Semi-Executable Specifications

Daniel Jackson & Craig Damon
November 1995
CMU-CS-95-216

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract. Executability is usually gained at the expense of implicit specification constructs. This need not be the case. By identifying explicit elements within a specification, and by employing reduction mechanisms to speed up the search required by the implicit elements, it is possible to execute specifications that are usually thought to be amenable only to proof. The paper explains the basic notions behind the Nitpick tool, a checker and simulator for specifications written in a relational language similar to Z: the uniform representation of states, operations and claims as formulae; the identification of derived variables by static analysis; and, briefly, the underlying case enumeration method.

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. Views and conclusions contained in this document are those of the author and should not be interpreted as necessarily representing official policies or endorsements, either express or implied, of ARPA, NSF, TRW or the United States Government.

Keywords: formal specification, operational specification, executable specification, prototyping, simulation, implicit definition, post-conditions, specification checking, relational calculus, Z.

Specifications are traditionally divided into two classes: those that can be executed like programs and those that cannot. Proponents of executability argue that specifications, like programs, can benefit from testing, especially to validate informal requirements. Proponents of implicit specification respond that the inevitable loss of expressive power in constraining a language to be executable outweighs any advantages. This crudely summarizes a long-standing debate in which many other factors (such as the risk of bias) play a role; see [1-5].

We are investigating an approach to specification that combines some of the benefits of both classes. Our specification language, Nitpick, is implicit in style, and is designed to support the pervasive use of invariants commonly found in Z specifications. Post-state variables need not be given by explicit constructions, thus allowing non-determinism and implicit preconditions. Moreover, specifications can be composed with conjunction in the standard fashion.

At the same time, we have built a tool that executes such specifications, essentially by a generate-and-test strategy. Efficient execution must constrain the specification language somehow, of course; the novelty of our approach is that, unlike executable subsets of VDM [6] and Z [7], we do not eliminate implicit constructs. Instead, we restrict the language's datatypes. Allowing only sets and binary relations over 'given' types (sets that have no internal structure) makes it possible to avoid generating values that are isomorphic to ones already produced. This gives an exponential increase in performance without compromising completeness: if a satisfying post-state of some cardinality exists, either that state or one of its isomorphs will be found [8].

Executability was not, initially at least, a goal of our research. The Nitpick tool generates counterexamples to formulae that represent claims about a specification. The claim

$$inv \wedge op \Rightarrow inv'$$

for instance, might say that the operation *op* preserves the invariant *inv*; a counterexample would be an assignment of values to the variables of *op* representing a transition that breaks the invariant. Experimenting with Nitpick, we would often find counterexamples to such claims that made no sense because they appeared not to correspond to specified executions of the operation. At other times, no counterexamples would be found even when the claim appeared to be false. Both kinds of surprises arose because the operation as specified was not the operation we thought we had specified. We were thus led to asserting claims of the form

$$not\ op$$

whose counterexamples are executions of the operation. If this generated bogus transitions, we then knew the transcription of the specification was at fault. If, examining the first few counterexamples, we could not find an expected transition, we would then attempt to force it by constraining the set of prestates

$$not\ (pre \wedge op)$$

or post-states, or both. This activity has proven so useful in practice that we now routinely execute operations before checking their properties; we also produce examples of states satisfying invariants by assertions such as

$$not\ inv$$

The primary benefit of executability is thus mundane but vital: it is extremely effective at catching errors of formalization. These are easy to make, and arguably even easier the more expressive the language. If they are not eliminated early, formal specification can be a retrograde step, itself introducing flaws into the development process.

Our approach blurs the distinction between the sampling of executions and the checking of specification properties, and between implicit and explicit specifications. We have found that constructing a specification with Nitpick in this exploratory style, interleaving writing with sampling and checking, is quite effective. It is also fun.

We have written elsewhere about using Nitpick to check theorems about a specification [9]. Our focus here is on how the blurring of sampling and checking and of implicit and explicit specifica-

tions is achieved. There are three essential ingredients to our approach:

· case enumeration, the underlying generate-and-test method;
· uniform representation of invariants, operations and claims as logical formulae; and
· determination of derived variables by static analysis.

