Efficient Search as a Means of Executing Specifications

Craig A. Damon*
Daniel Jackson

ABSTRACT The utility of directly executing formal specifications is briefly touched upon and the concept of exhaustive search as a means of execution is introduced. A mechanism for improving the efficiency of such searches is presented in some detail. Finally, the results of an implementation of the mechanism are presented.

KEYWORDS model generation, constraint satisfaction, exhaustive testing, formal specifications, executable specifications.

1 Introduction

One common prescription for curing the ills of software development involves the development of a formal specification in a language such as Z [Spi89]. However, beyond the clarity brought about by the very act of writing a formal specification, it is unclear what additional benefits the development of a specification actually brings.

One possible benefit arises if one is able to execute the specification. Such an execution can be used to validate and help debug a specification. Unfortunately, it is not obvious how to directly execute a Z specification. One approach [Val91] involves restricting Z to a constructive (and therefore executable) subset.

Hayes and Jones [HJ89] argue that forcing any form of determinism into the specification is a disservice, in that it may lead to restrictions in the programs that meet the specification. They provide numerous examples where this type of constructive style leads to over-specification.

Fuchs [Fu92], in response, argues that any finite specification can be executed,

* Authors' address: School of Computer Science, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213. WWW: http://www.cs.cmu.edu/~{cdamon,dnj}. Email:{craig.arthur.damon, daniel.jackson}@cs.cmu.edu. Fax: (412) 268-5576.

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.

albeit inefficiently, by performing a search over the possible results. Ignoring quantification for the moment, it is straightforward and efficient to test whether a particular case is a legal execution of a specification of some operation. Thus if a system could enumerate all possible cases, checking each case against the specification, it could find all possible executions. Such a search can be easily extended to support both universal and existential quantification.

We have built just such a system, called Nitpick[1] [JD96]. The goal of Nitpick is to allow interactive exploration of formal specifications by generating instances of states satisfying invariants, executions of operations and counterexamples of claims about a specification. As all of these scenarios amount to finding models of a formula, this paper will focus only the ability to discover valid executions.

To force a finite search space, we require the user to artificially restrict the scope of the search by limiting the number of elements of each type under consideration. Because of this limitation, Nitpick is not a verifier, since it can never search what is in general an infinite state space. Rather it is more like a debugger or tester, allowing the user to improve the specification, without ever certifying perfection.

Even for a small specification and small scope, the search can easily require checking 10^{20} total possible cases and we have seen examples (still simple) where the total space has approached 10^{40} total possible cases. The naive approach to fully searching these entire case spaces is clearly untenable.

Our approach in Nitpick is to employ a collection of sound case space reduction techniques. We currently have implemented three such techniques: derived variable detection [JD95], isomorph elimination [JJD96] and enumeration tree pruning. With derived variable detection, we detect relationships that must hold within the specification and can be determined constructively. This allows us to remove certain variables from the enumeration space, constructing their values from the other variables which are enumerated. Isomorph elimination removes cases which are only relabelings of each other. The remainder of this paper focuses on the concept, implementation and effectiveness of enumeration tree pruning, including its interaction with isomorph elimination.

2 Overview

Nitpick evaluates specifications and claims written in a simplified form of the Z specification language called NP, that retains the essential non-constructive nature of Z [Ja96]. In particular, formulas are built from simple relation and set operators, composed with **and**, **or**, **not**, $\Rightarrow$ (logical implication) and $\Leftrightarrow$ (logical equivalence). Formulas themselves are grouped into schemas, which are named collec-

[1] The Nitpick specification analyzer is freely available (in binary form for the Macintosh) from the authors or from http://www.cs.cmu.edu/~nitpick.

tions of formulas with associated variable declarations. Schemas are used to describe legal states of data structures as well as appropriate transitions of those data structures. Claims about the states are written in essentially the same formula language, allowing the user to make assertions about the specification.

Although the Nitpick tool is built around the Z specification language, this reduction technique is not limited just to Z specifications. Rather, it could be applied more broadly to find satisfying assignments for the variables in many logical expressions. All we rely on is the standard logical connectives and the ability to efficiently evaluate the terms for a given assignment.

An assignment is a mapping from each free variable found in the specification to a value. Typically, in the execution of an operation, some of these variables represent the state prior to execution while others represent the final state after the execution of the operation. An assignment which satisfies all of the restrictions of the specification is called a satisfying assignment. We use the term case to describe a single assignment, satisfying or not. When looking for sample executions, a satisfying assignment is an interesting case.

