Faster Checking of Software Specifications By Eliminating Isomorphs

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

Both software specifications and their intended properties can be expressed in a simple relational language. The claim that a specification satisfies a property becomes a relational formula that can be checked automatically by enumerating the formula's interpretations. Because the number of interpretations is usually huge, this approach has not been thought to be practical. But by eliminating isomorphic interpretations, the enumeration can be reduced substantially, with a factor of roughly $k!$ contributed by each type of k elements.

Keywords: validity checking, model enumeration, symmetry, relational calculus, formal specification, Z notation.

1 Introduction

The success of model checking in verifying hardware designs suggests that systematic enumeration, once considered impractical for all but toy problems, is a promising basis for practical design analysis tools. Enumeration provides two benefits over syntactic techiques: full automation (no need to invent lemmas or devise proof strategies), and the generation of counterexamples when a check fails.

We are investigating enumerative analyses for checking software specifications. Unfortunately, model checking methods that have been remarkably successful in verifying hardware designs and protocols (such as [BC+92, Kur94, Hol91]) have no obvious application to software. In software designs, state explosion arises more from the data structures of a single machine's state than from the product of the control states of several machines, making it hard to analyze even a simple sequential operation.

This paper describes a method for reducing the number of cases a checker must consider by eliminating isomorphic interpretations. It achieves this in two ways: first by noting that permutations of variables may be independent of one another, and second by exploiting symmetries in the underlying data values. These symmetries are independent of the specification being checked, and are trivially determined prior to checking. Detection of symmetry therefore has no runtime cost and imposes no burden on the user. The method gives a reduction, in the number of cases, that increases exponentially in the number of types and the sizes of the their carrier sets. Since the exploitation of symmetry has little runtime overhead, the reduction in cases translates into a corresponding reduction in execution time.

Our specification language is a relational subset of Z, an increasing popular notation for formalizing software designs and requirements [Spi92, Hay93]. Properties are expressed in the same language; checking that a design has a given property amounts to determining the validity of a relational formula.

The method has been implemented in a practical tool, the Nitpick specification checker. An example of the application of Nitpick to a realistic problem is given, along with the rationale underlying its design, in [JD96]. The principles guiding the form of the specification language are discussed in [Jac96]. This paper explains the isomorph elimination method and demonstrates its soundness.

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

1.1 Operations, Invariants and Claims

The transition relation of a state machine is commonly expressed as a formula, with unprimed and primed variables denoting values of state components before and after a transition respectively. The formula

$$x' = x + 1 \ \lor \ x' = x - 1$$

for example, describes a machine that non-deterministically increments and decrements x. In software designs, the transition relation is divided into operations

$$incr \mathrel{\widehat=} x' = x + 1$$
$$decr \mathrel{\widehat=} x' = x - 1$$

whose properties can be investigated independently. The increment operation, for example, preserves a lower bound on x:

$$incr \land (x > min) \ \Rightarrow \ (x' > min)$$

where *min* is some constant of unspecified value. It is convenient to introduce a name for the invariant

$$inv \mathrel{\widehat=} x > min$$

so that the claim can be written more concisely

$$claim \mathrel{\widehat=} incr \land inv \ \Rightarrow \ inv'$$

where F' is short for the formula F with its variables primed. Note that *incr* and *inv* are just names for formulae, and have no semantic significance: there is no notion of a label on a transition. Relationships between operations can also be cast as simple formulae using further syntactic conventions [Spi92]; the formula

$$(incr \,;\, decr) \ \Rightarrow \ x' = x$$

for example, says that decrementing undoes incrementing.

Representing operations, invariants and claims all as logical formulae greatly simplifies both our language and our checking tool. It also allows simulated execution to be incorporated smoothly; the counterexamples generated for the formula

$$not\ incr$$

are exactly the executions of the *incr* operation.

1.2 Relational Formulae

Software designs, unlike their hardware counterparts, involve more complex datatypes than integers and booleans. At a level of abstraction appropriate for design analysis, all datatypes can be expressed in terms of relations. The connections of a phone switch, for example, can be modelled as a relation on phones

$$conns : Phone \leftrightarrow Phone$$

where $(p, q) \in conns$ means that 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 calling operation with two arguments

$$from, to : Phone$$

might then be specified as

$$Call \mathrel{\widehat=} to \notin ran\ conns \ \land \ conns' = conns \cup \{(from, to)\}$$

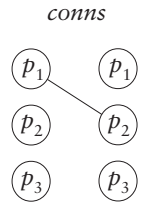

Figure 1a: Legal state satisfying both invariants

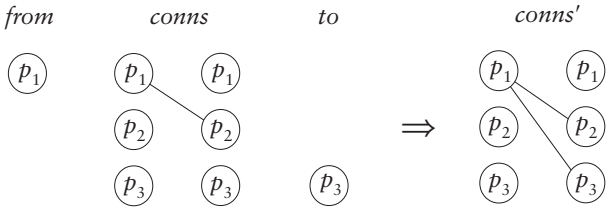

Figure 1b: Transition to legal state

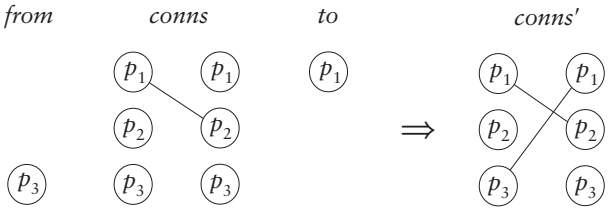

Figure 1c: Transition to illegal state

That is, a new call is constructed and added to the set of connections so long as the called party *to* is not already receiving a call.

Suppose we intend the state to satisfy two invariants. First, although *conns* is not a function (because of conference calls), its transpose should be, so that there is most one phone calling a given phone (and thus a single party to bill for each call):

$$inv_1 \mathrel{\widehat=} func\ (conns^{\sim})$$

Second, no party should both be making and receiving a call at once:

$$inv_2 \mathrel{\widehat=} dom\ conns \cap ran\ conns = \varnothing$$

The claims that the calling operation preserves these invariants are then:

$$Claim_1 \mathrel{\widehat=} Call \land inv_1 \ \Rightarrow \ inv_1'$$
$$Claim_2 \mathrel{\widehat=} Call \land inv_2 \ \Rightarrow \ inv_2'$$

The first claim is valid. The second is not, because the specification should have precluded not only *to* being called but also *to* being a caller (and *from* being called); it has the counterexample:

$$conns = \{(p_1, p_2)\}$$
$$from = p_3$$
$$to = p_1$$

Figure 1 illustrates some of these points. It shows an instance of the state satisfying both invariants, in which p_1 has established a call to p_2 (1a); the transition from this state corresponding to an execution of *Call* with arguments $from = p_1$ and $to = p_3$, resulting again in a legal state in which p_1 is conferenced to p_2 and p_3 (1b); and the counterexample, a transition leading to a state that violates inv_2 (1c).

1.3 Case Enumeration and State Explosion

A claim about a specification holds if it is true for every case, that is, every assignment of values to variables. A case may comprise one state (for a claim that the state space definition satisfies an invariant), two states (for a claim about an operation, as in the example above) or perhaps more (for a claim about a sequence of operations).

To check a claim, we might enumerate cases until one is found for which the formula evaluates to false. If the property does hold and there is no counterexample, then since the relations are generally unbounded, enumeration would not terminate. But this is rare. Most designs are flawed, and, in our experience so far, exhibit small counterexamples. Our tool therefore conducts the enumeration within a *scope* specified by the user (3 phones, say) and always terminates.

