Selective Enumeration

A Thesis Proposal

Craig A. Damon
School of Computer Science
Carnegie Mellon University
October 27, 1997

Abstract

Sets, functions and relations are powerful structures for modeling software systems. Relational specifications, built from these constructs, are the most common form of formal specification for software systems. However, in sharp contrast to other formal notations, there is an almost complete lack of automated tools for analyzing relational specifications.

A method for solving relational formulae must be at the core of any tool for analyzing relational specifications. In this this thesis, I am developing a method for efficiently solving relational formulae, which I call selective enumeration. Selective enumeration uses a generate-and-test approach, but it prevents the generation of the vast majority of cases, which can be considered duplicates.

I have implemented four selective enumeration techniques: isomorph elimination, bounded generation, short circuiting and derived variable construction. In this thesis I will provide a firm theoretical and empirical framework for understanding these techniques.

Thesis Committee
Daniel Jackson (co-chair)
Jeanette Wing (co-chair)
Gary Miller
Rance Cleaveland (North Carolina State University)

1. Introduction

Sets, functions, and binary relations combine to provide a convenient, yet rigorous, structure for modeling software systems. Z [Spi92], probably the most widely used formal notation for describing software systems, is based entirely on these constructs. As sets, functions, and relations can all be described using relational formulae, I use the term relational specification to describe any specification built on these constructs.

Other software description notations also draw much of their power from these constructs. Within the database community, the inter-relationships in a database schema are often specified using an entity-relationship diagram [Che76]. Given the name, it should not be surprising that entity-relationship diagrams can be clearly and succinctly described using relations.

More recently, UML [BJR97] has gathered great interest in the object community. Although UML combines several different notations to describe a single object design, many of these notations are simply applications of these same sets, functions, and binary relations.

Despite the broad appeal of these constructs, little automated support is available for analyzing relational specifications. Theorem provers [ES94; SM96] can help, but they require enormous manual effort. Model checkers [BC+92; CPS93] can analyze system specifications based on other formalisms, but no model checkers are available for relational specifications.

A method for solving relational formulae must lie at the core of any automated tool for analyzing relational specifications. My thesis research is to provide an efficient method for solving relational formulae as well as a tool for analyzing relational specifications of software systems. My approach solves relational equations by discovering, through search, satisfying assignments to those equations.

As with any search-based approach, my method will only discover solutions within some finite bounds. Although this finite limitation prevents any tool based on selective enumeration from being a true verifier, I think that this limitation is justified. By restricting the problem to finite assignments, a simpler solution is possible, giving the analyst a lightweight tool which can perform much, if not most, of the desired analysis. If, as I believe, most errors in specifications can be exhibited with a small case, most erroneous claims can be refuted using this approach.

Exhaustive enumeration, the simplest possible search-based approach, must consider far too many cases to ever be useful in solving relational formulae. The vast majority of these cases, however, can be considered duplicates. By preventing the generation of most of these duplicates, formulae can be solved efficiently. I call this approach *selective enumeration*.

For my thesis research, I will implement and analyze four techniques: isomorph elimination, derived-variable construction, bounded generation and short circuiting. Together, these techniques prevent the generation of most duplicate cases, making selective enumeration an effective technique for solving relational formulae.

As part of my work, I have implemented a prototype analysis tool, called Nitpick [JD96b], which embodies selective enumeration. Using Nitpick, a developer can analyze a design that has been described in NP [JD96a], a relational formal specification language similar to Z.

This research provides two concrete benefits. I am implementing and analyzing a collection of techniques that can efficiently solve relational formulae. I have also produced Nitpick, a tool that incorporates these techniques. Nitpick has already found a broad range of usages, ranging from purely pedagogical usage in course assignments to the analysis of an ongoing software design of a distributed file system, across applications ranging from the verification of the new mobile IP specification to the analysis of a word processor.

2. A Simple NP Example

Nitpick analyzes properties of specifications written in the specification language NP. NP provides a structuring for formulae written in the relational calculus [Tar41], extended with a transitive closure operator. For those unfamiliar with relational operators, the detailed semantics of each operator provided by NP is given in Figure 3 and Figure 4 at the end of this paper.

Figure 1 gives a simple example of an NP specification. This specification describes the behavior

```
[Addr, Value]

Heap =
[
  usage : Addr -> Value
  used : set Addr
|
  /* all currently mapped addresses are used */
  used = dom usage
]

Alloc(addr : Addr) =
[
  Heap
|
  /* Allocating a new address does not change the current allocation */
  used <: usage' = usage
  /* But addr is now mapped (to some value unknown) */
  used' = used U {addr}
 ]

uniqueAddrAlloc::
[
  Heap
  a : Addr
|
  /* A newly allocated address should not have been in use */
  Alloc(a) => a not in used
]
```

Figure 1: A trivial NP specification describing a heap allocation system. Addr and Value are the given types. Heap describes the basic structure being manipulated, Alloc describes an allocation operation, and uniqueAddrAlloc is a claim about the specification.

of a simple heap allocation system. I will illustrate the various techniques using this specification.

The first line of the example introduces the two *given types* used, `Addr` and `Value`. A given type is a set of elements, each of which has no structure. Every element considered by NP is contained in exactly one given type. All variables and expressions in NP are typed, indicating that they refer to one of three kinds of values. A variable may refer to an element of a given type, it may refer to a set of elements of a single given type, or it may refer to a relation, mapping elements of a given type (the domain) to elements of a given type (the range). Relations can be restricted to functions, injections or bijections and they can be restricted to total or surjective relations.

When using NP, specifiers describe their system using a series of *schemas*, which allow a simple structuring and composition of individual pieces of the specification, similar to the mechanism provided by Z. There are two independent characteristics that can be used jointly to classify schemas in NP. A schema is either a definition, which defines the system being specified, or a *claim*, which makes assertions about the system being specified. A schema, whether a definition or a claim, describes either a single state or a transition of the state. A transitional schema is called an *operation* and describes both a pre-state and a post-state. The specification given in Figure 1 contains examples of three of the four possible combinations of these characteristics, as explained in the following paragraphs.

