High Integrity Software Conference, Albuquerque, New Mexico, October 1997.

Meta-Amphion: Scaling up High-Assurance Deductive Program Synthesis

Steve Roach Recom Technologies NASA Ames Research Center Code IC, MS 269-2 Moffett Field, CA 94035 sroach@ptolemy.arc.nasa.gov	Jeff Van Baalen NASA Ames Research Center Code IC, MS 269-2 Moffett Field, CA 94035 jvb@ptolemy.arc.nasa.gov	Michael Lowry NASA Ames Research Center Code IC, MS 269-2 Moffett Field, CA 94035 lowry@ptolemy.arc.nasa.gov

Abstract

Amphion is a domain-independent program-synthesis system. It is specialized to specific applications through the creation of an operational domain theory. The Meta-Amphion system is being developed to empower domain experts to develop and maintain their own Amphion applications. *Operationalization*, a technology for automatically transforming declarative domain theories into efficient, domain-specific program synthesis systems, is described here.

A prototype of the system has been implemented in TOPS, *Theory Operationalization for Program Synthesis*. Sets of axioms in the domain theory are replaced by specialized procedures. TOPS uses partial deduction to augment the procedure with the capability to construct ground terms for deductive synthesis. The procedures are automatically interfaced to a resolution theorem prover.

Answers to deductive synthesis problem specifications can be generated using procedures synthesized by TOPS if and only if they can be generated without using the procedures. Experiments show that the procedures synthesized by TOPS can reduce theorem proving search at least as much as hand tuning of the deductive synthesis system.

1.0 Introduction

This paper is concerned with developing program synthesis systems, that is, software that automatically creates computer programs. Amphion is a domain-independent, high-assurance program synthesis system developed by NASA [Lowry et al. 94, Stickel et al. 94]. It is a generic system that is specialized to a particular application domain. This specialization requires the creation of a declarative domain theory consisting of a specification language, an output language, and knowledge relating the two languages. Amphion applications can be used to generate programs consisting of hundreds of lines of code, even by users who have no experience in program synthesis or even in programming. The programs are generated via deductive synthesis. An automated theorem prover mathematically proves a theorem, and a program is extracted from the proof. The program is a logical consequence of the domain theory. This provable correctness is an important feature of deductive program synthesis systems.

The term *combinatorial explosion* refers to the dramatic growth in the search space of automated theorem provers as problem specifications get larger. Amphion relies on the automated, first-order, resolution theorem prover, SNARK, to generate programs. In the past, Amphion applications have been limited by SNARK's combinatorial explosion. It is theoretically possible to build a declarative domain theory and use an automated theorem prover to find proofs. In practice, the combinatorial explosion makes it is difficult or impossible to obtain proofs for all but simple problems using such a domain theory. To make theorem proving tractable for moderate problems, the system must generally be tuned, a process called *operationalization.* Figure 1 shows that as the size of a specification increases, the search time for an un-tuned system grows exponentially, while the search time for a tuned system grows relatively slowly.

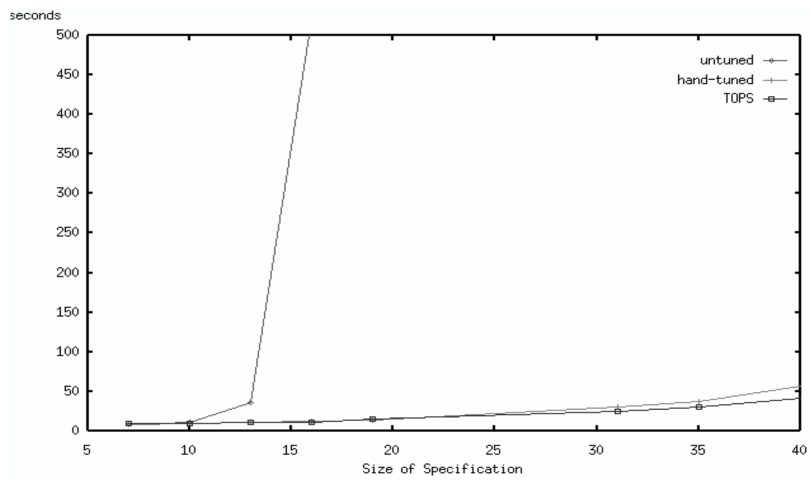

Figure 1: Time vs. Problem Size
Problem size increases along the x-axis. The time increases along the y-axis. The nearly vertical line is for an un-tuned system. The other two lines are for operationalized systems.

