# Using Decision Procedures to Build Domain-Specific Deductive Synthesis Systems[1]

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

## 1 Introduction

This paper describes a class of decision procedures that we have found useful for efficient, domain-specific deductive synthesis. These procedures are called *closure-based ground literal satisfiability procedures.* We argue that this is a large and interesting class of procedures and show how to interface these procedures to a theorem prover for efficient deductive synthesis. Finally, we describe some results we have observed from our implementation.

Amphion/NAIF [Stickel 94] 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 [Deville 94]) in which programs are synthesized as a byproduct of theorem proving. In this paradigm, problem specifications are of the form $\forall\bar{x}\exists\bar{y}[P(\bar{x},\bar{y})]$ , where $\bar{x}$ and $\bar{y}$ are vectors of variables. We are only interested in constructive proofs in which witnesses have been produced for each of the variables in $\bar{y}$.

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[2] are almost always unacceptably inefficient unless the domain theory and theorem prover are carefully tuned. This tuning process usually requires a large amount of time and expertise in automated reasoning and is therefore costly.

In order to assist in constructing efficient implementations, we are developing a tool, Meta-Amphion [Lowry 97], that takes a domain theory as input and automatically generates an efficient, specialized deductive synthesis engine such as Amphion/NAIF. The key is a technique that generates efficient decision procedures for subtheories of the domain theory and then integrates them with a general-purpose refutation-based theorem-prover.

A prototype of Meta-Amphion has been constructed [Roach 97]. This prototype has generated domain-specific deductive synthesis systems that achieve a significant speed improvement over non-optimized, general-purpose theorem provers. More importantly, these generated systems perform at least as well as, and often better than, expertly-tuned theorem provers for particular application domains. Figure 1 is a graph of the problem size (number of literals) vs. the number of inference steps required to find a proof for an unoptimized system, a hand tuned system, and a system generated by Meta-Amphion (Tops). Figure 2 com-

---

1. Extended Abstracts of LOPSTR'98, Eighth International Workshop on Logic-based Program Synthesis and Transformation, 15-19 June 1998, Manchester, UK. Technical Report Series, Department of Computer Science, University of Manchester, ISSN 1361-6161. Report number UMCS-98-6-1. http://www.cs.man.ac.uk/cstechrep/titles98.html

2. 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.

pares a hand-tuned system vs. the Meta-Amphion generated system (Tops).

This paper describes the underlying infrastructure used by Meta-Amphion. This includes the decision procedure interface and the properties that decision procedures must have to be interfaced. (We do not discuss the generation of these procedures here.) We have found that even with hand-generation of these procedures, this infrastructure dramatically reduces the time it takes to construct efficient domain-specific synthesis systems by enabling an automated reasoning expert to quickly identify where decision procedures can be used to improve the performance of the theorem prover without iteratively inspecting proof traces.

While much existing research on decision procedures has been either in isolation [N&O 79, Shostak 84, Cyrluk 96] or in the context of interfacing procedures to non-refutation-based theorem provers [PVS 92, B&M 88], we are unaware of any work done on decision procedures in the context of 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. The collection of extended inference rules enables procedures to be interfaced to the theorem prover in a straightforward and uniform manner. Combinations of procedures can be plugged in on a theory-by-theory basis, allowing the theorem prover to be tailored to particular theories. The class of decision procedures that can be interfaced