Even for a small finite scope, the number of cases can still be enormous. The primary challenge of our work, like that of model checking in hardware, is thus to overcome a state explosion problem, but one of different nature. It arises from the growth in the number of underlying data values as the scope increases: for k phones, there are $2^{k \times k}$ values of *conns*.

Each of the claims above ($Claim_1$ and $Claim_2$), for example, has a space of more than 2 million cases when *Phone* is restricted to have at most 3 values. Our tool determines, by static analysis, that the variable *conns'* need not be enumerated independently, but can be derived from the other variables. The actual formula checked for $Claim_1$ is thus

$$func\ (conns^{\sim}) \ \wedge\ to \notin ran\ conns \\ \Rightarrow\ func\ (conns \cup \{(from, to)\})^{\sim}$$

which has only 4608 cases for 3 phones. The method described in this paper reduces the space further to 167 cases. As we shall see, the reduction depends on the structure of the formula; a smaller reduction is obtained for $Claim_2$. But, encouragingly, as the scope is increased, the reduction factor increases exponentially (see Table 1, discussed in Section 9).

1.4 Overview of Isomorph Elimination

Our method works by eliminating isomorphic interpretations. Since the values of the basic type *Phone* have no structure, the labelling of a case such as

$$from = p_1, to = p_2, conns = \{(p_3, p_3)\}$$

has no significance; permuting $\{p_1, p_2, p_3\}$ can have no effect on the evaluation of the formula. There are 6 permutations of 3 elements, so at a stroke we can identify 5 cases equivalent to this one. But the space is not reduced by a factor of 6, because some cases have symmetries—permutations under which they are invariant. The permutation that exchanges p_1 and p_2 is a symmetry of

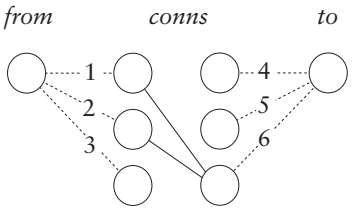

Figure 2: Generating assignments for $Claim_1$ for a fixed, canonical value of *conns* by varying its wirings to the other variables

$$from = p_3, to = p_3, conns = \{(p_1, p_3), (p_2, p_3)\}$$

for example, so of the 5 equivalent cases, 2 are identical anyway.

Our method does not eliminate isomorphs by computing permutations, but avoids their generation in the first place. Consider a particular instance of *conns* (such as any of those shown in Figure 1). The actual labels do not matter; what determines the value of the formula is simply the *relative* labelling of *from*, *to* and *conns*. Instead of generating different labellings, we therefore vary the relationships between variables. Only isomorphically distinct values of each variable are generated; the effect of labelling is accounted for by enumerating, additionally, bijections between the variables. The form of these bijections, which we call 'wirings', and how they are generated, are the focus of this paper.

Which variables must be related? This depends on the structure of the formula. $Claim_1$ tests whether *conns* is a function, but does not compare its domain and range. So the only relative labellings that matter are between *from* and the left side of *conns* and between *to* and the right side. $Claim_2$, on the other hand, tests whether the domain and range of *conns* intersect, and so their relationship will matter.

Figure 2 shows the assignments that can arise for $Claim_1$ when *conns* has a canonical value that maps two phones to one phone. There are two wirings, one relating *from* and the left side of *conns*, one relating *to* and the right side of *conns*. Each wiring has three values, shown as dotted lines, so there are at most 9 distinct interpretations for this canonical value of *conns*. The symmetry in *conns*, which (as noted above) seems to detract from the available savings, can now be turned to our advantage. It renders wiring 1 equivalent to wiring 2, and 4 equivalent to 5, so only 4 of the 9 assignments need be checked.

It may help to cast this argument in terms of explicit labels, even though the method works with canonical values and wirings. For the case

$$from = p_1, to = p_1, conns = \{(p_1, p_3), (p_2, p_3)\}$$

it is clear that exchanging p_1 and p_2 giving

$$from = p_2, to = p_2, conns = \{(p_1, p_3), (p_2, p_3)\}$$

will have no effect on the meaning of $Claim_1$. But permuting the elements of the basic type is not the only way to identify isomorphic cases. Less obviously, we can exchange the values of *from* and *to* independently, giving

$$from = p_2, to = p_1, conns = \{(p_1, p_3), (p_2, p_3)\}$$

for example. The reason is that the formula never compares *from* and *to*, nor does it compare the elements in the domain of *conns* to elements in its range. Since p_1 and p_2 play symmetrical roles in the domain of *conns* (because both are mapped to p_3) and in the range (because nothing is mapped to either), p_1 and p_2 can be exchanged in *from* and *to* independently. In $Claim_2$, on the other hand, the domain and range elements of *conns* are indeed compared, and the independent permutation of *from* and *to* does not generally give isomorphs.

In summary, the reduction arises both from the independence of wirings and from equivalences on wirings induced by the symmetry of the values of variables. In fact, the notion of wirings alone can increase the space, since permuting a relation on both sides simultaneously can leave it invariant. Exploiting symmetry both compensates for this and brings further reductions.

Whether a wiring is placed between two variables depends on whether their values are independent in the formula. This is easily determined by type inference. If two variables (or relation sides) have different types, their values are never compared, and they can be permuted independently; therefore no wiring is inserted between them. In $Claim_1$, for example, *conns* can be given the type

$$conns: Callers \leftrightarrow Receivers$$

and so the left and right sides of *conns* need not be wired together, but in $Claim_2$, they have matching types and are thus wired.

Our method consists of the following steps. In a preliminary static analysis, a type inference is applied to the formula to be checked, and for each type equivalence class obtained, a collection of wiring variables is introduced. Then the assignments are enumerated. Each variable is assigned every possible isomorphically distinct value; and for each combination of variable values, the values of the wiring variables are enumerated. The assignment of canonical values to the formula variables induces, by its underlying symmetries, equivalences on the values of wirings, so that not all wiring values need be generated.

2 Syntax of Relational Calculus

Our notation is a relational subset of Z. Variables may be scalars, sets or binary relations over unstructured types. Formulae and their variable declarations are combined into named 'schemas' which help structure the specification and its associated claims.

Here, for the purpose of explaining the isomorph elimination method, we consider a much simpler language—the relational calculus—that serves as a kernel into which the constructs of the practical notation can be readily translated. Its abstract syntax is:

$$f ::= e \subseteq e \mid func(e) \mid \neg f \mid f \wedge f$$
$$e ::= v \mid O \mid J \mid L \mid e\,;e \mid e \cup e \mid e \cap e \mid e^{\sim} \mid e^{\circ}$$

where f is a formula, e is a relational expression and v is a relational variable. O, J and L are constants denoting the empty, identity and universal relations respectively; $e^{\sim}$ (e°) is the transpose (complement) of e. The formula $func(e)$ is true when e denotes a function.*

Although purely relational, this language can express familiar notions of sets and scalars [SS93]. A subset of a set X can be modelled as a relation u on $X \times Y$ that pairs each element of u with every element of Y, so that a relation u denotes a set when

$$u\,;L = u$$

where $r = s$ is short for $r \subseteq s \wedge s \supseteq r$. The domain of a relation r, namely the set of elements it maps from ($\{a \mid \exists\, (a, b) \in r\}$), and its range, the set of elements it maps to ($\{b \mid \exists\, (a, b) \in r\}$), are then easily defined:

$$dom\ r \mathrel{\widehat=} r\,;L$$
$$ran\ r \mathrel{\widehat=} r^{\sim}\,;L$$

Similarly, a scalar x in X is a relation on $X \times Y$ that pairs the single element x with every element of Y, so that a relation x denotes a point when

$$x\,;L = x \ \wedge\ x\,;x^{\sim} \subseteq J \ \wedge\ x \neq O$$

A pair of points (x, y) is thus represented by the relation $x\,;y^{\sim}$.