The typical approach to addressing combinatorial explosion in deductive synthesis is to operationalize the synthesis system to specific problem classes. The paradigm employed is to start with a set of problem specifications and observe the behavior of the theorem prover for each case. The system is operationalized by manually reformulating the domain theory and manually tuning the parameters of the theorem prover. This can be an extremely difficult and time consuming process. Further, reformulating axioms results in changing the domain theory from a simple declarative axiom set to an axiom set that depends on the characteristics of the theorem prover being used. Hence, the axioms become more difficult to verify and maintain, since they adopt a more operational than declarative characteristic. In the past, Amphion applications have been constructed by experts in deductive synthesis and have required substantial operationalization for each new domain. This has been the major impediment to the construction of Amphion applications.

Adequate tools to assist in creating and maintaining deductive synthesis systems are not yet available. Meta-Amphion is currently being developed to fill this void [Lowry and Van Baalen 97, Roach 97]. The research described here has resulted in a prototype of a tool, *Theory Operationalization for Program Synthesis* (TOPS). TOPS takes a formal, declarative domain theory as input and automatically generates a domain-specific deductive synthesis system. It transforms an inefficient system based on general-purpose inference rules into an efficient system based on special-purpose algorithms and data structures comparable to a system that would be hand-crafted by an expert. Manual tuning of parameters and manual reformulation of domain theories is replaced by automatic generation of specialized procedures that eliminate much of the search associated with program synthesis.

In contrast to this usual paradigm for theorem proving applications, TOPS analyzes the domain theory and generates procedures based on this analysis. The analysis is independent of test cases and depends only on the properties of the axioms in the domain theory. TOPS demonstrates the feasibility of automatic operationalization of deductive synthesis domain theories.

2.0 Deductive Synthesis

Deductive synthesis systems take a specification of the form $\forall(\vec{x})\exists(\vec{y})(P(\vec{x}) \rightarrow R(\vec{x}, \vec{y}))$ and prove a theorem of the form $\forall(\vec{x})(P(\vec{x}) \rightarrow R(\vec{x}, f(\vec{x})))$ [Green 69, Manna and Waldinger 92]. Here, P is some set of preconditions on the input variables $\vec{x}$; $\vec{y}$ is a set of outputs; and R is a set of post conditions constraining the outputs in terms of the inputs. The theorem prover constructs the term $f(\vec{x})$ that computes the outputs. For example, suppose we have a domain theory with two axioms, $x = x$ and $g(u) = h(f(u))$ [1], and a specification of the form $\forall(in)\exists(out, x)(g(in) = x \wedge h(out) = x)$. This specification says that for some x, x is equal to both $g(in)$ and $h(out)$. The variable out is the output. Obviously the substitution $\{out \leftarrow f(IN)\}$ makes the specification a theorem of the domain theory. Furthermore, it shows how to compute the value of out from the universally quantified input in.

Many automated theorem provers work by refutation. A refutation proof is finished when it shows that the negation of the goal is unsatisfiable when combined with some theory. More precisely, a refutation proof of a formula ϕ from a set of formulas Φ derives false from $\Phi \cup \{\neg\phi\}$. The set of formulas Φ is the domain theory. Resolution is an inference rule that is used to derive new formulas that are logical consequences of a set of formulas in clause form. A resolution proof is obtained by repeated application of the resolution rule. By definition, the empty clause is equivalent to false, thus the proof is finished when the theorem prover derives the empty clause.

2.1 Amphion

Amphion is a general purpose, deductive, program synthesis system that greatly facilitates reuse of domain-oriented software libraries. It enables a user to state a problem in an abstract, domain-oriented vocabulary using a graphical notation. Then it automatically generates a program that implements a solution to the problem specification. The generated program consists of assignment statements and calls to subroutines from the software library. It takes significantly less time for an experienced user to develop a problem specification with Amphion than to manually generate and debug a program. More importantly, a user does not need to learn the details of the components in the library before using Amphion to create useful programs. This removes a significant barrier to the use of software libraries.

Amphion is described in detail in [Lowry et al. 94, Stickel et al. 94]; an overview is presented here. Amphion consists of three subsystems: a specification acquisition subsystem; a program synthesis subsystem; and a domain-specific subsystem. The specification acquisition and program synthesis subsystems are generic across domains.

Amphion's specification acquisition system includes a graphical editor that enables a user to interactively build a diagram representing a formal problem specification in first-order logic. The editor provides an intuitive interface that interactively guides the user in the creation of a specification. The menu system is driven by type information extracted from the domain theory.

