'91: Formal Software Development Methods*, Vol. 1, Lecture Notes in Computer Science 551, Springer-Verlag, 1991.

[McK81] Brendan D. McKay. Practical graph isomorphism. *Congressus Numerantium* 21 (1981), pp. 499–517.

[McK94] Brendan D. McKay. *Nauty User's Guide*, version 1.5. Computer Science Department, Australian National University, GPO Box 4, ACT 2601, Australia.

[McK96] Brendan D. McKay. *Isomorph-free exhaustive generation.* Unpublished manuscript. Computer Science Department, Australian National University, GPO Box 4, ACT 2601, Australia.

[MZ94] Peter Mataga and Pamela Zave. Formal specification of telephone features. *Proc. 8th Z User's Meeting*, pp. 29–50, Springer-Verlag, 1994.

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

[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.

[R+91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy and William Lorensen. *Object-Oriented Modeling and Design.* Prentice Hall, Englewood Cliffs, NJ, 1991.

[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.

[SM88] Sally Shlaer and Stephen Mellor. *Object-oriented Systems Analysis: Modeling the World in Data*. Yourdon Press/Prentice-Hall, 1988.

[SM96] Mark Saaltink and Irwin Meisels. *The Z/EVES Reference Manual (draft)*. Technical Report TR-96-5493-03, ORA Canada, Ottawa, Ontario, Canada, December 1995; revised April 1996, 104 pp.

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

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

[Sta91] P. Starke. Reachability analysis of Petri nets using symmetry. *Syst. Anal. Model. Simul.*, 8 (4/5), pp. 293–303, 1991.

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

[Val91] Samuel H. Valentine. Z^{--}, an executable subset of Z. In J.E. Nicholls (ed.), *Z User Workshop*, York, 1991. Springer-Verlag Workshops in Computing, 1992.

[WD96] Jim Woodcock and Jim Davies. *Using Z: Specification, Refinement, and Proof.* Prentice Hall International Series in Computer Science, 1996.

[WE92] M.M. West and B.M. Eaglestone. Software development: two approaches to animations of Z specifications using Prolog. *Software Engineering Journal*, 7(4), pp. 264–276, July 1992.

[WO80] E.J. Weyuker and T.J. Ostrand. Theories of program testing and the application of revealing subdomains. *IEEE Trans. on Software Engineering*, vol. SE-6, pp. 236–245, May 1980.

[Zha96] Jian Zhang. Constructing Finite Algebras with Falcon. Journal of Automated Reasoning 17(1), pp.1–22, 1996

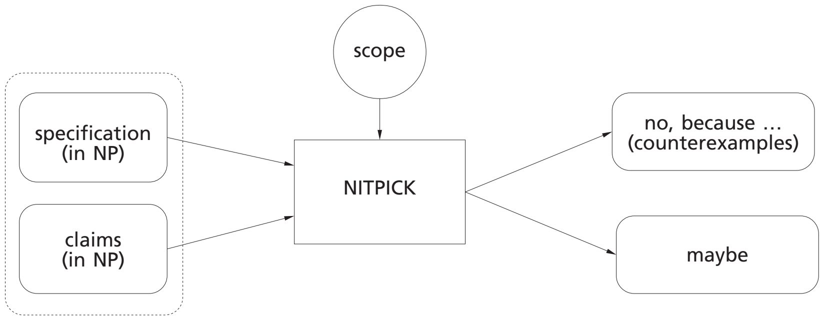


Figure 1: How Nitpick is used to check specifications

k	*#labelled*	*#unlabelled*
1	2	2
2	16	7
3	512	36
4	65536	317
5	3.4e7	5624
6	6.9e10	251610
7	5.6e14	33642660

Table 1: The number of $k \times k$ binary relations with isomorphs included (first column) and excluded (second column)

syntax
$s ::= d\,f \mid d\,s$
$d ::= v{:}\, \langle t, t\rangle$
$f ::= (e = e) \mid \neg f \mid f \wedge f$
$e ::= v \mid \emptyset \mid Id \mid Un \mid e \,;\, e \mid e \cup e \mid e \cap e \mid e \setminus e \mid e^{\sim} \mid e^{+}$
(s *in Spec*, d *in Decl*, f *in Formula*, e *in Term*, v *in Var*, t *in Type*)