All schemas have the same basic structure. The *body* of the schema comes after the name of the schema and is enclosed in square brackets (`[]`). The body is separated into two sections by a single vertical bar (|). The first of these sections defines the variables used in the schema, whereas the second section gives a series of relational formulae which must all be satisfied in any system described by this specification.

In the example given in Figure 1, `Heap` is a definitional schema that describes the basic structure of a heap. `Heap` introduces two variables, `usage` and `used`. The variable `usage` refers to a function mapping addresses (elements of `Addr`) to their values (elements of `Value`). The other variable, `used`, refers to a set which contains all of the addresses currently in use. `Heap` also defines a single formula that describes a relationship that must hold in all valid heaps; the set of addresses in use is exactly the set of addresses currently mapped, that is the domain of the function `usage`.

`Alloc` is an operation that describes the change in a heap when a new piece of memory is allocated. As `Alloc` refers to `Heap` in its declaration section, `Alloc` inherits all of the variables defined by `Heap`. Within `Alloc`, the pre-state is referenced using the simple variable names, whereas the post-state is referenced using primed variables, such as `usage`'. Operations are indicated by the presence of a (possibly empty) parameter list.The parameter list for `Alloc` defines a single parameter, `addr`, which is the newly allocated address.

There are two formulas within `Alloc`. The first (`used <: usage' = usage`)[1] guarantees that the allocation does not change any existing mappings. The second formula (`used' = used U Addr`) indicates that the newly allocated address is now considered to be in use (as well as any addresses already in use).

The third schema, `uniqueAddrAlloc`, is a claim that asserts that a newly allocated address is not initially in use.

3. Selective Enumeration in Nitpick

The primary focus of this research is to provide an efficient approach for solving relational formulae. In particular, I want to solve relational formulae which arise when attempting to analyze relational specifications. This section describes the concepts and techniques underlying my approach in the context of refuting the claim `uniqueAddrAlloc` from the example described in the pre-

---

[1] The `<:` operator is the domain restriction operator. The result of this expression is a relation that includes all of the pairs in the relation given as the second argument whose first element is contained in the set given as the first argument. For the formal definition of this and other operators, refer to Figure 3 and Figure 4 at the end of this proposal.

ceding section. Refuting claims is the intended primary usage of Nitpick

Nitpick attempts to solve a relational formula by constructing possible variable assignments. A *variable assignment* maps variables in the formula to values of the appropriate type. A *partial assignment* leaves some of the variables unmapped, whereas a *full assignment* maps every variable to a value. A formula evaluates to a boolean value when interpreted by any full assignment, as well as some partial assignments. A *satisfying assignment* for a formula is an assignment that causes that formula to be evaluated to true.

My approach finds satisfying assignments only if they have a small number of elements. When using Nitpick, an analyst defines a *scope* for the search, providing a finite bound for the number of elements in each given type.

A claim is *valid* if every possible full assignment is a satisfying assignment.When Nitpick refutes a claim, it provides a counterexample to guide the developer in fixing the specification. A *counterexample* is an assignment that demonstrates a case where the claim being investigated does not hold. Rather than directly searching for a counterexample, Nitpick searches for a satisfying assignment for the negation of the claim. If Nitpick is unable to find a satisfying assignment, the claim is valid (at least within the finite bounds of the scope).

As the first step, Nitpick must derive a single formula from the claim. The formula derived from the claim entitled `uniqueAddrAlloc` in Figure 1 is given by Formula 1.

```
used = dom usage and used' = dom usage'
            =>
      ( used = dom usage and used' = dom usage' and
        used <: usage' = usage and used' = used U {a}
            =>
        a not in used )
```

Formula 1: Formula derived from the claim `uniqueAddrAlloc` in Figure 1.

The first line of Formula 1 is simply the body of `Heap`; Nitpick is interested only in cases that satisfy `Heap` but violate the formula found in the claim itself. The third and fourth lines are the expansion of `Alloc`. The last line is the consequent of the formula from the claim itself.

When attempting to find a counterexample, Nitpick negates this formula, yielding Formula 2.

```
used = dom usage and used' = dom usage'
            and
      used = dom usage and used' = dom usage' and
      used <: usage' = usage and used' = used U {a}
            and
      a in used
```

Formula 2: Negation of the formula derived from the claim `uniqueAddrAlloc` in Figure 1.

The simplest approach to finding a satisfying assignment is called exhaustive enumeration. With exhaustive enumeration, every possible full assignment must be checked, typically enumerated using a depth first search. With a finite set of possible values for each variable, the number of possible assignments to search is bounded. Even for a scope with small bounds and a formula with few variables, however, exhaustive enumeration is nearly always prohibitively expensive. For the trivial example in Formula 2 with a minimal scope limiting both `Addr` and `Value` to three elements each, there are 786,432 possible assignments. As the scope grows to six elements each, the number of assignments grows to $3.4x10^{14}$.

For any given formula, the vast majority of these assignments are duplicates of another assignment. Informally, two assignments are duplicates if they both yield the same interpretation of the formula for the same essential reason. By preventing the generation of many of these duplicates, selective enumeration can solve many formulae that are infeasible for exhaustive enumeration. Avoiding the generation of duplicate assignments is the distinguishing property of selective enumeration.

As the elements of a single given type are themselves unstructured, many assignments are distinguished only by the actual choice of elements. A possible partial assignment binding all of the variables in the last term of Formula 2 (`a` **`in`** `used`) is given in Assignment 1.[2] As a_0 is

```
a = a0
used = { a0, a1 }
```

Assignment 1: A possible assignment for the term
`a` **`in`** `used.`

indistinguishable from a_1, choosing a_1 for `a` rather than a_0 cannot change the interpretation. Similarly, I could permute a_0 and a_2 in this assignment (as shown in Assignment 2) without changing

```
a = a2
used = { a1, a2 }
```

Assignment 2: Another assignment for the term
`a` **`in`** `used,`
obtained by permuting a_0 and a_2 in Assignment 1.

the interpretation. Any consistent permutation of an assignment yields a new assignment that has the same interpretation. Therefore any assignment that is a consistent permutation of another assignment is a duplicate, which I call a *permutation duplicate*.