In general, specifications are given at an *abstract* level, and programs are generated at a *concrete* level. Abstract objects are free from implementation details; thus, geometric points or lines

[1] For example, the function g could take a real number and interprets it as a distance in meters, and the function h could take a real number and interprets it as a distance in feet. Then f is a function that converts between feet and meters.

are abstract concepts while arrays of real numbers in FORTRAN are concrete, implementation level constructs.

In Amphion, graphical specification diagrams are equivalent to specifications of the form: $\forall(\vec{x})\exists(\vec{y})(C1 \wedge \ldots \wedge Cn)$. Here $\vec{x}$ is a vector of universally quantified input variables and $\vec{y}$ is a vector of output and intermediate variables existentially quantified within the scope of the input variables. The conjuncts are all expressed in the abstract specification language, except for conjuncts expressing the relationships between concrete input or output variables and the abstract variables they represent.

The program synthesis subsystem consists of the SNARK theorem prover and a translator that generates code in the syntax of the target programming language [Stickel et al. 94]. A functional (applicative) program is generated through deductive synthesis [Manna and Waldinger 92]. During a proof, substitutions are generated for the existential variables through unification and equality replacement. The substitutions for the output variables are constrained to be terms in the target language whose function symbols correspond to the components of the library. The functional program is translated into a target programming language, such as FORTRAN or C++, through program transformations.

The domain specific components of an Amphion application consist of a domain theory, a subroutine library, and a set of theorem proving tactics. An Amphion domain theory has three parts: an abstract theory whose language is suitable for problem specifications, a concrete theory that includes the target component descriptions, and an implementation relation between the abstract and concrete theories. These implementation relations are axiomatized through abstraction maps as described by Hoare [Hoare 73]. theorem proving tactics guide SNARK from abstract, specification-level constructs towards concrete, implementation-level constructs. The tactics are implemented by defining an agenda-ordering function. The agenda is an ordered list of supported clauses, that is, clauses that are descendants of the negated goal. The agenda-ordering function is a heuristic function that puts the clauses most likely to lead to a refutation at the top of the list. As shown in Figure 1, the theorem proving tactics are highly effective, reducing program synthesis times from hours, or sometimes days, to minutes. The tactics do not need to be tuned for individual problems, but often do require expert manual tuning when a domain theory is modified [Lowry et al. 94].

Amphion has been applied to three domains at NASA. Amphion/NAIF solves problems in solar system geometry. Another application, Amphion/CFD, generates programs that solve problems in computational fluid dynamics, while Amphion/TOT solves routine problems in space shuttle navigation. Amphion/NAIF has generated programs that are in use by space scientists, including programs that perform geometry calculations and construct animations to assist in planning for the upcoming Cassini mission to Saturn.

2.2 Meta-Amphion

Meta-Amphion is a set of tools which enable domain experts to create and maintain Amphion applications. In the past, applications have been created by deductive synthesis experts. This has been a significant bottleneck to the development of program synthesis systems. Empowering domain experts (instead of deductive synthesis experts) to maintain Amphion applications is the key capability that must be achieved in order for Amphion applications to become widely used. In order for a program synthesis system to have a language and abstractions intuitive to domain practitioners, the responsibility for the creation and maintenance of these systems must lie with the domain experts who create and maintain software libraries.

Meta-Amphion is currently under development. Its key components will be a user interface to guide domain experts in creating and extending a domain theory, a subsystem to check the consistency of axioms created by the domain expert, and a subsystem to automatically operationalize a domain theory. This last subsystem will eliminate the need for program synthesis experts to tune the domain theory. In the implemented prototype of TOPS, manual tuning of complicated agenda-ordering functions is replaced by automatic generation of specialized procedures. Manual reformulation of a domain theory is replaced by automatic reformulation of axioms during the process of procedure generation.

3.0 Procedures for Theorem Proving

Resolution refutation is universal in the sense that any provable first-order formula is provable using general resolution. However, resolution is a syntactic rule and is unable to make any use of semantic knowledge about the symbols it manipulates. A promising approach to operationalizing deductive synthesis systems is to add specialized procedures to the theorem prover to eliminate undirected search [Nelson and Oppen 79, Shostak 84, Van Baalen 92]. A *satisfiability procedure* for a logical theory T is an algorithm for determining whether or not a formula is satisfiable in T. *Theory resolution* and *RQ resolution* provide inference rules that utilize semantically based satisfiability procedures when constructing refutation proofs. These procedures can be identified based on axioms present in the domain theory, as opposed to tuning tactics based on the observed performance of the theorem prover.

