# Elements of Style: Analyzing a Software Design Feature with a Counterexample Detector

*Daniel Jackson and Craig A. Damon*
*School of Computer Science*
*Carnegie Mellon University*

*Abstract*

We illustrate the application of Nitpick, a specification checker, to the design of a style mechanism for a word processor. The design is cast, along with some expected properties, in a subset of Z. Nitpick checks a property by enumerating all possible cases within some finite bounds, displaying as a counterexample the first case for which the property fails to hold. Unlike animation or execution tools, Nitpick does not require state transitions to be expressed constructively, and unlike theorem provers, operates completely automatically without user intervention. Using a variety of reduction mechanisms, it can cover an enormous number of cases in a reasonable time, so that subtle flaws can be rapidly detected.

Keywords: requirements analysis, design analysis, functional specification, formal specification, Z notation, model checking, exhaustive testing.

## 1 Introduction

We are investigating the automated analysis of software designs. By 'design', we mean what others might call 'functional specification': namely determining the behaviour rather than the architecture of a software system. We prefer the term 'design' because, unless the specification is trivial, its development inevitably involves design decisions.

Our approach goes back to Guttag and Horning's paper 'Formal Specification as a Design Tool' [GH80], in which they show that, having characterized a design in a formal language, quite subtle questions can be framed about the design in the same formalism. Despite advances in theorem proving technology, however, it is still not possible just to feed such questions to a tool that will answer yes or no. Sobered by the apparent intractability of any specification language rich enough to describe interesting properties, researchers have since then turned their attention more to languages and away from analysis.

The analysis of hardware designs, in contrast, has advanced steadily since the early 1980s. Clarke's method, known as *temporal logic model checking* [CES86, BC+90], can now handle realistic designs, and has exposed errors in published protocol standards and contemporary chip designs. Initial skepticism that enumerative approaches are doomed to fail because of state explosion now seems misplaced, and the possibility that techniques such as model checking might apply to software has inspired a renewed interest in automatic analysis. Parnas's tabular specifications [PM90] in particular, on account of their finite nature and practical utility, have recently been the target of novel analysis techniques: by Atlee and Gannon [AG93], who showed how to translate the tables so that Clarke's method can be applied, and by Heitmeyer [Hei95] and Leveson [LH+94], who have devised local consistency checks that establish global properties (such as determinism) by induction. Wing has also applied Clarke's method to software [WV95], abstracting a network protocol by hand into an appropriate finite state machine.

Clarke's method does not, however, apply straightforwardly to most software specifications. Their complexity lies in state space explosion due not so much to concurrency, but rather to data structures. Adding a 3-by-3 binary relation to the state variables of a machine increases the state space by a factor of 512: clearly not many such variables are required to bring even a powerful model checker to its knees. Worse, standard model checkers assume that, given a state, the next state (or set of possible states) can be trivially generated in a single step. Good specification style in languages like Z, VDM and Larch, on the other hand, encourages implicit definition of the transition relation, so that even finding a state's successor calls for search.

The prospects for model checking of software specifica-

*Authors' address: School of Computer Science, Carnegie Mellon University, 5000 Forbes Ave., Pittsburgh, PA 15213. Email: {daniel.jackson, craig.damon}@cs.cmu.edu. WWW: http://www.cs.cmu.edu/~dnj. A free copy of Nitpick (in binary form for the Macintosh) may be obtained from the authors.*

*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.*

Figure 1 (top): Style mechanism in Microsoft Word. Figure 2 (below): Quark Xpress

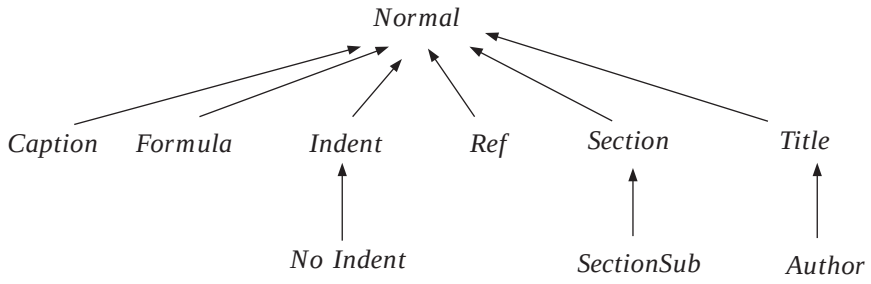


Figure 3: Style hierarchy for this document

tions become even dimmer when we consider that almost all software machines have unbounded state spaces. Some properties of an infinite machine can be evaluated in a finite search by applying an abstraction that partitions the infinite space into a finite number of equivalence classes that can be treated as abstract states, but this method is limited and demands some ingenuity in the choice of abstraction [Jac94a].

Our approach is simply to truncate the state space artificially, checking only within some finite bounds. This means, of course, that Nitpick, our checking tool, can refute but not verify a property of a design; and its output is not proofs but counterexamples. So far, this compromise seems to be paying off. Most designs are flawed, and Nitpick finds a counterexample—even in a huge space—surprisingly fast. Even if we were able to verify design properties automatically, we think that a tool like Nitpick would play a useful role. Good provers tend to be bad refuters; witness how hard it can be, when a theorem prover fails, to work out why, and whether the proof strategy or the theorem was at fault. Optimizing for the failing case is not a new idea, but has motivated the design of many software engineering tools, including theorem provers [GGH90].

