Using Decision Procedures to Accelerate Domain-Specific Deductive Synthesis Systems

Jeffrey Van Baalen
Steven Roach
M.S. 269-2
NASA Ames Research Center
Moffet Field, CA
{jvb, sroach}@ptolemy.arc.nasa.gov

Abstract. This paper describes a class of decision procedures that we have found useful for efficient, domain-specific deductive synthesis, and a method for integrating this type of procedure into a general-purpose refutation-based theorem prover. We suggest that this is a large and interesting class of procedures and show how to integrate these procedures to accelerate a general-purpose theorem prover doing deductive synthesis. While much existing research on decision procedures has been either in isolation or in the context of interfacing procedures to non-refutation-based theorem provers, this appears to be the first reported work on decision procedures in the context of refutation-based deductive synthesis where witnesses must be found.

1 Introduction

This paper describes a class of decision procedures that we have found useful for efficient, domain-specific deductive synthesis, and a method for integrating this type of procedure into a general-purpose refutation-based theorem prover. These procedures are called *closure-based ground literal satisfiability procedures.* We suggest that this is a large and interesting class of procedures and show how to integrate these procedures to accelerate a general-purpose theorem prover doing deductive synthesis. We also describe some results we have observed from our implementation.

Amphion/NAIF[17] is a domain-specific, high-assurance software synthesis system. It takes an abstract specification of a problem in solar system mechanics, such as "when will a signal sent from the Cassini spacecraft to Earth be blocked by the planet Saturn?", and automatically synthesizes a FORTRAN program to solve it. Amphion/NAIF uses deductive synthesis (a.k.a proofs-as-programs [6]) in which programs are synthesized as a byproduct of theorem proving. In this paradigm, problem specifications are of the form $\forall\vec{x}\exists\vec{y}[P(\vec{x}, \vec{y})]$, where $\vec{x}$ and $\vec{y}$ are vectors of variables. We are only interested in constructive proofs in which witnesses have been produced for each of the variables in $\vec{y}$. These witnesses are program fragments.

Deductive synthesis has two potential advantages over competing synthesis technologies. The first is the well-known but unrealized promise that developing a declarative domain theory is more cost-effective than developing a special-purpose synthesis engine. The second advantage is that since synthesized programs are correct relative to a domain theory, verification is confined to domain theories. Because declarative domain theories are simpler than programs, they are presumably easier to verify. This is of particular interest when synthesized code must be high-assurance.

The greatest disadvantage that general-purpose deductive synthesis systems have is that systems based on theorem proving[1] are almost always unacceptably inefficient unless the domain theory and theorem prover are carefully tuned. This tuning process consists of iteratively inspecting proof traces, reformulating the domain theory, and/or adjusting parameters of the theorem prover, a process that usually requires a large amount of time and expertise in automated reasoning.

In order to assist in constructing efficient synthesis systems, we are developing a tool, Meta-Amphion [10], the goal of which is: given a domain theory, automatically generate an efficient, specialized deductive synthesis system such as Amphion/NAIF. The key is a technique, that is under development, that generates efficient decision procedures for subtheories of the domain theory and then integrates them with a general-purpose refutation-based theorem prover. Some success has been achieved with this automated technique [14]. However, significantly more research is required to enhance the automated process of designing decision procedures to replace subsets of a theory.

However, we have found that deductive synthesis systems can be manually tuned very effectively by manually performing the process we are trying to automate. Hence, one can accelerate a deductive synthesis system by manually inspecting a domain theory and identifying subtheories for which we already have or can construct replacement decision procedures. This technique has, in our experience, been far easier and has often produced more efficient systems than the traditional technique of inspecting proof traces and tuning parameters. For instance, Figure 1 summarizes our experience with the Amphion/NAIF system. It is a graph of the performance of three different versions of Amphion/NAIF. It shows the number of inference steps required to find proofs as the input problem specification size (number of literals) grows for an un-optimized system, a traditionally hand-tuned system, and a system tuned by replacing subtheories with decision procedures (Tops). Figure 2 compares the traditionally hand-tuned system vs. the system tuned with decision procedures (Tops).

While much existing research on decision procedures has been either in isolation [11][16][5] or in the context of interfacing procedures to non-refutation-based theorem provers [13][2], we are unaware of any work done on decision procedures in the context of refutation-based deductive synthesis where witnesses must be found. This paper presents a decision procedure interface to a theorem prover with several inference rules including binary resolution and paramodulation. These inference rules have been extended to enable the class of ground literal satisfiability procedures to be integrated with the theorem prover in a straightforward and uniform manner. Procedures can be plugged in on a theory-by-theory basis, allowing the theorem prover to be tailored to particular theories. We show that when these procedures have the additional property of being *closure based*, they can be used to produce witnesses for deductive synthesis. The class of ground literal satisfiability procedures is such that the witnesses produced are

1. Amphion/NAIF utilizes a refutation-based theorem prover. We initially considered using Prolog; however, the domain theory required extensive use of equality, making the Prolog implementations available at the time inappropriate.

typically straight-line program fragments that are incorporated into larger programs produced by the general-purpose theorem prover.

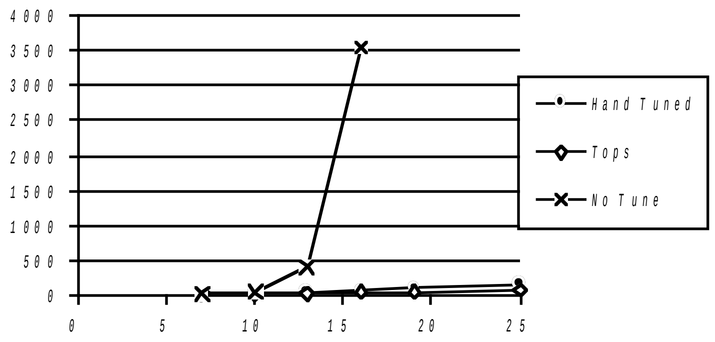

Figure 1: A comparison of the performance of three different versions of Amphion/NAIF.

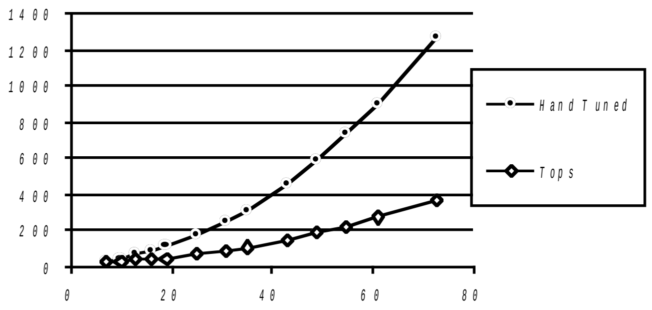

Figure 2: A closer comparison of the two tuned versions of Amphion/NAIF.

Section 2 introduces *separated clause notation*, the notation used by our extended inference rules. The motivation for these rules is that they enable decision procedures to be integrated with the theorem prover and used in place of subsets of a theory. The integration of procedures requires procedures that compute satisfiability and procedures that compute entailment. Section 3 describes procedures for testing satisfiability in a theory. Section 4 describes decision procedures that test entailment in a theory and that compute witnesses for deductive synthesis. The example used in Sections 2-4 is in the domain of LISP list structure and involves a decision procedure originally developed by Nelson and Oppen. This is used for simplicity of presentation. Then in Section 5, a decision procedure from the Amphion/NAIF domain is described. Section 6 describes the implementation of the interface and the results of using procedures like the one described in Section 5 for deductive synthesis in Amphion/NAIF. Finally, Section 7 discusses related work and concludes.