To find these assignments, Nitpick identifies the free variables involved in the specification, enumerates all of the possible values for each of those variables (based on the scope restrictions provided by the user), and constructs an enumeration tree, with each level corresponding to a single free variable. The specification is then evaluated against each leaf and the satisfying assignments are reported back to the user.

As a simple example, consider the formula

$$a = \{\} \text{ or } a = b \text{ or } b = c$$

with a, b and c each declared elsewhere to be sets of type T. For this example there are three free variables, a, b and c. Assuming that T is defined to consist only of the values t_1 and t_2, a possible enumeration tree can be seen in Figure 1.

Given this simple formula and this enumeration tree, the enumeration tree can be pruned in an obvious way. If the first portion of the formula ($a = \{\}$) is tested first for each value of a, it is clear that when a is empty the value of the entire formula is true and does not depend on which values of b or c are chosen. Because of this, the entire leftmost sub-tree below a can be pruned. Similarly, if a is equal to b, the value of the formula is independent of the value chosen for c and the appropriate sub-trees of b can be pruned. The reduced tree is shown in Figure 2.

Note that this reduction does not occur statically, that is merely by examining the specification. On the contrary, at each node of the enumeration tree, starting from the root, the specification is partially evaluated given the variables bound thus far. If this partial assignment gives a definite value to the formula as a whole, the tree traversal can be safely terminated and the tree can be pruned at that node. Otherwise, each child of that node must be evaluated for possible pruning.