Selective enumeration prevents the generation of most of the permutation duplicates using a technique called *isomorph elimination*. Our isomorph elimination technique [JDD98] consists of solving two distinct, but related, problems. The first of these problems involves establishing an isomorphism context, whereas the second problem is the actual generation of isomorph free values, given a context. The context captures the distinctions between otherwise indistinguishable elements based on other values already bound in the assignment.

Without a context, generating isomorph-free values is a well understood problem [McK94] and provides a substantial reduction in the number of values required. In this example, there are 64 ($(\#Value+1)^{\#Addr}$) possible values of `usage`, counting all permutation duplicates. However, enumerating the complete set of isomorph-free 3x3 functions yields only the 7 values shown in Table

2. In all example values, I will use lower case letters followed by a subscript to represent a particular element of the given type with the same first letter. Thus, a_0 refers to a particular element of the given type named `Addr`.

1.

{ }	{ a_0 -> v_0 }	{ a_0 -> v_0, a_1 -> v_0 }	{ a_0 -> v_0, a_1 -> v_1 }
{ a_0 -> v_0, a_1 -> v_0, a_2 -> v_0 }	{ a_0 -> v_0, a_1 -> v_0, a_2 -> v_1 }	{ a_0 -> v_0, a_1 -> v_1, a_2 -> v_2 }	

Table 1: The seven isomorph-free values for `usage, with a scope #Addr=3 and #Value=3`.

To better understand the role of the context, consider generating values for `a` and `usage` from Formula 2. In the absence of any context, the elements of Addr are indistinguishable from one and another. Therefore, a single, arbitrary element of `Addr` is sufficient to describe all possible, unique values of `a`. A naive enumeration of `usage` would then similarly consider all elements of `Addr` as still indistinguishable. However, once a value has been assigned to `a`, that value is now distinguished from all other elements of `Addr`. Therefore, if `a` is bound to a_0, a value for `usage` which maps only a_0 to v_0 is distinct from a value which maps only a_1 to v_0. The elements of `Value`, on the other hand, are still indistinguishable, so mapping a_0 to v_0 is duplicated when mapping a_0 to v_1.

Our implementation of isomorph elimination does not remove all permutation duplicates because the context only considers simple permutations[3] of the assignment . Although pairs of assignments do exist which are separated by multiple permutations, the number of these is small and the percentage of these pairs among all assignments decreases as the total number of assignments increases [Fag77].

Partial assignment duplication is the second source of duplication that I exploit. For most formulae, some partial assignments can determine the value of the formula as a whole. I call these assignments *determining partial assignments*. These assignments occur when the partial assignment fully binds the variables in a single term in a conjunction and that term evaluates to false under the partial assignment. Each term in Formula 2 includes only a subset of the all of the variables and determines the value of the formula if that term evaluates to false; if the variables `used` and `usage` are bound such that `used =` **`dom`** `usage` is false, then Formula 2 itself is clearly false for any assigment containing those bindings. Any two full assignments which contain that same determining partial assignment must give the same interpretation to that formula and are therefore partial assignment duplicates. I utilize three reduction techniques to prevent generation of these duplicates: derived-variable construction, bounded generation, and short circuiting.

Derived-variable construction [JD95] is the simplest of these techniques. By analyzing the speci-

3. A simple permutation is one that involves permuting only a single pair of elements. A multiple permutation, on the other hand, involves swapping multiple elements at once. Although any multiple permutation can be expressed as a product of simple permutations, isomorph elimination only considers permutations which are themselves meaning preserving. Therefore, any multiple permutation which cannot be constructed as a composition of meaning-preserving simple permutations will not be considered.

fication, selective enumeration can generally discover variables with values that can be constructed from the values of other variables in the specification. I call these variables *derived variables*. Rather than generating numerous non-satisfying assignments for each one of these variables, derived variables are excluded from the search completely, with their values computed constructively to complete the assignment.

The variables `used` and `used'` can be derived in Formula 2. The value of `used` is the same as the value of **dom** `usage` and the value of `used'` is the same as the value of **dom** `usage'`.[4] These variables can thus be dropped from the search, leaving only `a`, `usage` and `usage'`.

Bounded generation looks for constraints in the specification that limit the possible satisfying values for a variable. Some of these constraints are propagated directly to the generation part of the search, preventing the generation of any values not satisfying the constraint. Although not all constraints can be used to restrict generation in this way, enough can to significantly reduce the number of cases generated. Typical constraints that are be propagated involve set membership or subset relationships, where one or more elements can be determined to be included (or excluded) from every satisfying value.

The analysis for bounded generation notices that `a` **in** `used` must hold true for any satisfying assignments of Formula 2. Therefore, selective enumeration will only generate values for `a` that are elements contained in the set referred to by `used` (instead of considering all elements of `Addr`). Assuming a uniform distribution of values of `used`, this will halve the number of values of `a` that need to be considered on average.

By using derived-variable construction and bounded generation, selective enumeration generates only assignments that satisfy many of the terms of Formula 2. Therefore, these terms may be dropped from the formula being checked. The simplified formula is given in Formula 3.

```
used <: usage' = usage and used' = used U {a}
```

Formula 3: Simplified negation of the formula derived from the claim `uniqueAddrAlloc` in Figure 1. This is the only portion of the formula that must be checked during the search

Short circuiting [DJ96] identifies determining partial assignments missed by the other two techniques. During the search, selective enumeration evaluates all terms that have been fully bound by the assignment generated thus far. If any of these terms determine the value of the entire formula, selective enumeration chooses an arbitrary value for each of the remaining variables to complete the full assignment. This full assignment is necessarily a duplicate of any other full assignment that would be generated by continuing the search beyond the last variable enumerated.

Although the second term (`used' = used U {a}`) is only meaningful when all three remaining variables are bound, the first term (`used <: usage' = usage`) requires only `usage` and `usage'` to be bound. As soon as both `usage` and `usage'` are bound, short circuiting will evaluate the first term, pruning the search tree wherever the term evaluates to false.