The paper explains these notions in turn, the first only briefly, and then demonstrates their application in a small example. Although very different in flavour, these notions all involve the interplay between language design and checking technology, an area of research that we believe has yet to receive the attention it deserves.

2 *Case Enumeration*

A counterexample to a logical formula is an assignment of values to variables for which the formula evaluates to false. To check a formula then, we can simply enumerate all its possible assignments. In Nitpick, all values are assumed to be finite, so if a formula is not valid, this strategy is complete and must eventually find any counterexample. Of course, 'eventually' may be a long way off: the number of assignments explodes, both because relation variables have a huge number of values ($2^{k \times k}$ for k by k) and because if each variable has k values, there are k^n combinations for n variables.

Nevertheless, for many of the formulae that arise in practice, case enumeration is tractable. In Nitpick, the user selects a *scope* that bounds the size of the carrier set of each given type; the checker then enumerates only the relations and sets that are within the scope. Nitpick reduces the space of assignments principally by two means:

Isomorph elimination allows assignments to be skipped that are isomorphic to assignments already generated [8]; the notion of isomorphism exploits the type structure of the formula, so that different formulae give different reduction factors, typically ranging (for given types with 3–5 elements) from 100 to 10,000.

Short circuiting prunes the search tree by avoiding the generation of values for one variable when the values of others determine that the formula is true. For example, in generating assignments for

$$P(x, y) \Rightarrow Q(x, y, z)$$

where P is a subformula with variables x and y, and Q is a subformula with variables x, y and z, Nitpick will avoid generating values of z to accompany values of x and y that make P false. This reduction mechanism has the nice property that it works better the more complex the specification.

Together, these mechanisms have a dramatic effect. We used Nitpick to analyze the paragraph style feature of Microsoft Word. A scope of 3 styles and 2 formats sufficed to detect an interesting anomaly (both in the specification and the program itself). The search space contained just under a billion assignments, which was reduced to 9,000. The counterexample was found almost instantaneously after examining 4,800 states; exhausting the reduced space took 4 seconds [9].

The enumeration of values takes the type declaration of a variable into account. If a variable is declared as an injective function, for example, Nitpick generates only injective functions, rather than generating arbitrary relations and testing each one. Unlike in a Z schema, therefore, where the difference between constraining a variable in a declaration (above the line) and with a predicate (below the line) is of little consequence, in Nitpick the difference is immense, and may determine whether or not a specification can be checked at all. In addition to the standard repertoire of functions, Nitpick provides a number of other special declaration forms. For example,

a, b, c: *P part X*

declares a, b and c to be sets that partition the set X. P is the name of the partition itself; it can be thought of as the characteristic function that maps elements of X to a set of labels $_a$, $_b$, $_c$, so that the assertion

$S \lhd P = S \lhd P'$

says that the elements of the set S fall in the same equivalence classes in partitions P and P'. Similarly,

s: $seq\, X$

declares s to be a sequence of elements in X, useful for modelling duplicate-free sequences, such as

$\{a \mapsto b, b \mapsto c, c \mapsto d, d \mapsto d\}$

which corresponds to the sequence

$\langle a, b, c, d \rangle$

The declaration is equivalent semantically to declaring s to be a relation

s: $X \leftrightarrow X$

and then asserting a constraint that the relation is a total order on some subset of X (in which each element has a single successor, the last being mapped to itself). There are, however, 512 relations but only 16 sequences.

3 *Formulae*

Nitpicks adopts from Z the idea of representing invariants, operations and claims uniformly as logical formulae. Suppose we have an operation *op* specified as a formula with free variables are x, y, x' and y'. The primed variables by convention denote components of the post-state; the unprimed ones belong to the pre-state. To check that op preserves an invariant inv, specified as a formula whose variables are x and y, we would assert the claim

$op \wedge inv \Rightarrow inv'$

in which inv' denotes a copy of the formula *inv* with its variables primed. The claim is valid if it evaluates to true for all assignments of values to x, y, x' and y'. Instructed to check it, Nitpick will enumerate all assignments within a scope selected by the user, and stop when the first counterexample – an assignment for which the formula is false – is detected. This is the default; the user can choose how many counterexamples are to be detected before enumeration stops, and how many of these are to be displayed.