Example. Writing C for *conns*, and T and F for the relations representing the points *to* and *from*, the formula $Claim_1$

$$func\ (conns^{\sim}) \ \wedge\ to \notin ran\ conns \Rightarrow func\ (conns \cup \{(from, to)\})^{\sim}$$

becomes

$$\neg((func\ (C^{\sim}) \ \wedge\ \neg T \subseteq C^{\sim}\,;L) \ \wedge\ \neg func\ ((C \cup\ (F\,;T^{\sim}))^{\sim})$$

with the additional constraints

$$F\,;L = F \ \wedge\ F\,;F^{\sim} \subseteq J \ \wedge\ \neg F \subseteq O$$
$$\wedge\ T\,;L = T \ \wedge\ T\,;T^{\sim} \subseteq J \ \wedge\ \neg T \subseteq O.$$

The formula for $Claim_2$

$$dom\ conns \cap ran\ conns = \varnothing \wedge\ to \notin ran\ conns$$
$$\Rightarrow\ dom\ (conns \cup \{(from, to)\}) \cap ran\ (conns \cup \{(from, to)\}) = \varnothing$$

noting the simplification

$$dom\ conns \cap ran\ conns = \varnothing \ \Leftrightarrow\ conns\,;conns^{\sim} = \varnothing$$

becomes

$$\neg((C\,;C^{\sim} = O \ \wedge\ \neg T \subseteq C^{\sim}\,;L) \wedge\ \neg(C \cup\ (F\,;T^{\sim}))\,;(C \cup\ (F\,;T^{\sim}))^{\sim} = O)$$

with the same additional constraints. □

3 Types

Our typing scheme is a little unusual. Its purpose is to support the identification and construction of wirings between variables. By giving each variable a distinct type, we can express

** func(r) can be formulated, without the need for a special construct, as $r^{\sim}; r \subseteq J$, but this leads to an overconstraint in the type system: it suggests that the left and right elements of r need to be compared when in fact whether r is a function can be determined from its unlabelled shape.*

the placement of a wiring in its type structure alone, without saying explicitly which variables it relates. The standard notion of type is then cast as an equivalence on type names. If this equivalence is incompatible with the declared types of the variables, the formula is ill-typed. Since we are not concerned with type checking, however, we shall ignore the declared types and consider only the inferred equivalence.

A relational expression has a type $\langle A, B\rangle$ consisting of a left type A (the type from which domain elements are drawn) and a right type B (from which range elements are drawn). Each relation variable is given a unique type (that is, neither its left or right type appears in the type of another relation). The relational constants should be regarded as indexed sets, so that each instance of a constant in an expression is distinct, and has its own type.

An expression has a type that is derived from, and induces an equivalence on, the types of its constituent variables. Given expressions s and t with types

$$\begin{aligned} s &: \langle Ls, Rs\rangle \\ t &: \langle Lt, Rt\rangle \end{aligned}$$

the compound expressions have types

$$\begin{aligned} s\,;t &: \langle Ls, Rt\rangle \\ s \cup t &: \langle Ls, Rs\rangle \\ s \cap t &: \langle Ls, Rs\rangle \\ s^{\sim} &: \langle Rs, Ls\rangle \\ s^{\circ} &: \langle Ls, Rs\rangle \end{aligned}$$

The choice of the type of s (rather than of t) for the types of $s \cup t$ and $s \cap t$ is arbitrary and has no significance; the types of s and t will be deemed equivalent anyway.

Writing $e \vdash A \approx B$ for the judgment that in expression (or formula) e, the types A and B are equivalent, the type equivalence induced by an expression is defined to be the smallest equivalence relation that satisfies the matching rules

$$\begin{aligned} s\,;t &\vdash Rs \approx Lt \\ s \cup t &\vdash Ls \approx Lt,\ Rs \approx Rt \\ s \cap t &\vdash Ls \approx Lt,\ Rs \approx Rt \\ s \subseteq t &\vdash Ls \approx Lt,\ Rs \approx Rt \end{aligned}$$

incorporates the equivalences induced by its subexpressions

$$\frac{e \vdash A \approx B}{expr(e) \vdash A \approx B}$$

where $expr(e)$ is any expression in which e appears, and for any instance of the identity relation J with type $\langle Lj, Rj\rangle$, has $Lj \approx Rj$.

Example. Given the typing:

$$\begin{aligned} C &: \langle Lc, Rc\rangle \\ F &: \langle Lf, Rf\rangle \\ T &: \langle Lt, Rt\rangle \\ L &: \langle Ll, Rl\rangle \end{aligned}$$

the formula for $Claim_1$

$$\neg((func\ (C^{\sim}) \ \wedge\ \neg T \subseteq C^{\sim}\,;L) \ \wedge\ \neg func\ ((C \cup\ (F\,;T^{\sim}))^{\sim})$$

induces the equivalence classes

$$\{Lc, Lf, Ll\}\ \{Rc, Lt\}\ \{Rf, Rt, Rl\}$$

The formula for $Claim_2$

$$\neg((C\,;C^{\sim} = O \ \wedge\ \neg T \subseteq C^{\sim}\,;L) \\ \wedge\ \neg(C \cup\ (F\,;T^{\sim}))\,;(C \cup\ (F\,;T^{\sim}))^{\sim} = O)$$

on the other hand, induces

$$\{Lc, Lf, Ll, Rc, Lt\}\ \{Rf, Rt, Rl\}$$

Note that $Claim_2$, because its invariant compares the domain and range elements of *conns*, has a coarser type equivalence, collapsing the first two classes of $Claim_1$ into one. □

4 Conventional Semantics

The meaning of an expression (and thus a formula) is defined with respect to an interpretation that gives values to the relation variables. An interpretation has three components: a finite universe of atoms U, a type assignment τ and a variable assignment λ. The type assignment maps each type name to its carrier, a finite set of atoms from the universe

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

and maps equivalent types to the same sets:

$$S \approx T \ \Rightarrow\ \tau[\![S]\!] = \tau[\![T]\!]$$

The variable assignment

$$\lambda : Var \to \mathcal{P}(U \times U)$$

associates a value—a finite set of pairs—with each relation variable. This value must respect the variable's type, so that if r has type $\langle S, T\rangle$,

$$\lambda[\![r]\!] \subseteq \tau[\![S]\!] \times \tau[\![T]\!]$$

The meaning of relation expressions for a given interpretation $\langle U, \tau, \lambda\rangle$ is given by the function

$$E : Expr \to \mathcal{P}\ (U \times U)$$

defined inductively over the syntax:

$$\begin{aligned} E[\![v]\!] &= \lambda[\![v]\!] \\ E[\![O]\!] &= \varnothing \\ E[\![s\,;t]\!] &= \{(x,y) \mid \exists z.\ (x,z) \in E[\![s]\!] \wedge (z,y) \in E[\![t]\!]\} \\ E[\![s \cup t]\!] &= \{(x,y) \mid (x,y) \in E[\![s]\!] \vee (x,y) \in E[\![t]\!]\} \\ E[\![s \cap t]\!] &= \{(x,y) \mid (x,y) \in E[\![s]\!] \wedge (x,y) \in E[\![t]\!]\} \\ E[\![s^{\sim}]\!] &= \{(y,x) \mid (x,y) \in E[\![s]\!]\} \end{aligned}$$

