Experience Report on Automated Procedure Construction for Deductive Synthesis

Steve Roach
Department of Computer Science
University of Texas at El Paso
El Paso, TX, 79968
sroach@cs.utep.edu
and
Jeffrey Van Baalen
Department of Computer Science
University of Wyoming
Laramie, WY 82071
jvb@uwyo.edu

Abstract

Deductive program synthesis systems based on automated theorem proving offer the promise of "correct by construction" software. However, the difficulty encountered in constructing usable deductive synthesis systems has prevented their widespread use. Amphion is a real-world, domain-independent program synthesis system. It is specialized to specific applications through the creation of an operational domain theory and a specialized deductive engine. This paper describes an experiment aimed at making the construction of usable Amphion applications easier.

The software system Theory Operationalization for Program Synthesis (TOPS) has a library of decision procedures with a theory template for each procedure. TOPS identifies axioms in the domain theory that are an instance of a library of procedure and uses partial deduction to augment the procedure with the capability to construct ground terms for deductive synthesis. Synthesized procedures are interfaced to a resolution theorem prover. Axioms in the original domain theory that are implied by the synthesized procedures are removed.

During deductive synthesis, each procedure is invoked to test conjunctions of literals in the language of the theory of that procedure. When possible, the procedure generates ground terms and binds them to variables in a problem specification. These terms are program fragments. 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 Introduction

Deductive program synthesis technology has been known since the late 1960's [3], [9]. However, construction of usable deductive synthesis systems is difficult. In the past thirty years, a great deal of progress has been made in the development of program synthesis systems based on theorem proving, transformations, and logic programming [3], [12], [4]. However, in spite of this progress, these techniques are not in the mainstream of software development. This paper describes the results of an experiment aimed at making the construction of usable deductive synthesis systems easier.

Amphion [7], [14] is a real-world, domain-independent program synthesis system developed by NASA. A user of an Amphion application interactively creates a formal specification for a program by declaring the inputs, the outputs, and relationships between them. Amphion applications take these types of specifications and generate source-code-level computer programs consisting of calls to components of target subroutine libraries. Amphion applications have been used to generate programs, primarily consisting of assignments and procedure calls, containing hundreds of lines of code.

The first step in creating an Amphion application is to write a declarative domain theory consisting of an abstract specification language, a concrete output language, and knowledge relating the two. At this point, one could, in principle, generate programs by proving theorems. For instance, Amphion applications use SNARK [15], which is a refutation theorem prover. However, general-purpose theorem provers like SNARK are subject to exponential growth in the search space required to find a proof. To make an Amphion application usable, the system must generally be modified to reduce the search required for completing proofs, a process often called *operationalization*.

Operationalization is itself difficult, time consuming, and requires expertise in program synthesis, familiarity with the domain, and knowledge of the operational semantics of SNARK. In the past, Amphion applications have been constructed by experts in deductive synthesis and have required substantial operationalization for each new domain. Further, the addition or modification of axioms frequently entails re-operationalizing the system. Axioms in a domain theory may be coupled to other axioms in ways that are difficult to predict *apriori*. This has been the major impediment to the construction and maintenance of Amphion applications.

This paper describes our experience with a prototype of a software tool, *TOPS*, *Theory Operationalization for Program Synthesis*. TOPS takes a declarative domain theory as input and automatically generates an efficient, domain-specific synthesis system comparable to a system that would be hand-tuned by an expert.

The usual methods of operationalization fall into three categories: (1) providing general theorem proving strategies, tactics and heuristics; (2) reformulation of axioms in the domain theory; and (3) incorporating new inference rules and special purpose reasoning mechanisms.

While strategies and tactics are helpful, general methods are inadequate for all but simple problems [8]. Heuristics are *ad hoc* rules for selecting axioms upon which to act. Reformulation of axioms in the domain theory to force the theorem prover to apply inference rules in a specific order or in a specific way changes a simple declarative domain theory into one that depends on the characteristics of the theorem prover being used. The axioms become more difficult to verify and maintain, since they adopt a more operational than declarative character. Hence, we would rather leave the domain theory declarative and operationalize it by some automatic process.