Despite Nitpick's partial nature, it retains the spirit of model checking. Philosophically, it might be classed as a debugging or testing tool, since a successful run means only that a counterexample was not found in a finite portion of the infinite state space. But while tools for testing specifications (such as Kemmerer's Aslantest [DK94] or IFAD's animator for the explicit subset of VDM [LL91, ELL94]) are likely to run tens or hundreds of cases, our tool executes millions. Pragmatically then, our tool feels more like a model checker. We obtain very large searches by generating cases systematically (rather than depending on the user to provide them), and by employing a variety of state space reduction methods, principally isomorph elimination and short-circuiting, to skip states that are known in advance not to be counterexamples. In this case study, for example, $10^{23}$ cases are covered in 17 minutes on a PowerMac 7100.

This paper illustrates the application of Nitpick to a small but real example: the design of an operation in a paragraph style mechanism. We show that a variety of design alternatives all lead to violations of a simple and reasonable property. In each case, Nitpick finds a counterexample in a few seconds that sheds some light on where the fault lies. To evaluate how well the technique scales, we give figures for Nitpick's performance on state spaces that are larger than necessary to find these problems, showing that our reduction mechanisms extend its utility beyond the example at hand.

## 2 Paragraph Style Mechanisms

Most modern word processors and desktop publishing programs base formatting on the notion of *paragraph styles*. The exact behaviour of a style mechanism is not a technical detail, but determines quite fundamentally how useful the program will be: what kinds of formatting properties can be retained and reapplied, and how easy it is to undo or alter them.

The document is divided into a sequence of paragraphs, each terminated by a special character (usually carriage return). A paragraph starts on a new line and may be formatted independently of other paragraphs, the user choosing, for example, the face, size and weight of the type, the justification scheme, the addition of space before or after the paragraph, and—in more sophisticated programs—hyphenation policy, leading, etc. A *style* is a named collection of formatting rules that can be applied to a paragraph at once.

A collection of paragraph styles is called a *style sheet*. Here is where the notion of paragraph style brings the greatest benefits. By separating formatting from the tagging of paragraphs with style names, it becomes possible to alter the layout of the document by editing the style sheet alone. A change to a single style will then be applied uniformly to all paragraphs of that style.

Often, paragraphs share common formatting features. We might want to employ the same type face throughout the body of the document, for example, and use common justification and hyphenation policies. Many programs allow the styles to be arranged in an inheritance hierarchy, so that a change to a common feature may be made in a single style and is then propagated automatically to the others.

This paragraph, for example, has the style 'Indent'. The first paragraph of the section has the style 'No Indent', which is declared to be *based on* 'Indent', and has no formatting rules except that its first line indentation is zero. It therefore inherits all the formatting properties of 'Indent', but overrides the indentation. The inheritance hierarchy is usually a tree with a special style 'Normal' at its root that cannot be deleted (see Figure 3). Otherwise, users are free to add and delete styles, and change their places in the hierarchy.

Figures 1 and 2 show the dialogue boxes for editing styles in two widely used programs: a word processor (Microsoft Word), and a desktop publishing program (Quark Xpress). Although the programs perform very different functions—Word being designed primarily for editing and Quark for typographic layout—they share almost the identical paragraph style mechanism. (In fact, this is no accident: it allows a document to be structured in Word, with style names tagged to paragraphs, and then typeset in Quark, where the style names are associated with different formatting rules.) Our analysis will focus on the specification of an operation, *ChangeParent*, that models the sequence of actions performed by the user to change the parent of a style in the hierarchy: in both programs choosing a new parent from the pop-up menu marked 'Based on'. Figure 3 shows the hierarchy for the style sheet of this document; the edge from 'SectionSub' to 'Section' corresponds to the setting of the pop-up menus in the dialogue boxes.

## 3 Some Design Questions

The design of a style mechanism raises some tricky questions. Although different programs may appear to have identical style mechanisms, it is rare to find two with exactly the same behaviour. We have even found significant differences between releases of the same program, and between versions designed for different platforms. Many of the differences are characterized by answers to the following two questions:

*Is the inheritance of a formatting property inferred or declared explicitly by the user?* In most programs, when a style shares a property with its parent, the style sheet cannot distinguish whether the sharing is intentional or accidental. This can be inconvenient. Suppose we want our headings not to be justified (if they run over the end of a line), and so we define the style 'Section' to have the formatting property 'no justify'. If this style's parent, 'Normal' say, is defined to be justified, the 'no justify' property will be stored as an override in the definition of 'Section', shielding it, as desired, from changes to the justification of its parent.

Suppose, however, that the body of the text is set ragged right, so that 'Normal' has the 'no justify' property too. Then the accidental coincidence of parent and child's formatting will be taken as intentional: 'Section' will inherit 'no justify' from 'Normal', and if we now change 'Normal' to 'justify', 'Section' will change too. There is no way to say that 'Section' must be unjustified irrespective of the formatting of its ancestors. Some style mechanisms, such as Framemaker's, solve this problem by allowing the user to specify explicitly when formatting properties are to be inherited, but this solution is not widely adopted as it introduces other complications.

*What happens when the relationship between styles is changed?* In most style mechanisms, changing which style a style is based on has no effect on the formatting associated with styles. Rebasing a style on a different parent may therefore change the child's formatting property list, so that it can maintain the same formatting despite a different context. We might decide, for example, to make 'Ref' a child of 'Indent', to keep the typeface of regular text and references consistent. Performing the move would then cause 'Ref' to acquire the formatting rule 'zero first line indentation' to override the indentation defined for 'Indent'.

Other style mechanisms retain each style's explicit formatting properties, allowing the implicit formatting properties to change instead. Making 'Ref' based on 'Indent' would then have no compensating adjustment, and references would now become indented.

## 4 Modelling the Rudiments of the Style Hierarchy

Our aim in recording elements of the design will not be to *specify* the style mechanism but rather to *model* it at a level of detail just sufficient to allow useful analysis. For this purpose, a subset of Z [Spi89] will suffice, consisting of no more than a simple logic of sets and relations.