2 Separated Inference Rules

This section describes our extension to the inference rules in the SNARK [17] theorem prover enabling the use of decision procedures. The basic idea is that all clauses are separated into two parts: a part that is reasoned about by integrated decision procedures and a part that is reasoned about by SNARK. First, *separated clause* form is defined, and then the separated inference rules are described. SNARK has inference rules resolution, hyperresolution, paramodulation, and demodulation. The overall extension has been accomplished by extending each inference rule in a uniform manner. This paper only discusses separated binary resolution and separated paramodulation, but the other rules are extended similarly.

Separated binary resolution is similar to resolution with restricted quantifiers or *RQ-resolution* [3]. Recall that, given a first-order theory T and a first-order formula Φ, we prove $T \models \Phi$ by refutation by showing that $T \cup \{\neg\Phi\}$ is unsatisfiable. Assuming that T is satisfiable, this amounts to showing that no model of T is a model of $\neg\Phi$. The general idea of our binary resolution rule (as well as RQ-resolution) is as follows. If there is a method for determining satisfiability of a formula relative to a theory $T_2 \cup \{\neg\Phi\}$, we prove $T \models \Phi$ by showing that no model of T_1 can be extended to a model of $T_2 \cup \{\neg\Phi\}$, where $T_2 = T - T_1$. Our work is an extension of Burckert's in two ways. First, our resolution rule differs from RQ-resolution because it allows function symbols to appear in T_2 and $\neg\Phi$. In practice, this is extremely important because it drastically reduces the number of potential resolutions that must be considered. Second, we have extended the separation idea to inference rules other than resolution.

The separated rules work with clauses that are *separated relative* to a subtheory, called a *restriction theory*.

Definition 2.1 (Separated Clause) Let L be the language of a theory T, a first-order theory with equality. We treat equality as a logical symbol, so = is not in L. Let $L_1 \subseteq L$ be the language of $T_1 \subseteq T$. A clause C with the following properties is said to be *separated relative to* T_1:

1. C is arranged into $C_1 \vee C_2$, where both C_1 and C_2 are disjunctions of literals (i.e., clauses).
2. All the function and relation symbols in C_1 come from L_1 and all the function and relation symbols in C_2 come from L-L_1.

Notice that $C_1 \vee C_2$ can be written $\overline{C_1} \Rightarrow C_2$, where $\overline{C_1}$ is the negation of C_1. Since C_1 is a disjunction of literals, $\overline{C_1}$ is a conjunction of the negations of the literals in C_1. If $C = [\overline{C_1} \Rightarrow C_2]$ is a clause separated relative to some theory, $\overline{C_1}$ is called the *restriction* of C and C_2 is called the *matrix* of C. A set of clauses is separated relative to a theory if each of the clauses in the set is separated relative to the theory.

Constant (0-ary function) symbols are not discussed in the definition of separation because they may appear in C_1 or C_2 regardless of which language they are in. As explained below, this is a property of the mechanism for passing information between the matrix and the restriction of clauses as they are manipulated by our extended inference rules.

A clause is separated in two steps. In the first step, literals are placed in the restriction or matrix of a clause based on their predicate symbol. In the second step, each non-constant term t in the restriction whose head symbol is in L-L_1 is replaced by a new variable x, and $x \neq t$ is disjoined to the matrix. Similarly, each non-constant term in the matrix whose head symbol is in L_1 is replaced by a new variable x, and $x=t$ is conjoined to the restriction.

Example 2.1 Suppose we have a theory T_1 of LISP list structure whose non-logical symbols are the function symbols *head*, *tail*, *cons*, and *nil*.. Then the separation of the formula $tail(K) \neq nil$ relative to T_1 is $(x = tail(K)) \Rightarrow (x \neq nil)$.

Separated binary resolution computes a resolvant of two clauses, *C'* and *C''*, each separated relative to a theory T_1. This resolvant is also a clause separated relative to T_1. Informally, a resolvant is computed as follows. First, ordinary resolution is performed on the matrices (right hand sides) of *C'* and *C''* to form the matrix of the resolvant. The resulting substitution σ is used in forming the restriction of the resolvant which is the conjunction of the restrictions of *C'* and *C''* with the substitution σ applied. If the new restriction is unsatisfiable in T_1, the resolvant is *true* and, as a practical matter for resolution refutation, can be discarded.

Definition 2.2 (Separated Binary Resolution) Let *C'* and *C''* be variable disjoint clauses separated relative to a theory T_1. Let $C' = \alpha_1 \wedge \ldots \wedge \alpha_n \Rightarrow l_1 \vee Q$ and $C'' = \beta_1 \wedge \ldots \wedge \beta_p \Rightarrow l_2 \vee R$, where Q and R are (possibly empty) clauses and $n,p \geq 0$. If l_1 and $\overline{l_2}$ unify with most general unifier σ and $\exists[(\alpha_1 \wedge \ldots \wedge \alpha_n \wedge \beta_1 \wedge \ldots \wedge \beta_p)\sigma]$ is satisfiable in T_1, the *separated resolvant* of *C'* and *C''* is the separation[2] of $(\alpha_1 \wedge \ldots \wedge \alpha_n \wedge \beta_1 \wedge \ldots \wedge \beta_p)\sigma \Rightarrow (Q \vee R)\sigma$.

Example 2.2 The third clause below is a separated resolvant of the first two clauses.

$$(x = cons(u, v)) \wedge (K = cons(u, z)) \wedge (w = cons(z_1, nil)) \Rightarrow (z \neq append(v, w))$$

$$(x_1 = tail(K)) \wedge (y_1 = \mathrm{cons}(u_1, nil)) \Rightarrow ((x_1 = append(front(x_1), y_1)) \vee (u \neq \mathrm{last}(x_1)))$$

2. The separation of the resolvant does not have to be a separate step. However, it simplifies the presentation.

$$\left(\begin{array}{l}(x = cons(u, v)) \wedge (K = cons(u, z)) \wedge (w = cons(z_1, nil)) \\ \wedge ((z = tail(K)) \wedge (w = cons(u_1, nil)))\end{array}\right) \Rightarrow (u \neq last(z)) \vee (v \neq front(z))$$

The literal $z \neq append(v,w)$ (in the first clause) is unified with $x_1 = append(front(x_1),y_1)$ (in the second clause) producing the unifier $\{x_1 \leftarrow z, v \leftarrow front(z), w \leftarrow y_1\}$. The matrix of the resolvant is obtained as in ordinary binary resolution and the restriction is the result of applying the unifier above to the conjunction of the restrictions of the two parents. The resulting clause is separated, moving *front(z)* to the matrix.

Lemma 2.1 (Soundness of separated binary resolution) Let Ψ be a set of separated clauses, and let ψ be a clause derived from two elements of Ψ by separated binary resolution. If M is a model of Ψ, M is a model of $\Psi \cup \{\psi\}$.