Many properties of operations can be expressed as simple formulae. The claim

$op \,;\, op \Rightarrow op$

(in which the semicolon is Z's sequential composition) says that the operation is idempotent; doing it a second time has no effect. The claim

$op \,;\, u \;\Rightarrow (x' = x \wedge y' = y)$

says that the operation u is an undo for *op*. Simple kinds of refinement can be checked also;

$op1 \;\Rightarrow op2$

says that every execution of $op1$ is also an execution of $op2$. Properties of state invariants take a similar form. The claim

$inv \;\Rightarrow P$

for example, says that the property P of the state space follows from the invariant *inv*.

To generate transitions of an operation, we simply assert that it cannot happen:

not op

The counterexamples Nitpick generates will then be sample transitions. We can force Nitpick to generate transitions that satisfy stronger constraints (if they exist); if we want the prestates to satisfy a formula P we simply assert

not $(op \wedge P)$

If we want the post-states to satisfy P, we assert instead

$not\ (op \wedge P')$

To generate a sequence of transitions, we can assert, for example,

$op1\ ;\ op2\ ;\ op3$

whose counterexamples will assign values to 4 instances of each variable x and y (exposing the existentially quantified variables representing intermediate states). Sometimes we want to see how one version of an operation $op1$ differs from another version $op2$. The claim

$not\ (op1 \wedge not\ op2)$

will cause transitions of $op1$ but not $op2$ to be generated.

In summary, because of the nature of counterexample generation, we can use simple formulae with free variables for two very different purposes: to force executions that fit certain patterns, and to check properties of operations. The claim

$not\ (op1 \wedge not\ op2)$

is written in a form that suggests execution, but is formally identical to, and treated no differently from, the refinement claim

$op1\ \Rightarrow op2$

The use of a single formula as input to the Nitpick tool has a more subtle advantage also. The reduction mechanisms exploit the structure of the formula as a whole. In the formula

$not\ (op \wedge inv')$

which causes executions of op whose post-states satisfy inv, Nitpick might choose to generate the post-state variables first; if inv is strong, most values of the post-state will be inappropriate, and no search for matching pre-states will be required. Moreover, changing inv will generally lead to different searches even if op is unchanged, because isomorph elimination exploits the type structure of the entire formula; here a weaker invariant generally leads to bigger reductions [8].

4 *Derived Variables*

Most implicit specifications contain some explicit fragments. Consider, for example, an operation whose specification includes the formula

$e \in s\ \wedge\ t' = t \cup \{e\}$

The specification is certainly implicit; no construction for t', the new value of t, is given. Despite the non-determinism of the operation, however, once e has been chosen, the value of t' follows from the value of t. In generating assignments, then, there is no need to enumerate t' and check the equality; instead it can be *derived* from the other variables.

Nitpick performs this static analysis, partitioning variables into those to be enumerated and those to be derived. At one extreme – a fully implicit specification – all variables are enumerated; at the other extreme – a fully explicit specification – only the pre-state variables need be enumerated, and the rest are derived. Presented with a specification that happens to be executable, Nitpick will thus execute it as efficiently as a simulation tool would.

Interestingly, the analysis does not necessarily pick post-state variables to be derived. Suppose we want to sample operation executions whose post-states satisfy an invariant:

$not\ (op \wedge inv')$

A frame condition such as

$x' = x$

will then be executed backwards, with values of x' selected to satisfy the invariant, and values of x subsequently derived. Another place where derived variables are frequently identified is in state invariants. It is often convenient to add redundant components to the state; these are usually related explicitly to the basic state components. Having declared the calls of a phone system as a rela-

tion on phones, for example, we might define the active phones to be those engaged in calls:

$calls : phone \leftrightarrow phone$
$active : set\ phone$
$active = dom\ calls \cup ran\ calls$

Adding redundant state components contributes greatly to the clarity of the specification, but, thanks to the derived variable analysis, adds no run-time cost.

The static analysis is straightforward. The formula being checked is negated and put in conjunctive normal form. Any elementary formula that appears alone, unnegated, in a conjunct must be true of every counterexample. Of these, we extract the equalities which have a variable standing alone on one or both sides of the equals sign.