The meaning of two of the constants, and of complementation, depends on the type assignment:

$$\begin{aligned} J &: \langle S, T\rangle \ \models E[\![J]\!] = \{(x,x) \mid x \in \tau[\![S]\!]\} \\ L &: \langle S, T\rangle \ \models\ E[\![L]\!] = \tau[\![S]\!] \times \tau[\![T]\!] \\ s &: \langle S, T\rangle \ \models E[\![s^{\circ}]\!] = \{(x,y) \in\ \tau[\![S]\!] \times \tau[\![T]\!] \mid (x,y) \notin E[\![s]\!]\} \end{aligned}$$

The meaning of formulae is given by the function

$$M : Formula \to Boolean$$

defined in the obvious way:

$$\begin{aligned} M[\![s \subseteq t]\!] &= \forall (x,y) \in E[\![s]\!].\ (x,y) \in E[\![t]\!] \\ M[\![func\ (s)]\!] &= \forall (x,y), (x,z) \in E[\![s]\!].\ y = z \\ M[\![f \wedge g]\!] &= M[\![f]\!] \wedge M[\![g]\!] \\ M[\![\neg f]\!] &= \neg\, M[\![f]\!] \end{aligned}$$

When we want to make the interpretation explicit, we shall write $I[\![e]\!]$ for the meaning of e (that is, $E[\![e]\!]$) under interpretation I, and shall say that $I \models f$ holds when $M[\![f]\!]$ is true for I. When $I \models f$ holds for all interpretations I, the formula f is *valid*; an interpretation I for which $I \models f$ does not hold is a *counterexample* to f.

Example. Given the type equivalences of $Claim_1$

$$\{Lc, Lf, Ll\}\ \{Rc, Lt\}\ \{Rf, Rt, Rl\}$$

the type assignment

$$\tau[\![Lc]\!] = \tau[\![Lf]\!] = \tau[\![Ll]\!] = Callers = \{c_1, c_2, c_3\}$$
$$\tau[\![Rc]\!] = \tau[\![Rf]\!] = Receivers = \{r_1, r_2, r_3\}$$
$$\tau[\![Rf]\!] = \tau[\![Rt]\!] = \tau[\![Rl]\!] = Dummies = \{d_1, d_2, d_3\}$$

is appropriate for a universe U containing at least the elements of the three sets *Callers*, *Receivers* and *Dummies* (this last set being required to represent scalars as relations). Under the interpretation formed by this type assignment and the variable assignment

$$\lambda[\![C]\!] = \{(c_1, r_1)\}$$
$$\lambda[\![F]\!] = \{c_2\} \times Dummies$$
$$\lambda[\![T]\!] = \{r_2\} \times Dummies$$

the formula for $Claim_1$

$$\neg((func\ (C^{\sim}) \ \wedge\ \neg T \subseteq C^{\sim}\,;L) \ \wedge\ \neg func\ ((C \cup\ (F\,;T^{\sim}))^{\sim})$$

is true. The formula (even with the additional constraints that F and T be scalars) is true for any interpretation, and is thus valid.
The formula for $Claim_2$

$$\neg((C\,;C^{\sim} = O \ \wedge\ \neg T \subseteq C^{\sim}\,;L) \\ \wedge\ \neg(C \cup\ (F\,;T^{\sim}))\,;(C \cup\ (F\,;T^{\sim}))^{\sim} = O)$$

with typing

$$\{Lc, Lf, Ll, Rc, Lt\}\ \{Rf, Rt, Rl\}$$

requires a type assignment that does not distinguish callers and receivers:

$$\tau[\![Lc]\!] = \tau[\![Lf]\!] = ... = \tau[\![Lt]\!] = ... = Phones = \{p_1, p_2, p_3\}$$
$$\tau[\![Rf]\!] = \tau[\![Rt]\!] = \tau[\![Rl]\!] = Dummies = \{d_1, d_2, d_3\}$$

and is not valid, having the counterexample (Figure 1c)

$$\lambda[\![C]\!] = \{(p_1, p_2)\}$$
$$\lambda[\![F]\!] = \{p_3\} \times Dummies$$
$$\lambda[\![T]\!] = \{p_1\} \times Dummies$$ □

5 Wired Semantics

As sketched above (in section 1.4), our method works by enumerating assignments of relation variables to canonical values. Each value of a relation can be relabelled in many ways; the canonical value is any one of these arbitrarily selected. To account for relabellings that can affect the value of the formula, we enumerate bijections, called *wirings*, between the relations. Varying the wirings is equivalent to applying permutations independently to the different relations, but much cheaper, first because some relative permutations do not matter (and are thus not represented by wirings) and second because, due to the symmetry of the canonical values, some values of the wirings themselves can be ignored. To show that this method is sound, we give a semantics in terms of wirings, which is subsequently shown to be compatible with the conventional semantics.

Assume some infinite universe of places $\mathbb{P}$ and an infinite collection of finite relations $\mathbb{R}$ whose domain and range elements are drawn from $\mathbb{P}$. $\mathbb{R}$ contains one relation isomorphic to any finite relation; given any relation

$$q \subseteq U \times U$$

there are bijections v and w and exactly one $r \in \mathbb{R}$ such that

$$v^{\sim}\,;q\,;w = r$$

$\mathbb{R}$ may thus be viewed as a collection of 'unlabelled' relations.

In the conventional semantics, equivalent types were assigned equivalent carriers. This time, equivalent types may have distinct carrier sets, which are related instead by explicit bijections that 'wire' the relations together. A *wiring* $\sigma(S, T)$ is a bijection from the carrier of type S to the carrier of type T:

$$\sigma(S, T) \subseteq \tau[\![S]\!] \times \tau[\![T]\!]$$

The wirings must respect the type equivalence relation, so for all types S, T and U:

$$\sigma(S, T) = \sigma(T, S)^{\sim}$$
$$S \approx T \approx U \ \Rightarrow\ \sigma(S, U) = \sigma(S, T)\,;\sigma(T, U)$$

A wired interpretation $\langle \tau, \lambda, W \rangle$ consists of a type assignment that maps types to carriers

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

a variable assignment that maps variables to canonical relations

$$\lambda\colon Var \to \mathbb{R}$$

and, as before, is constrained to respect the variable's type, and a wiring set W, containing a wiring $\sigma(t, t')$ for every pair of equivalent types t and t'.

Let s and t be relational expressions with types

$$s : \langle Ls, Rs \rangle$$
$$t\colon \langle Lt, Rt \rangle$$

and let J and L be instances of the constants with types:

$$J : \langle Lj, Rj \rangle$$
$$L\colon \langle Ll, Rl \rangle$$

The rules defining the meaning of expressions in the wired semantics are no different to the conventional rules for expressions involving a single relation; for those involving two relations, the wirings must be inserted:

$$E[\![s\,;t]\!] = E[\![s]\!]\ ;\sigma(Rs, Lt)\,;E[\![t]\!]$$
$$E[\![s \cup t]\!] = E[\![s]\!] \cup (\sigma(Ls, Lt)\,;E[\![t]\!]\,;\sigma(Rt, Rs))$$
$$E[\![s \cap t]\!] = E[\![s]\!] \cap (\sigma(Ls, Lt)\,;E[\![t]\!]\,;\sigma(Rt, Rs))$$
$$E[\![J]\!] = \textit{the* } j \in \mathbb{R} \textit{ that is a bijection in } \tau[\![Lj]\!] \to \tau[\![Rj]\!]$$