Proof: Soundness follows immediately from the soundness of ordinary binary resolution. The satisfiability check on the restriction of the resolvant is not necessary for soundness of the rule overall. Rather, if the restriction of the resolvant is unsatisfiable, the separated clause is a tautology. []

Definition 2.3 (Separated Paramodulation) Let $l[t]$ be a literal with at least one occurrence of the term t. Let C' and C'' be variable disjoint clauses separated relative to a theory T_1. Let $C' = \alpha_1 \wedge \ldots \wedge \alpha_n \Rightarrow l[t] \vee Q$ and $C'' = \beta_1 \wedge \ldots \wedge \beta_p \Rightarrow (r = s) \vee R$, where Q and R are (possibly empty) clauses and $n,p \geq 0$. If t and r unify with most general unifier σ and $\exists[(\alpha_1 \wedge \ldots \wedge \alpha_n \wedge \beta_1 \wedge \ldots \wedge \beta_p)\sigma]$ is satisfiable in T_1, a *separated paramodulant* of C' and C'' is the separation of

$$(\alpha_1 \wedge \ldots \wedge \alpha_n \wedge \beta_1 \wedge \ldots \wedge \beta_p)\sigma \Rightarrow l\sigma[s\sigma] \vee Q\sigma \vee R\sigma$$

where $l\sigma[s\sigma]$ represents the result obtained by replacing a single occurrence of $t\sigma$ in $l\sigma$ by $s\sigma$.

As with resolution, soundness of separated paramodulation follows from the soundness of the ordinary paramodulation rule.

An ordinary resolution refutation of a set of clauses C consists of a sequence of clauses where each clause is an element of C or is derived from two preceding clauses in the sequence by binary resolution or paramodulation. An ordinary refutation is *closed* when the empty clause, which we denote [], is derived. A *separated refutation* is a sequence of separated clauses derived using the separated rules. Unlike an ordinary refutation, a separated refutation is not necessarily closed when a clause with an empty matrix is derived. Instead, in general, there is a set of clauses $\{C_1 \Rightarrow [], \ldots, C_n \Rightarrow []\}$ each of which has a separated refutation such that $T \models [\exists C_1 \vee \ldots \vee \exists C_n]$, where $\exists F$ is the existential closure of F. A proof of this fact can be found in [Burckert91] where it is also shown that this disjunction is finite so long as T_1 is first-order (this is a consequence of Compactness). Hence, a *closed separated refutation* is a separated refutation that ends with a collection of separated clauses all of whose matrices are empty such that the existential closure of the disjunction of their restrictions is a theorem of T_1.

Lemma 2.2 (Soundness of separated refutation) If the separation of a set of clauses *C* has a closed separated refutation, *C* is unsatisfiable.

Proof: This result follows immediately from soundness of separated binary resolution and separated paramodulation, and the fact that if a set of separated clauses is unsatisfiable, so is the unseparated clause set. []

An inference system with ordinary binary resolution and ordinary paramodulation is complete if reflexivity axioms are included. In order for a system including separated versions of these rules to be complete, separated versions of congruence axioms for some theory predicate and function symbols are added. An example of a predicate congruence axiom is $(x = y) \Rightarrow (P(x) \Leftrightarrow P(y))$. Space does not permit a proof of the completeness of separated resolution and paramodulation here.

[3] points out that for some restriction theories, closed separated refutations can always be obtained by considering the entailment of the restrictions of only individual clauses. For instance, it is proven that if T_1 is a definite theory, i.e., a theory that can be written as a set of definite clauses, closed separated refutations are guaranteed for query clauses whose restrictions contain only positive literals. This paper focuses on the case where entailment of only single restrictions needs to be checked. When this is not the case, getting a closed separated refutation requires an additional inference rule (such as consensus [7]) or it requires decision procedures to be used in a more complicated manner than presented here. Thus far, the simpler case has been sufficient in our work on deductive synthesis.

The definition of a separated clause often prevents the derivation of clauses with empty matrices when terms that are not in the restriction language appear. These terms keep getting separated back into the matrix. For instance in example 2.2 above because *front* is in $L\text{-}L_1$, instead of substituting *front(z)* into the restriction of the resolvant, $v \neq front(z)$ is disjoined in the matrix. In such cases, the matrix of a clause will end up containing only literals of the form $t \neq x$ for some variable *x* and some term *t* in the language $L\text{-}L_1$ not containing *x*, Such a clause can be viewed as having an empty matrix with the disequalities considered as substitutions for variables in the restriction. Our system completes refutations by applying these substitutions to the restriction (rendering the clause no longer separated) and then checking the entailment of the resultant restriction with symbols in $L\text{-}L_1$ uninterpreted. This can be seen in the last step of the following example.

Example 2.3. Let T_{LISP} be a theory of LISP list structure with function symbols *head*, *tail*, *cons*, and *nil*. Given the theory *T*:

$$(x = x) \wedge (K \neq nil) \wedge (tail(K) \neq nil)$$

$$(K \neq nil) \Rightarrow ((tail(K) \neq nil) \Rightarrow tail(K) = append(front(tail(K)), cons(last(tail(K), nil))))$$

$$append(cons(u, v), w) = cons(u, append(v, w))$$

$$(x \neq nil) \Rightarrow (x = cons(head(x), tail(x)))$$

$$(head(cons(x, y)) = x) \wedge (tail(cons(x, y)) = y) \wedge (cons(x, y) \neq nil)$$

we want to show that $\exists y_1, z_1[K = append(y_1, cons(z_1, nil))]$ is a theorem of *T*. In this theory, the functions *front* (which “computes” all but the last of a list) and *last* (which “computes” the last element of a list) are constrained in terms of the functions *append, cons,* and *tail.* Witnesses found in proving the theorem above can be viewed as synthesized definitions of the functions *front* (a witness for y_1) and *last* (a witness for z_1) under the assumption for an input list *K*, that $K \neq nil$ and $tail(K) \neq nil$. Note that we make these assumptions here so as to focus on the witnesses found by a decision procedure. When these assumptions are relaxed, the fragments generated by the decision procedure are incorporated into recursive definitions of *front* and *last*.

A refutation that is separated relative to T_{LISP} is given below. Clauses 1-5 below are the first three axioms above separated relative to T_{LISP}. Note that these are the clauses of T-T_{LISP}. The last two formulas above are axioms of T_{LISP} and are needed only in determining satisfiability and entailment of restrictions. Therefore, they do not appear in the proof. We give as justification for step 10 “substitute.” The clause of this step is obtained from step 9 by treating the disequalities in the matrix as a substitution that is applied to the restriction. Since the existential closure of the restriction of 10 is a theorem of T_{LISP} (regardless of the interpretation of *front* and *last*), the proof is finished. As discussed in Section 4, the witnesses for *front* and *last* are obtained from the consequences of the restriction of step 10.