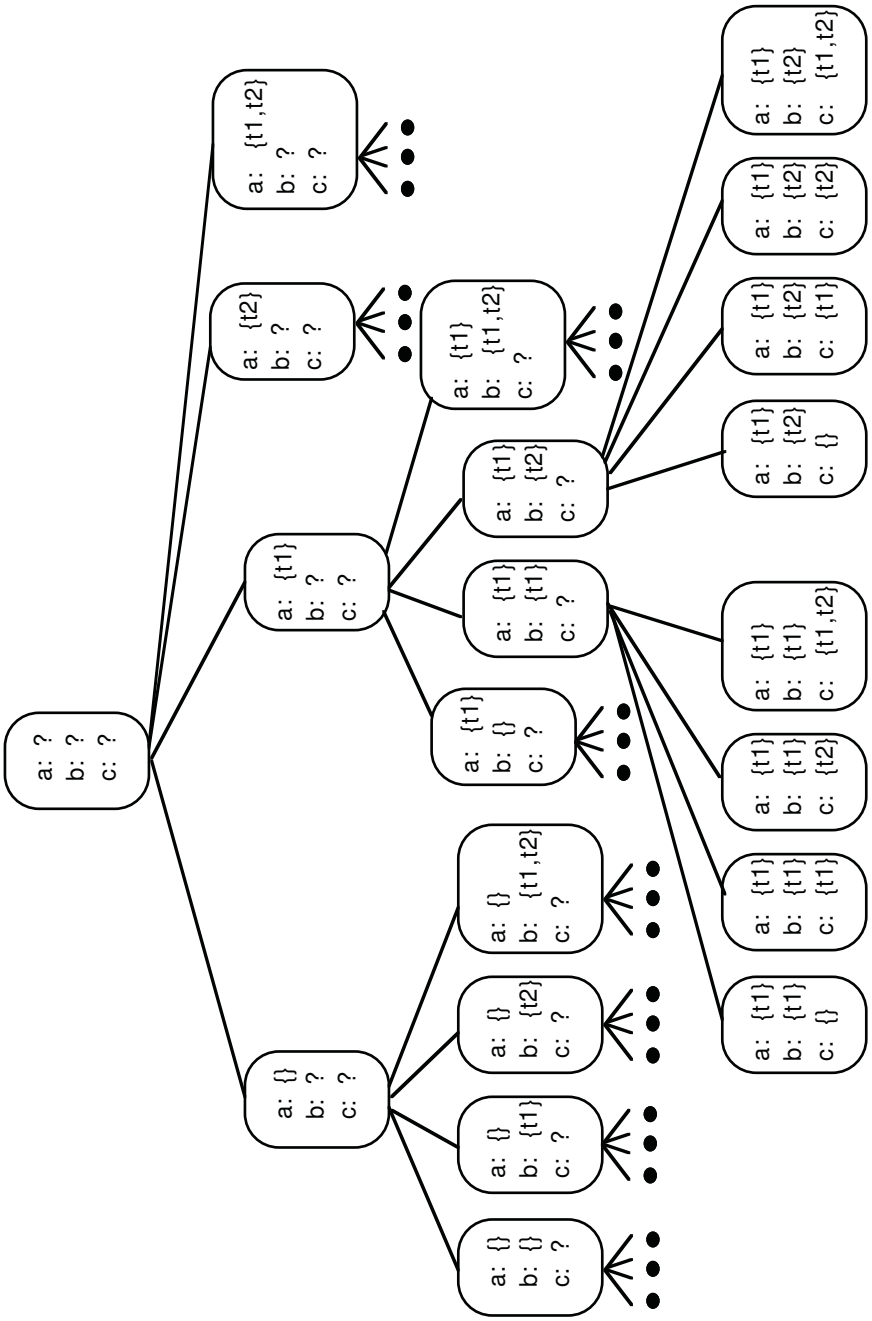

Figure 1: Full Enumeration Tree

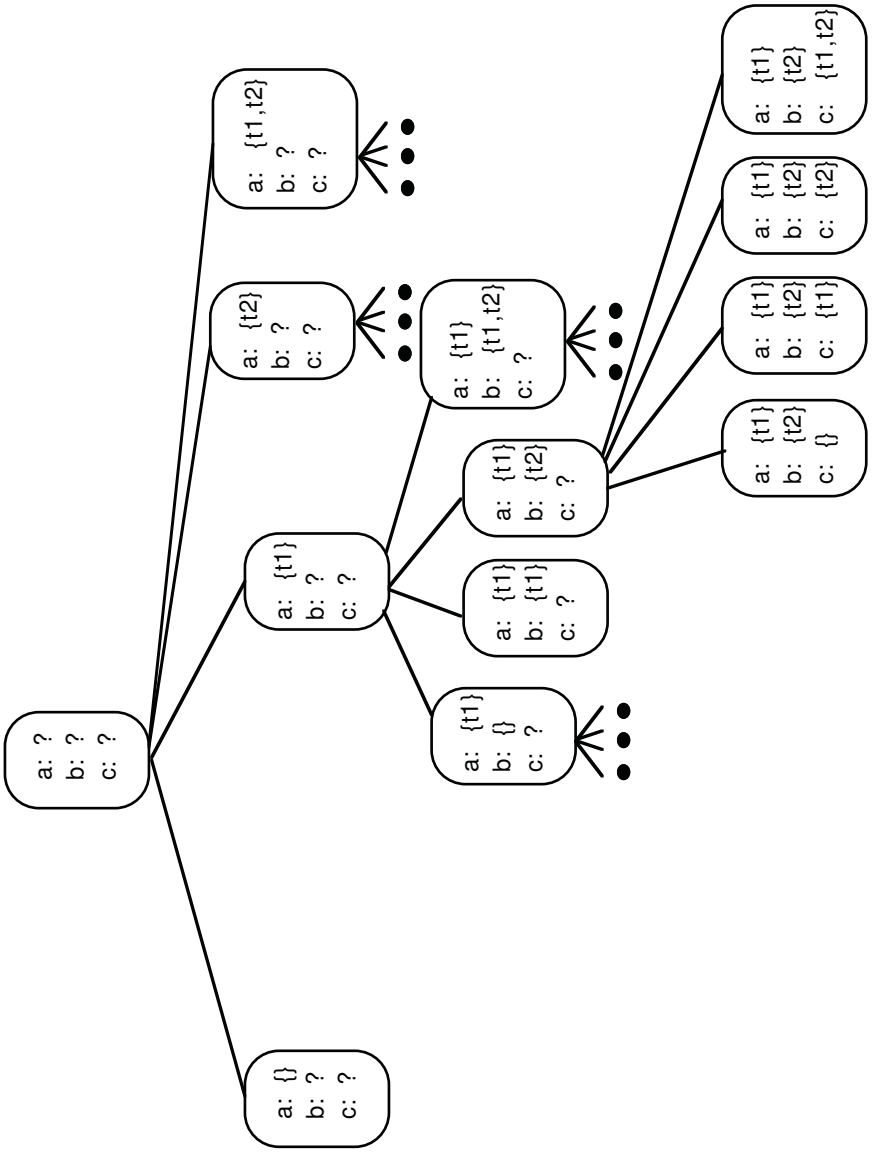

Figure 2: Pruned Enumeration Tree

Because of this, the pruning is done on the fly while the search is being performed.

Something could easily have gone wrong with this short circuiting of the tree. If *c* had been defined as the first level of the tree, only a small amount of pruning would have been possible.

