A Forward Inference Engine to Aid in Understanding Specifications*

Donald Cohen
USC Information Sciences Institute
4676 Admiralty Way
Marina del Rey, CA 90292

Abstract: An important part of understanding a specification is recognizing the consequences of what is stated. We describe a program that can help a user acquire this understanding. It does this by deriving interesting, though not deep consequences of a set of input axioms, while avoiding (a typically much larger set of) uninteresting consequences. The heuristics for obtaining that effect are described and justified. The program has been used in a symbolic evaluator that helps a user to understand and debug specifications written in the Gist specification language.

1. Introduction

A specification can be viewed as a set of facts describing an existing or desired system. This paper describes a program called FIE (for "Forward Inference Engine"), which finds interesting consequences of a set of input facts. Such a "kibitzer" program (a term suggested by Elliot Soloway) can help us to understand the system. This is useful either in designing a new system or in trying to understand an existing system. FIE is the underlying inference engine in a prototype symbolic evaluator for Gist specifications [Cohen 83].

Humans automatically draw consequences of new facts. A kibitzer does the same thing. To the extent that the results overlap, the kibitzer confirms the user's understanding. Results that he failed to anticipate may reveal important properties of the system. Results that contradict his beliefs reveal bugs, either in his mental model of the system or in the formal description.

2. Examples

Imagine a user trying to specify a domain, perhaps as a step in database design. The kibitzer prompts with ">" and announces results in upper case. The user types in lower case. We start with an example of the kibitzer finding an expected result:

```
>every person has exactly one sex.
>no person has a spouse with the same sex.
NO PERSON IS HIS OWN SPOUSE.
>why?
A PERSON WHO IS HIS OWN SPOUSE HAS THE SAME SEX
AS HIS SPOUSE.
>
```

The next example shows an unexpected result:

```
>ships may carry (any number of) cargo objects.
>no ship carries both grain and fuel (types of
cargo object).
>suppose some ship, s, is carrying some cargo
object, c.
IF S CARRIES ANY GRAIN THEN C IS NOT A FUEL.
IF S CARRIES ANY FUEL THEN C IS NOT A GRAIN.
C MUST NOT BE BOTH A GRAIN AND A FUEL.
>
```

The first two results are expected. The third seems trivial at first - after all, nothing is both a grain and fuel. However, this is *not* implied by the specification. (The axioms only imply that no cargo object *carried by a ship* is both grain and fuel!)

An intelligent kibitzer can be expected to refrain from reporting uninteresting consequences. Hence, an apparently trivial output is evidence of a misunderstanding - the kibitzer does not think the result is trivial. Typically, such a result follows trivially from a belief on the part of the user which is not shared by the kibitzer. If the specification is taken to be definitive, this indicates an unjustified assumption on the user's part. If the user is debugging the specification, this indicates an omission.

The ship example comes from a Gist specification which declared grain and fuel as subtypes of cargo, but failed to declare them as disjoint. Notice that this is not discovered until we suppose that there is a ship carrying cargo. FIE tends not to "speculate" very far by imagining situations. Rather the user guides its exploration by providing explicit suppositions.

Finally we present an example in which an expected result is not found:

```
>every party has exactly one candidate.
>the president is the candidate of the winning
party.
>if the republican party wins, reagan is the
president.
>
```

When I wrote this example, I expected to be told that Reagan was the Republican candidate. It turns out that this expectation (which is widely shared) rests on an interpretation of "if ... then ..." which does not correspond to classical implication: the formal specification does not match our intent.

The failure of an intelligent kibitzer to report an expected result suggests that the user may be wrong to expect it. This may be a symptom of the user's incorrect reasoning or of a missing axiom.