1	Given	$\Rightarrow K \neq nil$
2	Given	$(x = tail(K)) \Rightarrow x \neq nil$
3	Given	$(x = tail(K)) \wedge (y = cons(z, nil)) \wedge (w = tail(K)) \Rightarrow \begin{pmatrix} (x = nil) \vee (K = nil) \vee (z \neq last(w)) \vee \\ (x = append(front(x), y)) \end{pmatrix}$
4	Given	$(x = cons(u, v)) \wedge (y = cons(u, z)) \Rightarrow ((append(x, w) = y) \vee (z \neq append(v, w)))$
5	Negated conclusion	$(x_1 = cons(z_1, nil)) \Rightarrow (K \neq append(y_1, x_1))$
6	resolve 4 and 5 $\{y_1 \leftarrow x, x_1 \leftarrow w, y \leftarrow K\}$	$(x = cons(u, v)) \wedge (K = cons(u, z)) \wedge (w = cons(z_1, nil)) \Rightarrow (z \neq append(v, w))$
7	resolve 1 and 3	$(x = tail(K)) \wedge (y = cons(u, nil)) \Rightarrow (x = nil) \vee (x = append(front(x), y)) \vee (u \neq last(x))$
8	resolve 2 and 7	$(x_1 = tail(K)) \wedge (y_1 = cons(u_1, nil)) \Rightarrow (x_1 = append(front(x_1), y_1)) \vee (u \neq last(x_1))$
9	resolve 8 and 6 $\begin{Bmatrix} x_1 \leftarrow z, v \leftarrow front(z), \\ w \leftarrow y_1 \end{Bmatrix}$	$(x = cons(u, v)) \wedge (K = cons(u, z)) \wedge (w = cons(z_1, nil))$ $(z = tail(K)) \wedge (w = cons(u_1, nil)) \Rightarrow (u_1 = last(z))(v \neq front(z))$

10	substitute	$(x = cons(u, front(z))) \wedge (K = cons(u, z)) \wedge$ $(w = cons(z_1, nil)) \wedge (z = tail(K)) \wedge$ $(w = cons(last(z), nil))$ $\Rightarrow [\,]$

3 Procedures for Testing Satisfiability

The separated inference rules presented in the previous section depend on procedures which perform two actions. These are (1) a test for satisfiability of restrictions; and (2) a test for entailment of the restriction of any formula which has an empty matrix. This section describes procedures that test satisfiability of restrictions. We identify a class of procedures that can be used for this purpose.

When we have a procedure for deciding satisfiability of a conjunction of literals in a theory $T_1 \subseteq T$, we use separated inference rules to prove a theorem Φ in a theory T. The clauses of T-T_1 and $\neg\Phi$ are separated relative to T_1, and the procedure is used at each inference step to test the satisfiability of derived restrictions.

The restrictions are conjunctions of literals possibly containing variables. Even though these restrictions may have variables, it is possible to use *ground literal satisfiability procedures (GLSPs)* to determine satisfiability of the conjunctions in restrictions. We do this by replacing the variables in the restrictions by new constant symbols. The fact that we can use GLSPs to determine satisfiability here is established by Theorem 3.1.

Definition 3.1 (Ground Literal Satisfiability Procedure) A *ground literal satisfiability procedure for a theory T* is a procedure that decides whether or not a conjunction of ground literals *F* is satisfiable in *T*. The language of *F* must be the language of *T* but may be extended by a collection of uninterpreted function symbols (including constants).

Theorem 3.1 (Applicability of GLSPs) If *P* is a GLSP for a theory T_1, *P* can be used to decide the satisfiability in T_1 of the restriction of any clause separated relative to T_1.

Proof: Let $C = [\overline{C_1} \Rightarrow C_2]$ be a clause separated relative to T_1. Let $x_1,...,x_n$ be the variables in $\overline{C_1}$. Let σ be the substitution $\{x_1 \leftarrow c_1, \ldots, x_n \leftarrow c_n\}$, where the c_i are new uninterpreted constant symbols. Replace the restriction of C with $(x_1 = c_1) \wedge \ldots \wedge (x_n = c_n) \wedge \overline{C_1}\sigma$.

We show that (a) $(x_1 = c_1) \wedge \ldots \wedge (x_n = c_n) \wedge \overline{C_1}\sigma$ is satisfiable just in case $\overline{C_1}$ is and (b) the satisfiability of C implies the satisfiability of $((x_1 = c_1) \wedge \ldots \wedge (x_n = c_n) \wedge \overline{C_1}\sigma) \Rightarrow C_2$. Note that a conjunction of equalities constructed in this fashion is always satisfiable. Hence, we can replace any separated clause C with the clause $((x_1 = c_1) \wedge \ldots \wedge (x_n = c_n) \wedge \overline{C_1}\sigma) \Rightarrow C_2$ and decide the satisfiability of the restriction of such a clause by deciding the satisfiability of the ground conjunction $\overline{C_1}\sigma$.

(a) Since $\overline{C_1}$ is the generalization of a subconjunction of $(x_1 = c_1) \wedge \ldots \wedge (x_n = c_n) \wedge \overline{C_1}\sigma$, if $(x_1 = c_1) \wedge \ldots \wedge (x_n = c_n) \wedge \overline{C_1}\sigma$ is satisfiable, $\overline{C_1}$

is satisfiable. Now suppose $\overline{C_1}$ is satisfiable in a model M under the variable assignment I. Extend M to M^* which interprets each c_i the same as I assigns x_i. Then $<M^*, I> \models (x_1 = c_1) \wedge \ldots \wedge (x_n = c_n) \wedge \overline{C_1}\sigma$.

(b) If C is satisfiable, either C_2 is satisfiable or $\overline{C_1}$ is unsatisfiable. If C_2 is satisfiable, then $((x_1 = c_1) \wedge \ldots \wedge (x_n = c_n) \wedge \overline{C_1}\sigma) \Rightarrow C_2$ is satisfiable. Otherwise, since $\overline{C_1}$ is satisfiable just in case $(x_1 = c_1) \wedge \ldots \wedge (x_n = c_n) \wedge \overline{C_1}\sigma$ is, $(x_1 = c_1) \wedge \ldots \wedge (x_n = c_n) \wedge \overline{C_1}\sigma$ is also unsatisfiable. []

The fact that any GLSP can be interfaced to the separated inference rules is a fortunate situation because there appear to be a large number of useful GLSPs. The argument supporting this claim has three parts. First, a significant number of GLSPs have been identified and published [11][12][5]. Second, other work reports on techniques for extending some GLSPs that have been identified. Third, there are techniques that enable GLSPs to be combined.

Nelson & Oppen in [12] show how to extend a GLSP for the theory of equality with uninterpreted function symbols to the theory T_{LISP}. Their procedure can be integrated with our theorem prover and used to check satisfiability of restrictions in the running example, e.g., the restriction of the clause derived in step 9 of Example 2.1

$$(x = cons(u, v)) \wedge (K = cons(u, z)) \wedge (w = cons(z_1, nil)) \wedge (z = tail(K)) \wedge (w = cons(u_1, nil))$$

can be checked for satisfiability in the theory of LISP list structure using Nelson & Oppen's procedure considering all the variables to be constants.

We have used techniques similar to Nelson & Oppen to construct several new procedures by extending a GLSP for congruence closure (one such procedure is described in Section 5). Also, [8] gives techniques for constructing GLSPs based on congruence closure for conjunctions of ground literals containing predicates. The essential idea is to introduce boolean constants *True* and *False* and to represent $P(t_1, \ldots, t_n)$ as $P(t_1, \ldots, t_n) = True$ and $\neg P(t_1, \ldots, t_n)$ as $\neg P(t_1, \ldots, t_n) = False$. Then, if the congruence closure graph of a conjunction F contains *True*=*False*, F is unsatisfiable.

Finally, both [11] and[16] describe techniques for combining GLSPs with disjoint languages into a GLSP for the union of these languages. Much work has been done recently on the closely related topic of combining decision procedures for equational theories [1].