** A technicality: if no $j \in \mathbb{R}$ can be found as a meaning for J, the type assignment is not well-formed.*

Wirings do not affect the meaning of the logical connectives, nor are they required to determine if a relation is a function. So the only case for which the meaning function on formulae changes is:

$$M[\![s \subseteq t]\!] = E[\![s]\!] \subseteq (\sigma(Ls, Lt)\ ;\ E[\![t]\!]\ ;\ \sigma(Rt, Rs))$$

Example. Let $\mathbb{P} = \{\pi_1, \pi_2, \pi_3, ...\}$, and assume $\mathbb{R}$ contains $\{\pi_1\} \times \{\pi_1, \pi_2, \pi_3\}$ and $\{(\pi_1, \pi_1)\}$. Then a legal type assignment for $Claim_2$

$$\neg((C\ ;\ C^{\sim} = O\ \wedge\ \neg T \subseteq C^{\sim}\ ;\ L) \\ \wedge\ \neg(C \cup\ (F\ ;\ T^{\sim}))\ ;\ (C \cup\ (F\ ;\ T^{\sim}))^{\sim} = O)$$

maps all types to $\{\pi_1, \pi_2, \pi_3\}$. The variable assignment

$$\lambda[\![C]\!] = \{(\pi_1, \pi_1)\}.$$
$$\lambda[\![F]\!] = \lambda[\![T]\!] = \{\pi_1\} \times \{\pi_1, \pi_2, \pi_3\}$$

along with any wiring set that includes the wirings

$$\sigma(Lf, Lc) = \{(\pi_1, \pi_3), (\pi_3, \pi_1), (\pi_2, \pi_2)\}$$
$$\sigma(Lt, Rc) = \{(\pi_1, \pi_2), (\pi_2, \pi_1), (\pi_3, \pi_3)\}$$
$$\sigma(Lt, Lc) = \{(\pi_1, \pi_1), (\pi_2, \pi_2), (\pi_3, \pi_3)\}$$

is a counterexample. □

6 *Relating Semantics*

To relate the two semantics, we shall first show how to construct a wired interpretation from a conventional interpretation and vice versa, and then prove that, under corresponding interpretations, the two semantics are equivalent. Except when clear from the context, we shall subscript wired interpretations and their components (I_w, τ_w, λ_w) to set them apart from conventional ones.

Let the triple $\langle\sigma_1, R, \sigma_2\rangle$ denote the relation value obtained by applying the labelling bijection σ_1 to the left side of the canonical relation R and σ_2 to the right side. Let us write $\sigma(S, U)$ for the function* that labels elements of type S. Then if R has the type $\langle Lr, Rr\rangle$,

$$\langle\sigma(Lr, U), R, \sigma(Rr, U)\rangle = \sigma(U, Lr)\ ;\ R\ ;\ \sigma(Rr, U)$$

To obtain a wired from a conventional interpretation, each labelled relation value must be converted to a triple; the canonical relation is then the value of the relation in the wired assignment, and the labellings are used to construct a wiring. Suppose for some variable r with type $\langle Lr, Rr\rangle$

$$\lambda[\![r]\!] = \langle\sigma(Lr, U), R, \sigma(Rr, U)\rangle$$

where $R \in \mathbb{R}$ is an canonical relation. Then

$$\lambda_w[\![r]\!] = R$$

and the wiring set is obtained from the labellings; for example

$$\sigma(Lr,Rr) = \sigma(Lr, U)\ ;\ \sigma(U, Rr) = \sigma(Lr, U)\ ;\ \sigma(Rr, U)^{\sim}$$

is the wiring from Lr to Rr.

** This is a slight abuse of notation. The labelling function $\sigma(S, U)$ is a bijection, and should strictly be written $\sigma(S, U')$ where U' is an appropriate subset of U.*

To apply the translation in reverse, it is necessary to find a set of labellings $\sigma(T_i, U)$ for all types T_i that is consistent with the wiring, satisfying the constraints such as the one above. Since the left and right types of any relation are unique, this is trivial, each relation's labellings being determined independently.

Example. The conventional counterexample to $Claim_2$:

$$\lambda[\![C]\!] = \{(p_1, p_2)\}$$
$$\lambda[\![F]\!] = \{p_3\} \times Dummies$$
$$\lambda[\![T]\!] = \{p_1\} \times Dummies$$

is related to the wired counterexample

$$\lambda_w[\![C]\!] = \{(\pi_1, \pi_1)\}.$$
$$\lambda_w[\![F]\!] = \lambda[\![T]\!] = \{\pi_1\} \times \{\pi_1, \pi_2, \pi_3\}$$

$$\sigma(Lf, Lc) = \{(\pi_1, \pi_3), (\pi_3, \pi_1), (\pi_2, \pi_2)\}$$
$$\sigma(Lt, Rc) = \{(\pi_1, \pi_2), (\pi_2, \pi_1), (\pi_3, \pi_3)\}$$
$$\sigma(Lt, Lc) = \{(\pi_1, \pi_1), (\pi_2, \pi_2), (\pi_3, \pi_3)\}$$

by the labellings

$$\sigma(Lf, U) = \{(\pi_1, p_3), (\pi_2, p_2), (\pi_3, p_1)\}$$
$$\sigma(Lt, U) = \sigma(Lc, U) = \{(\pi_1, p_1), (\pi_2, p_2), (\pi_3, p_3)\}$$
$$\sigma(Rc, U) = \{(\pi_1, p_2), (\pi_2, p_1), (\pi_3, p_3)\}.$$

Notice that $\lambda[\![C]\!] = \sigma(U, Lc)\ ;\ \lambda_w[\![C]\!]\ ;\ \sigma(U, Rc)$. □

The relationship between the values of the variables in the two semantics applies identically to the values of the expressions:

Lemma 1. For any pair of corresponding interpretations I and I_w, and for any expression e of type $\langle Le, Re\rangle$

$$I[\![e]\!] = \langle\sigma(Le, U), R, \sigma(Re, U)\rangle \Leftrightarrow I_w[\![e]\!] = R$$

From this, it follows that a formula holds under a conventional interpretation exactly when it holds under the corresponding wired interpretation:

Lemma 2. For any formula f and corresponding interpretations I and I_w,

$$I \models f \Leftrightarrow I_w \models f$$

We have demonstrated above how to translate any conventional interpretation into a wired one and vice versa. Consequently, the notions of validity in the two semantics match exactly:

Theorem 1. A formula f is valid in the wired semantics when it is valid in the conventional semantics.

This theorem is the fundamental justification of our checking method. Instead of enumerating assignments that bind relations to labelled relation values and interpreting the formula conventionally, we can enumerate assignments to canonical values, and for each assignment, enumerate wirings and interpret the formula for each combination of assignment and wiring set.

Since, in the translation between semantics, labelled relations are obtained from canonical relations by applying bijec-

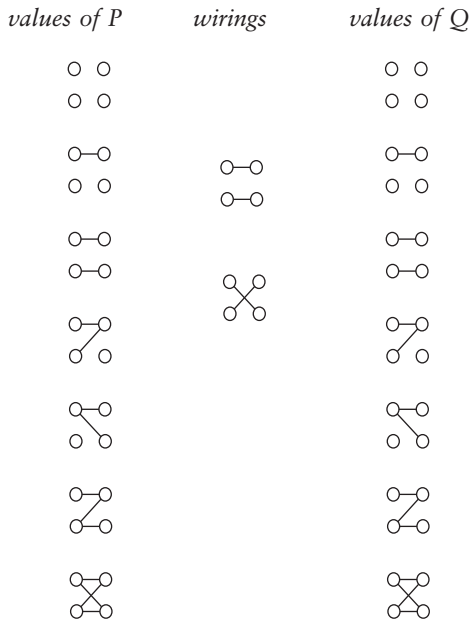

Figure 3: Wired interpretations

tions, a counterexample in the conventional semantics corresponds to one of the same size in the wired semantics. A *scope*

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

associates a bound on the size of the carrier set of each type; an assignment is within a scope Σ if its type assignment τ satisfies, for all types t,

$$\#\tau[\![t]\!] \leq \Sigma[\![t]\!]$$

Formally then, a formula has a wired counterexample in some scope Σ exactly when it has a conventional counterexample in the same scope.