We start by declaring style names and formats to be sets of values without structure:

$$[Style, Format]$$

The normal style is represented by a constant:

$$normal : Style$$

Each style is based on at most one other style, so the inheritance graph can be modelled as a partial function *based* from child to parent. With the exception of the normal style, any style on which another style is based must itself be based on some style. These constraints are given in a schema:

$$\begin{array}{l} StyleHierarchy = \\ \quad [based : Style \pfun Style \mid \\ \qquad normal \notin dom\ based \\ \qquad (ran\ based) \setminus \{normal\} \subseteq dom\ based] \end{array}$$

Our intention is that all styles should be mapped until normal is reached, so that the hierarchy forms a tree. But is this true? The hierarchy is connected if every style based on something is directly or indirectly based on *normal*†:

$$\begin{array}{l} Connected = \\ \quad [StyleHierarchy \mid \\ \qquad dom\ (based^+ \rhd \{normal\}) = dom\ based] \end{array}$$

To check this, we invoke Nitpick to read these two schemas along with the claim

$$StyleHierarchy \Rightarrow Connected$$

and the instruction to consider *Style* to have 3 values at most. Nitpick's output is shown in Figure 4, with an adjacent dia-

*† Readers unfamiliar with relational operators may need some help here. $based^+$ is the transitive closure of the based function: it's a relation that maps each style to every one of its ancestors. The range restriction operator $\rhd$ trims away all pairs except those whose second element is in the restricting set (in this case just the singleton $\{normal\}$). Taking the domain of this relation gives all styles that have normal as an ancestor. Finally, the assertion says that these are exactly the styles that are mapped, so that any style mapped at all is mapped directly or indirectly to normal. The relational operators used in the paper are formally defined in Appendix 1.*

```
Checking claim StyleHierarchy => Connected...

 using short circuiting to reduce search
 computing derived variables to reduce search
 using isomorph elimination to reduce search
 restricting Format to elements { f0, f1 }
 restricting Style to elements { normal, s1, s2 }

claim was contradicted in case:
based: Style -> Style is
   { s2 -> s2 }

Finished evaluating claim
After checking 5 cases of 64 possible
 (2 unlabeled)
 (skipped 0 unlabeled cases by short-circuiting)
1 counter example found
Executed 67 instructions checking claim
Elapsed time was 0:00.00
```

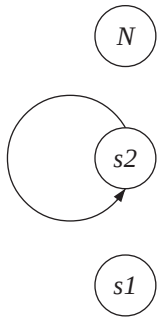


Figure 4: Inheritance hierarchy not connected if cyclic

gram depicting the case found. It stops at the first counterexample, in this case obtained after examining only 5 states. We had neglected the possibility that there be a cycle in the inheritance hierarchy: the style *s2* is based on itself, and, because *based* is a function, cannot therefore be indirectly based on *normal*. So we add a new constraint

$$based^+ \cap Id = \{\}$$

to the *StyleHierarchy* schema, saying that the hierarchy is acyclic. Rechecking the property with the new schema

$$\begin{array}{l} StyleHierarchy = \\ \quad [based: Style \pfun Style \mid \\ \qquad normal \notin dom\ based \\ \qquad (ran\ based) \setminus \{normal\} \subseteq dom\ based \\ \qquad based+ \cap Id = \{\}\ ] \end{array}$$

now yields no counterexamples.

## 5 Adding Formats

Elaborating the state, we now introduce the relationship between styles and formats:

$$\begin{array}{l} Sheet = \\ \quad [StyleHierarchy \\ \qquad delta, assoc : Style \pfun Format \mid \\ \qquad \{normal\} \lhd assoc = \{normal\} \lhd delta \\ \qquad \{normal\} \ndres assoc = \\ \qquad\quad \{normal\} \ndres ((based\ ;\ assoc) \oplus delta)] \end{array}$$

There are two new state components here, each a partial function from styles to formats: *delta* models the explicit definitions, and *assoc* models the implicit association that takes account of inheritance. The two constraints express the relationship between them. The first says that *assoc* and *delta* agree at the style *normal*: that is, the implicit format of *normal* is just its defined format. The second says that, everywhere else, the implicit format of a style is the implicit format of the style it is based on, overridden with its own defined format.

The type *Format*, recall, has no structure. Think of a format as a type face: the state invariant says, for example, that if a style has Palatino as its defined face, its child will also have Palatino, unless it overrides it explicitly with a different choice of face.

This is a gross oversimplification, of course. A format is more accurately modelled as a list of formatting rules ("Palatino, 12pt, Italic, Indented") or a function from property names to properties ("Face: Palatino, Size: 12pt, Style: Italic, Indent: Yes"). Moreover, the combination of formats can be quite complicated. In Microsoft Word (5.0 for the Macintosh), for example, italic is a toggle but underline is not (try setting both in some style, overriding both in a child, and then unsetting both in the parent – the child will end up with italic on but underline off). These complications need not concern us, however, since they play no role in the anomalous behaviours we shall investigate.

Consider now what happens when the parent of a style is changed: that is, we take a style *s* that is based on some style *from*, sever its link and make it based on a different style *to* instead.

$$\begin{array}{l} ChangeParent = \\ \quad [\Delta\ Sheet \\ \qquad s, from, to: Style \mid \\ \qquad\quad from = based\ (s) \\ \qquad\quad s \in dom\ based \\ \qquad\quad based' = based \oplus \{s \mapsto to\} \\ \qquad\quad assoc = assoc' \\ \qquad\quad \{s\} \ndres delta = \{s\} \ndres delta'\ ] \end{array}$$

The first constraint of the schema† defines *from* to be the old parent. The second updates the inheritance relation: the relational override operator $\oplus$ replaces the old link $s \mapsto from$ with the new link $s \mapsto to$, and leaves *based* unchanged elsewhere. So far, there are no other tenable choices. Note, by the way, that the requirement that *based* remain acyclic (an invariant from *StyleHierarchy* imported with *Sheet*) restricts the application of this operation, with an implicit precondition that prevents *to* from being a child of *s*.