Resolution allows the derivation of a resolvent formula from two given formulas. This resolvent is logically implied by the conjunction of its parents, that is, whenever the parent formulas are both true, the resolvent formula is also. As shown in the left column of Table 1, given two parent formulas (in clause form) containing complementary literals (R in the first parent and the negation of R in the second), a resolvent can be formed. A refutation proof is a proof by contradiction. A proof starts with the axioms of the theory and the negation of the goal. Resolution is used to derive an unsatisfiable clause (the contradiction).

Theory resolution generalizes resolution [Stickel 85]. Theory resolution relaxes the requirement that the literals be syntactically complementary. Instead, theory resolution finds the smallest conjunction of literals that is unsatisfiable with respect to some theory. For example, suppose we have a theory that enforces a strict ordering on the domain of the predicate symbol '>'. Then we know that it is not possible for both $a>b$ and $b>a$ to be true. Thus the theory resolvent $P \vee Q$ is derived for the two parent clauses in the center column of Table 1 because *(a>b) and (b>a)* is unsatisfiable in the background theory of '>'. In practice, a special procedure would be invoked to identify the unsatisfiable conjunction of literals.

Table 1: Resolution, Theory Resolution, and RQ Resolution

	Resolution	Theory Resolution	RQ Resolution
parent	$R \vee P$	$(a > b) \vee P$	$S1 \rightarrow R \vee P$
parent	$\neg R \vee Q$	$(b > a) \vee Q$	$S2 \rightarrow \neg R \vee Q$
resolvent	$P \vee Q$	$P \vee Q$	$(S1 \wedge S2) \rightarrow P \vee Q$

In RQ resolution (also called constrained resolution), a clause is written as an implication [Burckert 91]. The antecedent is called the *constraints* (or *restrictions*). Recall that for arbitrary first-order formulas α and β, the clause $\neg\alpha \vee \beta$ is equivalent to the formula $\alpha \rightarrow \beta$. For RQ resolution, the clause form of $S \cup \{\neg\phi\}$ is rewritten as a set of formulas of the form $\alpha \rightarrow \beta$ where α is a conjunction of literals and β is a disjunction of literals. A refutation proof is finished when a formula is derived for which β is unsatisfiable (empty) and α is valid. An example is given in the right column of Table 1. The two given clauses are written in implication form. The RQ resolvent $(S1 \wedge S2) \rightarrow P \vee Q$ is formed when it is determined that $(S1 \wedge S2)$ is satisfiable with respect to the background theory. The resolvent $P \vee Q$ can be formed if $(S1 \wedge S2)$ is valid[2]. In practice, a special procedure would determine the satisfiability and validity of the antecedent.

3.1 DRAT

A novel approach to the incorporation of procedures into a theorem prover was demonstrated by DRAT, Designing Representations for Analytical Tasks [Van Baalen 91, Van Baalen 92]. DRAT automatically identifies instances of theories within a domain theory and replaces the corresponding sets of axioms with instances of specialized procedures taken from a library of procedures. A problem posed to DRAT is a pair <T,Φ>, where T is a first-order theory and Φ is a set of queries. Technically, DRAT designs a ground literal satisfiability procedure, that decides for a theory whether a conjunction of ground literals is satisfiable. A literal satisfiability procedure is used to solve a problem by converting each query in Φ into a satisfiability question.

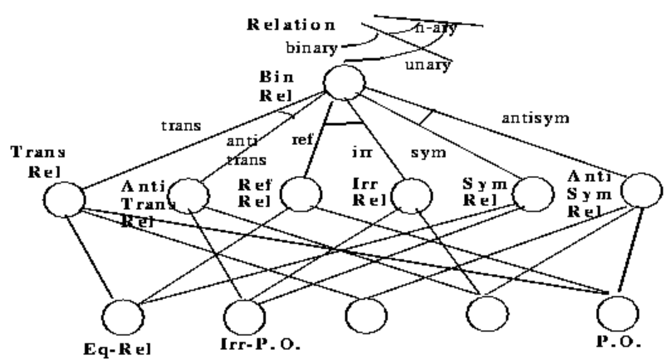

Figure 2: DRAT's Binary Relation Hierarchy

Given a problem, DRAT performs a semantic analysis of the non-ground axioms in T and replaces as many as possible with instances of specialized decision procedures. These decision procedures are interfaced to a general purpose theorem prover through theory resolution. The combined theorem prover/decision procedures and the remaining axioms perform the same inferences as the general purpose theorem prover with all the axioms, but much more efficiently. For example, one representative problem took over three hours to solve with a general-purpose resolution theorem prover. By contrast, DRAT automatically produced a theorem prover/decision procedure combination that solves the same problem in a few seconds.

