Using Decision Procedures to Build 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

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 in which programs are synthesized as a byproduct of theorem proving from a domain theory. 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, and 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.

There are several reasons why, despite these potential advantages, the number of deductive synthesis systems remains small. Perhaps the most serious reason is that systems built using this technology are almost always unacceptably inefficient unless the domain theory and theorem prover are carefully tuned. This tuning process requires a large amount of automated reasoning expertise, and even with this expertise, the process is iterative and extremely time consuming.

In our attempts to construct an efficient deductive synthesis system for Amphion/NAIF, we initially considered using Prolog. However, due to the extensive need for equality in the domain theory, Prolog was inappropriate. So we moved to a more general paradigm employing a refutation-based theorem prover. Constructing an efficient implementation in this setting was very time consuming.

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 through an interface to 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 compares a hand-tuned system vs. the Meta-Amphion generated system (Tops).

This paper describes the underlying infrastructure used by Meta-Amphion, i.e. the interface and the properties of the procedures. (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.

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 satisfiability 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.

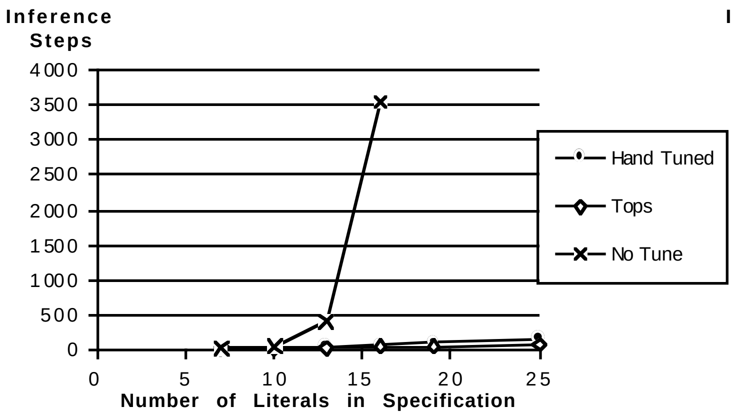

Figure 1

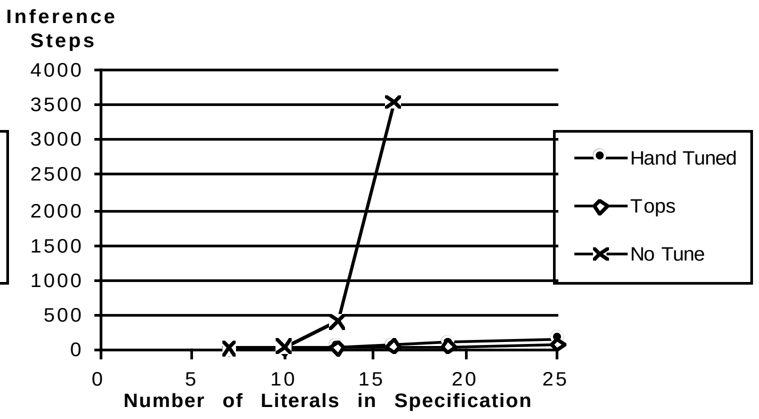

Figure 2

Section 2 introduces separated clause notation, the notation used by the separated resolution and paramodulation 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 domain. Section 3 describes the decision procedure interface to the theorem prover. 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 decision procedures to be interfaced for deductive synthesis. 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. The other rules are extended similarly.

Separated binary resolution is similar to resolution with restricted quantifiers or *RQ-resolution* [Burckert91]. Recall that we prove $T|=\Phi$ 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 $T|=\Phi$ by showing that no model of T_1 can be extended to a model of $T_2\cup\{\neg\Phi\}$, where $T_2=T-T_1$ (where a theory is a set of sentences closed under implication).

The separated rules work with clauses that are *separated relative* to a subtheory, called a *restriction theory* (also often called a *constraint theory)*. Separated clause form is similar to RQ-clause form in [Burckert 91].

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.

Constant symbols may appear in C_1 or C_2 regardless of what 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, 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 $tail(L) \neq nil$ relative to T_1 is $x = tail(L) \Rightarrow (x \neq nil)$.

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. If l_1 and $\overline{l_2}$ unify with most general unifier σ and $\exists\left[(\alpha_1 \wedge \ldots \wedge \alpha_n \wedge \beta_1 \wedge \ldots \wedge \beta_p)\sigma\right]$ is satisfiable in T_1, the *separated resolvant* of C' and C'' is the separation[1] 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 In step 8 of example 2.3, the existential closure of the derived restriction $(x = tail(w)) \wedge (y = cons(u, z))$ is satisfiable in the theory of LISP list structure. Therefore, the resolvant is retained.

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 here 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. If t and r unify with most

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

general unifier σ 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 $l\sigma[s\sigma]$ is the literal l with the substitution σ applied and one occurrence of $t\sigma$ replaced with $s\sigma$.

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

An ordinary 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 parmodulation, 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. In practice, completeness has not been an issue in our work on deductive synthesis, so we do not discuss it further here.