Before the actual search can begin, selective enumeration must choose the order in which it will consider the variables during the search. For this example, bounded generation imposes one con-

4. Equally acceptable alternative derivations do exist. Instead of deriving `used`, `usage` could be derived, with the same value as the expression `used <: usage'`. An alternative value for `used'` is the value of the expression `used U {a}`.

straint on the search: used must be available before a is enumerated. As used is derived from usage, usage must be enumerated before a. The first term of Formula 3 can be used to prune the search tree, so short circuiting would like usage and usage' enumerated as early as possible in the search. < usage, usage', a> is an ordering that satisfies the constraints of both bounded generation and short circuiting.

Assignment 3 gives a satisfying assignment found by this search, which can then be displayed to

$a = a_0$
$used = \{ a_0 \}$
$used' = \{ a_0 \}$
$usage = \{ a_0 \rightarrow v_0 \}$
$usage' = \{ a_0 \rightarrow v_0 \}$

Assignment 3: A counterexample to the claim uniqueAddrAlloc (from Figure 1).

the user as a counterexample. This counterexample shows a possible execution of Alloc where the newly allocated address (a_0) is already in the set used. The original specification was too weak; it did not require that the newly allocated address is not already in use.

These four techniques have given very promising preliminary results. For this simple example, Nitpick discovers the counterexample after examing only three cases. To cover the entire space, with a scope setting #Addr=3 and #Value=3, requires 273 cases, instead of the nominal 786,432. Much more notably, as the scope grows to #Addr=5 and #Value=5, the number of nominal cases grows to 3.1e11, whereas the number of cases required for Nitpick to cover the entire space is only 43,786.

I have formally defined two forms of duplication of assignments, which I call permutation duplication and partial assignment duplication. Other forms of duplication may also exist; removing these additional duplicates would improve selective enumeration's ability to reduce the search space.

4. Work Remaining

The four reduction techniques — isomorph elimination, short circuiting, derived-variable construction, and bounded generation — are all implemented and work reliably. However, our inability to predict their performance accurately for any particular specification shows that our understanding of these techniques is still far from complete.

Two primary areas of work remain. Most importantly, I need to do a thorough analysis of the techniques, both theoretical and analytical. Secondly, I plan to do a handful of remaining implementation tasks, most notably adding support for two new language features.

I will perform two different forms of analysis: empirical and theoretical. The empirical analyses are necessarily dependent on the implementation changes, thus these analyses must be performed after the implementation work has been completed. The theoretical analyses do not share this dependency. They will generally be completed prior to or concurrently with the implementation changes. In several cases, work performed in the theoretical analysis will determine the final implementation details. Table 2 gives an overview of the analysis tasks.

I will perform four kinds of theoretical analyses: constructing precise formal definitions for each reduction technique, demonstrating the soundness of each technique based on its formal definition, computing the complexity of each technique and computing the expected reduction gained from each technique, including best, worst, and average (mean) cases. For the average-case analysis, I will assume a uniform distribution of values for each variable.

Time Estimate	Task
Theoretical analyses	
6 weeks	Definitions
3 weeks	Soundness
3 weeks	Complexity
3 weeks	Expected reduction
15 weeks	
Empirical analyses	
1 week	Specification suite
1 week	Benchmark harness
2 weeks	Individual measurements
4 weeks	Combined measurements
2 weeks	Performance measurements
2 weeks	Approximation measurements
2 weeks	Normalization measurements
3 weeks	Measurements of other tools
17 weeks	
32 weeks	Total analyses

Table 2: Remaining Analysis Tasks with Time Estimates.
The time estimates are minimum elapsed time. For most of the measurement tasks, only a small amount of effort is required during this elapsed time.

The formal definitions of the techniques form the basis for all the subsequent theoretical analyses, as well as supporting many of the final implementation changes. Current definitions exist for all the techniques, but these definitions vary in their levels of completeness and correctness.

From these formal definitions, I will construct a soundness argument for each technique. In the context of these reduction techniques, soundness means that the search will always find at least one satisfying assignment if any exist. There are existing soundness arguments for each technique on which I will base this additional work.

I will compute the (worst-case) algorithmic complexity of the mechanisms used. There are several mechanisms involved with any one reduction technique and some mechanisms are shared among more than one reduction technique. I will find the overall complexity of each reduction technique by appropriately composing the complexity computed for each relevant mechanism.

The expected reduction of the search space obtained from each technique is generally more interesting to the user than is the algorithmic complexity of that technique. The reductions are, of course, not expected to be constant across different specifications. Instead, I will develop parametric expected-reduction equations. The parameterization will differ for each technique. For short circuiting, the expected reduction will depend on the opportunities discovered, whereas for isomorph enumeration, the expected reduction will depend on the number and kind of other variables of the same type. The best- and worst-case expected reductions will bound the possible reductions, whereas the average-case expected reduction should give realistic expectations for most specifications.

In addition to estimates of each technique individually, I will also consider the interactions among reduction techniques. For example, bounded generation significantly limits the opportunities for short circuiting, whereas bounded generation minimally affects the reduction opportunities for isomorph elimination. I will produce a best- , worst- , and average-case change in the expected reduction for each technique with each combination of enabling and disabling the other techniques.

The empirical analyses are the counterparts of the complexity and expected reduction analyses, measured against a number of actual specifications. I will select several specifications, on the order of a dozen, to use consistently across all the empirical analyses. These specifications will be representative of a broad category of possible specifications, chosen to exhibit a set of desired characteristics. I will define the desired characteristics and choose the actual set of specifications.

One of the goals of the empirical analysis is to derive a model of the techniques that can be used to predict, with reasonable accuracy, the expected number of cases required for any formula. In my initial analysis of short circuiting and its interaction with isomorph elimination, I noted that the log ratio of the reduction is one such possible model. The log ratio is the ratio of the logarithm of the number of assignments needed to cover the entire space to the logarithm of the total number of possible assignments. This ratio therefore represents the exponential reduction in the size of the total search space.