DRAT uses a hierarchy of theories to classify the sort, relation and function symbols from T. The root nodes of the hierarchy are defined syntactically (e.g., unary function, symmetric binary relation), nodes lower down are semantically specialized (e.g., one-to-one function, partial order). The theory of a procedure at a node is the union of the axioms labeling the edges on paths from the root to the node. The hierarchy for binary relations is shown in Figure 2.

2. Recall that a formula is *satisfiable* if it is true in *some* model; it is *valid* if it is true in *all* models.

The theories are parameterized by nonlogical symbols. These are instantiated when DRAT selects a sort, relation, or function symbol from T. Symbols are classified by exploring edges from a root node. DRAT invokes a theorem prover to attempt to prove the instantiated axioms associated with an edge, given the theory of a problem. When the axioms can be proved, the edge is followed to the next node. Some edges are mutually exclusive, e.g., reflexive and irreflexive edges for binary relations. DRAT follows the edges as deeply as possible into the hierarchy, since more specialized theories are associated with more efficient decision procedures.

3.2 Procedures for Deductive Synthesis

SNARK has been extended to incorporate inference rules which facilitate the use of special procedures in finding a proof of a specification [Van Baalen and Cowles 97, Roach 97]. These procedures have two principle functions. First, they determine the satisfiability and validity of conjunctions of literals during deductive synthesis. Second, they construct ground terms which are bound to existential variables in the original specification. These bindings are propagated to SNARK and are incorporated in the answer term. All of SNARK's inference rules have been extended. These rules are resolution, paramodulation, subsumption, and rewriting. The resolution rule is similar to theory resolution and RQ resolution. Clauses are separated and written as constrained clauses, and a specialized procedure determines the satisfiability and validity of the constraints in the antecedent of a resolvent.

The separation of formulas is based on the language of the theory of the procedure and the language of the domain theory. Let Σ be the language of the domain theory and Γ be the language of the theory of a procedure. Then $\Gamma \subseteq \Sigma$. We want to separate each formula ϕ into a logically equivalent formula $\alpha \rightarrow \beta$ where the relation symbols of α are in the language Γ, and β is in the language $\Sigma - \Gamma$. Assume that ϕ is in clause form. For every literal δ in ϕ whose head symbol is a relation symbol in Γ, move the negation of δ into the conjunction α. Move the remaining literals into β. Now for each term τ in β that is in the language Γ, create a new variable x and a new equality $x=\tau$. Add this equality to α, and replace τ in β by x. This separation allows terms in the language $\Sigma - \Gamma$ to appear in α. The procedures will treat these terms as uninterpreted constants. For example, suppose there is a procedure for a theory whose language includes the symbols R and f. The separated form of the formula $\neg R(f(a), x) \vee \neg Q(y, f(z))$ is $R(f(a), x) \wedge (v1 = f(z)) \rightarrow \neg Q(y, v1)$ .This separation is similar to the separation in [Nelson and Oppen 79] and is described in [Roach 97].

The deductive synthesis procedures answer queries for deductive synthesis. Informally, a deductive synthesis query (DSQ) is a query of the form $\forall(\vec{x})\exists(\vec{y})(P(\vec{x}) \rightarrow R(\vec{x}, \vec{y}))$ where the signature of the query is restricted to the abstract portion of the domain theory. P and R are quantifier free formulas. R contains variables in the union of the inputs $\vec{x}$ and the outputs $\vec{y}$. All of the output variables have concrete sorts. Concrete function symbols are not allowed to appear in queries. Terms with concrete function symbols are program fragments, so ultimately this restriction prevents queries from mentioning partially instantiated programs. Note that a deductive synthesis procedure is not a decision procedure. Rather than determine the validity of arbitrary formulas, a deductive synthesis procedure only answers the types of queries encountered during deductive synthesis.

The operation of a deductive synthesis procedure is summarized as follows. The theorem prover is given a DSQ. The DSQ is negated, since a proof is constructed by refutation. Each formula is rewritten as an implication with the antecedent restricted to the language of the theory of the procedure. During resolution, the deductive synthesis procedure combines antecedents from parent formulas and decides the satisfiability of the new antecedent. When possible, the procedure generates ground terms which are bound to the output variables in the antecedent and passes those bindings to SNARK as substitutions. The refutation is found when a formula is derived for which the antecedents are valid and the consequents are unsatisfiable.

The process is complicated by the fact that the deductive synthesis procedure may itself be composed of multiple sub-procedures. In this case, the languages of the sub-procedures must be separable. If the concrete language of the theory of a sub-procedure is disjoint from the concrete language of the rest of the theory, then any term in the concrete language of the sub-theory generated for an existential variable is uninterpreted with respect to the remaining theory. Thus this term will work as well as any other such term. Therefore, the procedure does not need to generate any other terms.