typing rules
$e_1{:}\, \langle s, t\rangle, e_2{:}\, \langle t, u\rangle \vdash e_1 \,;\, e_2{:}\, \langle s, u\rangle$
$e_1{:}\, \langle s, t\rangle, e_2{:}\, \langle s, t\rangle \vdash e_1 \circ e_2{:}\, \langle s, t\rangle$ ($\circ$ *is* $\cup, \cap, \setminus$)
$e{:}\, \langle s, t\rangle \vdash e^{\sim}{:}\, \langle t, s\rangle$
$e{:}\, \langle t, t\rangle \vdash e^{+}{:}\, \langle t, t\rangle$
$Id{:}\, \langle t, t\rangle, Un{:}\, \langle s, t\rangle, \emptyset{:}\, \langle s, t\rangle$

semantics of terms
$\#[\![\#]\!]{:}\ Interpretation \times Term \to \mathcal{P}\,(U \times U)$
$I\,[\![v]\!] = A[\![v]\!]$
$I\,[\![\emptyset]\!] = \{\}$
$I\,[\![Id{:}\,\langle s, s\rangle]\!] = \{(x, x) \mid x \in \tau[\![s]\!]\}$
$I\,[\![Un{:}\,\langle s, t\rangle]\!] = \tau[\![s]\!] \times \tau[\![t]\!]$

each operator o has a meaning O
$I\,[\![o(e_1, ..., e_n)]\!] = O\,(I\,[\![e_1]\!], ..., I\,[\![e_n]\!])$

meanings of operators
$p \,;\, q = \{(x, y) \mid \exists z.\ (x, z) \in p \wedge (z, y) \in q\}$
$p \cup q = \{(x, y) \mid (x, y) \in p \vee (x, y) \in q\}$
$p \cap q = \{(x, y) \mid (x, y) \in p \wedge (x, y) \in q\}$
$p \setminus q = \{(x, y) \mid (x, y) \in p \wedge (x, y) \notin q\}$
$p^{\sim} = \{(y, x) \mid (x, y) \in p\}$
$p^{+} = \bigcap \{q \mid p \subseteq q \wedge q \,;\, q \subseteq q\} = p \cup (p \,;\, p) \cup ...$

semantics of formulae
$\#[\![\#]\!]{:}\ Interpretation \times Formula \to Bool$
$I\,[\![s = t]\!] = (I\,[\![s]\!] = I\,[\![t]\!])$
$I\,[\![f \wedge g]\!] = I\,[\![f]\!] \wedge I\,[\![g]\!]$
$I\,[\![\neg f]\!] = \neg\, I\,[\![f]\!]$

Figure 2: Syntax, typing and semantics of a simple relational language

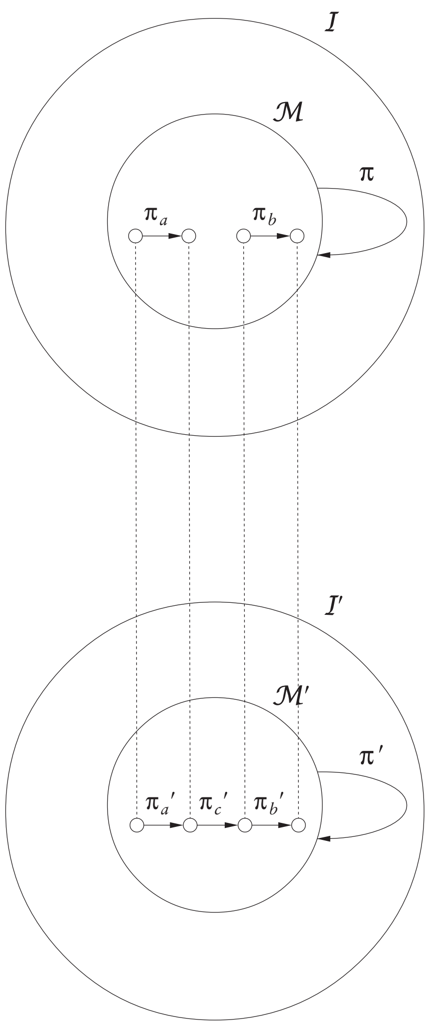
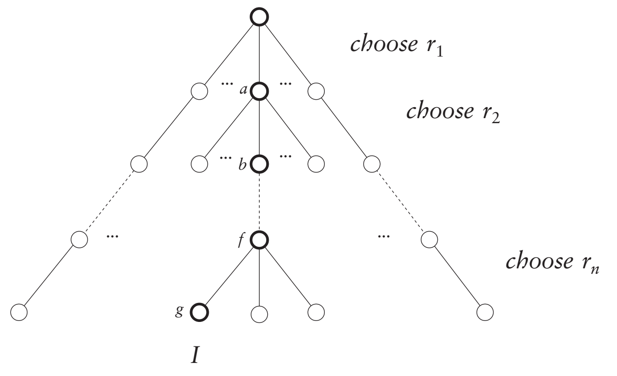