At this point a user would probably like to ask why the expected result does not hold (or whether it does). Another way to phrase this question, is under what circumstances would the expected result *not* apply. This could be (but has not yet been) implemented by supposing that the result is false, and reporting any interesting consequences, i.e., using the kibitzer as a weak refutation theorem prover:

```
>when wouldn't reagan be the republican
candidate?
SUPPOSE THE REPUBLICANS DO NOT WIN.
>
```

*This research was supported by the Air Force Systems Command, Rome Air Development Center, under contrace No. F30602 81 K 0056, and by the Defense Advanced Research Projects Agency under contract No. MDA903 81 C 0335. Views and conclusions contained in this report are the author's and should not be interpreted as representing the official opinion or policy of RADC, DARPA, the U.S. Government, or any person or agency connected with them.

With the exception of English input, all the pieces of the above system exist in prototype form. Instead of English input, we currently use either Gist or predicate calculus. FIE discovers the consequences. In the examples above (except for the unimplemented "Why not?" segment) it reported all the results shown and no others. The Gist behavior explainer [Swartout 83] is capable of converting the results to English, and to some extent can explain proofs.

For an extended example that illustrates FIE's role in symbolic execution of Gist specifications [Cohen 83], see [Swartout 83].

3. Requirements for an Effective Kibbitzer

FIE is the theorem proving component of the kibitzer illustrated above. Unlike a conventional theorem prover, it has no specific target theorem, but rather the more general goal of finding the interesting consequences of its input axioms.

We will not attempt to formally define interestingness. Informally, a user should feel that it's worth his time to read the output. One heuristic is that a result is *not* interesting if it follows trivially from other known results. FIE therefore tries to suppress closely related results. The meaning of "closely related" and the user's influence over it are described later.

Other heuristics are related to symbolic execution. For example, Gist allows descriptive reference. It is therefore important to know whether two descriptions refer to the same object. We feel that these heuristics will not prevent FIE from being useful for other applications, though additional heuristics might well be appropriate.

In order to provide useful interactive assistance, FIE must be fairly fast - it should rarely take more than a minute or so. This precludes the sort of "deep" consequences that challenge today's theorem provers. However, shallow consequences may still surprise or interest a user. (After all, human kibitzing is useful even in the absence of deep consequences.)

4. Overview

FIE can be viewed as a function that accepts a set of "old" facts, modeling a state of understanding, and a set of "new" facts to be integrated into that model. It returns a set of facts equivalent (in the sense of two way implication) to the union of these input sets. The "interesting results" are the output facts that were not in either of the input sets.

Initially the "old" set is empty. Subsequently it contains the results of previous calls. The advantage of dividing the input into two sets is that FIE need not consider interactions among already integrated facts. It simply integrates one new fact at a time. (For efficiency, FIE integrates simpler facts first.)

We now describe how FIE integrates new facts. (If you don't want to see technical details, skip to the end of the paper.) We use terminology common in the literature of logic and automatic theorem proving. Definitions of these terms can be found in [Loveland 78]. FIE relies heavily on well known techniques from resolution theorem proving. Most of this paper describes additions and alterations to these techniques that have been useful in the kibitzing application.

Facts are represented as clauses. A clause, *current*, is added to the set of old clauses, *old*, in three phases, which are described in detail in the following sections:

1. Consider *current* in isolation: it is simplified and canonicalized, and its factors are found (they will also be added).
2. Consider interactions between *current* and members of *old* which justify simplifications (of either). Whenever a clause is simplified, the unsimplified version is discarded and the simplified version is put into the set of clauses to be integrated.
3. Consider interactions between *current* and members of *old* to generate new consequences.

4.1. Logical language

FIE uses a typed version of first order logic: every variable and object has a type. It is assumed that there is at least one object of each type. (If not, the type should be replaced by a new predicate on objects of a supertype and all inputs should be changed accordingly.) FIE relies on external decision procedures to tell whether two types are disjoint and whether one type is a subtype of another. Some objects are further classified as *literals* which are assumed to be distinct objects. All other objects (including skolem functions) are essentially names which may or may not refer to distinct objects. In the examples below we will use letters near the end of the alphabet (e.g., x, y, z) for variables. Function and predicate names can be distinguished by position. Literals will be capitalized. Where appropriate, terms will be subscripted to indicate type.