3.3 Extending DRAT to Deductive Synthesis

The approach to operationalization taken by TOPS is to analyze a domain theory and replace axiom sets in the theory with specialized procedures. In this manner, TOPS replaces general purpose reasoning with directed, special purpose reasoning specifically targeted towards program synthesis. Using this system, a declarative domain theory is automatically converted to an operational domain theory using mathematically sound transformations.

TOPS extends DRAT to program synthesis. Like DRAT, the TOPS library contains parameterized procedure templates. An instance of a procedure (or simply "a procedure") is generated by providing values for the parameters. TOPS analyzes a domain theory to identify sets of axioms that are instances of the theory of a library procedure template. An instance of a procedure is specialized to generate ground instantiations for existential variables specifically in the context of deductive synthesis. The axioms captured by the procedure are removed from the domain theory, and the procedure is interfaced to the theorem prover. The theorem prover serves as an integrating framework for the sets of procedures as well as a procedure for axioms that are not captured by procedures.

DRAT was designed to produce procedures that solve analytical reasoning problems. TOPS is designed to produce procedures for deductive synthesis. Analytical reasoning problems are ground satisfiability problems (i.e., is a ground formula satisfiable in a given theory?). In contrast, deductive synthesis problems for Amphion are specifications given as pre- and post-conditions. In addition, TOPS' library includes procedures for computing the satisfiability of conjunctions of literals containing existentially quantified variables, as opposed to the satisfiability of ground literals.

In contrast to analytical reasoning problems, an important consideration for problems in deductive synthesis is the algebraic structure of output terms, that is, the equivalence classes of terms bound to output variables. For deductive synthesis we want the output terms to represent the "best" program in their equivalence class.[3]

Amphion domain theories also differ from analytical reasoning domain theories. The latter are unstructured and primarily relational. In contrast, Amphion domain theories are structured into an abstract and a concrete level, with abstraction maps between these levels. Conceptually, each TOPS satisfiability procedure solves a class of small program synthesis problems. Each procedure generates inferences that are logically implied by the theory of the procedure.

4.0 TOPS: Automatic Generation of Procedures

Through experience with the NAIF domain, several types of axioms have been identified as leading to combinatorial explosions in the theorem prover's search space. These axioms are used

3. Amphion assumes that the best programs are represented by the ground terms with the smallest number of function applications.

during deductive synthesis to construct terms that are bound to existential variables in the specification. TOPS is designed to identify these axiom sets. TOPS instantiates and composes satisfiability procedures from a library of procedure templates. Since axioms implied by the procedures are removed from the domain theory, the theorem prover does not generate multiple child formulas from a single parent, as it would without the procedures.

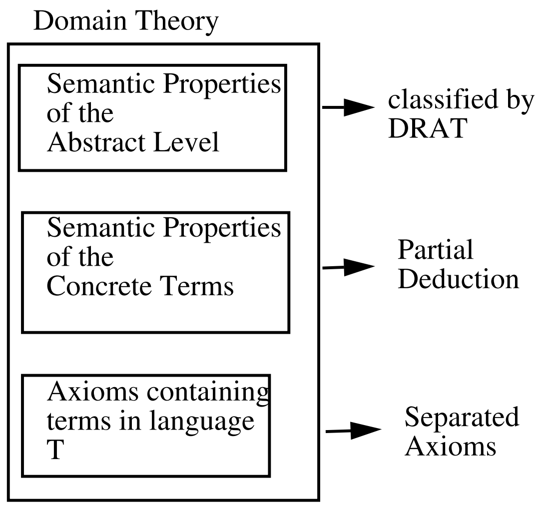

Figure 3: TOPS Processing Domain Theory Axioms

The basic function of TOPS is depicted in Figure 3. The DRAT hierarchy is used to classify abstract relation symbols in the domain theory. For each relation symbol classified, a procedure is selected from the DRAT library of procedures. TOPS augments the procedure by pre-computing terms using partial deduction and knowledge compilation [Komorowski 92, Selman and Kautz 96]. Axioms captured (implied) by the TOPS procedure are removed from the domain theory. Remaining axioms that contain terms in the language of the theory of the procedure are separated relative to this language. The six steps below outline the TOPS algorithm.

(1) Each abstract relation symbol is classified using the DRAT hierarchy. To classify a symbol, the theorem prover is called to prove each property in the hierarchy. For example, to classify a binary relation R as symmetric, the theorem prover is invoked to prove $\forall((xy)(Rxy) \rightarrow (Ryx))$. A procedure template is selected from the library of procedure templates.