Based on limited empirical evidence, the log ratios appear to remain approximately linear, for a given specification, as the size of the total search space is varied (by varying the scope limiting the search). The log ratio for each of the reduction techniques individually tends towards 0.5. Given four reduction techniques and a tester that is capable of checking approximately 10^4 assignments per second, Nitpick should be able to consider spaces of up to 10^{64} possible assignments in a single second. Unfortunately, such rosy projections are not born out in reality, where handling 10^{15} to 10^{20} possible assignments per second is more common. Availability of additional empirical data, as well as a thorough analysis of the data, should refine this model to give a more realistic prediction.

In addition to the number of cases required, I will measure the computation time required for the search, with and without each technique. We have investigated techniques that turned out to cost more to compute than they provided in savings from a reduced search space. Our current techniques may exhibit this undesirable behavior for some specifications. If I understand these considerations, I may be able to target future optimization work or to develop heuristics that can automatically disable inappropriate techniques.

Another useful metric is how much of the total space must be searched before the first satisfying assignment is found. This measurement is most relevant when the reduction techniques are used to verify properties of relational specifications. We believe that most specifications will contain bugs, so many searches can be terminated prematurely. This analysis will help evaluate the new clause ordering heuristics to be used by the normalization, as well as providing a more realistic view of the current use of the technology.

The current implementation uses approximate values where an exact value cannot be computed effectively or efficiently, most notably in choosing both a variable ordering and a clause ordering. I will measure the accuracy of these approximations for a handful of the specifications under consideration. I will then determine the effects of the inaccuracies found in these approximations, using the same specifications.

5. Implementation Tasks

The major new language feature to be implemented is limited support for natural numbers. Manip-

Estimated Time		Task
New features		
	2 weeks	Natural numbers and sequences
	1 weeks	Quantifiers
3 weeks		
Structural improvements		
	1 week	Share consequence closure mechanism
	2 weeks	Implement and utilize DNF conversion
3 weeks		
Algorithmic changes		
	2 weeks	Improved variable ordering
	1 week	Correctly handle same-type relations
	2 weeks	Repair or re-implement isomorph-free generators
	1 week	Repair progress bar
6 weeks		
12 weeks		All implementation changes

Table 3: Remaining Implementation Tasks with Time Estimates.

ulation of natural numbers will be confined to a handful of new operators: addition (and subtraction), comparison, and the cardinality operator (#), which yields the number of elements in a set or edges in a relation. Natural numbers will be allowed anywhere that elements of given types are currently allowed. This ability introduces the major difficulty in supporting natural numbers — setting enumeration bounds that will neither unduly mask satisfying assignments nor unreasonably hamper the ability to search efficiently for satisfying assignments. I am still investigating the appropriate mechanisms to define these bounds.

Support for natural numbers also gives direct support for sequences. I will build special generators that will efficiently support the use of sequences.

The other feature to be added, quantifiers, is not truly new. I previously built a prototype implementation that gives support for both universal and existential quantifiers in the relational formulae. Although the prototype's basic approach and implementation are valid, the concepts behind the implementation need to be clarified and formalized, and their soundness demonstrated. Subsequent new features and implementation changes have introduced problems with the prototype implementation that need to be addressed.

There are a number of other miscellaneous cleanup tasks that also remain. Some of these tasks fix bugs in the existing implementation, whereas others will offer further incremental improvements in the reductions gained by the various techniques. One task, the normalization to disjunctive normal form (DNF), will improve the reliability and predictability of any measurements taken of the techniques. A complete listing of the remaining implementation tasks is give in Table 3.

6. Related Work

There are two basic families of related work: methods that solve similar problems and seemingly unrelated research that has contributed, directly or indirectly, to mechanisms supporting selective enumeration. A few methods have been introduced that can solve relational formulae. Some of these methods offer a more general solution, providing support for domains other than relational formulae. Although these general solutions can be applied to solving relational formulae, they suffer from a loss of information. Most of the remaining methods solve problems that are similar, but not strictly related. These solutions may provide some insight, but frequently offer little other gain in solving the relational formulae considered here.

The model-generation community [Zha96; ZZ95; Sla94] has addressed a problem that is similar to the relational formula satisfaction problem addressed by selective enumeration. A model-generation tool searches for an assignment, called a model in their context, which satisfies a formula. The logic supported by the model-generation tools varies from the logic supported in Nitpick. Variables are allowed to be arbitrary arity functions in most of these tools, whereas NP directly supports only unary functions. (Arbitrary arity functions can be expressed in NP through careful encoding, but the resultant formula is hard to understand and Nitpick is particularly inefficient at analyzing such formulae.) Nitpick, on the other hand, adds support for quantifiers and transitive closure, both of which are difficult or impossible to express in the model-generation languages. There is also a difference in apparent goals: Nitpick is best suited for solving formulae with several variables using a relatively small scope, whereas the model generators appear targeted towards formulae with few variables (frequently one) using a larger scope.

Despite these differences, some of the model generators use an approach similar to the one used in Nitpick. Zhang [Zha96], in particular, used a similar set of reduction techniques in FALCON. FALCON includes a simpler form of isomorph elimination, a direct equivalent to derived-variable construction and a backtracking feature that is similar to short circuiting. Slaney [Sla94] also uses a backtracking approach in Finder; he achieves reductions similar to those gained with bounded generation by separating the enumeration of functions into separate boolean variables, each representing a single maplet. As noted in the preceding section, I will measure the performance of these tools on formulae that can be expressed for both these tools and Nitpick, which will provide a more concrete comparison.

Jipsen's approach [Jip92] finds a Boolean algebra with operators (BAO) that satisfies a set of first-order equations. His approach does not require finite bounds, and thus can be used as a true verifier. As the relational calculus can be embedded in BAO, this approach will also solve relational formulae. However, there are a number of difficulties with Jipsen's work. His approach has never been proved complete — it may never terminate for an unsatisfiable system of equations. As transitive closure is not supported within BAO, most of our specifications could not be expressed in full generality. No experimental results are provided, so it is not possible to compare his approach to Nitpick, even for the finite domain.