5. Processing a Clause in Isolation

In the first phase, *current* (the new clause to be integrated) is simplified. This is important, but mostly mundane from a technical standpoint, e.g., ~True → False, (P ∨ False ∨ Q) → (P ∨ Q), (P ∨ Q ∨ P) → (P ∨ Q), (P ∨ Q ∨ ~P) → True.

5.1. Equality Simplification

The algorithm for simplifying equalities makes use of the type structure decision procedures. If the types of the two objects are incompatible the equality is False. If the two objects are different literals the equality is False. Other cases specific to symbolic execution are also recognized. For example, in Gist it is possible to create and destroy objects. An object that is created must be distinct from any object that was known to exist before. If none of these apply, the equality is ordered so as to make it preferable to substitute the first term for the second: constants are preferred to variables, terms of more specific type are preferred to terms of more general type, etc. As a last resort, all expressions are ordered alphabetically.**

5.2. Substitution for Restricted Variables

The next two steps simplify results that don't seem to arise very often in normal theorem proving. The first substitutes for variables in inequality literals, e.g., ~(= a x) ∨ (P x) is rewritten as (P a). In particular, if the inequality is between a variable, x, of type t1, and a term, a, of type t2, where a contains no variables and t2 is a subtype of t1, then the inequality literal is discarded from the clause, and all occurrences of x in the remaining literals are replaced by a. In part, this rule is used to apply substitutions computed in the generalized resolution procedure described below.

5.3. Equality Canonicalization

The next step is analogous but allows inequalities between non-variables, e.g., ~(= a b) ∨ (P b) becomes ~(= a b) ∨ (P a). Intuitively, someone who knows "if a = b then (P b)" also knows

** Substituting the first item for the second is a common operation below. However, the substitution only affects top-level terms, e.g., even if (f x) were to be substituted for x, there would be no infinite loop. Also, it is possible to mark certain literals as "already canonical", in which case they will not be simplified.

"if a = b then (P a)". In this case the inequality literals must be kept. While the previous rule is a clear simplification, this one is a canonicalization, allowing clauses to be recognized as equivalent.

6. Interactions that Simplify

The second phase uses information in one clause to simplify another clause. We describe modifications to the standard procedures for subsumption and equality substitution.

6.1. Conditional Equality Canonicalization

First the clauses in *old* are used to further simplify and canonicalize *current* via substitution of equalities. This may be regarded as an efficient restriction of paramodulation - doing in parallel all paramodulations which can be viewed as simplifying. A special case is demodulation: if we know (= a b), the clause (P b) is rewritten as (P a).

More generally R ∨ (= a b) can be used to rewrite R ∨ (Q b) as R ∨ (Q a). The intuitive justification is that someone who knows "if P then a = b" will consider "if P then Q(b)" equivalent to "if P then Q(a)". As the name suggests, conditional equality canonicalization is especially valuable to FIE in dealing with conditional statements.

In general, we argue that if the clauses C and D combine to yield E, then A ∨ C should combine with A ∨ D to yield A ∨ E. This holds for FIE's generation of new consequences (described later), and we think that it would be appropriate for resolution theorem provers in general. (Note that this is not true of ordinary demodulation.)

Incidentally, these substitutions do not always yield unique results. Given P ∨ (= a c) and Q ∨ (= b c) we can rewrite P ∨ Q ∨ (R c) in two different ways. We have done nothing about this.

The final generalization is that if R1 ⊃ R2, then R1 ∨ (= a b) can be used to rewrite R2 ∨ (Q b) as R2 ∨ (Q a). The previous rule is obtained if implication is only recognized between identical formulae. Subsumption is the obvious candidate for a stronger recognizer of implication.