The third and fourth constraints embody a design decision. On the assumption that this operation should have no effect on the formatting of the document, and is performed solely with the intent of altering the arrangement of styles in the hierarchy, we have decided to make *assoc* invariant (third constraint). To maintain the implicit association of formats with styles, while allowing a change in the hierarchy, we must allow the explicit definition of formats to change appropriately. The fourth constraint does not say how *delta* changes, but does confine any changes to its value at *s* alone.

Having specified the operation, we can investigate its properties. An obvious property to check is that changing a style's parent and immediately changing it back again should have no

*† The primed variables here denote values after execution of the operation, an the unprimed variables denote values before. $\Delta$ Sheet is a shorthand for two copies of the schema Sheet, one in which the state variables are primed, and one in which they are unprimed. Including it thus adds not only the definitions of the state variables, but also the invariants of the schema, on both pre- and post-states.*

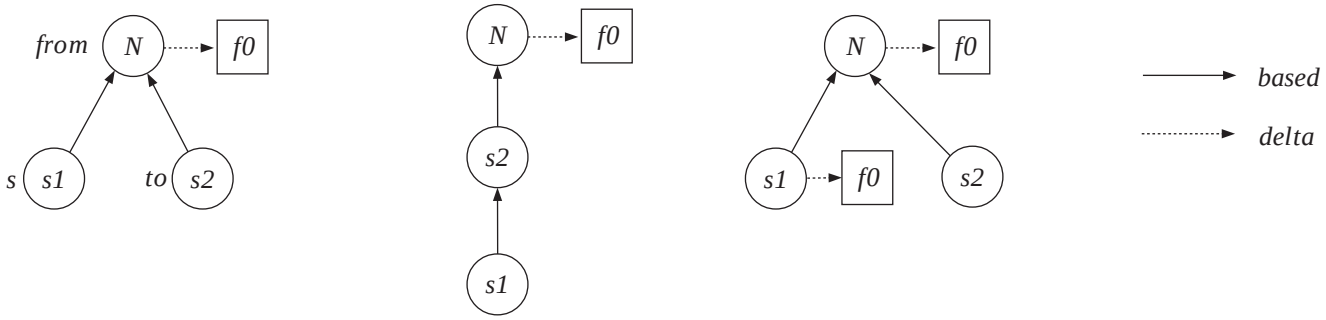


Figure 5: Incomplete specification allows introduction of spurious formatting

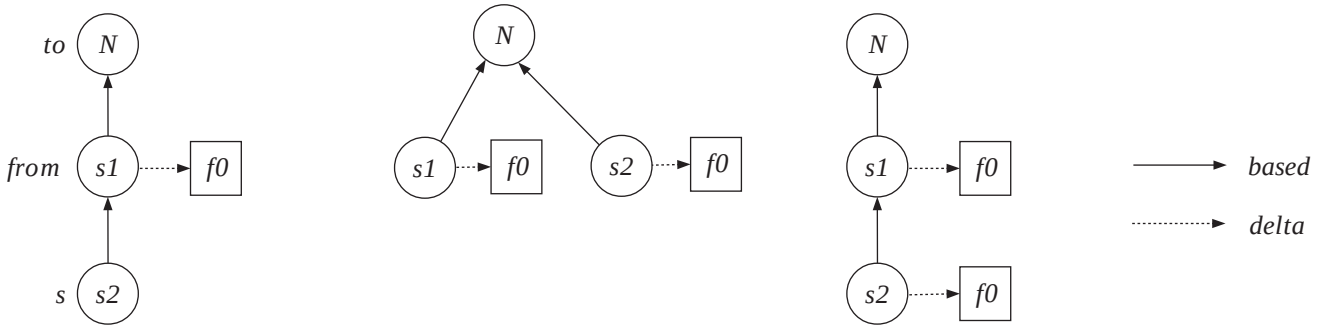


Figure 6: Spurious formatting introduced to retain implicit formatting

effect†:

$$ChangeParent \;;\; ChangeParent\,[from/to, to/from] \Rightarrow \Xi Sheet$$

We invoke Nitpick instructing it to consider at most 3 styles and 2 formats. Figure 5 illustrates the counterexample found, with the graphs representing the progressions through the three states. *Based* is shown with plain arrows; *delta* is shown with dotted arrows (and *assoc* is not shown at all). (See Appendix 2 for the checker's actual output.)

By not constraining *delta* at *s*, we have allowed it to acquire an explicit formatting property that was previously inherited. The counterexample shows this happening in the second execution of *ChangeParent* but it might equally well have happened in the first. Our model is therefore incomplete; we need to say more about the value of *delta* at *s*. One reasonable alternative is to demand that *delta* be unchanged if the implicit formatting of *to* and *from* are the same

$$assoc\,(to) = assoc\,(from) \Rightarrow delta = delta'$$

since in this case *assoc* will remain the same anyway without a change to *delta*.

Adding this constraint to *ChangeParent* and rerunning Nitpick gives a new counterexample (Figure 6). This one cannot be dismissed so easily. The style *s2* must acquire the format *f0* since it no longer inherits it from *s1*. On switching back, it retains the acquired format. This is certainly an undesirable design feature, because it means that alterations to the style hierarchy can cause styles to acquire spurious formatting. But it has no obvious fix, since without extra state, there is no way to distinguish formatting acquired automatically as a byproduct of altering the style hierarchy from formatting defined by the user.