Hence, we are in the convenient situation of being able to combine GLSPs to create a GLSP for a restriction theory. Given a theory T, we can design from components a decision procedure for a restriction theory. (See [10] or [14] for examples of techniques for automatically designing decision procedures from components.)

4 Deductive Synthesis Decision Procedures

This section shows that if a GLSP has the additional property of being *closure-based* it can be used not only to check satisfiability but also to check for entailment and to produce witnesses for deductive synthesis. All of the procedures mentioned in Section 3 as well as all of the procedures we have used in our work on deductive synthesis are closure based.

As discussed in Section 2, producing a closed separated refutation requires decision procedures for computing both satisfiability and entailment of restrictions. For the entailment problem, we need decision procedures that check the entailment of clauses containing existentially quantified variables and possibly also to produce witnesses for those variables. We call such procedures *literal entailment procedures*.

Definition 4.1 A *literal entailment procedure (LEP)* for a theory T is a procedure that decides for a conjunction of literals F in the language of T (possibly containing free variables) whether or not $T \models \exists F$.

While in general the satisfiability procedure and the entailment procedure for a restriction theory are separate procedures, we have found that closure-based GLSPs can also be used as LEPs.

Definition 4.2 (Closure-based satisfiability procedure) A *closure-based* satisfiability procedure for a theory T computes satisfiability in T of a conjunction of formulas Φ by constructing a finite set Ψ of ground consequences of $T \cup \{\Phi\}$ such that Ψ contains a ground literal and its negation just in case Φ is unsatisfiable in T.

The congruence closure procedure is a closure-based satisfiability procedure for the theory of equality with uninterpreted function symbols. It constructs a congruence closure graph [8] and in so doing computes a finite set of ground consequences of a conjunction of input ground equalities. As new equalities are added to a conjunction, new nodes representing terms are added to the graph and/or congruence classes are merged. Many GLSPs that extend congruence closure are also closure-based satisfiability procedures.

A closure-based GLSP with theory T can be used as a LEP as follows. Given a conjunction of literals $F(x_1,...,x_n)$, where the x_i are the existentially quantified variables in F, the procedure is used to check the satisfiability of $T \cup \{F(c_1,...,c_n)\}$, where the c_i are new constant symbols substituted for the corresponding variables. Since the procedure is closure based, in computing satisfiability, it computes consequences of $T \cup \{F(c_1,...,c_n)\}$. If consequences of the form $c_i=t_i$ are computed for all $i=1,...,n$, the procedure is run again on $\neg F(t_1,...,t_n)$. If this clause is unsatisfiable in T, witnesses have been identified for the original variables x_i. The idea of the first call to the procedure is to determine if in every model of $T \cup \{F(c_1,...,c_n)\}$, $c_i=t_i$, $i=1,...,n$. The idea of the second call is to determine if $F(t_1,...,t_n)$ is true in every model of T, i.e., $T \models \exists F$.

We illustrate how a closure-based satisfiability procedure is used as a LEP with Nelson & Oppen's GLSP for T_{LISP}.

Example 4.1 In step 10 of example 2.3, it must be shown that the existential closure of

$$F(x, u, z, w, z_1) = \begin{bmatrix} (x = cons(u, front(z))) \wedge (K = cons(u, z)) \wedge (w = cons(z_1, nil)) \wedge \\ (z = cons(u, front(z))) \wedge (w = cons(last(z), nil)) \end{bmatrix}$$

is a theorem of T_{LISP}. First, the Nelson & Oppen GLSP is used to check the satisfiability of this conjunction (assuming that the variables are uninterpreted constants). In doing so, the procedure computes the following additional equalities. From K=*cons*(u,z), we get u=*head*(K) and z=*tail*(K). Hence, x=*cons*(*head*(K),*front*(*tail*(K)). From w=*cons*(z_1,*nil*) and w=*cons*(*last*(z),*nil*), we get

$$head(cons(z_1,nil))=head(cons(last(z),nil)).$$

Hence, z_1=*last*(z) and z_1=*last*(*tail*(K)).

What the procedure has shown is that if we treat the variables in the above formula as constants, in every model of T that is also a model of this formula, x=*cons*(*head*(K),*front*(*tail*(K))) and z_1=*last*(*tail*(K)). That is, that $F(c_x, c_u, c_z, c_w, c_{z_1})$ is satisfiable in the theory T_{LISP} and that c_x=*cons*(*head*(K),*front*(*tail*(K))) and $c_{z_1} = last(tail(K))$. Next, to establish that *cons*(*head*(K),*front*(*tail*(K)), is a witness for x and that *last*(*tail*(K)) is a witness for z_1, the procedure is used to check the unsatisfiability of $\neg F(t_1, \ldots, t_n)$. Since this is a disjunction that is unsatisfiable just in case all of its disjuncts are, each literal of $\neg F(t_1, \ldots, t_n)$ can be checked separately. If $\neg F(t_1, \ldots, t_n)$ is unsatisfiable, $T \models F(t_1, \ldots, t_n)$ and $T \models \exists F$. We have exploited the following fact in this analysis.

Lemma 4.1 Let $F(c_1,...,c_n)$ be a conjunction of ground literals that is satisfiable in a theory T. Further, suppose that the constant symbols $c_1,...,c_n$ do not occur in T. If $T \cup F(c_1, \ldots, c_n) \models ((c_1 = t_1) \wedge \ldots \wedge (c_n = t_n))$ where each t_i is a term not containing any of the c_js, $T \models \forall x_1, \ldots, x_n(F(x_1, \ldots, x_n) \Rightarrow ((x_1 = t_1) \wedge \ldots \wedge (x_n = t_n)))$.

Proof: Suppose $T \cup F(c_1, \ldots, c_n) \models ((c_1 = t_1) \wedge \ldots \wedge (c_n = t_n))$. Then, by the deduction theorem, $T \models (F(c_1, \ldots, c_n) \Rightarrow ((c_1 = t_1) \wedge \ldots \wedge (c_n = t_n)))$, Also, since the c_i do not appear in T, the first-order law of universal generalization gives us $T \models \forall x_1, \ldots, x_n(F(x_1, \ldots, x_n) \Rightarrow ((x_1 = t_1) \wedge \ldots \wedge (x_n = t_n)))$.[]

Lemma 4.1 gives us license to use a GLSP to find potential witnesses for existentially quantified variables, i.e., terms that make F true in every model of $T \cup \{F\}$. The GLSP is then used to check that these potential witnesses are, in fact, witnesses, i.e., that they make F true in every model of T.

We have used the separated refutation system in the context of deductive synthesis where we are only interested in constructive proofs in which witnesses have been produced for existentially quantified variables in a theorem. In this context, decision procedures may be required to produce witnesses. Closure-based GLSP have an added benefit in deductive synthesis, namely that such a GLSP establishes that the existential closure of a restriction is a theorem by constructing witnesses. These witnesses can be extracted to produce programs in deductive synthesis. For example, in proving the theorem $\exists y_1, z_1[K = append(y_1, cons(z_1, nil))]$ in example 2.3, the Nelson & Oppen GLSP produces witnesses for y_1 and z_1. These are *cons*(*head*(K),*front*(*tail*(K)) and *last*(*tail*(K)) respectively, which are the synthesized programs for *front*(K) and *last*(K) under the assumption that $K \neq nil$ and *tail*(K)$\neq nil$.