(2) For each abstract relation symbol classified in the previous step, the functions mapping concrete sorts to the abstract sorts are identified, again by calling the theorem prover. These are used to construct program fragments from the abstract, specification-level terms. The construction of these terms leads to combinatorial explosion. The mappings identified here restrict TOPS to just "problem" axiom sets.

(3) At program synthesis time, each procedure must be able to determine satisfiability of formulas with respect to a theory. The language of this theory consists of the abstract relation symbol, the functions mapping concrete to abstract sorts, and concrete constants. In addition to determining satisfiability, the procedure must be able to construct ground, concrete terms. Therefore, the procedure must encapsulate enough information to be able to construct terms that are answers to any DSQ in the language of the theory of the procedure. TOPS captures this information by using partial deduction (invoking the theorem prover) to generate universally quantified partial answers to a set of DSQs. The set of DSQs has the property that every DSQ in the language of the theory of the procedure is an instance of some DSQ in the set.

(4) A procedure instance is created and integrated with the theorem prover. The procedure instance includes the abstract relation symbol, the functions mapping concrete to abstract sorts, and the set of partially deduced answers. TOPS uses the theorem prover/procedure combination to prove as many axioms in the domain theory as possible. Axioms proved using the procedure are "captured" by the procedure and are removed from the domain theory [Van Baalen 91].

(5) Axioms not captured by the procedure but that mention terms in the abstract language of the procedure are separated into antecedent/consequent form. This separation facilitates the use of the new inference rules for deductive synthesis.

(6) TOPS verifies that all of the necessary axioms have been captured by the procedure. Any axiom mentioning terms in the concrete portion of the language of the procedure must have been captured. If any remain, then the necessary separation of languages has not been obtained and TOPS will not produce a procedure.

4.1 Correctness of TOPS

TOPS is considered to generate correct procedures if the procedures are sound and complete. By "sound" we mean that any program generated using TOPS synthesized procedures can be generated without using those procedures. The properties and terms used by each procedure are derived from the domain theory by a sequence of proofs. Axioms removed from the domain theory are proved using the procedure. Thus the procedure is as strong, but only as strong as the removed axioms. More formally, any answer derived using the TOPS procedures is a logical consequence of the original domain theory.

By "complete" we mean complete with respect to DSQs. Given a DSQ for which an answer can be derived without using the procedures, an answer can be derived using the procedures. This does not imply that the answers are identical, just that the theorem prover will find some answer using the TOPS procedures. The procedure from the DRAT library provides inferences at the abstract level. For example, the procedure for a symmetric binary relation takes advantage of the fact that the order of the related terms can be reversed. The set of partially deduced answers allows the procedure to construct concrete-level terms. Since DSQs do not contain program fragments, the procedure only needs to construct concrete-level terms. It does not need to reason about them. For example, it is not necessary for the procedure to determine whether two concrete terms are equal in a given theory. Since the languages are separated, ground terms generated by any procedure are uninterpreted outside the procedure.

5.0 Results

TOPS' compilation algorithm is highly effective. The results are better than even those obtained by manual tuning of theorem proving tactics. A prototype of TOPS was applied to one release of the NAIF domain theory. This domain theory consists of 330 first-order axioms that define the abstract specification language, the pre- and post-conditions for a set of FORTRAN routines in the NAIF tool kit, and the abstraction mappings between the concrete and abstract sorts. To test TOPS, we compared the performance of three deductive synthesis systems: a TOPS-generated system, an untuned system, and a system manually tuned to the NAIF domain by program synthesis experts.

A series of 27 specifications were used. These specifications ranged from trivial with only a few literals to fairly complex with dozens of literals. Half of the specifications were obtained from domain experts, thus this set is representative of the problems encountered during real-world use. Figure 1 compares the time required by SNARK to find an answer for a specification to the number of literals in the specification. The untuned system shows exponential behavior, and the search for a solution quickly became intractable. The hand-tuned system and the TOPS-synthesized system both show more reasonable behavior, with the TOPS system finding proofs somewhat faster than the hand-tuned system.