The most general related problem is the well known boolean-satisfiability problem. Although the problem itself is NP-complete, researchers have attempted two major approaches to achieve an effective solution in realistic time for many formulae. One approach, taken in [SLM92] among others, provides an incomplete solver, which may fail to return a solution even if one exists. In the other widespread approach, a structure or algorithm provides significantly reduced exponential growth for common formulae, although the worst-case performance may lag even the most naive approach. Binary decision diagrams (or BDDs) [Bry92] are a popular structure that provides this generally reduced exponential growth.

There is a straightforward translation from relational formulae to boolean formulae, so any of the boolean satisfiability approaches can be applied to solve any relational formulae. This conversion loses much of the higher level semantics of the relational formulae. Selective enumeration uses these semantics to produce its reductions. As given in [DJJ96], translating a relational formula into the corresponding boolean one and solving the boolean formula using BDDs required approximately the same time as solving the relational formula directly using selective enumeration. The introduction of bounded generation and an improved isomorph-elimination technique has moved the balance significantly towards selective enumeration. Although it is possible that additional effort on the BDD version could similarly reduce the time required, I believe that the additional semantics available in the relational formula will give an advantage to selective enumeration.

Although the techniques employed by other formal-methods tools [CW+96] are quite different from selective enumeration, many of these tools also attempt to verify a formula. The primary research challenge in model checking, introduced in [CE81], has been to reduce the state space explosion. One successful approach translates the problem into a more general, but well understood one, such as boolean satisfiability. As noted in the preceding paragraphs, such solutions come at the cost of sacrificing higher level structure that might be usefully exploited. Other successful model checkers exploit this structure. The mechanism for exploiting this structure, however, is only applicable to the specific problem under consideration. None of these tools have focused on relational formulae, so little, if any, of this work can be directly applied to solving relational formulae.

There have also been tools that use a theorem proving approach to solve relational formulae [BH94] or verify properties of relational specifications [ES94; SM96].These tools remove the finite bounds required by Nitpick, thereby providing a true verification or refutation. Unlike Nitpick (or model checkers), theorem proving still requires enormous human effort, generally from an expert user, and provides no counterexamples for investigating invalid claims.

Solving a relational formula could be structured as a constraint satisfaction problem. Traditional constraint satisfaction approaches [Kum92; Mac92], used to solve problems such as shape recognition [Wal75] or job shop scheduling [SF91], support only a much more limited constraint language. Finite constraint satisfaction, the area most similar to Nitpick, allows only a restricted subset of Horn clauses to express the constraints. For most of the existing constraint satisfaction algorithms, all constraints must be binary (involve no more than two variables). Constraint satisfaction algorithms also typically require a complete enumeration of possible values for each variable, which can be prohibitively expensive for the relation-typed variables commonly found in NP specifications.

There are significant similarities between the approaches taken in constraint satisfaction and the approach taken in Nitpick. Backtracking in constraint satisfaction is a direct equivalent of short circuiting in Nitpick, requiring the same care in selecting a variable ordering. The standard constraint propagation algorithms, including arc consistency and k-consistency, are strongly reminiscent of bounded generation. They are limited, however, to the weaker constraint language.

Isomorph elimination draws heavily on many well developed principles of graph theory. McKay [McK81; McK94], in particular, has extensively investigated issues surrounding isomorph-free generation of bi-partite graphs, which include relations and functions. Unfortunately, his techniques do not easily allow for an arbitrary coloring context, as required by our current isomorph-elimination technique.

The concept of using symmetries of the problem to reduce the search space for model checking has been investigated by Jha [Jha96], among others. Whereas some of this work has been directly incorporated, existing model checkers use a fundamentally different approach to perform their search, limiting the applicability of the work for solving relational formulae.

Some of the mechanisms used in selective enumeration draw directly or indirectly from existing compiler optimizations [ASU86]. The mechanisms for selecting a partial assignment for consideration in short circuiting draw both from traditional common-sub-expression elimination and from short-circuiting optimization for languages without strict ordering requirements, such as Pascal. Past research in range checking analysis provides many useful mechanisms for bounded generation, particularly for the analysis of natural number expressions.

7. Schedule

In broad terms, I will perform the work in the following order:

(1) theoretical analyses (15 weeks)

(2) implementation changes (12 weeks)

(3) empirical analyses (17 weeks)

(4) write dissertation (20 weeks)

I will undertake some of the implementation changes while still performing the theoretical analyses. If an implementation change is directly related to a theoretical analysis, the change will be implemented immediately following the completion of the corresponding analysis task.

As some of the empirical measurements require extensive time to complete, but do not require significant attention while they are running, I will begin writing the dissertation in parallel with these measurements. I believe that, once the test harness has been completed and the other tools are working on the translated specifications, the empirical measurements will require approximately one fifth of my time to run and evaluate. The remainder of this time will be spent writing. Therefore, the first 12 weeks of writing time should be completed during the measurement work, leaving 8 weeks of additional writing time. A task oriented high level schedule is given in Figure 2.