Figure 3: Interpretations, models and orbits



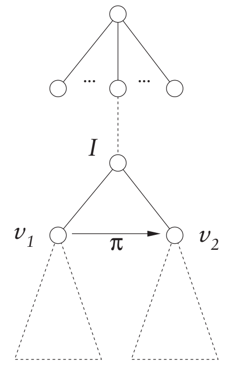


Figure 4 (left): Part of a full enumeration tree
Figure 5 (right): Isomorph elimination at an internal node

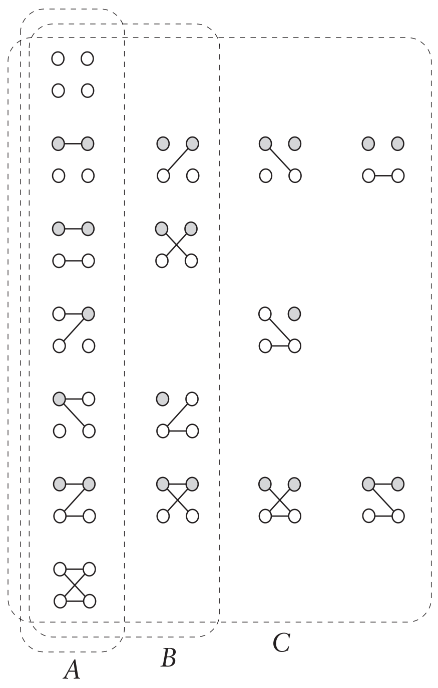


Figure 6: The 2× 2 relations

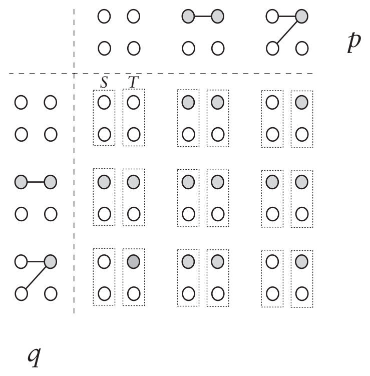


Figure 7: An example of how type colourings are obtained from relation colourings

```
fun enumerate (A, eq, i)
    if $i > n$
        if eval(F, A) then display (A)
        return
    s := left[i]
    t := right[i]
    forall (x, eql, eqr) in gen (tau[s], tau[t], eq[s], eq[t])
        $A'$ := (var [i], x) :: A
        eq [s] := eql
        eq [t] := eqr
        enumerate (A, eq, $i + 1$)
```

Figure 8: Basic Algorithm

```
$fun\ enumerateR\ (A, leq, req, i)$
    $if\ i > n$
        $if\ eval(F, A)\ then\ display\ (A)$
        $return$
    $forall\ j\ in\ termvars\ [i]$
        $v := evalterm\ (term\ [j], A)$
        $leq\ [j], req\ [j] := colour\ (v)$
    $eql := coarsest$
    $eqr := coarsest$
    $forall\ j\ in\ relvars\ [i]$
        $if\ left\ [i] = left\ [j]$
            $eql := join\ (eql, leq\ [j])$
        $if\ right\ [i] = right\ [j]$
            $eqr := join\ (eqr, req\ [j])$
    $forall\ x\ in\ genR\ (tau\ [left[i]], tau\ [right\ [i]], leq, req)$
        $leq\ [i], req\ [i] := colour\ (x)$
        $A := (var\ [i], x) :: A$
        $enumerateR\ (A, leq, req, i + 1)$
```

Figure 9: Refined Algorithm

$[Phone]$

$Switch = [conns: Phone \leftrightarrow Phone]$

$Call\ (from, to: Phone) =$
$\quad [Switch \mid to \notin ran\ conns \ \wedge\ conns' = \ conns \cup \{(from, to)\}]$