7 Independence of Wirings

Within a given scope, there are generally fewer wired interpretations than conventional ones. The elements of inequivalent types are not related by wirings, so the wired semantics effectively ignores relative permutations that correspond to different interpretations in the conventional semantics.

Example. Take the valid formula

$$\mathit{func}(P) \wedge \mathit{func}(Q) \Rightarrow \mathit{func}(P\,;Q)$$

with the scope $\Sigma[\![Lp]\!] = \Sigma[\![Rp]\!] = \Sigma[\![Rq]\!] = 2$. P and Q have 7 canonical values each, and the wiring $\sigma(Rp, Lq)$ has 2 values, so there are $7 \times 7 \times 2 = 98$ wired interpretations (Figure 3). In contrast, there are 64 labelled values of each relation and thus $64 \times 64 = 256$ conventional interpretations. The reduction here arises because the domain of P and range of Q can be permuted independently without affecting the meaning of the formula. □

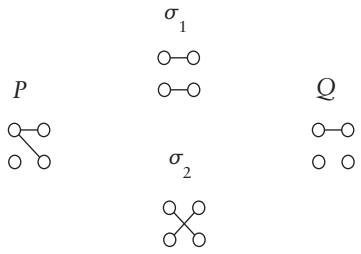

Figure 4: Equivalence of wirings induced by symmetry of values

Usually not even all the wired interpretations are necessary. When a canonical relation value is symmetrical in some domain or range elements, two wirings that differ only in a permutation of those elements will be indistinguishable.

Theorem 2. Suppose that, for two wired interpretations $I_1 = \langle \tau, \lambda, W_1 \rangle$ and $I_2 = \langle \tau, \lambda, W_2 \rangle$ differing only in their wiring sets, any wiring $\sigma_1(S, T)$ in W_1 is equivalent to its counterpart $\sigma_2(S, T)$ in W_2 in the following sense: for the pair of relation values R_a, R_b in the variable assignment (or their transposes) which have the appropriate types

$$R_a\,;\sigma_1\,;R_b = R_a\,;\sigma_2\,;R_b$$

Then the interpretations themselves are equivalent:

$$I_1 \models f \Leftrightarrow I_2 \models f\,.$$

This completes the justification of our method. Instead of enumerating conventional interpretations, we enumerate wired interpretations. Each assignment of canonical relation values to variables has symmetries that induce equivalences in the wirings, so that not all wirings need be generated, and the number of wired interpretations is thus dramatically reduced.

Example. For the same formula, if P has the value $\{(\pi_1, \pi_1), (\pi_1, \pi_2)\}$ and Q has the value $\{(\pi_1, \pi_1)\}$, the wirings

$$\sigma_1(Rp, Lq) = \{(\pi_1, \pi_1), (\pi_2, \pi_2)\}$$
$$\sigma_2 Rp, Lq) = \{(\pi_1, \pi_2), (\pi_2, \pi_1)\}$$

are indistinguishable, since the value of P has the symmetry $(\pi_1\pi_2)$ (Figure 4). Excluding all such unnecessary wirings reduces the number of wired interpretations from 98 to 67. □

Example. Consider enumerating wired interpretations of Claim_1 with $\Sigma[\![Lc]\!] = \Sigma[\![Rc]\!] = 3$ (that is, considering three phones). The wirings shown in Figure 2 are $\sigma(Lf, Lc)$ and $\sigma(Lt, Rc)$. The symmetry of this value of *conns* reduces the number of combinations of these wirings from 9 to 4. □

8 Extensions of the Method

Sometimes enumerating wirings, even modulo symmetry, will produce two assignments that are actually equivalent. To see why, consider a canonical value of a 2×2 relation that is a 2-edge bijection (such as $\{(a,c), (b,d)\}$). According to our method, this value has no symmetry, since there is no permutation which, when applied to one side of the relation, leaves it

scope	*# cases*	*Claim1*	*Claim2*
2	64	16 (4.0)	32 (2.0)
3	4608	167 (24)	713 (5.7)
4	1048576	2707 (387)	33306 (31)
5	8.39 E8	82432 (1.0e5)	4.0e6 (208)

Table 1: Reduction factors for the formulae introduced in Section 1.2. 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.

scope	*# cases*	*Claim1*	*Claim2*
2	576	80 (7.2)	144 (4.0)
3	294912	2060 (143)	6780 (43)
4	6.6e8	85118 (7699)	580433 (1129)
5	6.5e12	6.7e6 (9.8e5)	1.1e8 (5.7e4)

Table 2: Reductions for the elaborated example of Figure 5 (below). A scope of k means that the enumeration was restricted to cases involving at most k phones and k numbers ($\Sigma[\![Phone]\!] = \Sigma[\![Number]\!] = k$).

$$Called : Phone \leftrightarrow Number$$
$$Net : Number \rightarrow Phone$$
$$Conns : Phone \leftrightarrow Phone$$
$$from : Phone$$
$$to : Number$$

$$Conns \mathrel{\widehat{=}} Called \,;\, Net$$
$$Call \mathrel{\widehat{=}} to \notin ran\, Called \;\wedge\; Called' = Called \cup \{(from, to)\} \;\wedge\; Net' = Net$$

$$inv_1 \mathrel{\widehat{=}} func\ (Conns^{\sim})$$
$$inv_2 \mathrel{\widehat{=}} dom\, Conns \cap ran\, Conns = \varnothing$$

$$Claim_1 \mathrel{\widehat{=}} Call \wedge inv_1 \Rightarrow inv_1'$$
$$Claim_2 \mathrel{\widehat{=}} Call \wedge inv_2 \Rightarrow inv_2'$$

Figure 5: An elaborated version of the example of Section 1.2 that distinguishes telephones and their directory numbers. In this case, $Claim_1$ is also invalid, because a single phone may have more than one number (*Net* is not injective).

invariant. Yet permuting *both* sides simultaneously has no effect, and so of 4 possible combinations of the two wirings, two are redundant. In the implemented version of our method, we account for this kind of symmetry by arbitrarily picking the left side of the relation and marking its two places as equivalent. As a result, only the wiring on the right side is varied, and the spurious cases are not generated.

There are symmetries in composite values that might also be exploited. An expression such as P ; Q, for example, often has symmetries when its consistuent values P and Q do not; by identifying such symmetries higher in the parse tree, it should be possible to obtain even greater reductions.

9 Implementation

The method has been implemented within the Nitpick specification checker. It is written in about 30,000 lines of C and runs (so far only) on Macintosh computers. The user loads a file containing the specification and associated claims and selects a scope by choosing a bound for each type; the tool then runs until a counterexample is found or the space is exhausted. Isomorph elimination, in addition to a variety of other reduction mechanisms, may be toggled on and off. This feature is purely for research; we have yet to come across a claim that runs faster when a reduction is turned off.

Generation of canonical relations is delegated to the public-domain Nauty tool [McK81, McK94a, McK94b]. For efficiency, scalars, sets, functions and domain/range operators are implemented directly (rather than being translated into the relational calculus as the formalization above suggests). Because there are relatively few relation values—there are 5624 canonical 5×5 relations—they can usually be cached with their symmetries for an entire session.