The goal of this reduction is to automatically remove as many leaves from the enumeration tree as possible. This should occur regardless of the order and structure of the specification that the user provides to Nitpick.

To maximize the pruning, the implementation must the optimal ordering of the variables in the enumeration tree. This is non-trivial to compute, so we approximate this, seeking at least a "good" ordering. This is in addition to the work required to actually prune the enumeration tree.

To understand this analysis, it is helpful to understand more about how Nitpick actually executes specifications. Logically, there are three steps Nitpick performs to evaluate a specification. First, the formula is analyzed and compiled into a simple function which takes a full assignment of the variables as its input and returns a boolean indicating whether this assignment satisfies the specification. Next, the full set of potential values for each variable is computed[2] and these values are used to define the enumeration tree. Finally all combinations of these values are tested by checking partial and full assignments for each node in the enumeration tree.

The enumeration tree, of course, need never be held in memory in its entirety. Using a depth first search, only an indicator of the current value for each variable need be maintained. Because of this, the memory utilization of a Nitpick execution is minimal.

3 Finding Determinants

Discovering the variable ordering begins with discovering the determinants of the formula. Informally, a determinant is a sub-formula which can, by itself, determine the value of the entire formula. Some determinants only determine the value of the formula if they are true and others only when they are false. We call these true and false value determinants respectively.

To give a more formal definition of determinants, it is necessary to give a more formal definition of a formula. Although the following grammar is obviously simplified, it is sufficient for purposes here, since this reduction is independent of the term structure of the language. This simplified grammar is

Formula = Formula **and** Formula | Formula **or** Formula |
not Formula | **predicate**(x_1, ... , x_n)

[2] To further reduce our memory requirements, we do not fully enumerate the set of values for most variables. Instead we initially compute the number of possible values and provide simple calculations which can compute each value quickly from a simple ordinal.

where **predicate** represents a function which takes *n* variables as its arguments and returns a boolean.

The sets of true and false value determinants for a formula F can then be defined as the smallest sets of sub-formulas of F satisfying:

F∈ trueDeterms **and** F ∈ falseDeterms

if f ∈ trueDeterms **and** f = g **or** h

then g∈ trueDeterms **and** h ∈ trueDeterms

if f ∈ falseDeterms **and** f = g **and** h

then g∈ falseDeterms **and** h ∈ falseDeterms

if f ∈ trueDeterms **and** f = **not** g

then g∈ falseDeterms

if f ∈ falseDeterms **and** f = **not** g

then g∈ trueDeterms

This formalism can easily be extended to include the entire NP language (as well as many others). The logical implication operator (⇒) can be converted into **not** and **or** in the standard manner, f ⇒ g = **not** f **or** g. All of the standard relational and set tests can be modelled as predicates, with specialized predicates to handle more complicated expressions.

For our example above, a = {} **or** (a = b **or** b = c), the sets of determinants are

trueDeterms = { a = {}, a = b, b = c }

falseDeterms = { a = {} **or** (a = b **or** b = c) }

Note that it is possible for a formula to be both a true value and a false value determinant. Although not true in the example above, it is even possible for a formula to be a member of both miminal determinant sets. When this occurs, all other portions of the formula are extraneous since they do not effect the ultimate evaluation of the formula. As will be seen later, however, these other determinants may be useful to reduce the number of cases needed to be considered.

4 Variable and Formula Ordering

To find a good ordering of the variables, two special functions are used: one to estimate the chance of a sub-formula being known to be true given values for a subset of the variables and a similar one to estimate the chance of it being known to be false. These functions, called *known value estimators*, estimate what percentage of all assignments will make a given formula true or false, given knowledge of only a subset of variables.

```
selectedVariables := empty
variableOrdering := empty
remainingVariables = all free variables
while not remainingVariables = {}
    bestChance := 0
    foreach variable v in remainingVariables
        chance := 0
        testSet := selectedVariables ∪ {v}
        foreach true determinant d
            chance := 1.0 - ((1.0  - chance) *
                    (1.0 - trueValueEstimator(d,testSet)))
        end foreach
        foreach false determinant d
            chance := 1.0 - ((1.0  - chance) *
                    (1.0 - falseValueEstimator(d,testSet)))
        end foreach
        if chance > bestChance
            bestVariable := v
            bestChance := chance
        end if
    end foreach
    if  bestChance = 0
            /* no best selected yet, so choose an element */
        bestVariable := choose (remainingVariables)
    end if
    remainingVariables := remainingVariables \ {bestVariable}
    selectedVariables := selectedVariables ∪ {bestVariable}
    variableOrdering := append(variableOrdering,bestVariable)
end while
```

Figure 3. Simple Variable Ordering Algorithm

For example, the true known value estimator may estimate that the formula $a = \{\}$ will be true 12.5% of the time if the set of known variables contains a. If the set of known variables was $\{b, c\}$ however, it would estimate a 0% chance of knowing that the formula was true.