Thus far in our deductive synthesis work, all the GLSPs we have developed can be used to generate witnesses in this manner.

5 Amphion/NAIF Decision Procedures

This section discusses the implementation of procedures for Amphion/NAIF. Amphion/NAIF is a deductive-synthesis system that facilitates the construction of programs that solve problems in solar system mechanics. Programs generated by Amphion/NAIF consist of straight-line code using assignment statements and calls to elements of the SPICE subroutine library. The SPICE library was constructed to solve problems related to observation planning and interpretation of data received by deep space probes.

An Amphion domain theory has three parts: an abstract theory whose language is suitable for problem specifications, a concrete theory that includes the specification of the target components, and an implementation relation between the abstract and concrete theories. Specifications are given in the abstract language, and programs are generated in the concrete language. Abstract objects are free from implementation details. For example, a point is an abstract concept, while a FORTRAN array of three real numbers is a concrete, implementation level construct.

At the abstract level, the Amphion/NAIF domain theory includes types for objects in Euclidean geometry such as points, rays, planes, and ellipsoids, augmented with astronomical constructs such as planets, spacecraft, and time. The abstract functions and relations include geometric constraints such as whether one geometric object intersects another. The concrete portion of the Amphion/NAIF domain theory defines types used in implementing a program or in defining representations, and it defines the subroutines and functions that are elements of the target subroutine library.

The implementation relations are axiomatized through abstraction maps using a method described by Hoare [9].These are maps from concrete types to abstract types. The function *abs* is used to apply an abstraction map to a concrete object. For example, *abs*(*coordinates-to-time*(*TS*), *tc*) denotes the application of the abstraction map *coordinates-to-time*, parameterized on the time system *TS*, to the time coordinate *tc*, i.e., this term maps a concrete time coordinate *tc* in time system *TS* to an abstract time.

In the NAIF theory, many implementation relations are axiomatized as equalities. For example, an abstract time may be encoded by any of several data representations such as Julian date or Calendar date. An example of an equality axiom relating two concrete objects to a single abstract object is

$$\forall ts_1, ts_2, tc\left(\begin{array}{l} abs(\mathit{coordinates\text{-}to\text{-}time}(ts_1), tc) \ = \\ abs(\mathit{coordinates\text{-}to\text{-}time}(ts_2), \mathit{convert\text{-}time}(ts_1, ts_2, tc)) \end{array}\right)$$

This axiom says that two abstract times are equivalent. The first abstract time is derived from time coordinate *tc* in time system ts_1. The second abstract time is the time conversion function *convert-time* applied to the first time coordinate to convert it from one system to another. This axiom is used to introduce invocations of the *convert-time* function into a synthesized program. These synthesized code fragments convert time data from one representation to another.

The terms in a specification usually do not match perfectly with axioms in the domain theory. Inference steps are required to generate matching terms. When these inference steps include resolution or paramodulation, choice points are introduced into the theorem prover's search space. For example, a specification may state that an input time is in the Julian time system, but it may require that the time be in the Ephemeris time system for the appropriate inferences to carry through. The theorem prover will apply the *convert-time* axiom (using paramodulation) to convert the input time coordinate from Julian to Ephemeris, then apply the appropriate inferences. Each paramodulation represents a multi-branched choice point in the search space. As a practical matter, this branching usually causes a combinatorial explosion in theorem proving. One class of decision procedure interfaced to the theorem prover in Amphion/NAIF performs representation conversions, like the time conversion just described, without search. In Amphion/NAIF, inserting decision procedures to eliminate choicepoints has a dramatic speedup effect.

5.1 A Procedure for Time Conversion

The procedure for time conversion is the simplest example of a representation conversion decision procedure used in Amphion/NAIF. When this procedure is interfaced to the theorem prover, the domain theory is separated relative to the NAIF subtheory of time. For example, an axiom such as

$$\forall t_1, t_2 \begin{pmatrix} \mathit{sum\text{-}of\text{-}times}(abs(\mathit{coordinates\text{-}to\text{-}time}(\mathit{EPHEMERIS}), t_1), \\ \qquad abs(\mathit{coordinates\text{-}to\text{-}time}(\mathit{EPHEMERIS}), t_2)) = \\ abs(\mathit{coordinates\text{-}to\text{-}time}(\mathit{EPHEMERIS}), \mathit{sum\text{-}time\text{-}coords}(t_1, t_2)) \end{pmatrix}$$

is separated, yielding

$$\begin{array}{l} (X_1 = abs(\mathit{coordinates\text{-}to\text{-}time}(\mathit{EPHEMERIS}), U_2)) \wedge \\ (X_2 = abs(\mathit{coordinates\text{-}to\text{-}time}(\mathit{EPHEMERIS}), U_3)) \wedge \\ (X_3 = abs(\mathit{coordinates\text{-}to\text{-}time}(\mathit{EPHEMERIS}), \mathit{sum\text{-}time\text{-}coords}(U_2, U_3))) \\ \qquad \Rightarrow (\mathit{sum\text{-}of\text{-}times}(X_1, X_2) \neq X_3) \end{array}$$

The decision procedure implements congruence closure to compute the consequences of a set of equalities some of which are between abstract time terms, i.e., it is used on the restriction of clauses like the one above. The congruence closure algorithm computes the consequences of a conjunction of ground equalities by maintaining equivalence classes of terms in a congruence closure graph [8]. As described in Sections 3 and 4, for this purpose, the variables in the restriction are treated as constants. These constants are specially marked to distinguish them from constants appearing either in the domain theory or an input specification. The equivalence classes of two terms are merged either when an equality between those terms is introduced or when two terms are found, by congruence, to be equal, i.e., if $t_i = s_i$, $i=1,..,n$, then $f(t_1, \ldots, t_n) = f(s_1, \ldots, s_n)$. Hence, the equalities involving *abs* terms appearing in the restriction of a clause and the consequences of those equalities are represented in a congruence closure graph, rather than as literals of a clause.

The time conversion procedure is, in fact, an extension of congruence closure because it also finds witnesses for some of the specially marked constants (those that were substituted for variables) in *abs* terms. When a term in an equivalence class of *abs* terms is ground, witnesses are produced for variables in the other terms in the class when those other terms contain (non variable) time systems. There are two cases. Case 1 is when the time systems in the two *abs* terms are the same. In this case witnesses are generated based on the domain theory axiom

$$\forall ts, tc_1, tc_2(abs(\textit{coordinates-to-time}(ts), tc_1) = abs(\textit{coordinates-to-time}(ts), tc_2)) \\ \Rightarrow (tc_1 = tc_2)$$

Case 2 is when the time systems are different. These witnesses are produced based on the domain theory axiom

$$\forall ts_1, ts_2, tc(abs(\textit{coordinates-to-time}(ts_1), tc) \\ = abs(\textit{coordinates-to-time}(ts_2), \textit{convert-time}(ts_1, ts_2, tc)))$$

For example, if there is an equivalence class containing the term *abs*(*coordinates-to-time*(*EPHEMERIS*),*A*) and containing the term *abs*(*coordinates-to-time*(*JULIAN*),*tc*), the procedure produces the witness *convert-time*(*EPHEMERIS*,*JULIAN*,*A*) for *tc*.

When the *convert-time* procedure is used in Amphion/NAIF, the axioms above are no longer needed. Hence, they are removed from the domain theory.