We then construct a graph, with each node representing a variable and each edge an equality and its direction of "execution". The equality

$E1 : y = f(p, q)$

for example, would contribute edges from p and q to y, all labelled $E1$, meaning that y can be derived from p and q by executing the equality E in the normal direction (like an assignment, from right to left). The equality

$E2 : y = x$

would contribute two edges, one from x to y labelled $E2$, and one from y to x labelled $E2'$, the prime indicating that the equality is ordered from left to right.

Any cycles in the graph are eliminated by dropping the edges associated with one or more oriented equalities. Remaining nodes that have incoming edges represent variables that can be derived; the nodes without incoming edges must be enumerated. From the edges of the graph, we read off the equalities that are executed (and their directions). In choosing which equalities to drop when breaking cycles, the number of derived variables is maximized. Also, given the choice of identifying one of two variables as derived, Nitpick selects the one with the larger set of possible values.

Suppose, for example, we have the formula

$(e \in s \ \wedge \ t' = t \cup \{e\} \ \wedge \ r' = t') \Rightarrow (r' = t)$

Negating and putting into conjunctive normal form, we get

$(e \in s) \ \wedge \ (t' = t \cup \{e\}) \ \wedge \ (r' = t') \wedge \ (\neg\, r' = t)$

The relevant equalities are then

$E1 : t' = t \cup \{e\}$
$E2 : r' = t'$

which give a graph with three edges:

$E1 : t \mapsto t'$
$E2 : t' \mapsto r'$
$E2' : r' \mapsto t'$

Dropping either $E2$ or $E2'$ eliminates the cycle, but dropping $E2'$ maximizes the number of derived variables. From the resulting graph

$E1 : t \mapsto t'$
$E2 : t' \mapsto r'$

we determine that t' and r' are derived, and t is enumerated.

5 A Nitpick Session

Let's take a trip back to the 1980s and imagine specifying the directory structure of a file system as it appears to the user for a new icon-based operating system: the Macintosh finder.

We start by declaring a given type that models the objects in the system:

```
OBJ == {drive, trash, ...}
```

Two elements of the type are named explicitly; the ellipsis indicates that there may be other elements also. The basic state space is defined:

```
simpleFinder = [
 parent: OBJ -> OBJ
 files, folders: kinds part OBJ
 trashed, stored: set OBJ
 |
 {drive, trash} <= folders
 ran parent <= folders
 trashed = parent~+.{trash}
 stored = parent~+.{drive}
 not drive in trashed]
```

The syntax is essentially an ascii version of Z's horizontal schema. The partial function *parent* maps objects to the objects in which they are contained; *kinds* is a partition of the set of objects into two sets, *files* and *folders* – for simplicity we are ignoring the creation and destruction of objects; *trashed* is the set of objects in the trash can, and *stored* is the set of objects in the drive. The objects that are not in the domain of *parent* (that is, not contained in any other object) are interpreted as being on the desktop.

The constraints following the declarations are straightforward: that the drive and trash are folders (<= means subset); that all objects that contain other objects are folders too; that *trashed* is the relational image (denoted with the dot) of the closure (+) of the transpose (~) of *parent*; and so on.

To see some instances of the state, we select a scope of 3 objects and run Nitpick with the claim

```
not simpleFinder
```

It immediately displays

```
parent = {}
folders = {drive, trash, o2}
files = {}
stored = {}
trashed = {}
```

Not a very interesting case: the drive, the trash, and some other object *o*2, all on the desktop. So we ask Nitpick to provide 2 more states. The first simply makes *o*2 a file; the second has

```
parent = {drive -> drive}
```

This exposes an error; we did not intend the drive to be contained in any other folder. So we change the first constraint of the state schema to

```
{drive, trash} <= folders \ dom parent
```

hoping to eliminate cycles, since *parent* is a function. To check that we have succeeded, we define

```
acyclic = [simpleFinder | parent+ & Id = {}]
```

(& is intersection), and assert

```
simpleFinder => acyclic
```

Nitpick responds with

```
parent = {o2 -> o2}
folders = {drive, trash, o2}
files = {}
stored = {}
trashed = {}
```