With these two functions, Nitpick performs a simple search to discover a good ordering of the variables. The algorithm for this search is given in Figure 3. The search is begun by using these functions to check each of the determinants against each single variable. The single variable with the aggregate best chance to induce tree pruning is thus identified. This process is then repeated, checking each remaining variable, in combination with the variables already selected, for their chance of pruning each determinant.

If at any point, no single variable provides any additional opportunities for prun-

ing the enumeration tree, the full algorithm also checks all pairs of variables. If any pair gives opportunities for pruning while neither individual variable offers any, either variable can be chosen without preference. For performance reasons, we do not currently consider triples or larger groups of variables.

If no best variable is found, we apply some heuristics to determine the best candidate, implemented by our *choose* function from Figure 3. Our best heuristic to date is to take the variable with the smallest set of possible values, as this will create the cheapest tree if pruning does occur. Finally, we just arbitrarily choose one.

Currently, we use a very rough approximation for the known value estimators. The first check, of course, verifies that enough variables are defined to have any knowledge of the value of the formula. Where sufficient variables are defined, we use some very rough hardwired approximations, such as estimating boolean equality as 1/2 true and 1/2 false, while set equality is estimated at 1/8 true and 7/8 false[3].

Coming back to our original example, $a = \{\}$ **or** $a = b$ **or** $b = c$, the ordering process can be easily demonstrated. During the first iteration, it is determined that a has a 1/8 chance of being true for the determinant $a = \{\}$, while no other single variable could give a value to any other determinant. Therefore, a will be the first variable enumerated. During the second iteration, it is seen that the pair of variables $\{a, b\}$ has an approximate 1/8 chance of making $a = b$ true (in addition to the $a = \{\}$ chance), whereas the pair $\{a, c\}$ could not determine any new determinants. The variable ordering is now determined to be $<a, b, ?>$. Since there is only a single variable remaining, no additional iterations are required and the final ordering is $<a, b, c>$, which is in fact the optimal variable ordering for this simple example.

Given a variable ordering, the appropriate ordering of determinants can now be defined. To test a single case, the determinants will be checked in this order, halting when the value of the formula is first known (i.e. when a determinant has been evaluated to its determining value).To maximize pruning, we need to evaluate those determinants which are well defined higher in the enumeration tree before those that are only well defined lower in the tree.

To implement this ordering, each determinant, whether true or false value, is tagged with the level of the lowest variable in the enumeration tree which is required by the determinant. They are then sorted, highest level first, based on these tags so that they may be evaluated in that order.

3 The values for structured types, such as sets and relations, are based on an assumed cardinality of three, which is the default for the scope settings. It remains to be seen whether refining these values by using the actual scope restrictions will improve the orderings.

```
foreach variable v
    resetValue (v)

variableLevel := 1
done := false
repeat
    while variableLevel <= numVariables
        /* Check Determinants */
        foreach determinant d where tag (d) =  variableLevel
            if eval (d,case) = determingValue (d)
                goto valueDetermined
        /* Nothing determined yet */
        variableLevel := variableLevel + 1
        continue while loop

    valueDetermined:
        displayCase (case,formula)
        repeat
            v := variables[variableLevel]
            if moreValues (v)
                nextValue (v)
                exit repeat
            else
                resetValue (v)
                variableLevel := variableLevel - 1
        until variableLevel = 0
        if variableLevel = 0
            done := true
    end while

    /* Got to a leaf - Start back up */
    variableLevel := variableLevel + 1
    repeat
        v := variables[variableLevel]
        if moreValues (v)
            nextValue (v)
            exit repeat
        else
            resetValue (v)
            variableLevel := variableLevel - 1
    until variableLevel = 0
    if variableLevel = 0
        done := true
until done
```