5.2 A Proof Using the Time Conversion Procedure

The axiom shown below indicates how a sum of two times is computed. In this axiom, the final time is computed as a sum only if both times are in Ephemeris representation

$$\forall t_1, t_2\begin{pmatrix} \textit{sum-of-times}(abs(\textit{coordinates-to-time}(EPHEMERIS), t_1), \\ \qquad abs(\textit{coordinates-to-time}(EPHEMERIS), t_2)) = \\ abs(\textit{coordinates-to-time}(EPHEMERIS), \textit{sum-time-coords}(t_1, t_2)) \end{pmatrix}$$

The operation of the *convert-time* procedure can be demonstrated by supposing SNARK is given the following specification.

$$\forall jt, et, \exists ut\begin{pmatrix} \textit{sum-of-times}(abs(\textit{coordinates-to-time}(JULIAN), jt), \\ \qquad abs(\textit{coordinates-to-time}(EPHEMERIS), et)) = \\ abs(\textit{coordinates-to-time}(UTC), ut) \end{pmatrix}$$

This specification asks: when given two distances, one represented in JULIAN format and one in EPHEMERIS format, find the sum of those distances in UTC format. The first step in the proof of this specification is to negate the conclusion and generate the Skolem form. Then *jt* and *et* become new Skolem constants, and *ut* becomes universally quantified. This formula separated relative to the NAIF theory of time is:

$$(Y_1 = abs(\textit{coordinates-to-time}(JULIAN), jt)) \wedge \\ (Y_2 = abs(\textit{coordinates-to-time}(EPHEMERIS), et)) \wedge \\ (Y_3 = abs(\textit{coordinates-to-time}(UTC), ut)) \\ \Rightarrow (\textit{sum-of-times}(Y_1, Y_2) \neq Y_3)$$

The conjunction in the antecedent is the restriction which is stored as a congruence closure graph. The disequality is the matrix available to SNARK's extended inference rules. SNARK unifies the matrix of the above formula with the complementary literal in the (now rewritten) *sum-of-times* axiom, shown below.

$$(X_1 = abs(\textit{coordinates-to-time}(EPHEMERIS), U_2)) \wedge$$
$$(X_2 = abs(\textit{coordinates-to-time}(EPHEMERIS), U_3)) \wedge$$
$$(X_3 = abs(\textit{coordinates-to-time}(EPHEMERIS), \textit{sum-time-coords}(U_2, U_3)))$$
$$\Rightarrow (\textit{sum-of-times}(X_1, X_2) \neq X_3)$$

The unifier $\{Y_1 \leftarrow X_1, Y_2 \leftarrow X_2, Y_3 \leftarrow X_3\}$ is obtained. The matrix of the resolvant is the empty clause. The unifier is then passed to the *convert-time* procedure which now attempts to merge the closure graphs of the two restrictions. This generates the following three equalities:

$$abs(\textit{coordinates-to-time}(JULIAN), jt) = abs(\textit{coordinates-to-time}(EPHEMERIS), U_2)$$

$$abs(\textit{coordinates-to-time}(EPHEMERIS), et) = abs(\textit{coordinates-to-time}(EPHEMERIS), U_3)$$

$$abs(\textit{coordinates-to-time}(UTC), ut) =$$
$$abs(\textit{coordinates-to-time}(EPHEMERIS), \textit{sum-time-coords}(U_2, U_3))$$

Since *jt* and *et* are constants, the procedure determines that a binding can be generated for variables U_2 and U_3. The procedure then generates the witness $U_2 \leftarrow$*convert-time*(*JULIAN, EPHEMERIS, jt*). Also the binding $U_3 \leftarrow et$ is generated for the second equality literal.

Finally, the variable *ut* is bound to the term *convert-time*(*EPHEMERIS, UTC, sum-time-coords*(*convert-time*(*JULIAN, EPHEMERIS, jt*), *et*)). This term is the answer term bound to the existentially quantified variable in the original specification. This term is a program that converts *jt* to ephemeris time, adds this converted time to *et* (already in the EPHEMERIS time system), and converts the resultant time to the UTC time system.

6 Implementation

We have augmented the SNARK resolution theorem prover with the separated inference rules described previously. We have also added a set of decision procedures to SNARK specifically for Amphion/NAIF and used the resulting system to generate programs. This section describes the results of this work. This work has been motivated by research into Meta-Amphion; however, a detailed discussion of Meta-Amphion is outside the scope of this paper. We believe that the work presented here is of general interest in its own right because it shows a new way of accelerating deductive synthesis engines, specifically full first-order refutation-based theorem provers, using decision procedures.

Our experience with Amphion/NAIF has been that the performance of the theorem prover has been an important consideration in the maintenance of the system. As with all general purpose theorem provers, SNARK is subject to combinatorial explosion in the search for a solution to a problem. As shown in Figure 1, the search space for all but the smallest problems is unacceptable when using an untuned system.

A great deal of time and effort has been devoted to tuning Amphion/NAIF. Figure 1 shows that the hand tuning (done over the course of a year and a half by an expert in deductive synthesis) was very effective in reducing the synthesis time, frequently reducing the time from hours or days to minutes. The central problem has not been the lack of success in tuning Amphion/NAIF; it has been the cost in time and effort required to tune the system particularly after the domain theory has been modified. The goal of Meta-Amphion is to assist in the construction and maintenance of deductive synthesis domain theories, and in particular, to assist in tuning declarative domain theories.

In general, a deductive synthesis system is tuned by generating example problems, observing the system as it attempts to solve the problems, then tuning the system to direct it towards solutions for the example problems. There are two primary methods of tuning the system: (1) changing the agenda ordering, and (2) reformulating the domain theory axioms.

SNARK selects an unprocessed formula from the set of supported formulas, then applies every applicable inference rule to the formula, possibly constructing many new (unprocessed) formulas. The agenda-ordering function orders formulas after each inference step, in effect choosing the next supported formula to be processed.

The original, general-purpose agenda-ordering function sorted formulas on the basis of the size (number of sub-terms) and the number of abstract terms in the formula. Thus SNARK favored formulas with fewer abstract terms. It also searched for solutions with a smaller number of terms before looking at larger terms. (SNARK only completes its search when a ground, concrete term is found for each output variable.) Since smaller answer terms represent shorter programs, shorter programs are generated before longer ones.

The hand-tuned agenda-ordering function weighs each formula according to several factors. In addition to the number of abstract terms in a formula, the hand-tuned agenda-ordering function also counts the number of non-ground convert-time terms. Formulas with more non-ground convert-time terms appear lower on the agenda. This prevents SNARK from generating terms such as

convert-time(*Ephemeris, Julian, convert-time* (*Julian, Ephemeris, tc*)).

This term results in the conversion of a time coordinate from the Ephemeris time system to the Julian time system and back again.[3] The agenda ordering function has similar weighting schemes for other terms for which tuning has been necessary.

When the decision procedures are used, the procedure for *coordinates-to-time* collects all the abstract time terms. It delays generating a *convert-time* term until a ground *convert-time* can be constructed.

We compared the performance of three Amphion/NAIF systems: an untuned system with a simple agenda-ordering function; an extensively hand-tuned system; and a system which uses several decision procedures. The untuned system describes the state of Amphion/NAIF prior to expert tuning. This is exactly the type of system we expect

3.In this case, this formula is rewritten by two rules into an identity. However, with paramodulation, new variables are introduced. No rewriting occurs since the new variables do not match. The result is that chains of convert-time terms may be generated. Each of these terms is then a target for resolution or paramodulation to generate more formulas, none of which lead to a solution.