There is an alternative. When the changed style's new parent already has the format it defines, we can eliminate the child's explicit format in favour of the inherited format. This solves the problem of spurious formatting appearing, because when the child is switched back to its old parent, the new formatting will disappear. To specify this behaviour, we add two constraints to *ChangeParent*. The first dictates that an unnecessary format is absorbed:

$$assoc(to) = delta\,(s) \Rightarrow s \notin dom\ delta'$$

The second is a weaker version of the constraint used before; it says that the explicit formatting must be maintained when it can be, and when absorption is not possible:

$$assoc(to) = assoc(from) \land assoc(to) \neq delta(s) \Rightarrow delta = delta'$$

Running Nitpick again, we obtain a third counterexample

*† This formula uses a mass of convenient (if arcane) syntactic conventions. The big semicolon (not to be confused with relational composition) indicates sequential composition, equivalent to conjoining two instances of ChangeParent in which the primed state variables of the first and the unprimed variables of the second are renamed to coincide. The explicit renaming applied to the second instance just switches to and from, so*

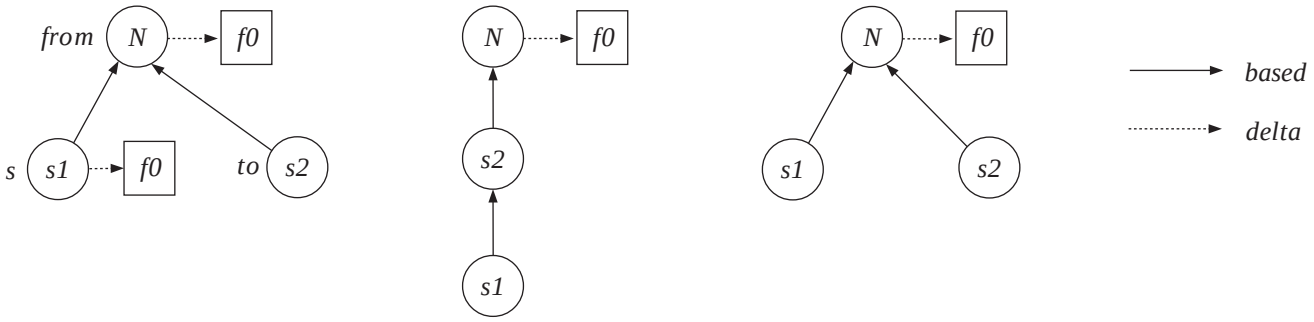


Figure 7: Formatting dropped accidentally

(Figure 7). This time the absorption of the supposedly unnecessary formatting is the problem: it happens not only to formatting that is introduced automatically, but also to the original formatting. So now instead of spurious formatting appearing, genuine formatting is lost.

Microsoft Word (5.0 for the Macintosh) exhibits this behaviour. Again, there is no obvious fix, although the anomaly is perhaps more palatable since with implicit inheritance it is not easy to define explicitly a format that is shared with the parent style. In Word, the difficulty arises specifically because the user changes *assoc* directly and *delta* is automatically changed accordingly. So to ensure that *delta* explicitly contains a property that is shared with the parent, one must first remove the property from the parent; then define it explicitly for the child; and finally restore it for the parent. (Word does not, incidentally, display *delta* in its Style dialogue box, but rather the difference between child's and parent's *assoc*. The value of *delta* can only be found by experiment.)

## 6 The Nitpick Tool

The output reports shown in this paper were generated by Nitpick, our prototype checker. The checker is implemented in about 30,000 lines of C and currently runs on Macintosh computers.

A typical Nitpick session starts with the loading of a text file containing the specification and some claims to be evaluated. The user opens a dialogue box that lists all the given sets and sets a "scope" by selecting a bound for the size of each. A claim is then chosen, and checking invoked; without any further interaction, it displays a counterexample or terminates without finding one. The timings given throughout the paper are in seconds; each of these checks took less than 5 seconds on a PowerMac 7100.

To speed up the search for a counterexample, Nitpick provides a repertoire of reduction mechanisms that can be switched on or off individually. The user also has control over when execution terminates: at the first counterexample, after some fixed number of counterexamples, or when the entire space has been exhausted.

The basic method underlying checking is simple. Nitpick enumerates every possible combination of values for all variables in the claim, and evaluates the claim for each one. When the claim evaluates to false, a counterexample has been found. Because the bounds on the size of any given set are small, even though there may be a huge number of possible combinations, each combination to be checked involves only small objects. This allows efficient bit-based representations. We represent relations as arrays of bitstrings, for example, and can typically compute a composition or closure in 10 or 20 machine instructions.

Naive enumeration works only on the most trivial examples. Finding the counterexample to the claim that *based* is connected (Figure 4) involved a search of at most 64 cases: one for each possible value of the *based* function. As variables are added, the state space grows exponentially; in the *ChangeParent* examples, there are about $10^{12}$ cases to consider. Even evaluating at 20,000 cases/second (the fastest we have observed), the checker would take 2 years to reach the counterexample.