Figure 4. Partial Depth Pruning Search Algorithm

```
foreach variable v
    resetValue (v)

variableLevel := numVariables
done := false
repeat
    /* Check Determinants */
    foreach determinant d in order
        if eval (determinant,case) = determiningValue (d)
            variableLevel := tag (d)
            exit foreach

    displayCase (case,formula)

    /* Find next unpruned case */
    repeat
        v := variables[variableLevel]
        if moreValues (v)
            nextValue (v)
            exit repeat
        else
            resetValue (v)
            variableLevel := variableLevel - 1
    until variableLevel = 0
    if variableLevel = 0
        done := true
until done
```

Figure 5. Full Depth Pruning Search Algorithm

5 Pruning The Tree

To prune the tree, we want to find the highest nodes in the enumeration tree for which the value of the formula is known. This can be easily determined using a pre-order walk of the tree. At each node of the enumeration tree, a subset of the determinants is evaluated. Only the determinants that have been associated with this level of the tree need to be evaluated, i.e. those determinants which use the variable corresponding to this level, but do not reference any lower variables. Having already sorted the determinants, this set is trivially recognized.

If any of these determinants do in fact determine the value of the formula, the enumeration tree can be pruned here, ignoring all descendent nodes. If the formula is not resolved at this node, each descendant node in turn must be similarly checked.

As noted earlier, the full enumeration tree is never, of course, maintained in memory. Instead, only a means for determining the next value for each variable is nec-

essary. Pruning the tree at a given level simply involves choosing the next value for the variable at the appropriate level, and resetting (starting back at the first value) the variable at each subsequent level in the "tree". The algorithm for searching the tree is given in Figure 4.

Partly for implementation reasons, Nitpick actually does a full enumeration down to a leaf before evaluatuing any of the determinants. This approach has the side effect of always providing a full assignment of variables, even if their values are not significant to the example discovered.

To make the evaluation of a single case as efficient as possible, Nitpick compiles the formula (currently into a simple interpretable stream). The code is generated in the order that was determined above. The state of each logical connective is modelled by position in the code, with a single branch in the code mapping to each logical connective (**and**, **or**, etc.). To determine where pruning can occur, the generated code actually tracks which variables have been used.

This variable tracking is done by tagging each branch in the generated code with the level of the lowest variable used in determining this branch. The modified algorithm for doing full depth searches with pruning is given in Figure 5.

6 Measured Reductions

This approach to reduce the state explosion has been implemented as a part of Nitpick. Nitpick consists of about 40,000 lines of C code and runs on Macintosh computers. Although we have not found instances where it is advantageous to turn off any of the space reductions techniques, we do allow them to be disabled for research purposes.

To illustrate the effectiveness of this approach, we measured the search space with and without the tree pruning reduction for three different small to moderate example specifications, each run with a smaller and larger scope setting. The Style example is a simple abstraction of a word processing paragraph style mechanism; the Phone example is a toy specification modelling call forwarding in a phone system; and the Desktop example models the move operation for the Macintosh desktop in the face of aliases. These examples range in size from fifty to a hundred lines of NP specification. For all of these runs, the total memory utilized by Nitpick was less than a megabyte all of which was allocated statically at compile time.

The results of these measurements are shown in Table 1. In this table, the *# Vars* column gives the number of free variables enumerated in the example. *Space Size* gives the total state size that would have to be searched without pruning. *Search Size* gives the number of states that remained after pruning. *Time* give the time in seconds (s), minutes (m) or hours (h) that the pruned search required when run on a Power Macintosh 7500. *Improv* is the (linear) improvement factor gained from the pruning. *Log Ratio* is the ratio of the logs of the *Search Size* and *Space Size*

Example	*Scope*	*# Vars*	*Space Size*	*Search Size*	*Time*	*Improv*	*Log Ratio*
Style	Small	8	9.8E8	4.6 E3	0.9 s	105	0.41
Style	Large	8	1.7E14	6.5 E6	12m	107	0.46
Phone	Small	3	2.9E5	5.2 E4	31s	6	0.87
Phone	Large	3	6.5E8	5.1E7	4h	10	0.94
Desktop	Small	6	7.9E6	1.1E3	1.7s	104	0.49
Desktop	Large	6	1.2E13	5.3E6	36m	106	0.51

Table 1. Results Of Pruning For Various Examples.

columns. All of these examples make use of derived variable detection, but do not utilize isomorph elimination. Since both the *Space Size* and*Search Size* columns include the derived variable detection reduction, the *Improv* and *Log Ratio* columns reflect only the gain from the tree pruning.

As the table shows, the gains from this tree pruning can be huge, with a 10^7 linear reduction in the large style example. The most interesting column, however, is the rightmost one, *Log Ratio*. This indicates the exponential reduction gained by the pruning.