So we were wrong; we forgot the possibility of a desktop object. Is this the only circumstance in which a cycle can arise? We assert

```
simpleFinder and (stored U trashed = files U folders) => acyclic
```

and Nitpick finds no counterexamples. To gain more confidence, we increase the scope to 4 objects, then 5, then 6, but no counterexamples are found. Until now, Nitpick's response time has been dominated by the time to display counterexamples. For 5 objects, however, there are 248,832 cases to consider. Nitpick reduces the space to 11,469 cases and exhausts it in just under 5 seconds (all timings on a Macintosh Duo 280c). 6 objects entail about 8 million cases; Nitpick exhausts these in 58,040 cases and 30 seconds. This seems a reasonable point to stop; intuitively, it is unlikely that a cycle can only be formed with 7 or more objects. Relieved that Dijkstra is not around to press us on the point, we continue.

Enriching the state to include aliases, we define

```
finder = [
        simpleFinder
        acyclic
        links : OBJ -> OBJ
        aliases: set OBJ
|
        aliases <= files
        aliases = dom links]
```

Aliases are treated as files, since they can never contain other objects. The *links* function maps an alias to the object it points to; note that this object may itself be an alias.

We now define the fundamental operation for changing the directory structure:

```
move = [
        finder finder'
        obj, to, fto: OBJ
|
        not to in parent~*.{obj}
        (not to in aliases) => fto = to
        (to in aliases) => {fto} = links+.{to}\ aliases
        parent' = parent (+) {obj -> fto}
        links' = links and kinds' = kinds]
```

The user drags *obj* to *to*, but its final destination, *fto*, may be different if *to* is an alias. The constraints below the declarations are as expected: that you can't move an object to itself or one of its descendants; that *fto* is the same as *to* if *to* is not an alias; that otherwise, of the objects reachable from to by repeated application of *links*, *fto* is the one that is not an alias; that the new value of *parent* is obtained from the old one by overriding the old association of *obj* with a mapping to *fto*; and that aliasing and the division into files and folders is unchanged. To sample an execution, we assert

```
not move
```

and Nitpick displays

```
parent = {}
obj = o2
to = drive
fto = drive
links = {}
folders = {drive, trash, o2, o3}
...
parent' = {o2 -> drive}
...
```

Wanting to see a more interesting case, we prod Nitpick to move an object into a folder in the trash

```
not (move and to in trashed)
```

and it dutifully generates

```
parent = {o2 -> trash}
obj = o3
```

```
to = o2
fto = o2
links = {}
folders = {drive, trash, o2, o3}
...
parent' = {o2 -> trash, o3 -> o2}
...
```

Wondering whether every object moved to the trash ends up trashed, we assert:

```
move and to in trashed => obj in trashed'
```

After examining 1705 states in a couple of seconds, Nitpick finds a counterexample:

```
parent = {o2 -> trash}
obj = o3
to = o2
fto = drive
links = {o2 -> drive}
folders = {drive, trash, o2, o3}
...
parent' = {o2 -> trash, o3 -> o2}
...
```

Our specification says that an object moved to an alias in the trash will end up back in the drive if that is where the alias points. The property we asserted is quite desirable; not surprisingly, in this scenario the Macintosh displays the message "The alias foo could not be used because it is in the trash".

The designers of the finder did, however, allow the reverse scenario. In response to

```
not (move and to in stored and obj in trashed')
```

Nitpick generates

```
parent = {o2 -> drive}
obj = o3
to = o2
fto = drive
links = {o2 -> trash}
folders = {drive, trash, o2, o3}
...
parent' = {o3 -> trash, o2 -> trash}
```

suggesting a nasty trick: make an alias for the trash and insert it somewhere in the drive. (The alias, incidentally, will share the trash icon but only the actual trash object will bulge when filled.) By changing the alias name and its icon, it will appear to be a regular folder, but any object moved to it will end up in the trash.

6 *Further Work*

It has been assumed that implicit specifications are good for doing proofs, and explicit specifications are good for simulations: that executability, with its appeal of immediate feedback, comes at the expense of eliminating the most powerful specification constructs.