Nitpick therefore employs mechanisms to reduce the search dramatically. These reduction mechanisms have been the focus of our research for the past year, for without them interactive analysis is infeasible. They work by skipping cases that are known to be equivalent to cases already examined. They are fully automatic, sound—no cases are erroneously skipped—and entirely deterministic. The reduction may be viewed as a partitioning of the space of cases into equivalence classes, with each class containing either only counterexamples or containing no counterexamples; the checker may then pick just one case from each class. Three mechanisms have been implemented in the checker:

*Discovery of derived variables*. Implicit specifications cannot be executed in the obvious sense, because a post-condition, unlike an assignment, does not give a prescription for constructing the post-value from the pre-value. Nevertheless, it is common to find that the values of some variables can be derived constructively. In the claim about *ChangeParent*, for example, the variable *delta'* is constrained implicitly, whereas *based'* can be obtained directly from *based*. This means that *based'*, unlike *delta'*, need not be enumerated: its value can be determined from the other enumerated variables. Nitpick automatically discovers such variables by static analysis of the claim. In the claim

$$(a = b) \Rightarrow F$$

for example, either $a$ or $b$ can be derived from the other, but if

| reduction mechanisms | 3 S, 2F | 3S, 3F | 4S, 2F | 4S, 3F | 5S, 3F |
|---|---|---|---|---|---|
| all reductions off | *1.6E12* | *6.9E13* | *2.0E16* | *3.0E18* | *9.9E22* |
| derived variables only | *9.2E8* | *2.9E10* | *1.7E12* | *1.7E14* | *1.1E18* |
| isomorphs & der. var | 9.7E7, 285m | – | – | – | – |
| short-circuiting & der. var | 7.2E3, 3s | 3.3E4, 12s | 4.7E5, 4m | 3.5E6, 25m | – |
| all reductions on | 4.6E3, 2s | 7.2E3, 4s | 6.7E4, 41s | 1.3E5, 83s | 1.6E6, 17m |

Table 1: Performance of the checker on a variety of scopes. The column (mS, nF) is for a scope of m styles and n formats. Italicized entries were calculated; the rest are measured and consist of the number of cases enumerated and the time taken in minutes and seconds.

the equality appears on the other side

$$F \Rightarrow (a = b)$$

neither can be derived. The number of variables that can be derived depends on the style of specification; in addition to explicitly constrained post-state variables, many Z specifications make copious use of redundant state components, which can almost always be derived.

*Short-circuit enumeration*. The variable values are enumerated in a depth-first search of a tree, with one variable enumerated at each level. The variables are ordered so that, if possible, some subexpressions can be evaluated before all variable values have been assigned. If the claim has the structure

$$F(x) \Rightarrow G(x, y, z)$$

for example, where $F(x)$ is a formula involving only $x$, the checker would enumerate $x$ at a higher level of the tree than $y$ or $z$. Now suppose that, during enumeration, a value of $x$ is encountered for which $F(x)$ is false. In this case, the claim is vacuously true. There is no need to compute the value of the consequent $G(x, y, z)$. More significantly, there is no point enumerating *any* values of $y$ and $z$ in combination with this value of $x$, so the entire subtree can be pruned.

*Isomorph elimination*. Consider a counterexample such as that shown in Figure 7. Clearly, since the actual values of the styles and formats are irrelevant, a systematic renaming of styles and formats will yield an equivalent counterexample. For any given case (that is, assignment of values to variables), whether a counterexample or not, there is a huge number of cases that are merely permutations, having the identical shape but labelled differently. The number of permutations increases exponentially with the size of the underlying objects. There are, for example, only 317 distinct 4-by-4 binary relations when labelling is ignored, but 65,536 labelled relations.

We have developed a technique to eliminate these isomorphs. Essentially it involves generating unlabelled relations, functions, sets, etc., and then enumerating possible labellings. Since a global renaming has no effect, we generate only local relabellings that alter the value of one object with respect to the others. By observing symmetries within the objects themselves, we are often able to eliminate a high proportion of these relative labellings too. Our method is described elsewhere [Jac94b, JJ96a]. Its salient features are that the detection and elimination of isomorphs is completely automatic; that the symmetry comes not from the structure of the formula (as in [CFJ93, ID93]), but from the structure of the underlying objects—and thus increases exponentially with the scope, and is not dependent on regularities in the structure of the specification; and that the cost of eliminating isomorphs during execution is linear and small (typically increasing the time per case by a factor of 2 at most).

Nitpick also employs techniques to reduce the time taken to evaluate a case. The claim is compiled into a program executed on each case; by applying standard compiler optimizations, its execution time can be reduced. The checker uses *short-circuit evaluation*, for instance, to halt evaluation when an expression's value is already known from the value of a subexpression: $F \wedge G$, for example, must evaluate to false when $F$ is false irrespective of the value of $G$.

The effect of the reduction mechanisms is shown in Table 1. We ran Nitpick on the last of the *ChangeParent* examples, instructing it not to stop at the first counterexample but to exhaust the entire space of cases. This gives a measure of how many cases might have to be examined in the worst case, given that a counterexample exists within a given scope, even though we often find that counterexamples (especially for simple errors) occur early in the enumeration. A number of conclusions can be drawn from these figures:

• Without the reduction mechanisms, even the smallest scope (3 styles and 2 formats) is infeasible, but with them it is possible to check a much larger scope (5 styles, 3 formats) with a vast number of cases ($10^{23}$).

• For all the scopes, short-circuiting gives the greatest reductions. This has not been our universal experience. Short-circuiting seems to work best with many variables and small scopes; isomorph elimination, in contrast, prefers fewer variables and larger scopes.

• Isomorph elimination gives a reduction of a factor of about 10 in the smallest scope. The effect of short-circuiting is so dramatic that we were unable to measure the reduction due to isomorph elimination alone for larger scopes. We have found, in other experiments, that isomorph elimination increases exponentially with the size of the scope [Jac95], and we have measured reductions of 6 orders of magnitude.

• Isomorph elimination does not work so well in the presence of short-circuiting. For the smallest scope, it reduces the space by less than factor of 2 when short-circuiting is on, as opposed to a factor of 10 when it is off. We suspect this is because isomorph elimination and short-circuiting compete, eliminating many of the same cases.