[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-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.

Example 2.3. Given the theory (with L a constant symbol):

$$x = x \wedge L \neq nil \wedge tail(L) \neq nil$$
$$L \neq nil \rightarrow (tail(L) \neq nil \rightarrow tail(L) = append(front(tail(L)), cons(last(tail(L)), nil))$$
$$append(cons(u,v),w) = cons(u, append(v,w))$$
$$(x \neq nil) \rightarrow x = cons(head(x), tail(x))$$
$$head(cons(x,y)) = x \wedge tail(cons(x,y)) = y \wedge cons(x,y) \neq nil$$

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*) and *last* (a witness for *z*) under the assumption for an input list *L*, that $L \neq nil$ and $tail(L) \neq nil$. Let T_I be a theory of HEAD, TAIL, CONS and NIL. 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.

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

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.

3 The Decision Procedure Interface

This section describes how decision procedures are interfaced to the theorem prover through the separated inference rules presented in the previous section. We identify the properties a decision procedure must have in order for it to be interfaced. We also show that there is a large and interesting class of decision procedures that meet this requirement.

A decision procedure is interfaced to the theorem prover for proving a theorem Φ in a theory T by identifying a restriction theory $T_1 \subseteq T$ for which the procedure decides satisfiability. The clauses of T-T_1 and $\neg\Phi$ are separated relative to T_1 and the decision procedure checks the satisfiability of derived restrictions.

An important question is, "What is the utility of our technique for interfacing decision procedures to a theorem prover?" The answer to this question turns on the following question: "How large is the class of decision procedures that can be interfaced?" To be interfaced, a procedure must decide satisfiability of clause restrictions. As mentioned, restrictions are conjunctions of literals possibly containing variables. One class of procedures that appears to be large is the class of *ground literal satisfiability procedures*. These are procedures that decide satisfiability of conjunctions of ground literals. It turns out that, even though restrictions of first-order clauses have variables in them, it is possible to interface any ground literal satisfiability procedure. We now establish this fact.

Definition 3.1 (Ground Literal Satisfiability Procedure) A *ground literal satisfiability procedure(GLSP) 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 Sketch: 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 $\{x_1 \leftarrow c_1,...,x_n \leftarrow c_n\}$, where the c_i are new uninterpreted constant symbols. Replace the restriction of *C* with $x_1 = c_1 \wedge ... \wedge x_n = c_n \wedge \overline{C_1}\sigma$.

The full proof shows 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 $\overline{C_1}\sigma$. []

The fact that any GLSP can be interfaced to the separated inference rules is a fortunate situation because many GLSPs have been identified [N&O 79][N&O 80] [Cyrluk 96]. In addition, there is reason to believe that there are many more procedures in this class. For instance, [N&O 80] shows 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

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

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.

Perhaps more interestingly, both [N&O79] and [Cyrluk 96] describe techniques for combining GLSPs with disjoint languages into a GLSP for the union of these languages and 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 that the disjunction of the restrictions from a set of clauses with empty matrices be checked for entailment. Recall that this paper is focused on restriction theories in which only single restrictions must be checked for entailment. Hence, automated separated refutations require decision procedures for computing both satisfiability and entailment of restrictions.

For the entailment problem, we cannot use the technique of replacing universally quantified variables in a restriction with uninterpreted constants. Instead, these variables are replaced by existentially quantified variables. (An argument similar to the proof of lemma 4.1 can be used to justify this.) For the entailment check, we need decision procedures that are *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 Φ by constructing a finite set Ψ of ground consequences of Φ such that Ψ contains a ground literal and its negation just in case Φ 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.

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

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

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 $L = cons(u,z)$, we get $u = head(L)$. Hence, $x = cons(head(L), front(tail(L)))$. 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 $z_1 = last(tail(L))$. 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, $T|\!=F(t_1,...,t_n)$ and $T|\!=\exists F$. 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 $T \cup F \models (c_1 = t_1 \wedge \cdots \wedge c_n = t_n)$, where each t_i is a term not containing any of the c_js, $T \models \forall x_1,...,x_n(F(x_1,...,x_n) \Rightarrow (x_1 = t_1 \wedge \cdots \wedge x_n = t_n))$.

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

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 in which F is true. 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 must 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 $\exists(y,z)(L = append(y, cons(z,nil)))$ in example 2.3, the Nelson & Oppen GLSP produces witnesses for y and z. These are *cons(head(L),front(tail(L))* and *last(tail(L))* respectively which are the synthesized programs for *front(L)* and *last(L)* under the assumption that $L \neq nil$ and $tail(L) \neq nil$.

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

5 Implementation

We compared the performance of three deductive synthesis systems: an untuned system, a system manually tuned by a program synthesis expert, and a system tuned by using several decision procedures to the theorem prover. The untuned system describes the state of Amphion/NAIF before the theorem proving expert tuned that system. This untuned system is exactly the type of system we expect Meta-Amphion will 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.

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.

[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.