As an example, suppose 1. no box has two distinct locations, and 2. every box is at location1. These imply 3. no box is at any location other than location1. One feels intuitively that fact 3 implies fact 1. The general rule allows fact 3 to rewrite fact 1 and subsume the result:

1. $(= y_{loc}\ z_{loc}) \vee \sim(Loc\ x_{box}\ y_{loc})) \vee \sim(Loc\ x_{box}\ z_{loc})$

is canonicalized by

2. $(= loc1\ y_{loc}) \vee \sim(Loc\ x_{box}\ y_{loc})$

to yield (we have arranged to use the null substitution)

$(= loc1\ z_{loc}) \vee \sim(Loc\ x_{box}\ y_{loc})) \vee \sim(Loc\ x_{box}\ z_{loc})$

which is subsumed by clause 2 (using x_{loc} for y_{loc})

This final generalization is relatively expensive in execution time. For example, one subsuming substitution may fail while another succeeds, e.g., Px ∨ a = b can be used to rewrite Pa ∨ Pb only by substituting a for x. However, it is easy to devise cheap algorithms that obtain part of the benefit. The current version of FIE requires R1 and R2 above to be identical. This is usually sufficient, because they are typically (e.g., in the case of branch conditions from conditional statements) single literals immune from substitution.

6.2. Subsumption

Next, FIE checks to see if *current* is subsumed by any clauses in *old*. In general, FIE deletes any clause subsumed by a known clause. Thus FIE should recognize C1 as subsuming C2 in just those cases where a person who knows C1 will consider C2 as redundant.

To test whether clause C1 subsumes clause C2, the inequality literals of C2 are first used to rewrite C1 as in equality canonicalization. This, in combination with equality canonicalization allows (P t1) to subsume any clause (regardless of equality ordering) of the form "if t1 = t2 then (P t2)".

The resulting clause C1' subsumes C2 if it has no more literals than C2 and some substitution maps each literal of C1' to a literal of C2 (a standard definition). In particular, a clause does not subsume its factors. Factors are often not obvious to humans, and thus constitute interesting results.

6.2.1. Reordering Arguments

When a relation is intuitively commutative, such as the Spouse relation, the commutative variants of known facts cease to be interesting. The user can declare a predicate to be intuitively symmetric (and other properties corresponding to permuting arguments), so that FIE will consider the variants to be "obvious" consequences of each other. The subsumption algorithm computes the variants of each literal and accepts a substitution for any of them. Perhaps other common properties would also be worth recognizing, but we have not had to deal with them yet.

6.2.2. Uniqueness Properties

We have described some additions to a large bag of previously known tricks for dealing with equality. The way FIE deals with commutativity is important for its application, but nothing new. In contrast, uniqueness does not seem to have been much studied. We feel we have made progress in building an understanding of uniqueness into FIE. In each case, the ability to discard a result that is too easily derived requires compensation (described later) the component that finds new consequences must be strengthened to avoid losing the consequences of what has been discarded.

When a relation is intuitively single valued, such as the Location relation, negative instances become uninteresting in the face of positive instances. If we know where an object is, there is no need to list all the other locations as places where it is not. The user can tell FIE that he understands certain uniqueness properties of a predicate. (In the case of symbolic execution, this information is already in the specification and the user need not restate it.)

The uniqueness properties are of the form: $\forall\ \underline{x},\underline{y},\underline{z},u,v\ (P\langle \underline{x}\underline{y}u\rangle \wedge P\langle \underline{x}\underline{z}v\rangle) \supset u = v$

where $\underline{x}$,$\underline{y}$ and $\underline{z}$ represent vectors of variables distinct from each other and from u and v ($\underline{y}$ and $\underline{z}$ of the same length), and $\langle \underline{x}\underline{y}u\rangle$ is some permutation of the variables in the concatenation of $\underline{x}$, $\underline{y}$ and (the single variable) u. A given uniqueness declaration must specify the predicate P, the permutation $\langle\rangle$ and the size of $\underline{x}$. The literal $P\langle \underline{a}\underline{b}c\rangle$ is considered to subsume the literal $\sim P\langle \underline{d}\underline{e}f\rangle$ (here we use c and f to stand for terms and $\underline{a}$, $\underline{b}$, $\underline{d}$ and $\underline{e}$ as vectors of terms) if there is a substitution θ which maps $\underline{a}$ to $\underline{d}$ and θ maps c to a term known not to be equal to f. (The current implementation just checks whether (= c f) simplifies to False. We have seen cases where this was inadequate.)