Table 1 shows results for the examples discussed above. The tool was set to exhaust the entire space even if counterexamples were found. Each type equivalence class whose types have k elements might be expected to contribute a reduction factor of $k!$, the number of ways to permute those nodes. (The actual reduction factor is lower for small scopes but grows more rapidly, because of symmetry.) Consequently, adding constraints (which tends to collapse type classes together) leads to smaller reductions—witness the effect of the invariant in $Claim_2$—but adding more variables (when it adds new type classes) leads to larger reductions. Table 2 illustrates this latter effect, showing reductions for an elaborated version of our example that distinguishes phones and their numbers (see Figure 3).

On a PowerMac 7100, Nitpick enumerates and checks 2,000–9,000 cases/second; the space of a billion cases for 5 phones is covered in 14 seconds for $Claim_1$ and just over 8 minutes for $Claim_1$. With short-circuiting also activated, greater reductions are obtained. In the elaborated example, for 5 phones and numbers, the space is reduced further to 5.0 E5 cases for $Claim_1$ and 5.2 E6 cases for $Claim_2$, which are covered in 3 and 32 minutes respectively.

10 Discussion

Small data structures have huge numbers of values. For this reason, enumerative analysis of software specifications has been regarded as infeasible.

A specification can be executed if limited to a constructive subset of the language (see [LL91, ELL94] for VDM, [Val91] for Z, and [DK94]). This deprives the specifier of conjunction—arguably the most useful specification construct—but not necessarily non-determinism [LL91]. Aside from our method, theorem proving is the only approach that can accommodate implicit specifications (see [GGH90] for Larch, [B+94] for VDM, [Jon92, BG94, ES94] for Z, and [BH94] for

the relational calculus). For checking safety-critical algorithms, where proof is needed to give perfect assurance, theorem proving will remain indispensable, but its cost rules out its use for everyday specification work.

Abstraction can reduce a huge (and even infinite) space of interpretations to a relatively small number of cases [Jac94], but it has limited applicability and demands ingenuity from the specifier in the choice of abstraction.

Techniques for finding satisfying assignments of formulae have been investigated before. The FINDER tool [Sla94], for example, uses backtracking to find models of a logic with functions and equality. Our example can be translated into its input language, albeit somewhat tediously; relational composition is handled by introducing Skolem constants for the existentially quantified variables. On the examples of this paper, FINDER is much slower than Nitpick. If its input is manually tweaked, however, by adding extra constraints and directives to order the search and break symmetries, FINDER can match Nitpick's performance, and beat it in some cases. FINDER constructs function values element-wise, and so its backtracking prunes the search more efficiently than Nitpick's short-circuiting mechanism, which examines entire relation values. Unfortunately, isomorph elimination and element-wise construction seem to be incompatible and it is unlikely that a tool could incorporate both.

Symmetry has been investigated in the context of automated deduction: [BS92], for example, shows how to take advantage of symmetries in a formula by exchanging one variable for another. But as far as we know, no other method uses symmetry in the assignable values themselves.

Isomorph elimination is related to a number of model checking techniques that exploit symmetry in the transition relation [Sta91, CFJ93, ES93, ID93]. The symmetries identified by our method, in contrast, are not in the entire formula but rather in the individual relation values. Consequently, the symmetries need not be provided by the specifier, and, because each type contributes an exponential factor, our method tends to give much larger reductions.

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, 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 most errors.

Isomorph elimination is only one reduction mechanism in Nitpick's repertoire. Although its reduction factor increases with the scope, it tends to decrease with the complexity of the formula. Reassuringly, another mechanism—short-circuit enumeration—tends to produce better reductions the more complex the formula, so in combination we are able to handle complex formulae and larger scopes. This approach has already made feasible the analysis of small specifications; Nitpick found an anomaly in Microsoft Word's style mechanism in 4 seconds for which a straightforward enumeration would have required 70 hours [JD95]. We are now embarking on the analysis of larger specifications.

Our reductions, although exponential, do not grow as fast as the space of cases, so the checking problem for finite scopes remains fundamentally intractable. Whatever successes we achieve, enumerative checking of software is likely to be a game of brinksmanship, teetering on the edge of intractability.

Acknowledgments

Many thanks to Brendan McKay for providing us with code (not included in his Nauty program) for isomorph-free generation. Thanks also to Merrick Furst and Steve Rudich for some helpful early discussions about graph isomorphism; to Anthony Hall and the referees for their comments on the paper; to Jeannette Wing, Darrell Kindred and the other members of the software group for their criticism of our approach; to John Slaney for running his FINDER tool on our examples; and to Jacob McGuire for collecting the performance numbers of a previous draft.

Appendix: Proofs of Lemmas

Lemma 1. For any pair of corresponding interpretations I and I_w, and for any expression e of type $\langle Le, Re\rangle$

$$I[\![e]\!] = \langle \sigma(Le, U), R, \sigma(Re, U)\rangle \;\Leftrightarrow\; I_w[\![e]\!] = R$$

Proof. By structural induction. Throughout assume that expression s has type $\langle Ls, Rs\rangle$, t has type $\langle Lt, Rt\rangle$ and let

$$\begin{aligned} I[\![s]\!] &= \langle \sigma(Ls, U), RS, \sigma(Rs, U)\rangle \,, I[\![t]\!] \\ &= \langle \sigma(Lt, U), RT, \sigma(Rt, U)\rangle \end{aligned}$$

Case 1: e is a variable (obvious from construction).
Case 2: e is a constant (easy).
Case 3: $e = s\,;t$

$$\begin{aligned} I[\![s\,;t]\!] &= \sigma(U, Ls)\,;\, RS\,;\, \sigma(Rs, U)\,;\, \sigma(U, Lt)\,;\, RT\,;\, \sigma(Rt, U) \\ &\quad \langle \textit{by conventional semantics}\rangle \\ &= \sigma(U, Ls)\,;\; RS\,;\, \sigma(Rs, Lt)\,;\, RT\,;\, \sigma(Rt, U) \\ &\quad \langle \textit{by property of wirings}\rangle \\ &= \sigma(U, Ls)\,;\; I_w[\![s]\!]\,;\, \sigma(Rs, Lt)\,;\, I_w[\![t]\!]\,;\, \sigma(Rt, U) \\ &\quad \langle \textit{by hypothesis}\rangle \\ &= \sigma(U, Ls)\,;\; I_w[\![s\,;t]\!]\,;\, \sigma(Rt, U) \\ &\quad \langle \textit{by wired semantics}\rangle \end{aligned}$$

Case 4: $e = s \cup t$.

$$\begin{aligned} I[\![s \cup t]\!] &= \sigma(U, Ls)\,;\, RS\,;\, \sigma(Rs, U) \\ &\quad \cup\; \sigma(U, Lt)\,;\, RT\,;\, \sigma(Rt, U) \\ &\quad \langle \textit{by conventional semantics}\rangle \\ &= \sigma(U, Ls) \\ &\quad ;\, (RS \cup\; \sigma(Ls, U)\,;\, \sigma(U, Lt)\,;\, RT\,;\, \sigma(Rt, U)\,;\, \sigma(U, Rs)) \\ &\quad ;\, \sigma(Rs, U) \\ &\quad \langle \textit{by property of wirings}\rangle \\ &= \sigma(U, Ls)\;\; ;\, (RS \cup\; \sigma(Ls, Lt)\,;\, RT\,;\, \sigma(Rt, Rs))\,;\, \sigma(Rs, U) \\ &\quad \langle \textit{by property of wirings}\rangle \\ &= \sigma(U, Ls) \\ &\quad ;\, (I_w[\![s]\!] \;\cup\; \sigma(Ls, Lt)\,;\, I_w[\![t]\!]\,;\, \sigma(Rt, Rs)) \\ &\quad ;\, \sigma(Rs, U) \\ &\quad \langle \textit{by hypothesis}\rangle \\ &= \sigma(U, Ls)\,;\; I_w[\![s \cup t]\!]\,;\, \sigma(Rt, U) \\ &\quad \langle \textit{by wired semantics}\rangle \end{aligned}$$