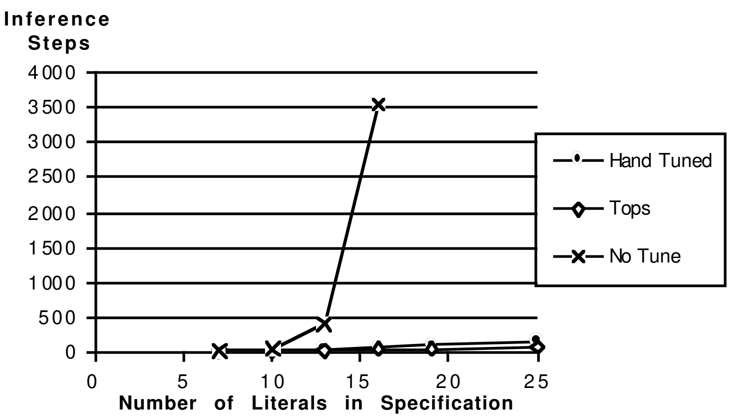

Figure 1

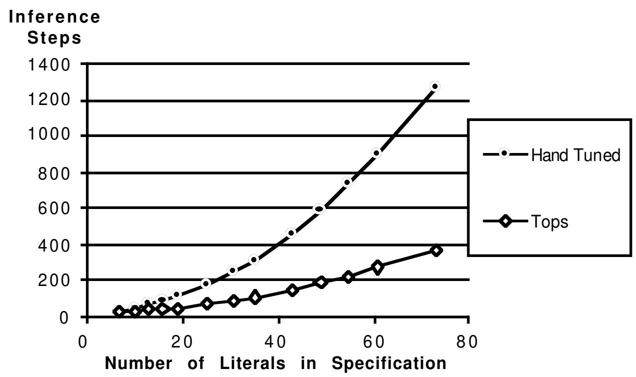

Figure 2

is such that the witnesses produced are typically straight-line program fragments that are incorporated into larger programs produced by the theorem prover.

Section 2 introduces separated clause notation, the notation used by our extended inference rules. The motivation for these rules is that they facilitate the use of decision procedures to replace general purpose theorem proving over a subset of a theory. Section 3 describes procedures for testing satisfiability. Section 4 describes decision procedures used specifically for deductive synthesis. Section 5 describes the implementation of the interface and the results of using these procedures for deductive synthesis for Amphion/NAIF.

## 2 Separated Inference Rules

This section describes our extension to the inference rules in the SNARK [Stickel 94] 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 interfaced 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* [Burckert91]. Recall that, given a first-order theory $T$ and a first-order formula $\Phi$, 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 $T_1 \subseteq T$, 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 functions 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* (also called a *constraint 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 $L\text{-}L_1$.

Constant (0-ary function) symbols may appear in $C_1$ or $C_2$ regardless of which language they are in. 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.

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 $t$ whose head symbol is in the "wrong" language is replaced by a new variable $x$. Then if $t$ appeared in the matrix, $x=t$ is conjoined to the restriction and if $t$ appeared in the restriction, $x \neq t$ is disjoined to the matrix.

**Example 2.1** Suppose we have a theory $T_1$ of LISP list structure whose non-logical symbols are the function symbols HEAD, TAIL, and CONS. 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 $\sigma$ is used in forming the restriction of the resolvant which is the conjunction of the restrictions of $C'$ and $C''$ with the substitution $\sigma$ 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. If $l_1$ and $\overline{l_2}$ unify with most general unifier $\sigma$ 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[3] 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** Step 10 of Example 2.3 (page 5) is a good illustration of separated binary resolution. In this step, the literal $z \neq append(v,w)$ (from step 7) is unified with $x_1 = append(front(x_1),y_1)$ (from step 9) 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.

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

**Lemma 2.1 (Soundness of separated binary resolution)** Let $\Psi$ be a set of separated clauses and let $\psi$ be a clause derived from two elements of $\Psi$ by separated binary resolution. If $M$ is a model of $\Psi$, $M$ 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 $T_I$. 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. If $t$ and $r$ unify with most general unifier $\sigma$ and $\exists\left[\left(\alpha_1 \wedge \ldots \wedge \alpha_n \wedge \beta_1 \wedge \ldots \wedge \beta_p\right)\sigma\right]$ is satisfiable in $T_I$, a *separated paramodulant* of $C'$ and $C''$ is the separation of $\left(\alpha_1 \wedge \ldots \wedge \alpha_n \wedge \beta_1 \wedge \ldots \wedge \beta_p\right)\sigma \Rightarrow l\sigma[s\sigma] \vee Q\sigma \vee R\sigma$, 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 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 $T_I \models \exists C_1 \vee \ldots \vee \exists C_n$. A proof of this fact can be found in [Burckert91] where it is also shown that this disjunction is finite so long as $T_I$ is first-order (this is a consequence of Compactness). Hence, a *closed* separated refutation 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 $T_I$.