One goal with Nitpick is to develop a number of reductions which each approach a log ratio of 0.5. As can be seen from the table, the pruning for all but the Phone example is near or beyond this goal. Such factors can drastically slow the exponential growth of state spaces. Although the growth of course remains exponential, reducing the exponent by a relatively constant small fraction flattens and shifts the growth curve to the right, yielding manageable search sizes for small to moderate specifications. Derived variable detection also typically gives a ratio around 0.5 (although this varies substantially depending on the style and structure of the specification). Like pruning, isomorph elimination, will meet this goal on many specifications, while occasionally dropping to the 0.9 range.

In general, and as could be expected, the gains from pruning seem to be most significant with more variables and smaller individual scopes. The positive correlation with the number of variables bodes well for the ability to scale into larger and more complex specifications, since larger specifications will typically have a greater number of variables.

This also blends nicely with the isomorph elimination work, which yields the largest reductions with fewer variables and larger scopes. Because of this combination, the product of the log ratios of these two mechanisms is nearly always better than 0.5, even if one or the other is significantly over that target.

7 Future Work

Based on earlier, more approximate orderings, it is clear to us that the gain is

heavily dependent on choosing a good ordering. As was noted earlier, our current estimation functions for truth and false values are quite rough. Over time we may need to investigate more exacting measures, although these are likely to be more computationally expensive than the current tests.

Another future research area is the possibility of adding additional determinants which can be computed more cheaply, in terms of the number of variables used, than the existing ones. For example, consider the following formula, where *a* and *b* are sets of type T and e is an element of T:

$$a \setminus \{e\} = b$$

(The \ operator is set difference.) The smaller formula

$$e \notin b$$

can be derived from this formula. This additional formula can be conjoined to the first without changing its meaning, thus is seemingly redundant. This change does however allow enumeration tree pruning to occur before evaluating *a* (when *e* is in *b*).

One final possibility is to record some information about the nature of the pruning at the point that it is discovered and then use this information to quickly find the next appropriate value. For example, consider a formula where the sub-formula $a \in b$ is a determinant and the enumeration tree has *a* above *b*. If $a \in b$ caused tree pruning with *a* having the value t_2 and *b* having the value {}, the tree walker could go directly to the value $\{t_2\}$ for *b,* skipping over $\{t_1\}$. If such value selections were sufficiently cheap, this significantly improve the search time.

8 Related Work

The first author's background with traditional compiler optimization [ASU86] was the basis for this work. However, there are fundamental differences between the expression re-ordering done by compilers and this work. Most notable among these is the goal itself. Compilers sort the executable trying to minimize the number of instructions required to compute an expression. Thus they sort the cheapest determinants to the front. And few programming languages give the compiler writer the freedom to re-order evaluation on the scale required here.

Related approaches have also been tried in search problems over the last two decades in the AI field [KK88], particularly game playing and optimization problems. These approaches tend to be much less rewarding in game playing since the problems tend to be much less absolute. In particular, no single sub-expression can generally definitively determine the outcome of the entire search. Rather a numeric weighting is typically established for each case investigated and the weightings compared. So related pruning strategies can only remove cases which are clearly inferior to ones already established, thus markedly reducing the effec-

tiveness of the pruning. In addition, searches for winning strategies in game playing, by their very nature, cannot re-order the evaluation tree, further limiting the effectiveness of this approach.

Recently, work has been done on variable ordering heuristics for speeding searches for solving constraint satisfaction problems[SF91]. While many of these heuristics are tailored to help with specific constraints, generally job shop scheduling, there are some general approaches which Nitpick may be able to exploit. Most interesting is the use of dynamic reordering of the search tree during the search. This is computationally very expensive and would only be useful if a significant further reduction in the search space was gained.

Slaney's FINDER program [Sla94] uses similar techniques for different goals. Its purpose is to find models of formulas in first-order logic with uninterpreted function symbols. Unlike Nitpick, FINDER constructs functions point-wise, and can thus exploit more fine-grained backtracking. By considering entire relation values, Nitpick, on the other hand, can combine backtracking search with isomorph elimination. FINDER depends on the user to provide variable orderings, and thus, although it may outperform this pruning algorithm when used by a skilled practioner, is less automatic.

9 Conclusion

The common belief has been that an "executable" specification language was necessary in order to be able to directly execute formal specifications. However, as Fuchs [Fu92] pointed out, most specifications, even non-constructive and non-deterministic ones, can be used as the test for an exhaustive search. This can then be viewed as a very inefficient form of executing the specification.