Our experience with Nitpick suggests that this particular tradeoff may not be necessary: an implicit specification can indeed be executed. There are other tradeoffs, of course. The expressiveness of the Nitpick language is limited in some respects, primarily in the lack of higher-order relations. Although unable to execute implicit constructs or search for counterexamples, the IFAD VDM tool [10], for example, can simulate operations over more elaborate data types. We think of Nitpick as complementary to existing tools; it might be used to perform a deeper analysis of an abstraction of a full specification.

Nitpick is not the first tool to blur the distinction between sampling executions and checking

properties. Aslantest [11] supports both concrete and symbolic execution of operations. Unlike Nitpick, however, it does not check properties by exhaustive search within a finite scope, but instead uses theorem-proving to validate the results of a symbolic execution.

We are planning a number of improvements to Nitpick. The most pressing is the addition of existential quantification. With only (implicit) universal quantification, Nitpick can only detect errors of underconstraint. Although the omission of an execution may alert the specifier to overconstraint, there is no systematic way to catch it.

Overconstraint is doubly insidious because it masks errors of underconstraint. This problem is familiar from model checking. An assertion that a machine never reaches a bad state will be vacuously true if the machine is deadlocked. It therefore makes sense always to accompany a safety check with a liveness check; in a language such as CTL, to assert *EFp* as well as *AGp* [12].

Existential quantification will allow such claims. In addition to regular precondition checks (that an operation has at least one transition for each prestate in some specified set), we plan to implement an 'exists exactly one' quantifier to support determinism claims (that an operation has at most one transition for the given states).

The derived variable analysis is effective but crude. In several situations it fails where it might reasonably be expected not to. In the *move* operation above, for example, the final destination *fto* of the object is defined by

```
(not to in aliases) => fto = to
(to in aliases) => {fto} = links+.{to}\ aliases
```

The variable *fto* is not identified as derived because it appears in two different equalities, and because, in the second, it appears on the left not as a variable but as the singleton set. Had we declared *fto* to be a set and defined it by

```
fto = links*.{to} \ aliases
singleton (fto)
```

instead – which has exactly the same meaning – Nitpick would have successfully identified *fto* as derived.

References

1. I. J. Hayes and C. B. Jones. Specifications are not (necessarily) executable. *Software Engineering Journal*, vol. 4, no. 6, pp. 330–338, 1989.
2. N. E. Fuchs. Specifications are (preferably) executable. *Software Engineering Journal*, vol. 7, no. 5, pp. 323–334, 1992.
3. P. Zave. An Operational Approach to Requirements Specification for Embedded Systems. *IEEE Transactions on Software Engineering*, vol. SE-8, no. 3, pp. 250–269, 1982.
4. P. Zave. An Insider's Evaluation of PAISLey. *IEEE Transactions on Software Engineering*, vol. 17, no. 3, pp. 212–225, 1991.
5. M. Jackson. Constructive Methods of Program Design. *Proc. First Conference of the European Cooperation in Informatics*, Amsterdam, 1976.
6. P. G. Larsen and P. B. Lassen. An executable subset of Meta-IV with loose specification. *Proc. VDM'91: Formal Software Development Methods*, 1991.
7. S. H. Valentine. Z--, an executable subset of Z. *Proc. Z User Workshop*, York, England, 1991.
8. D. Jackson and S. Jha. Faster Checking of Software Specifications By Eliminating Isomorphs. *Proc. Principles of Programming Languages (to appear)*, St. Petersburg Beach, FL, 1996.
9. D. Jackson and C. A. Damon. Elements of Style: Analyzing a Software Design Feature with a Counterexample Detector. *Proc. International Symposium on Software Testing and Analysis (to appear)*, San Diego, CA, 1996.
10. R. Elmstrom, P. G. Larsen, and P. B. Lassen. The IFAD VDM-SL toolbox: a practical approach to formal specifications. *ACM SIGPLAN Notices*, vol. 29, no. 9, 1994.

11. J. Douglas and R. Kemmerer. Aslantest: a symbolic execution tool for testing Aslan formal specifications. *Proc. International Symposium on Software Testing and Analysis*, Seattle, WA, 1994.

12. E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, vol. 8, no. 2, pp. 244–263, 1986.