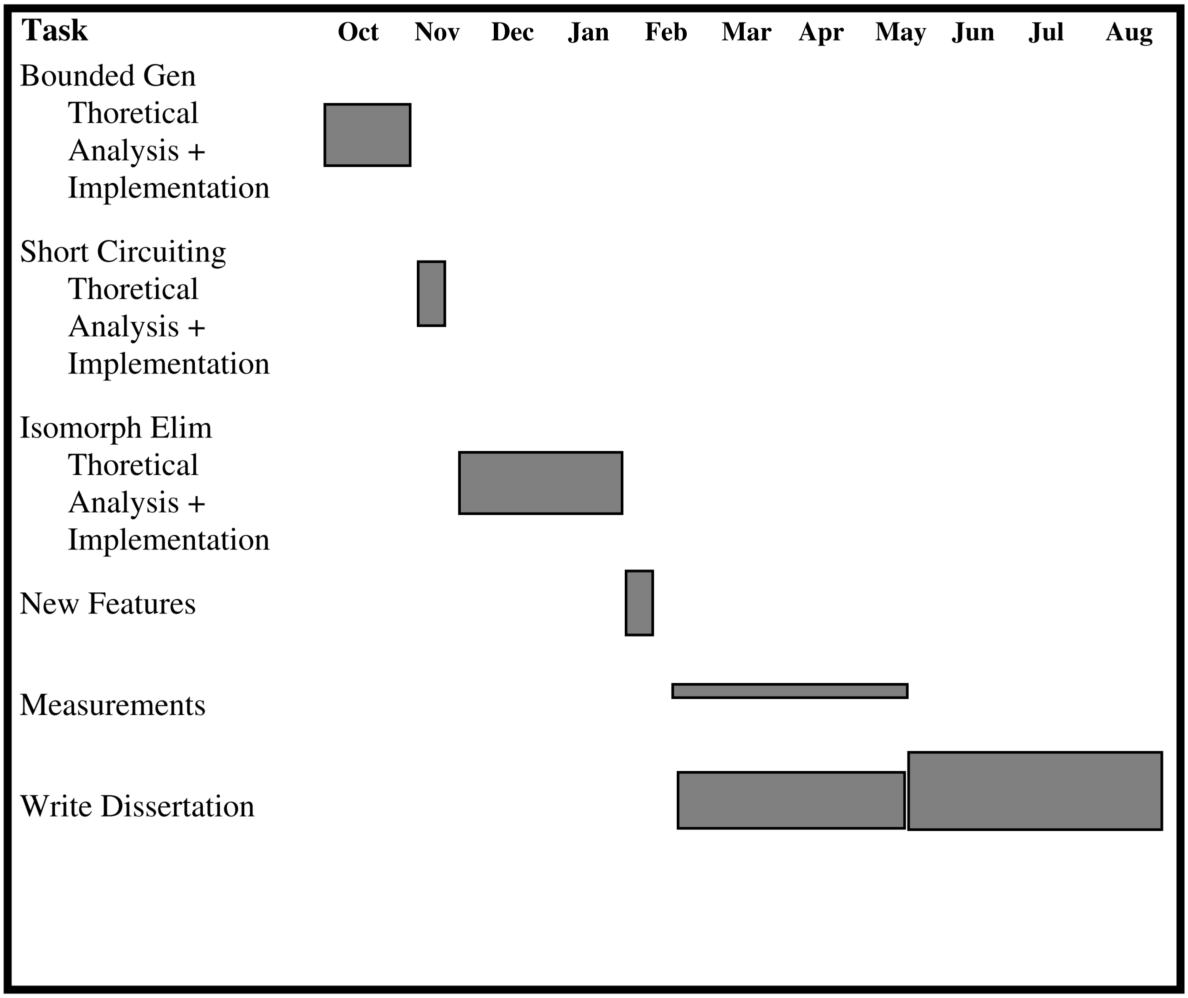

Figure 2: Schedule overview for the remaining work. All tasks except the measurements and writing the dissertation are full time. From February through May, the measurements will take approximately 1/5 of the time, with the remainder going to writing the dissertation.

Variables Used in Definitions

a, a2	:	A
b, b2	:	B
c	:	C
f	:	A -> B
p	:	B <-> C
q,r	:	A <-> B
s,t	:	set A
w	:	set B

Predicates

fun q	(Functional)	$\forall(a,b) \in q.\ (a,b2) \in q \Rightarrow b = b2$
inj q	(Injective)	$\forall(a,b) \in q.\ (a2,b) \in q \Rightarrow a = a2$
tot q	(Totality)	dom q = A
suj q	(Surjective)	ran q = B

Comparisons

s < t	(Proper subset)	$(a \in s \Rightarrow a \in t)$ and not s = t
s <= t	(Subset)	$a \in s \Rightarrow a \in t$
q < r	(Proper subset)	$((a,b) \in q \Rightarrow (a,b) \in r)$ and not q = r
q <= r	(Subset)	$(a,b) \in q \Rightarrow (a,b) \in r$
a in s	(Membership)	$a \in s$

Set operators

s U t	(Union)	$\{ a \mid a \in s$ or $a \in t \}$
s & t	(Intersection)	$\{ a \mid a \in s$ and $a \in t \}$
s \ t	(Difference)	$\{ a \mid a \in s$ and $a \notin t \}$

Special Set Values

{ }	(Empty Set)	$\{ \}$
Un	(Universal)	$\{ a \mid a \in$ given-type $\}$

Figure 3: Definition of operators supported in NP (Part 1).

Relational operators

$q \cup r$	(Union)	$\{ (a,b) \mid (a,b) \in q \text{ or } (a,b) \in r \}$
q & r	(Intersection)	$\{ (a,b) \mid (a,b) \in q \text{ and } (a,b) \in r \}$
q \ r	(Difference)	$\{ (a,b) \mid (a,b) \in q \text{ and } (a,b) \notin r \}$
q ; p	(Compose)	$\{ (a,c) \mid \exists b.\ (a,b) \in q \text{ and } (b,c) \in p \}$
q (+) r	(Override)	$\{ (a,b) \mid (a,b) \in r \text{ or } ((a,b) \in q \text{ and } \forall b2.(a,b2) \notin r) \}$
s <: q	(Domain restrict)	$\{ (a,b) \mid (a,b) \in q \text{ and } a \in s \}$
s <; q	(Neg. dom. restr.)	$\{ (a,b) \mid (a,b) \in q \text{ and } a \notin s \}$
q :> w	(Range restrict)	$\{ (a,b) \mid (a,b) \in q \text{ and } b \in w \}$
q ;> w	(Neg. ran. restr.)	$\{ (a,b) \mid (a,b) \in q \text{ and } b \notin w \}$
dom q	(Domain)	$\{ a \mid \exists b.\ (a,b) \in q \}$
ran q	(Range)	$\{ b \mid \exists a.\ (a,b) \in q \}$
f.a	(Application)	b where $(a,b) \in f$ or undefined
q.s	(Image)	$\{ b \mid \exists a.\ ((a,b) \in q \text{ and } a \notin s) \}$
q~	(Transpose)	$\{ (b,a) \mid (a,b) \in q \}$
q+	(Transitive closure)	$q \cup (q;q) \cup (q;q;q) \cup \ldots$
q*	(Refl. trans. closure)	$q+ \cup Id$

Special Relational Values

{}	(Empty Set)	$\{ \}$
Un	(Universal)	$\{ (a,b) \mid a \in$ given-type of domain and $b \in$ given-type of range $\}$
Id	(Identity)	$\{ (a,a) \mid a \in$ given-type $\}$

Figure 4: Definition of operators supported in NP (Part 2).