TOPS employs a combination of methods 2 and 3 to automatically operationalize a declarative domain theory. It automatically incorporates special-purpose reasoning mechanisms into the general-purpose theorem prover based on axioms it finds in a domain theory and reformulates the remaining domain theory. Suppose the original domain theory is T and SNARK can, in principle, perform proofs of the form $T \vdash \Phi$, where Φ is a formula of restricted form (to be defined later) in the language of T. Then, TOPS designs a special-purpose inference mechanism $\vdash_{T_1}$ that can perform proofs of the form $\mathit{reform}(T - T_1) \vdash_{T_1} \mathit{reform}(\Phi)$, where *reform* represents the reformulation process.

TOPS designs a special-purpose inference mechanism from a library of parameterized, special-purpose inference procedures. Each such procedure is annotated with a parameterized theory. TOPS identifies subtheories in a domain theory that imply instances of procedure theories. Each such subtheory is removed from the domain theory and, in its place, an instance of the identified procedure is interfaced to the theorem prover. Hence, $\vdash_{T_1}$ is built by instantiating and composing library procedures and interfacing these with SNARK.

Some previous efforts at incorporating special procedures into automated theorem provers have resulted in extremely complex procedures with a large amount of communication required between the theorem prover and the procedures [1]. The complexity of the interface between the theorem prover and the procedures can negate much of the performance gained by the procedures. In contrast, TOPS' synthesized procedures limit the interaction between the procedures and the theorem prover by separating the non-logical symbols of the theories of the procedures. This is possible because the procedures are fine-grained: they decide relatively simple theories.

The next section describes DRAT, a technique for designing procedures for ground satisfiability problems, and then explains how TOPS extends DRAT to automatically operationalize a domain theory for deductive synthesis. Section 3 provides some background on the NAIF domain for solving solar-system geometry programs. Section 4 presents TOPS, and Section 5 presents experimental results of using TOPS on the NAIF domain. Finally, Section 6 summarizes.

2 Interfacing Decision Procedures

2.1 DRAT

The technique described here was first introduced by DRAT, Designing Representations for Analytical Tasks [16], [17]. A problem posed to DRAT is a pair $\langle T, F\rangle$, where T is a sorted, first-order theory and F is a set of queries. A *possibility* query asks whether $T \cup f$ is satisfiable, where $f \in F$ is a ground formula. *Ne essity* queries ask whether $T \Rightarrow f$ or, equivalen tly,whether $T \cup \neg f$ is unsatisfiable.

Given a first-order specification of a problem, DRAT analyses the non-ground axioms in T and replaces as many as possible with instances of decision procedures for ground satisfiability. These procedures decide for some theory whether or not a ground formula is satisfiable in that theory. DRAT has a library of parameterized decision procedures which it instantiates and in terfaces to a general-purpose theorem-prov er. Because many of the axioms of the original theory are replaced by efficient and directed decision procedures, problems are solved muc h more efficiently by the theorem prov er/decision procedure combination designed by DRAT.

The parameters of DRAT's library procedures are the nonlogical symbols of the procedure's theory. F or example, one procedure decides the satisfiability of ground formulas in v olving an equivalence relation R. Hence, the procedure would indicate that the conjunction $R(a, b) \wedge R(b, c) \wedge \neg R(a, c)$ is unsatisfiable. In this case, R is this procedure's parameter and is instantiated with a relation symbol from the domain theory when DRAT interfaces the procedure to SNARK. The procedures decide satisfiability very efficiently by represen ting ground facts in a data structure that encodes the relevant properties. F or example, the procedure for R abov e represents ground fact inv olving R in a directed graph that encodes reflexivity, symmetry, transitivit y in its structure.

Different strategies are employ ed to construct graphs for different type of formulas. For example, when the formula is a conjunction of literals, a graph is constructed from the positive literals in the conjunction and then the formula is satisfiable just in case the graph does not contain a positive occurrence of one of the negative literals in the conjunction.

Decision procedures are interfaced to SNARK through *separated* inference rules which are based on R Q resolution [2]. They w ork with clauses that are separated relative to a subtheory.