$OneCaller = [Switch \mid inj\ conns]$

$OneRole = [Switch \mid dom\ conns \cap ran\ conns = \emptyset]$

$Claim1\ (from, to: Phone) :: Call\ (from, to) \wedge OneCaller \Rightarrow OneCaller'$

$Claim2\ (from, to: Phone) :: Call\ (from, to) \wedge OneRole \Rightarrow OneRole'$

Figure 10: Example specification

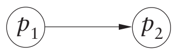


Figure 11a: An instance of the schema *OneRole.*
The graph represents the relation *Conns.*

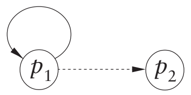


Figure 11b: An instance of the schema *Call*: *p1* is connected to itself, and calls *p2.*
The dotted arc is the new connection; the relation *Conns* is represented by the solid arc, and *Conns'* by both arcs. The variables *from* and *to* have the values *p1* and *p2* respectively.

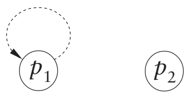


Figure 11c: A counterexample of *Claim2: p1* calls itself.

$[\mathit{Phone}, \mathit{Number}]$

$\mathit{Switch} = [$
$\quad \mathit{Called}\colon \mathit{Phone} \leftrightarrow \mathit{Number}$
$\quad \mathit{Net}\colon \mathit{Number} \rightarrow \mathit{Phone}$
$\quad \mathit{Conns}\colon \mathit{Phone} \leftrightarrow \mathit{Phone}$
$\quad |$
$\quad \mathit{Conns} = \mathit{Called} \mathbin{;} \mathit{Net}]$

$\mathit{Call}\ (\mathit{from}\colon \mathit{Phone}; \mathit{to}\colon \mathit{Number}) = [$
$\quad \mathit{Switch}$
$\quad |$
$\quad \mathit{to} \notin \mathit{ran\ Called}$
$\quad \mathit{Called}' = \mathit{Called} \cup \{(\mathit{from}, \mathit{to})\}$
$\quad \mathit{Net}' = \mathit{Net}]$

$\mathit{OneCaller} = [\mathit{Switch} \mid \mathit{inj\ conns}]$

$\mathit{OneRole} = [\mathit{Switch} \mid \mathit{dom\ conns} \cap \mathit{ran\ conns} = \emptyset]$

$\mathit{Claim1}\ (\mathit{from}, \mathit{to}\colon \mathit{Phone}) :: \mathit{Call}\ (\mathit{from}, \mathit{to}) \wedge \mathit{OneCaller} \Rightarrow \mathit{OneCaller}'$

$\mathit{Claim2}\ (\mathit{from}, \mathit{to}\colon \mathit{Phone}) :: \mathit{Call}\ (\mathit{from}, \mathit{to}) \wedge \mathit{OneRole} \Rightarrow \mathit{OneRole}'$

Figure 12: An elaborated version of the example of Figure 10

scope	*# cases*	*Claim1*	*Claim2*
2	64	16 (4.0)	32 (2.0)
3	4608	168 (27)	692 (7)
4	1048576	2816 (372)	22944 (46)
5	8.39 E8	91986 (9.1e4)	1.43e6 (590)

Table 2: Performance for the example of Figure 10. The reductions are due entirely to isomorph elimination. A scope of *k* means that the enumeration was restricted to cases involving *k* phones or fewer ($\Sigma[\![Phone]\!] = k$). The second column gives the number of cases in the unreduced space. The others give, for the two claims checked, the number of cases, and the reduction factor (in parentheses), for an enumeration of the same space using isomorph elimination. Term symmetry has no effect for this example.

scope	*# cases*	*Claim1*	*Claim2*
2	576	80 (7.2)	144 (4)
		72 (8)	118 (4.9)
3	294,912	2,320 (127)	8,832 (33.4)
		2,248 (131)	8,014 (36.8)
4	6.55e8	130,864 (5e3)	1.29e6 (508)
		130,528 (5e3)	1.23e6 (533)
5	6.52e12	1.69e7 (3.9e5)	5.16e8 (1.3e4)
		1.68e7 (3.9e5)	5.01e8(1.3e4)

Table 3: Reductions for the elaborated example of Figure 12. A scope of *k* means that the enumeration was restricted to cases involving at most *k* phones and *k* numbers ($\Sigma[\![Phone]\!] = \Sigma[\![Number]\!] = k$). The first line for each scope gives the results with term symmetry switched off, the second line with term symmetry on.