6.3. New Facts Simplify Old

If *current* (the clause being integrated) is still not simplified, it is used to simplify all the other clauses of *old*. Any clauses that are rewritten (where we regard subsumption as rewriting to True) are removed from *old* and the new version is put into the list of clauses to be added. Finally *current* is inserted into *old*.

7. Deriving New Consequences

In the final phase, *current* is combined with each clause of *old* to find new results. This is done with two rules, both closely related to binary resolution. Note that if clauses C1 and C2 are known to be true, and clause C3 is a resolvent of C1 and C2, then C3 must be true.

7.1. Resolution

The major difference between normal resolution and the first rule (which we will refer to simply as resolution) stems from our interest in equality. In normal resolution, it is impossible to resolve P(a) with ~P(b). In our case this is allowed, with the result of ~(= a b). Deriving inequalities may seem odd from a theorem proving point of view, but the results can be interesting to a person. They also serve as a communication mechanism in symbolic execution: the distinctness of two objects may be important at one time, but only derivable at another time.

In general, from the clauses P ∨ (R t_1 ... t_n) and Q ∨ ~(R u_1 ... u_n) (where t_i and u_i are terms and P and Q are clauses), we derive P ∨ Q ∨ ~(= t_1 u_1) ∨ ... ∨ ~(= t_n u_n). Notice that the substitution is stored in the inequality literals of the clause (which, of course, tend to simplify).

7.2. Uniqueness Resolution

The other rule uses the uniqueness information described above. Given that box1 is at N.Y. and box2 is at L.A., it directly derives that box1 and box2 are distinct. Also, given that Joe is at location1 and that Joe is at location2, it directly derives that location1 and location2 are identical. In fact, given a uniqueness declaration, an explicit axiom, such as ~(Loc x y) ∨ ~(Loc x z) ∨ (= y z) adds very little in terms of interesting consequences.

From the clauses P ∨ R<$\underline{x}$ y $\underline{u}$> and Q ∨ R<$\underline{w}$ z $\underline{v}$> uniqueness resolution derives
P ∨ Q ∨ ~(= $\underline{x}$ $\underline{w}$) ∨ (= u v),
where ~(= $\underline{x}$ $\underline{w}$) means ~(= $\underline{x}_1$ $\underline{w}_1$) ∨ ... ∨ ~(= $\underline{x}_m$ $\underline{w}_m$).

In the case of Location, $\underline{y}$ and $\underline{z}$ are empty, $\underline{x}$ and $\underline{w}$ are the terms representing objects and u and v are the terms representing locations.

`(Loc box1 N.Y.)` combines with

`(Loc box2 L.A.)` to yield

`~(= box1 box2) ∨ (= N.Y. L.A.)`

which simplifies (assuming L.A. and N.Y. are known to be distinct) to

`~(= box1 box2)`

A more impressive example:

`(Loc` x_{box} `N.Y.) ∨ (= box1` x_{box}`)` *every box other than box1 is at N.Y.*

`(Loc box2 L.A.)` yields by uniqueness resolution

`~(= box2` x_{box}`) ∨ (= N.Y L.A.) ∨ (= box1` x_{box}`)`

which simplifies in two steps to `(= box1 box2)`

The reordering properties of predicates (e.g., commutativity) are used in resolution (and uniqueness resolution) in the same way as in subsumption.

8. Filtering New Consequences

The two rules above can, of course, generate many new consequences. Some of these will be recognized as closely related to known facts, but in general this is not sufficient to prevent an explosion in the number of clauses. FIE adopts a very simple (and severe) strategy to ensure termination: it considers any resolvent that is more complex than either parent to be uninteresting. Higher complexity is defined as greater nesting of functions*** or more literals****

Different "versions" of FIE, corresponding to a tradeoff between power and selectivity may be obtained by varying some implementation parameters. The first of these is where to draw the boundary between "more literals" (uninteresting) and "fewer" (interesting). We have tried three solutions:

- the result must contain strictly fewer literals than one parent
- the result must contain fewer or the same number of literals as one parent
- the result must either contain strictly fewer literals, or the same number of literals but strictly more equality (inequality) literals than one parent.