Definition (Separated Clause) Let L be the language of a theory T, a sorted first-order theory with equality. We treat equality as a logical symbol, so $= \notin L$. Let $L_1 \subseteq L$ be the language of $T_1 \subseteq T$. A clause C with the follo wing properties is said to be *separated relative to* T_1:

1. C is arranged into $C_1 \vee C_2$, where both C_1 and C_2 are disjunctions of literals (i.e., clauses).

2. All the function and relation symbols in C_1 come from L_1 and all the function and relation symbols in C_2 come from $L - L_1$.

Notice that $C_1 \vee C_2$ can be written $\overline{C_1} \Rightarrow C_2$, where $\overline{C_1}$ is the negation of C_1. Since C_1 is a disjunction of literals, $\overline{C_1}$ is a conjunction of literals. $\overline{C_1}$ is called the *ante cedent* of the separated clause and C_2 is called the *consequent.*

DRAT designs a collection of decision procedures for T_1 a subset of a theory T, removes the axioms of T_1, and separates the remaining axioms relative to T_1.

A separated refutation of a set of separated clauses T is a sequence of separated clauses K where each $K_i \in K$ is an element of T or is derived from one or more of the previous K_i b y *separate d resolution* or *separate d paramodulation.* Separated resolution is an extension of standard binary resolution. Two literals of opposite polarit y in the consequents of separated clauses are unified. The unifier is applied to the antecedents, and the an tecedents are conjoined. If the antecedent is satisfiable, the consequent is formed in the standard w ay. Separated paramodulation is a similar extension to paramodulation.

A separated refutation is *closed* when a set of clauses of the form $\{\overline{C_i} \Rightarrow \Box, \ldots, \overline{C_{i+m}} \Rightarrow \Box\}$ is derived such that $T_1 \models \exists(\overline{C_i} \vee \cdots \vee \overline{C_{i+m}})$, where $\Box$ denotes the empty clause and $\exists$ denotes existential closure. In some cases, the structure of the theory T_1 and antecedents of the separated clauses in $T - T_1$ is suc h that, whenever T is unsatisfiable, a separated refutation can be found that ends in a single clause. One such case is when T_1 is Horn and the an tecedents of all clauses con tain only positive literals. In this case, we need only check for a single clause whose antecedent (a conjunction of literals) is a theorem of T_1 and whose consequent is the empty clause [2, p. 71]. The theories of DRAT's library procedures are all Horn, and so we restrict our considerations in this paper to this case.

The theorem prover with interfaced decision procedures is used to solve a problem by attempting to find a separated refutation. The procedures can decide the satisfiability of ground antecedents. When a new separated clause is derived b y SNARK and its antecedent is ground, the decision procedures are inv oked to c heck satisfiability. If the an tecedent is unsatisfiable in T_1, the clause is discarded.

When SNARK derives a separated clause whose consequent is empty and whose antecedent is ground, the decision procedures are inv oked to check that the antecedent is a theorem of T_1 by checking that its negated an tecedent is unsatisfiable in T_1.

Note that since the queries given in a problem specification to DRAT are all ground and since SNARK uses set-of-support strategy to deriv e clauses with empty consequents, a clause with empty consequent and ground antecedent will always be derived when $T \cup \neg f$ is unsatisfiable. As explained in the next section, this will not be the case when deductive synthesis problems are considered.

2.2 Extending DRAT for Deductive Synthesis

TOPS extends DRAT to program synthesis. Like DRAT, TOPS analyzes a domain theory to identify sets of axioms that are instances of the theory of a library procedure. These axioms are removed from the domain theory, and synthesized procedures are interfaced to the theorem prover.

DRAT was designed to produce ground satisfiability procedures. In contrast, TOPS is designed to produce procedures for deductive synthesis. In this case instead of being ground, the antecedents of separated clauses can contain variables that must be "filled in" with ground terms. More precisely, when a separated clause with empty consequent is derived, the procedures must prove that the existential closure of the antecedent is a theorem of T_1. It is also required that witnesses be produced for the variables in the antecedent.