We have shown one mechanism which can be used to significantly reduce the search space required to execute a specification. While this mechanism alone does not allow efficient execution of specifications through searching, in combination with other such mechanisms it does allow a large body of specifications to be executed reasonably efficiently. This mechanism is particularly valuable here, since it becomes more effective as the specification becomes more complex (and would thus be thought to be more expensive to analyze).

With such executions, a more exploratory and interactive environment can be provided to the specification writer. Nitpick is a first step towards such an environment, allowing less sophisticated users a means to become more comfortable with formal specifications.

Further research is still needed to find more of these reductions, which in turn will enlarge the potential range of specifications which can be executed. As the suite of reductions becomes more powerful, the dream of executing arbitrary specifications may perhaps be realized without sacrificing the enormous advantages of non-constructive specifications.

Appendix: Interaction with Isomorph Elimination

There is an interaction between enumeration tree pruning and the isomorph elimination technique [JJD96] also employed by Nitpick. When using isomorph elimination, only isomorphically distinct values are used in the enumeration tree, with each value being "unlabeled". In the earlier enumeration example, four distinct values were enumerated for the sets *a* and *b*: {}, $\{t_1\}$, $\{t_2\}$, $\{t_1,t_2\}$. The middle two values are isomorphic, so the isomorphic enumeration yields only three values: {}, {•}, and {•,*}, where • and * are canonical values to be bound to specific values in the labeling.

Before labeling, it is undetermined whether • or * maps to t_1 (or to t_2) in any given variable. The appropriate detection and generation of these labelings is beyond the scope of this paper. The reader here only needs to understand that the final labelings will generate a complete set of isomorphically distinct cases utilizing actual values.

At each leaf of the original enumeration tree, the complete necessary set of labelings of these canonical values is evaluated. These additional values can be thought of as a new tree rooted at each leaf of the enumeration tree, where each level of this new tree corresponds to a variable which must be labeled. The labeling tree labels each relevant value used in each variable, so that an actual value can be tested for each variable.

In this context, there are really two opportunities for pruning, one for each kind of tree. The pruning of the labeling trees looks much like the original pruning of the enumeration tree. It can be pruned if the value of the formula at some leaf of the labeling tree is independent of the lowest levels of the labeling tree. To ensure that the most effective chance of pruning can be obtained, the labeling trees always utilize the same variable ordering discovered earlier for use by the enumeration tree.

The enumeration tree can only be pruned if the value of the entire formula is known without considering any of the pruned variables for all leaves of the associated labeling trees. In other words, the enumeration tree can only be pruned back to the lowest pruning performed across all of the paths of the associated labeling tree. Note that it does not have to be the same determinant causing each pruning of the labeling tree, just that each pruning of the labeling tree must occur at the same (or higher) level as the pruning of the enumeration tree. As can be expected, this greatly reduces the effectiveness of pruning of the enumeration tree, although most of the loss is recovered by the pruning of the labeling tree.

References

[ASU86] A. Aho, R. Sethi and J. Ulman, *Compilers Principles, Techniques and Tools*, Addison Wesley 1986.

[Fu92] N. E. Fuchs. Specifications are (preferably) executable. *Software Engineering Journal,* 1992, 7, (5), pp 323-334.

[HJ89] I. J. Hayes and C. B. Jones, Specifications are not (necessarily) executable, *Software Engineering Journal*, 1989, **4**, (6), pp 330-338.

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

[Ja96] D. Jackson, Nitpick: A Checkable Specification Language, Workshop of Formal Methods in Software Practice (FMSP '96), San Diego, 1996.

[JD96] D. Jackson and C. A. Damon. Elements of Style: Analyzing a Software Design Feature with a Counterexample Detector. *Proc. ISSTA 96*, San Diego, CA, 1996.

[JJD96] D. Jackson, S. Jha and C. Damon. Faster Checking of Software Specifications by Eliminating Isomorphs. *Proc. Principles of Programming* St. Petersburg, Florida, USA, 1996.

[KK88] L. Kanal and V. Kumar, Ed. Search in Artificial Intelligence, Springer-Verlag, New York, 1988.

[SF91] N. M. Sadeh and M. S. Fox. *Variable and Value ordering Heuristics for Hard Constraint Satisfaction Problems: An Application to Job Shop Scheduling*, Technical Report CMU-RI-TR-91-23, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, 1991.

[Sla94] J. 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.

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

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