• With or without isomorph elimination, Nitpick evaluates approximately 5,000 cases/second. The overhead of the elimination mechanism is thus small, and grows only slowly with the scope. The cost of evaluation also grows slightly, since the number of instructions required to multiply two relations, for instance, varies with their size.

A number of other tools are available that can execute formal specifications: the B-toolkit for the AMN language [BCo95], IFADs tool for VDM-SL [LL91, ELL94], Aslantest for Aslan [DK94], and Valentine's interpreter for Z-- [Val91]. None of these generate cases automatically, however, and all of them simulate concrete executions only by requiring adherence to a constructive subset of the language.

## 7 Conclusions

The real benefit of formalizing a design is that it makes automatic analysis possible. Since currently little automated support is available beyond type checking, it is perhaps not surprising that practitioners have not taken up formal specification techniques as enthusiastically as researchers might have hoped. Type and syntax checkers are useful but do not in themselves make formalization worthwhile; and theorem proving, at the other end of the spectrum, demands far too much time and expertise to have everyday application. Certainly the very activity of formalization helps clarify design issues, but its cost is often too high to justify on these grounds alone.

We think that a tool like Nitpick might provide enough benefit to make formalization attractive in everyday design work. The idea of checking a claim by exhaustive enumeration of cases is hardly novel, and we suspect has not been tried because it seems so stupid. After all, the answer to the question "how many cases must be tested to check a property of a real specification?" can only be "too many", since most software designs have unbounded state spaces anyway. But the question "how many cases must be tested before an error is detected?" is very different, and admits only empirical answers. The example in this paper suggests that the answer may still be a very large number, but that, with appropriate reduction mechanisms, errors may be detected in a reasonable time.

Many research issues are yet to be investigated. Selecting the right scope is tricky: too big a scope can give unnecessarily elaborate counterexamples (and may prevent a counterexample from being detected), and too small a scope may exclude counterexamples altogether. Clearly, the right scope cannot be determined algorithmically—for then we would have a decision procedure for an undecidable language—but there may be some approximate analysis that could suggest a good scope. It might even be possible to determine, for some class of claims, that a certain scope is adequate, and for that class the checker would then be an effective verifier.

Whether our approach will scale remains to be seen. We certainly do not imagine that it will ever be possible simply to feed a formal specification of an entire system into Nitpick and obtain useful results. As in our small case study, we expect to have to form appropriate abstractions, perhaps splitting the function of a system into views that can be analyzed independently [JJ95]. The specification must be constructed from the start with checking in mind; not only in the choice of notation, but also in determining which aspects of the function are to be modelled and which are to abstracted away. Apprehension of the kinds of faults that might arise and bear investigation should guide the process, rather than any absolute measure of completeness.

The example described here is remarkable neither in its complexity nor in its commercial impact. Our purpose has been to demonstrate that a class of specifications widely considered immune to automatic analysis can in fact be checked quite effectively. Furthermore, the size of the specification and the effort required in its construction is not proportional to the size of the object being specified. Useful and signficant results about a large program can be obtained by analyzing a much smaller artifact: a specification that models an aspect of its behaviour.


## Acknowledgments

Thanks to Bill Griswold, Anthony Hall and Michael Jackson for their extensive and helpful comments on this work and its presentation; to Gregory Abowd, who joined the first author in some early experiments on Microsoft Word; to the masters of software engineering class at Carnegie Mellon of 1995 who suffered the paragraph style example as a homework exercise; and to Leslie Damon and the anonymous referees for their suggestions.


## Appendix 1: Relational Operators

$S \nrightarrow T$ is the set of partial functions from $S$ to $T$. For relations $P$ and $Q$, and set $S$, the operators used in this paper are defined as follows:

$$\{a \mapsto b\} = \{(a, b)\}$$
$$P \cap Q = \{(a, b) \in P \mid (a, b) \in Q\}$$
$$P \cup Q = \{(a, b) \mid (a, b) \in P \vee (a, b) \in Q\}$$
$$P ; Q = \{(a, b) \mid \exists c.\ (a, c) \in P \wedge (c, b) \in Q\}$$
$$P^+ = P \cup (P ; P) \cup (P ; P ; P) \cup \ldots$$
$$dom\ P = \{a \mid \exists b.\ (a, b) \in P\}$$
$$ran\ P = \{b \mid \exists a.\ (a, b) \in P\}$$
$$S \lhd P = \{(a, b) \mid a \in S\}$$
$$P \rhd S = \{(a, b) \mid b \in S\}$$
$$S \lhd\!\!\!- P = \{(a, b) \mid a \notin S\}$$
$$P \rhd\!\!\!- S = \{(a, b) \mid b \notin S\}$$
$$P \oplus Q = (dom\ Q \lhd\!\!\!- P) \cup Q$$

## Appendix 2: Checker Transcript (Figure 5)

```
Loaded claims: ChangeParent then ChangeParent [from/to, to/from] => Xi
Sheet

Checking claim...

 using short circuiting to reduce search
 computing derived variables to reduce search
 using isomorph elimination to reduce search
 restricting Format to elements { f0, f1 }
 restricting Style to elements { normal, s1, s2 }
```

```
claim was contradicted in case:
based: Style -> Style is
   { s1 -> normal
     s2 -> normal }
based-0: Style -> Style is
   { s1 -> s2
     s2 -> normal }
based': Style -> Style is
   { s1 -> normal
     s2 -> normal }
assoc: Style -> Format is
   { normal -> f0
     s1 -> f0
     s2 -> f0 }
assoc-0: Style -> Format is
   { normal -> f0
     s1 -> f0
     s2 -> f0 }
assoc': Style -> Format is
   { normal -> f0
     s1 -> f0
     s2 -> f0 }
delta: Style -> Format is
   { normal -> f0 }
delta-0: Style -> Format is
   { normal -> f0 }
delta': Style -> Format is
   { normal -> f0
     s1 -> f0 }
s: Style is s1
to: Style is s2
from: Style is normal

Finished evaluating claim
After checking 3252 cases of 9.1833e+08 possible
     (216 unlabeled)
     (skipped 3285 (unlabeled) cases due to short-circuiting)
1 counter example found
Executed 267745 instructions checking claim
Elapsed time was 0:04.20
```