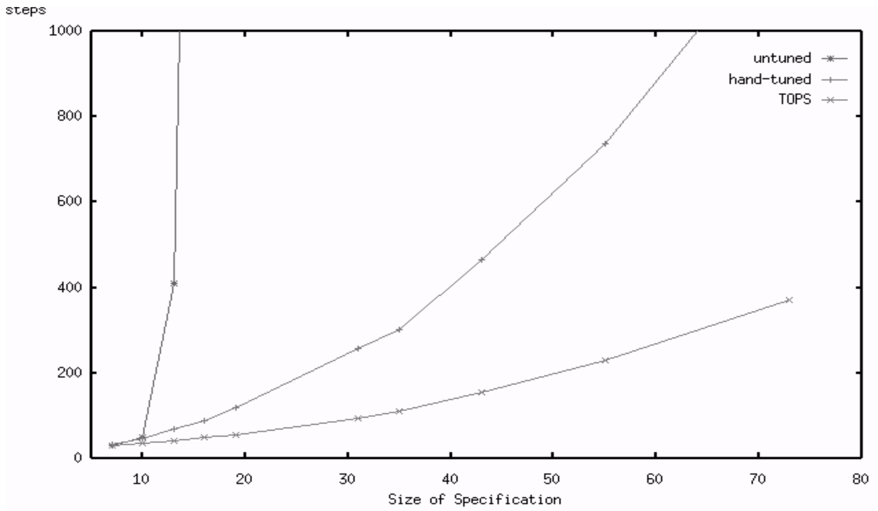

Figure 4: Inference steps vs. Problem Size.
Problem size increases along the x-axis. The number of inference steps required to find a solution increases along the y-axis. The TOPS-generated procedures perform better than even the hand-tuned system.

Figure 4 compares the number of inference steps that each configuration required to find a proof of specifications (the y axis) with varying numbers of literals in the specification (the x-axis). This graph clearly shows the exponential nature of deductive synthesis when using only general-purpose tactics. The manually-tuned and TOPS-generated configurations scale relatively well. This also shows that the TOPS system outperformed even the hand-tuned system, and that the number of steps that the TOPS system required to find a proof grew at about one third the rate of the hand-tuned system.

6.0 Conclusions

Program synthesis tools based on deductive synthesis are high assurance, but they have been severely limited by their inefficiency and inability to scale up to larger specifications. Manual methods for increasing their efficiency by tuning of tactics and strategies makes it very difficult to maintain them in the context of a changing target domain.

We have presented empirical results based on a prototype system for automatically compiling domain theories into procedures for deductive synthesis. This system generates procedures specifically for deductive synthesis in Amphion-structured domain theories. The results show that the compiled system is more efficient than even systems hand-tuned by theorem proving experts.

Acknowledgments

Thanks to Tom Pressburger and John Cowles for assisting with the technical content and presentation of this paper.

References

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

[Green 69] C. Green, "Applications of Theorem Proving," *IJCAI 69*, 1969, pp. 219-239.

[Hoare 73] C.A.R. Hoare, "Proof of Correctness of Data Representations," *Acta Informatica*, 1973, pp. 271-281.

[Komorowski 92] J. Komorowski, "An Introduction to Partial Deduction Framework," in *Meta-Programming in Logic*, *Lecture Notes in Artificial Intelligence*, Vol. 649, Springer-Verlag, 1992, pp. 49-69.

[Lowry *et al.* 94] M. Lowry, A. Philpot, T. Pressburger, and I. Underwood, "A Formal Approach to Domain-Oriented Software Design Environments," *KBSE*, 1994.

[Lowry and Van Baalen 97] M. Lowry and J. Van Baalen, "META-Amphion: Synthesis of Efficient Domain-Specific Program Synthesis Systems", *Automated Software Engineering*, vol. 4, 1997, pp. 199-241.

[Manna and Waldinger 92] Z. Manna and R. Waldinger, "Fundamentals of Deductive Program Synthesis," *IEEE Transactions on Software Engineering*, Vol. 18, No. 8, August 1992, pp. 674-704.

[Nelson and Oppen 79] G. Nelson, and D. Oppen, "Simplification By Cooperating Decision Procedures," *ACM Transactions on Programming Languages and Systems*, No. 1, 1979, pp. 245-257.

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

[Selman and Kautz 96] B. Selman and H. Kautz, "Knowledge Compilation and Theory Approximation", *JACM*, Vol. 43, No. 2, March 1996, pp. 193-224.

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

[Stickel 85] M. Stickel, "Automated Deduction by Theory Resolution," *Journal of Automated Reasoning*, Vol. 1, 1985, pp. 333-355.

[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.

[Van Baalen 91] J. Van Baalen, "The Completeness of DRAT, A Technique for Automatic Design of Satisfiability Procedures," *International Conference of Knowledge Representation and Reasoning*, 1991.

[Van Baalen 92] J. Van Baalen, "Automated Design Of Specialized Representations," *Artificial Intelligence*, Vol. 54, 1992.

[Van Baalen and Cowles 97] J. Van Baalen, J. Cowles, private communication, 1997.