**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, a number of additional types of axioms must be added such as separated versions of predicate congruence axioms. Further discussion of completeness is outside the scope of this paper.

[Burckert 91] points out that for some restriction theories, closed separated refutations can always be obtained by considering the validity of only the restrictions of individual clauses. For instance, it is proven that if $T_I$ 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 validity 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 [Dunham 63]) 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 prohibits the derivation of clauses with empty matrices when terms that are not in the restriction language appear, i.e., these terms keep getting separated back into the matrix. In this case, 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 $L\text{-}L_I$ 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 validity of the resultant restriction with symbols in $L\text{-}L_I$ uninterpreted.

**Example 2.3**. Let $T_I$ 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.* The theorem (proved below) can be viewed as a simple deductive synthesis query where we are attempting to derive 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 $T_1$ is given below. Clauses 1-5 below are the first three axioms above separated relative to $T_1$. Note that these are the clauses of $T$-$T_1$. The last two axioms above are incorporated into the decision procedure and, therefore, do not appear in the proof. Since the existential closure of the restriction of 11 is a theorem of the CONS-HEAD-TAIL theory (regardless of the interpretation of *front* and *last*), the proof is finished.

| | | |
|---|---|---|
| 1 | Given | $\Rightarrow x = x$ |
| 2 | Given | $\Rightarrow K \neq nil$ |
| 3 | Given | $x = tail(K) \Rightarrow (x \neq nil)$ |
| 4 | Given | $(x = tail(K) \wedge y = cons(z, nil) \wedge w = tail(K)) \Rightarrow (x = nil \vee K = nil \vee z \neq last(w) \vee x = append(front(x), y))$ |
| 5 | Given | $(x = cons(u, v)) \wedge (y = cons(u, z)) \Rightarrow append(x, w) = y \vee z \neq append(v, w)$ |
| 6 | Negated conclusion | $x_1 = cons(z_1, nil) \Rightarrow (K \neq append(y_1, x_1))$ |
| 7 | paramodulate 5 into 6 | $(x = cons(u, v)) \wedge (K = cons(u, z)) \wedge (w = cons(z_1, nil)) \Rightarrow (z \neq append(v, w))$ |
| 8 | resolve 2 and 4 | $(x = tail(K)) \wedge (y = cons(u, nil)) \Rightarrow (x = nil) \vee (x = append(front(x), y)) \vee (u \neq last(x))$ |
| 9 | resolve 3 and 8 | $(x_1 = tail(K)) \wedge (y_1 = cons(u_1, nil)) \Rightarrow (x_1 = append(front(x_1), y_1)) \vee (u \neq last(x_1))$ |
| 10 | resolve 9 and 7 | $(x = cons(u, front(z))) \wedge (K = cons(u, z)) \wedge (w = cons(z_1, nil)) \wedge (z = tail(K)) \wedge (w = cons(u_1, nil)) \Rightarrow (u_1 \neq last(z))$ |
| 11 | resolve 10 and 1 | $(x = cons(u, front(z))) \wedge (K = cons(u, z)) \wedge (w = cons(z_1, nil)) \wedge (z = tail(K)) \wedge (w = cons(last(z), nil)) \Rightarrow []$ |

## 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 of satisfiability of restrictions of derived formulae; and (2) a test of validity of the restriction of any formula which has an empty matrix. This section describes procedures that test satisfiability of restrictions. We identify a large class of procedures that can be used for this purpose.

To use separated inference rules to prove a theorem $\Phi$ in a theory $T$, we assume that a procedure exists for deciding the satisfiability of conjunctions of clauses in a restriction theory $T_1 \subseteq T$. 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 the restrictions of separated clauses. We do this by replacing the variables in the restrictions by new constant symbols. The fact that we can use GLSPs to determine satisfiability 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 $\sigma$ be the substitution $\{x_1 \leftarrow c_1, ..., x_n \leftarrow c_n\}$, where the $c_i$ are new uninterpreted constant symbols. Replace the restriction of $C$ with $x_1 = c_1 \wedge ... \wedge x_n = c_n \wedge \overline{C_1}\sigma$.