8. Bibliography

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ulman, *Compilers: Principles, Techniques and Tools,* Second Edition, Addison Wesley, 1986.

[BH94] Rudolf Berghammer and Claudia Hattensperger. *Computer-Aided Manipulation of Relational Expressions and Formulae Using RALF.* Technical Report, Institut fur Informatik und Praktische Mathematik, Christian-Albrechts Universitat Zu Kiel, Kiel, Germany, 1994.

[BJR97] Grady Booch, Ivar Jacobson, and James Rumbaugh. *The Unified Modelling Language for Object-Oriented Development.* Documentation Set, version 1.0, Rational Software Corporation, Available at <http://www.rational.com>.

[Bry92] Randal E. Bryant. Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams. *ACM Computing Surveys,* Vol. 24, No. 3, September 1992, pp. 293–318.

[BC+92] J. R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. Symbolic Model Checking: 1030 States and Beyond. *Information and Computation***,** Vol. 98, No. 2 (June 1992), pp. 143–170.

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

[CE81] Edmund M. Clarke and E. Allen Emerson. Synthesis of synchronization skeletons for branching time temporal logic. *Logic of Programs: Workshop* (Yorktown Heights, NY), *Lecture Notes in Computer Science*, Vol. 131, Springer Verlag, 1981, pp 52–71.

[CW+96] Edmund M. Clarke, Jeanette M. Wing, et al. Formal Methods: State of the Art and Future Directions. *ACM Computing Surveys,* Vol. 28, No. 4, December 1996, pp. 626–643.

[CPS93] Rance Cleaveland, Joachim Parrow, and Bernhard Steffen. The Concurrency Workbench: A Semantics-Based Tool for the Verification of Concurrent Systems. *ACM Transactions on Programming Languages and Systems,* Vol. 15, No. 1, January 1993, pp. 36–72.

[DJ96] Craig A. Damon and Daniel Jackson. Efficient Search as a Means of Executing Specifications. *Proceedings Second International Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '96)*, Passau, Germany, *Lecture Notes in Computer Science,* Vol. 1055, Springer Verlag, March 1996, pp. 70–86.

[DJJ96] Craig A. Damon, Daniel Jackson, and Somesh Jha. Checking Relational Specifications with Binary Decision Diagrams. *Proceedings 4th ACM SIGSOFT Conference on Foundations of Software Engineering,* San Francisco, CA, October 1996, pp. 70–81.

[ES94] Marcin Engel and Jens Ulrik Skakkeback. *Applying PVS to Z.* Technical Report ID/DTU ME 3/1, ProCos Project, Department of Computer Science, Technical University of Denmark, Lyngby, Denmark, 1994.

[Fag77] Ronald Fagin. The Number of Finite Relational Structures. Discrete Mathematics, 1977, Vol. 19, pp 50-58.

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

[JD96a] Daniel Jackson and Craig A. Damon. *Nitpick: A Checker for Software Specifications (Reference Manual).* Technical Report CMU-CS-96-109, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, January 1996.

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

[JJD96] Daniel Jackson, Somesh Jha, and Craig A. Damon. Faster Checking of Software Specifications by Eliminating Isomorphs. *Proceedings of ACM Symposium on Principles of Programming Languages (POPL '96),* St. Petersburg Beach, FL, January 1996, pp. 79–90.

[JJD98] Daniel Jackson, Somesh Jha, and Craig A. Damon. Isomorph-free Model Enumeration: A New Method for Checking Relational Specifications. *ACM Transactions on Programming Languages and Systems,* To appear.

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

[Jip92] Peter Jipsen. *Computer-aided Investigations of Relation Algebras.* Doctoral thesis, Dept. of Mathematics, Vanderbilt University, Nashville, Tennessee, May 1992.

[Kum92] Vipin Kumar. Algorithms for constraint satisfaction, A survey. *AI Magazine,* Vol. 13, No. 1, Spring 1992, pp. 32–44.

[Mac92] Alan K. Mackworth. The logic of constraint satisfaction. *Artificial Intelligence,* Vol. 58, December 1992, pp. 3–20.

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

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

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

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

[SLM92] Bart Selman, Hector Levesque, and David Mitchell. A new method for solving hard satisfiability problems. *Proceedings 10th National Conference on Artificial Intelligence (AAI-92),* San Jose, CA, July 1992, pp. 440–446.

[Sla94] John K. Slaney. Finder: Finite Domain Enumerator, System Description. *Proceedings of 12th International Conference on Automated Deduction,* Nancy, France, Alan Bundy (ed.), *Lecture Notes in Artificial Intelligence*, Vol. 814, Springer Verlag, Berlin, 1994, pp. 798–801.

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

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

[Wal75] D. Waltz. Understanding Line Drawings of Scenes with Shadows. In *The Psychology of Computer Vision,* ed. Patrick H. Winston, McGraw Hill, 1975, pp. 19–91.

[ZZ95] Jian Zhang and Hantao Zhang. Constraint Propagation in Model Generation. *Proceedings Principles and Practice of Constraint Programming - CP'95,* Ugo Montanari, Francesca Rossi (Eds.), Cassis, France, September 1995, Lecture Notes in Computer Science, Vol. 976, Springer 1995, pp. 398–414.

[Zha96] Jian Zhang. Constructing Finite Algebras with Falcon. *Journal of Automated Reasoning,* Vol. 17, No. 1, August 1996, pp. 1–22.