The standard setting for symbolic execution has been the third. How much the results differ, and whether the difference is for better or worse depends on the problem.

The other, and perhaps more interesting parameter, is how much simplification is done before deciding whether a result is interesting. FIE's results would be fairly predicatable (and dull from a theorem proving point of view, though perhaps still interesting to the user) if the decision were made directly on the results of resolution. Often, however, complex results simplify enough to be considered "interesting". So far, we have only processed resolvents in isolation before deciding whether to keep them, but we have seen cases where interactions with other known clauses would have allowed resolvents to be kept. The most complete version (classified as future work) would be to keep "uninteresting" clauses for simplification, but not resolve them unless (until) they were simplified to the point of "interestingness".

It must be mentioned that FIE still can not guarantee a small number of consequences. Knowing of n boxes at different locations will generate n^2 inequalities, all of which FIE considers interesting. Actually, it is proper to consider as interesting the fact that these n boxes are all distinct. The "problem" is that predicate calculus cannot express that fact succinctly. One could imagine building a new representation for such a fact, extending the subsumption algorithm to recognize it, and building a special inference mechanism to use it. This would be a useful addition for some domains, but other forms of the same problem would remain, such as transitivity: given the axiom that a relation R is transitive, a set of axioms of the form (R a_i a_{i+1}) implies all results of the form (R a_i a_j) for i<j.

*** Notice that FIE depends on the inability of its simplifier to generate new terms. If we had as an axiom (P n) ⊃ (P n + 1) and the simplifier knew how to add 1 to a number (generating a term of "equal complexity") then (P 1) would have infinitely many "interesting consequences".

**** We mention in passing that the complexity cutoff can be programmed to some extent by altering the input clauses. For example, given a clause C containing the literal L, and another clause D which does not contain a literal that unifies with L, one can disjoin L to D to effectively raise the complexity limit without losing termination.

We have not tried to deal with the problems above (or related problems). This is partly due to the fact that in exploring sets of axioms (including symbolic execution, where individual examples are normally small), one rarely needs many instances: in order to explore the axioms of ordering, we would probably consider three or four objects, not twenty. However, other problems have arisen in practice. These have been solved without new representations or inference mechanisms.

8.1. Conditionality, again

In symbolic execution, a conditional statement (If P then S1 else S2) can be intuitively understood as two possible worlds. There is one set of theorems of the form ~P V C_i (for results of S1) and another set of the form P V C_j (for results of S2). Resolving on P produces a cross product of clauses. These always seem uninteresting - they intuitively amount to the case split: P V ~P.

The symbolic evaluator generates a new literal, L, meaning "the THEN branch was taken". (It needs a way to refer to that bit of history anyway - the expression P is insufficient since it refers to mutable domain relations.) FIE discards any result of resolving on L unless it has strictly fewer literals than one of its parents. This accepts consequences that can be derived independent of L, and if L can be proven True or False it allows the appropriate set of clauses to be deconditionalized (and the others to be subsumed). In another setting, the user could tell FIE which literals intuitively correspond to case splits.

8.2. Skolem functions

FIE filters out a large class of consequences that contain skolem functions. In essence, the skolem functions offer alternative representations for certain facts. We prefer to represent these facts only once in the original (and more natural) way.

As an example, consider the clause that every box has a location, (Loc x (fx)). This clause is useful, e.g., if we find a box with no location it would be nice to notice the contradiction. However, it interacts with ~(Loc box1 L.A.) to yield ~(= L.A. (f box1)), which a person would consider redundant. Given that locations are unique, (Loc box1 N.Y.) implies (= N.Y. (f box1)). As another example, given that every person has a Gender, (Gen x (gx)), and that spouses cannot share a gender, we can derive ~(Sp x y) V ~(Gen y (gx)), which is hard to explain in English in any terms other than the original axiom, that spouses cannot share a gender. FIE discards all of these (and other similar) results.