We show that (a) $x_1 = c_1 \wedge ... \wedge x_n = c_n \wedge \overline{C_1}\sigma$ and $\overline{C_1}$ are cosatisfiable and (b) the satisfiability of $C$ implies the satisfiability of $(x_1 = c_1 \wedge ... \wedge x_n = c_n \wedge \overline{C_1}\sigma) \Rightarrow C_2$. Hence, we can replace any separated clause $C$ with the clause $(x_1 = c_1 \wedge ... \wedge x_n = c_n \wedge \overline{C_1}\sigma) \Rightarrow C_2$ 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 $x_1 = c_1 \wedge ... \wedge x_n = c_n \wedge \overline{C_1}\sigma$, if $x_1 = c_1 \wedge ... \wedge x_n = c_n \wedge \overline{C_1}\sigma$ 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 $\langle M^*, I \rangle \models x_1 = c_1 \wedge ... \wedge x_n = c_n \wedge \overline{C_1}\sigma$.

(b) If C is satisfiable, either $C_2$ is satisfiable or $\overline{C_1}$ is unsatisfiable. If $C_2$ is satisfiable, then $(x_1 = c_1 \wedge ... \wedge x_n = c_n \wedge \overline{C_1}\sigma) \Rightarrow C_2$ is satisfiable. Otherwise, since $\overline{C_1}$ and $x_1 = c_1 \wedge ... \wedge x_n = c_n \wedge \overline{C_1}\sigma$ are cosatisfiable, $x_1 = c_1 \wedge ... \wedge x_n = c_n \wedge \overline{C_1}\sigma$ 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 [N&O 79][N&O 80] [Cyrluk 96]. 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 [N&O 80] show how to extend a GLSP for the theory of equality with uninterpreted function symbols to a theory of LISP list structure, i.e., a theory in which the function symbols HEAD, TAIL, CONS and NIL are interpreted. Their procedure can be interfaced to our system and used to check satisfiability of restrictions in our running example, i.e., the restriction of the clause derived in step 9 of Example 2.1

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

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

We have also constructed several new procedures by extending a GLSP for uninterpreted function symbols. Also, [Gallier 86, ch 10.6] 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 $P(t_1,...,t_n)$ as $P(t_1,...,t_n)=True$ and $\neg P(t_1,...,t_n)$ as $P(t_1,...,t_n)=False$. Then, if the congruence closure graph of a conjunction $F$ contains $True=False$, $F$ is unsatisfiable.

Finally, both [N&O79] and [Cyrluk 96] 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 [Baader 97].

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 an integrated decision procedure for a restriction theory. (See [Lowry 97] or [Roach 97] for examples of techniques for 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 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 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 computes satisfiability of a set of formulas $\Phi$ by constructing a finite set $\Psi$ of ground consequences of $\Phi$ such that $\Psi$ contains a ground literal and its negation just in case $\Phi$ is unsatisfiable.

GLSPs based on congruence closure are examples of closure-based satisfiability procedures. They construct a congruence graph to check satisfiability of a conjunction of literals. As new literals are added to a conjunction, new nodes representing terms are added to the graph and/or congruence classes are merged.

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 HEAD-TAIL-CONS GLSP.

**Example 4.1** In step 11 of example 2.3, it must be shown that the existential closure of

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

is a theorem of the HEAD-TAIL-CONS theory. 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 $u = head(K)$. 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 in every model of $T$ in which $F(c_1,...,c_n)$ is satisfiable, $F(t_1,...,t_n)$ is satisfiable. Next, the procedure is used to check the unsatisfiability of $\neg F(t_1,...,t_n)$. Since $\neg F(t_1,...,t_n)$ is a disjunction which is unsatisfiable just in case all of its disjuncts are, each literal of $\neg F(t_1,...,t_n)$ can be checked separately. If $\neg F(t_1,...,t_n)$ is unsatisfiable, $T \models F(t_1,...,t_n)$ 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 $c_1,\dots,c_n$ do not occur in $T$. If $T \cup F \models (c_1 = t_1 \wedge \cdots \wedge c_n = t_n)$, where each $t_i$ is a term not containing any of the $c_j$s, $T \models \forall x_1,\dots,x_n (F(x_1,\dots,x_n) \Rightarrow (x_1 = t_1 \wedge \cdots \wedge x_n = t_n))$.