Case 5: $e = s \cap t$. Same as case 4.
Case 6: $e = s^{\sim}$.

$$\begin{aligned} I[\![s^{\sim}]\!] &= (\sigma(U, Ls)\,;\, RS\,;\, \sigma(Rs, U))^{\sim} \\ &\quad \langle \textit{by conventional semantics}\rangle \\ &= \sigma(Ls, U)\,;\, RS^{\sim}\,;\, \sigma(U, Rs) \\ &= I_w[\![s]\!]^{\sim} \\ &\quad \langle \textit{by hypothesis}\rangle \\ &= I_w[\![s^{\sim}]\!] \\ &\quad \langle \textit{by wired semantics}\rangle \end{aligned}$$

Case 7: $e = s^{\circ}$. Same as case 6.

Lemma 2. For any formula f and corresponding interpretations I and I_w,

$$I_w \models f \;\Leftrightarrow\; I_w \models f$$

Proof. By structural induction; only the base cases are non-trivial. The cases of the logical connectives are trivial. Assume that expression s has type $\langle Ls, Rs\rangle$, t has type $\langle Lt, Rt\rangle$ and let

$$\begin{aligned} I[\![s]\!] &= \langle \sigma(Ls, U), RS, \sigma(Rs, U)\rangle \\ I[\![t]\!] &= \langle \sigma(Lt, U), RT, \sigma(Rt, U)\rangle \end{aligned}$$

Case 1: $f = s \subseteq t$.

$$\begin{aligned} I[\![t]\!] &= \langle \sigma(Lt, U), RT, \sigma(Rt, U)\rangle \\ &= \sigma(U, Ls)\,;\, \sigma(Ls, U)\,;\, \sigma(U, Lt) \\ &\quad ;\, RT\,;\, \sigma(Rt, U)\,;\, \sigma(U, Rs)\,;\, \sigma(Rs, U) \\ &\quad \langle \textit{by property of wirings}\rangle \\ &= \langle \sigma(Ls, U)\,, \\ &\quad (\sigma(Ls, U)\,;\, \sigma(U, Lt)\,;\, RT\,;\, \sigma(Rt, U)\,;\, \sigma(U, Rs)), \\ &\quad \sigma(Rs, U)\rangle \\ &\quad \langle \textit{by definition of triple}\rangle \\ &= \langle \sigma(Ls, U)\,, (\sigma(Ls, Lt)\,;\, RT\,;\, \sigma(Rt, Rs)),\; \sigma(Rs, U)\rangle \\ &\quad \langle \textit{by property of wirings}\rangle \\ &= \langle \sigma(Ls, U)\,,\; (\sigma(Ls, Lt)\,;\, I_w[\![t]\!]\,;\, \sigma(Rt, Rs)),\; \sigma(Rs, U)\rangle \\ &\quad \langle \textit{by lemma } 1 \textit{ on } RT\rangle \end{aligned}$$

$$\begin{aligned} I[\![s]\!] &= \langle \sigma(Ls, U), RS, \sigma(Rs, U)\rangle \\ &\quad \langle \textit{by definition}\rangle \\ &= \langle \sigma(Ls, U), I_w[\![s]\!], \sigma(Rs, U)\rangle \\ &\quad \langle \textit{by lemma } 1 \textit{ on } RS\rangle \end{aligned}$$

$$\begin{aligned} M[\![s \subseteq t]\!] &= I[\![s]\!] \subseteq I[\![t]\!] \\ &= \langle \sigma(Ls, U),\; I_w[\![s]\!], \sigma(Rs, U)\rangle \\ &\quad \subseteq\; \langle \sigma(Ls, U)\,, (\sigma(Ls, Lt)\,;\, I_w[\![t]\!]\,;\, \sigma(Rt, Rs)), \sigma(Rs, U)\rangle \end{aligned}$$

Now $\langle \sigma, R, \sigma'\rangle \subseteq\; \langle \sigma, R', \sigma'\rangle$ iff $R \subseteq R'$ (since σ and σ' are bijections), so

$$M[\![s \subseteq t]\!] = I_w[\![s]\!] \subseteq\; \sigma(Ls, Lt)\,;\, I_w[\![t]\!]\,;\, \sigma(Rt, Rs) = M_w[\![s \subseteq t]\!]$$

Case 2: $f = \mathit{func}(s)$. $I[\![s]\!] = \langle \sigma(Ls, U), I_w[\![s]\!], \sigma(Rs, U)\rangle$ and the wirings are total bijections, so $I[\![s]\!]$ is a function exactly when $I_w[\![s]\!]$ is.

References

[B+94] J. Bicarregui, J.S. Fitzgerald, P.A. Lindsay, R. Moore and B. Ritchie. *Proof in VDM: A Practitioner's Guide*. FACIT, Springer-Verlag, 1994.

[BC+92] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill and L.J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. Information and Computation, Vol. 98, No. 2, pp. 142–170, June 1992.

[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.

[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.

[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.

[DK94] Jeffrey Douglas and Richard A. Kemmerer. Aslantest: a symbolic execution tool for testing Aslan formal specifications. *Proc. of International Symposium on Software Testing and Analysis*, Seattle, August 1994.

[ELL94] Rene Elmstrom, Peter Gorm Larsen and Poul Bogh Lassen. The IFAD VDM-SL toolbox: a practical approach to formal specifications. *ACM SIGPLAN Notices*, Vol. 29, No. 9, September 1994.

[ES93] E. Allen Emerson and A. Prasad Sistla. Symmetry and Model Checking. *Proc. 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.

[GGH90] Stephen Garland, John Guttag and James Horning. Debugging Larch Shared Language Specifications. *IEEE Transactions on Software Engineering,* Vol 16, No. 9, 1990.

[Hay93] Ian Hayes. *Specification Case Studies*. Second ed. Prentice Hall International (UK) Ltd, 1993.

[Hol91] Gerard J. Holtzmann. *Design and Validation of Computer Protocols*. Prentice Hall Software Series, Prentice Hall, 1991.

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

[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.

[JD96] Daniel Jackson and Craig A. Damon. Elements of Style: Analyzing a Software Design Feature with a Counterexample Detector. *Proc. International Symposium on Software Testing and Analysis*. San Diego, CA, 1996.

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

[Kur86] R.P. Kurshan. *Testing Containment of Omega-Regular Languages*. AT&T Bell Laboratories, Technical Report 1121-861010-33 (1986).

[Kur94] R.P. Kurshan. *Computer-Aided Verification of Coordinating Processes:The Automata-Theoretic Approach*. Princeton University Press, Princeton, New Jersey, 1994.

[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.

[McK94a] Brendan D. McKay. *Nauty User's Guide*, version 1.5. Computer Science Department, Australian National University, GPO Box 4, ACT 2601, Australia.

[McK94b] Brendan D. McKay. *Isomorph-free exhaustive generation*. Unpublished manuscript. Computer Science Department, Australian National University, GPO Box 4, ACT 2601, Australia.

[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.

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

[SS93] Gunther Schmidt and Thomas Strohlein. *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.

[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.

[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.