9. Related Work

FIE resembles Eurisko [Lenat 83] (and AM [Lenat 76]) in that it searches for interesting extensions to an initial knowledge base. However, Eurisko is meant to find deep results in many hours of exploration, whereas FIE is meant to quickly point out a few consequences that the user should probably know. Furthermore, Eurisko's results are empirically justified conjectures, whereas FIE's results are theorems.

Forward inference is quite rare in programs with general theorem proving capability, except on a special case basis. Programs like [Nevins 75] use forward inference rules whose form is carefully designed to generate certain types of results. On the other hand, [Bledsoe 78] provides various types of forward reasoning as user options, with no guarantee that they will lead to reasonable behavior - the responsibility belongs to the human user. [Cohen 81] is much closer in spirit to FIE in that it takes total responsibility for deciding which inferences to make. However, it uses forward reasoning purely to integrate new knowledge into its own database. It is not trying to interest an external user.

FIE has strong ties to the large body of work on resolution theorem proving [Loveland 78]. It uses clause representation, and resolution is its primary rule of inference [Robinson 65]. Also, we share with much of this work an emphasis on techniques for recognizing and deleting useless or redundant information, e.g., canonicalization and subsumption.

10. Conclusions

FIE *automatically generates interesting consequences from a set of input axioms.* One measure of success is that the symbolic evaluations we have tried have reported nearly all the results we expected, and some that were not expected.

FIE *works hard to avoid uninteresting consequences.* The notion of interestingness is heavily dependent on context. In particular, a fact is considered uninteresting if it is too easy to derive from other known facts.

FIE *finds mostly shallow consequences, but finds them quickly.* In the longest symbolic execution to date of a Gist specification, the average call to FIE integrated 10 new clauses with 23 old ones to yield 27 clauses in less than 10 CPU seconds on a VAX750 running Interlisp.

FIE has been used successfully in a symbolic evaluator. The specified domains have included a file system, a package router, a world of people (marriages, children, etc.), and a world of ships (cargoes, ports etc.). In the future we hope to adapt it to other purposes (e.g., debugging and explaining database schemas).

Acknowledgements: This work was done in the context of a larger effort by the Gist group at ISI. Also, the presentation of this paper was greatly improved by the suggestions of members of the group, especially Jack Mostow.

References

[Bledsoe 78] W. W. Bledsoe and Mabry Tyson, *The UT Interactive Prover*, University of Texas, Technical Report, 1978. Math. Dept. Memo ATP-17A

[Cohen 81] Donald Cohen, *Knowledge Based Theorem Proving and Learning*, UMI Research Press, 1981.

[Cohen 83] Donald Cohen, "Symbolic Execution of the Gist Specification Language," in *IJCAI*, 1983.

[Lenat 76] Douglas Lenat, *AM: An Artificial Intelligence Approach to Discovery in Mathematics as Heuristic Search*, Ph.D. thesis, Stanford University, July 1976.

[Lenat 83] Douglas B. Lenat, "EURISKO: A Program that Learns New Heuristics and Domain Concepts; The nature of Heuristics III: Program design and results," *Artificial Intelligence* 21, (1,2), March 1983, 61-98.

[Loveland 78] Donald W. Loveland, *Fundamental Series in Computer Science.* Volume 6: *Automated Theorem Proving: A Logical Basis*, North-Holland Publishing Company, 1978. This is cited only as one representative of a large body of work on resolution theorem proving.

[Nevins 75] Arthur J. Nevins, "Plane Geometry Theorem Proving Using Forward Chaining," *Artificial Intelligence* 6, (1), 1975, 1-23.

[Robinson 65] J. A. Robinson, "A Machine-Oriented Logic Based on the Resolution Principle," *JACM* 12, (1), Jan. 1965, 23-41.

[Swartout 83] Bill Swarout, "The GIST Behavior Explainer," in *NCAI*, 1983.