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

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 $\exists(y,z)(K = append(y, cons(z, nil)))$ in example 2.3, the Nelson & Oppen GLSP produces witnesses for $y$ and $z$. 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 in this manner.

# 5 Implementation

We have augmented the SNARK resolution theorem prover with the separated inference rules described previously. We then hand-coded a set of decision procedures and added them to the theorem prover. This section describes the results of this work. Amphion/NAIF is a deductive synthesis system that facilitates the construction of programs that solve programs in solar system mechanics. Programs generated by Amphion/NAIF consist of straight-line code using assingment statements and calls to elements of the SPICE subroutine library. The SPICE library was constructed to solve problems related to the planning and interpretation of data received by deep space probes.

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. There are two primary elements to tuning the system: the agenda ordering and reformulation of domain theory axioms. A sophisticated agenda-ordering function is used to order formulas after each inference step.

We compared the performance of three Amphion systems: an untuned system with a simple agenda ordering function; an extensively hand-tuned system; and a system which uses several decision procedures. (The performance of the system using the decision procedures was unaffected by changing between the simple and sophisticated agenda ordering functions.) The untuned system describes the state of Amphion/NAIF prior to expert tuning. This is exactly the type of system we expect Meta-Amphion to be given as input.

The domain theory for the hand-tuned system consisted of 330 first-order axioms. In the untuned and Meta-Amphion tuned systems there were approximately 320 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.

Five procedures were created and used to prove each of the 27 specifications. Each of these procedures was interfaced to the SNARK resolution theorem prover using the inference rules described in Section 2. Although the programs generated were not always identical, it was shown that solutions to the same problem specification were always programs that computed the same values.

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 TOPS-generated systems both grew much less rapidly, with the TOPS-generated 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.

**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**

[Baader 97] 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.

[Boyer and Moore 88] 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.

[Burckert 91] Burckert, H. J., “A Resolution Principle for a Logic With Restricted Quantifiers,” *Lecture Notes in Artificial Intelligence*, Vol. 568, Springer-Verlag, 1991.

[Chang & Lee 73] Chang, C & Lee, R.C*., Symbolic Logic and Mechanical Theorem Proving*, Academic Press, New York, 1973.

[Cyrluk 96] 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, p463-477, 1996.

[Deville 94] Deville, Y. and Lau, K., “Logic Program Synthesis,” *Journal of Logic Programming*, 19,20: 321-350, 1994.

[Dunham 63] 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.

[Gallier 86] Gallier, J. H., *Logic for Computer Science: Foundations of Automatic Theorem Proving,* Harper and Row, 1986.

[Lowry 97] M. Lowry and J. Van Baalen, “META-Amphion: Synthesis of Efficient Domain-Specific Program Synthesis Systems”, *Automated Software Engineering*, vol 4, pp199-241, 1997.

[N& O 79] Nelson, G., and Oppen, D., “Simplification By Cooperating Decision Procedures,” *ACM Transactions on Programming Languages and Systems*, No. 1, pp245-257, 1979.

[N&O 80] Nelson, G., and Oppen, D., “Fast decision procedures based on congruence closure,” *Journal of the ACM*, 27, 2, pp. 356-364, 1980.

[PVS 92] Owre, S., Rushby, M., and Shankar, N., “PVS: A Prototype Verification System,” CADE-11, *LNAI* Vol 607, pp 748-752, 1992.

[Roach 97] Roach, S., “TOPS: Theory Operationalization for Program Synthesis,” Ph.D. Thesis at University of Wyoming, 1997.

[Shostak 79] Shostak, R., “A practical decision procedure for arithmetic with function symbols,” *Journal of the ACM*, Vol. 26, pp. 351-360, 1979.

[Shostak 84] Shostak, R., “Deciding Combinations of Theories,” *Journal of the ACM*, Vol. 31, pp. 1-12, 1984.

[Stickel *et al* 94] 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