Meta-Amphion to be given as input. The hand-tuned system was the result of extensive tuning by a theorem proving expert. Not only was a specialized agenda ordering function developed, but several of the axioms were reformulated to force the theorem prover to behave in a particular manner. Such reformulation depends on in-depth knowledge of the workings of the theorem prover. For the decision procedure system, a set of five decision procedures was written and used. Each of these procedures was interfaced to SNARK using the inference rules described previously.

The domain theories for each of these systems consisted of approximately 325 first-order axioms. Many of these axioms are equalities, some of which are oriented and used as rewrite rules. A series of 27 specifications was used to test these synthesis systems. These specifications ranged from trivial with only a few literals to fairly complex with dozens of literals. Thirteen of the specifications were obtained as solutions to problem specifications given by domain experts, thus this set is representative of the problems encountered during real-world use.

As shown in Figure 1, the untuned system showed exponential behavior with respect to the specification size for the number of inference steps (and the CPU time) required to generate a program. The hand-tuned and decision-procedure-tuned (TOPS) systems both grew much less rapidly, with the decision-procedure-tuned system growing at about one third the rate of the hand-tuned system in the number of inference steps required to obtain a proof, as shown in Figure 2.

The performance of the system using the decision procedures was unaffected by changing between the simple and sophisticated agenda-ordering functions. This is not surprising since the decision procedures and the hand tuning both targeted the same inferences, and when using the procedures, the terms counted by the agenda ordering function are hidden inside the data structures of the procedures.

Although the programs generated using the decision procedures were not always identical to programs generated without them, for each case we proved that the programs computed the same input/output pairs.

7 Conclusion

A major hindrance to the construction and use of deductive synthesis systems is the cost associated with constructing and maintaining the domain theories associated with them. A primary cost of maintaining a domain theory is the cost of tuning the deductive synthesis system to the theory. Our work continues on the Meta-Amphion system whose goal is to automatically design specialized deductive synthesis systems from untuned domain theories. Much of our effort is on techniques to automate the process of replacing subtheories with automatically generated decision procedures. The decision procedure design process is one of design from components, where the components are parameterized procedures. The process involves combination and instantiation. The component procedures turn out to be closure-based ground literal satisfiability procedures. As described in this paper, we have also found this class of procedures useful in a new type of hand-tuning of deductive synthesis systems. We have defined the class of closure-based ground literal satisfiability procedures, introduced extensions of a refutation-based general-purpose theorem prover to enable any procedure in this class to be integrated with the theorem prover to accelerate deductive synthesis, and proven that these extensions are correct.

We have shown how we have used the manual decision procedure insertion technique to accelerate Amphion/NAIF, a system that is in regular use by NASA space scientists. Also, we are using the technique in the development of other "real-world" systems such as a system to synthesize three dimensional grid layouts in Computational Fluid Dynamics and a system to automatically generate schedulers for Space Shuttle payloads.

The research methodology we are employing is to manually identify sets of axioms that give rise to combinatorial explosion problems in theorem proving, just as we do when tuning manually. Then we generalize these problems into problem classes and create generalized solutions in the form of parameterized decision procedures. These parameterized procedures are added a library in Meta-Amphion. Meta-Amphion also has an evolving library of techniques that enable it to analyze a domain theory, identify instances of problem classes for which it has a decision procedure, and automatically instantiate the procedure for the problem class. Meta-Amphion also proves that the procedure is a solution to the specific problem, then modifies the domain theory appropriately.

In general, the soundness of solutions found using decision procedures very much depends on the soundness of the procedures themselves. Another ongoing activity in the Meta-Amphion project is to develop methods for proving the correctness of parameterized decision procedures.

ACKNOWLEDGEMENTS

Tom Pressburger read and helped with useful discussions of several drafts of this paper. Mahadevan Subramaniam, Mike Lowry, John Cowles and the anonymous reviewers also provided helpful comments.

REFERENCES

[1] Baader, F. & Tinelli, C., "A New Approach for Combining Decision Procedures for the Word Problem, and its Connection to the Nelson-Oppen Combination Method," CADE14, pp. 19-33, 1997.

[2] R. Boyer and Moore, J, *Integrating Decision Procedures into Heuristic Theorem Provers: A Case Study of Linear Arithmetic*, Institute for Computing Science and Computer Applications, University of Texas as Austin, 1988.

[3] Burckert, H. J., "A Resolution Principle for a Logic With Restricted Quantifiers," *Lecture Notes in Artificial Intelligence*, Vol. 568, Springer-Verlag, 1991.

[4] Chang, C & Lee, R.C., *Symbolic Logic and Mechanical Theorem Proving*, Academic Press, New York, 1973.

[5] Cyrluk, D., Lincoln, P., Shankar, N. "On Shostak's decision procedures for combinations of theories," *Automated Deduction--CADE-13* in *Lecture Notes in AI* 1104, (M. A. McRobbie and J. K. Slaney Eds), Springer, pp. 463-477, 1996.

[6] Deville, Y. and Lau, K., "Logic Program Synthesis," *Journal of Logic Programming*, 19,20: 321-350, 1994.

[7] Dunham, B. and North, J., “Theorem Testing by Computer,” *Proceedings of the Symposium on Mathematical Theory of Automata*, Polytechnic Press, Brooklyn, N. Y., pp. 173-177, 1963.

[8] Gallier, J. H., *Logic for Computer Science: Foundations of Automatic Theorem Proving*, Harper and Row, 1986.

[9] C.A.R. Hoare, “Proof of Correctness of Data Representations,” *Acta Infomatica*, Vol. 1, pp. 271-281, 1973.

[10] M. Lowry and J. Van Baalen, “META-Amphion: Synthesis of Efficient Domain-Specific Program Synthesis Systems”, *Automated Software Engineering*, vol 4, pp. 199-241, 1997.

[11] Nelson, G., and Oppen, D., “Simplification By Cooperating Decision Procedures,” *ACM Transactions on Programming Languages and Systems*, No. 1, pp. 245-257, 1979.

[12] Nelson, G., and Oppen, D., “Fast decision procedures based on congruence closure,” *Journal of the ACM*, 27, 2, pp. 356-364, 1980.

[13] Owre, S., Rushby, M., and Shankar, N., “PVS: A Prototype Verification System,” CADE-11, *LNAI* Vol. 607, pp. 748-752, 1992.

[14] Roach, S., “TOPS: Theory Operationalization for Program Synthesis,” Ph.D. Thesis at University of Wyoming, 1997.

[15] Shostak, R., “A practical decision procedure for arithmetic with function symbols,” *Journal of the ACM*, Vol. 26, pp. 351-360, 1979.

[16] Shostak, R., “Deciding Combinations of Theories,” *Journal of the ACM*, Vol. 31, pp. 1-12, 1984.

[17] M. Stickel, R. Waldinger, M. Lowry, T. Pressburger, and I. Underwood, “Deductive Composition of Astronomical Software from Subroutine Libraries,” *CADE-12*, 1994. See http://ic-www.arc.nasa.gov/ic/projects/amphion/docs/amphion.html

PostScript error (undefinedfilename, fontfile)