## *Appendix 3: Checker Transcript (Figure 6)*

```
Checking claim...

 using short circuiting to reduce search
 computing derived variables to reduce search
 using isomorph elimination to reduce search
 restricting Format to elements { f0, f1 }
 restricting Style to elements { normal, s1, s2 }

claim was contradicted in case:
based: Style -> Style is
   { s1 -> normal
     s2 -> s1 }
based-0: Style -> Style is
   { s1 -> normal
     s2 -> normal }
based': Style -> Style is
   { s1 -> normal
     s2 -> s1 }
assoc: Style -> Format is
   { s1 -> f0
     s2 -> f0 }
assoc-0: Style -> Format is
   { s1 -> f0
     s2 -> f0 }
assoc': Style -> Format is
   { s1 -> f0
     s2 -> f0 }
delta: Style -> Format is
   { s1 -> f0 }
delta-0: Style -> Format is
   { s1 -> f0
     s2 -> f0 }
delta': Style -> Format is
   { s1 -> f0
     s2 -> f0 }
s: Style is s2
to: Style is normal
from: Style is s1

Finished evaluating claim
After checking 4947 cases of 9.1833e+08 possible
     (296 unlabeled)
     (skipped 4075 (unlabeled) cases due to short-circuiting)
1 counter example found
Executed 424374 instructions checking claim
Elapsed time was 0:04.40
```

## *Appendix 4: Checker Transcript (Figure 7)*

```
Checking claim...

 using short circuiting to reduce search
 computing derived variables to reduce search
 using isomorph elimination to reduce search
 restricting Format to elements { f0, f1 }
 restricting Style to elements { normal, s1, s2 }

claim was contradicted in case:
based: Style -> Style is
   { s1 -> normal
     s2 -> normal }
based-0: Style -> Style is
   { s1 -> s2
     s2 -> normal }
based': Style -> Style is
   { s1 -> normal
     s2 -> normal }
assoc: Style -> Format is
   { normal -> f0
     s1 -> f0
     s2 -> f0 }
assoc-0: Style -> Format is
   { normal -> f0
     s1 -> f0
     s2 -> f0 }
assoc': Style -> Format is
   { normal -> f0
     s1 -> f0
     s2 -> f0 }
delta: Style -> Format is
   { normal -> f0
     s1 -> f0 }
delta-0: Style -> Format is
   { normal -> f0 }
delta': Style -> Format is
   { normal -> f0 }
s: Style is s1
to: Style is s2
from: Style is normal

Finished evaluating claim
After checking 3269 cases of 9.1833e+08 possible
     (221 unlabeled)
     (skipped 3315 (unlabeled) cases due to short-circuiting)
1 counter example found
Executed 268390 instructions checking claim
Elapsed time was 0:04.20
```

*References*


[BCo95] *The B-Technologies: a system for computer aided programming*. B-Core (UK) Limited, Oxford, England, 1995.

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

[BC+90] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill and J. Hwang. Symbolic model checking: $10^{20}$ states and beyond. *Proc. 5th Annual Symposium on Logic in Computer Science*, IEEE Computer Society Press, June 1990.

[CES86] E.M. Clarke, E.A. Emerson and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions of Programming Languages and Systems*, 8(2), pp. 244–263, 1986.

[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. *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.

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

[GH80] John Guttag and James Horning. Formal specification as a design tool. *7th Symposium on Principles of Programming Languages*, Las Vegas, Nevada, Jan. 1980.

[Hei95] Constance Heitmeyer, Bruce Labaw and Daniel Kiskis. Consistency checking of SCR-style requirements specifications. *Proc. RE '95: 2nd IEEE International Symposium on Requirements Engineering*, York, England, March 1995, pp. 56–63.

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

[Jac94a] Daniel Jackson. Abstract model checking of infinite specifications. *Proceedings of Formal Methods Europe Conference*, Barcelona, 1994.

[Jac94b] Daniel Jackson. *Exploiting Symmetry in the Model Checking of Relational Specifications*, Technical Report CMU-CS-94-219, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, December 1994.

[JJ96a] Daniel Jackson and Somesh Jha. Faster Checking of Software Specifications by Eliminating Isomorphs. *Proc. ACM Symp. on Principles of Programming Languages*, St. Petersburg Beach, FL, January 1996.

[JJ96b] Daniel Jackson and Michael Jackson. Problem Decomposition for Reuse. to appear, *Software Engineering Journal, (special issue on viewpoints).*

[LH+94] N.G. Leveson, M.P.E. Heimdahl, H. Hildreth and J.D. Reese. Requirements specification for process-control systems. *IEEE Trans. on Software Engineering*, September 1994, Vol. 20, No. 9, pp. 684–707.

[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.

[PM90] D. Parnas and J. Madey. *Functional documentation for computer systems engineering.* Technical Report TR-90-287, Queen's University, Kingston, Ontario, September 1990.

[Spi89] J.M. Spivey, *The Z Notation: A Reference Manual*, Prentice Hall International, 1989.

[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.

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