To see why variables appear in antecedents when proving theorems for deductive synthesis, note that in a deductive synthesis problem, theorems 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 P is a quantifier-free formula. The variables in $\bar{x}$ represent the inputs to the program that is synthesized. By the first-order rule of universal generalization, we can substitute freshly generated constant symbols $\bar{c}$ for the $\bar{x}$, obtaining $\exists \bar{y}[P(\bar{c}, \bar{y})]$. Hence, theorems of this later form are what must be proved, and witnesses that serve as program fragments must be identified for each $y \in \bar{y}$.

TOPS uses the same procedure library as DRAT. However, during the procedure identification and instantiation process, TOPS generates a wrapper around the DRAT procedure so that it can answer two types of queries. First, the extended procedure can determine that the existential closure of a clause's antecedent (a conjunction of literals) is unsatisfiable in its theory. Second, it can generate witnesses demonstrating that such a closure is a theorem of its theory. In the later case, the procedure is actually used to demonstrate the unsatisfiability of the negation of the existential closure of a conjunction of literals. In this case, the problem is to find witnesses for the variables that demonstrate that a disjunction of literals is unsatisfiable.

TOPS generates the extentions to find witnesses using a type of partial deduction explained in Section 4. Before this is explained, some background on the NAIF domain is helpful.

3 Amphion/NAIF

Amphion/NAIF is an Amphion application that solves problems in solar system geometry. It has been used to generate programs that are in use by space scientists, including programs that perform geometry calculations used to assist in planning for the Cassini mission to Saturn. This section describes Amphion/NAIF and provides background for the description of TOPS.

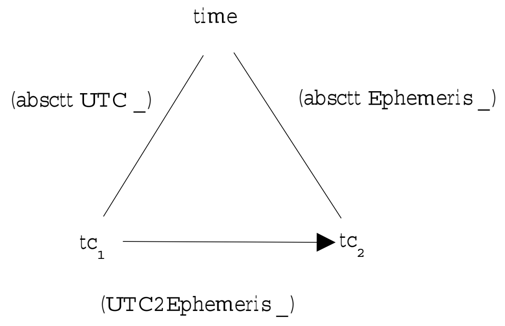
Amphion synthesizes programs from specifications written in a domain-specific first-order language. As mentioned before, the specifications have the form $\forall \bar{x} \exists \bar{y}[P(\bar{x}, \bar{y})]$, where $\bar{x}$ and $\bar{y}$ are vectors of variables and P is a quantifier-free formula.

The synthesized programs are composed of subroutines from the NAIF SPICE[10] subroutine library. The specification language is designed for domain experts who may not be familiar with the SPICE library. It has elements such as points, rays, planes, and celestial bodies (e.g. Saturn). This language is referred to as *abstract* and is independent of the program representation of these elements. During program synthesis, witnesses are constructed for the existential terms in the specification. These witnesses are restricted to terms that correspond to elements of the SPICE subroutine library. These terms are called *concrete*. The only time a user is concerned about the concrete level is when defining the program input and output. We refer to the signature of the abstract and concrete language of a domain theory as ΣA_T and ΣC_T respectively.

The domain theory also contains axioms that relate abstract and concrete terms. Some of the functions in the domain theory have concrete arguments, but have abstract sorts. These functions are called *abstraction functions* as they map concrete objects to abstract objects.

In the remainder of this section, we introduce a part of the NAIF domain theory that will be used in examples in the following sections. Experience with the NAIF application has revealed two types of axioms that lead to combinatorial explosions in the theorem prover's search space. These axioms are the representation conversions and abstract constraints that have concrete functions associated with them.

A representation conversion is a function that takes a concrete representation and produces an equivalent concrete representation. For example, SPICE supports a number of time formats such as UTC-Calendar (a string of the form "YYYY MMM DD HH:SS") and ephemeris time (a double precision floating point number). Certain SPICE routines require specific formats. SPICE provides routines for converting between formats. The diagram below describes one such common conversion.



In this diagram, tc_1 and tc_2 are concrete time coordinates. In a program, these might be variables holding a representation of a point in time. [1] *Time* is abstract. It is free of representation details. In a specification, *time* might be used to specify the time of an event such as when an observation is made.

The function *absctt* is an abstraction function that takes a time coordinate and a time system and returns an abstract time. Thus ($absctt\ UTC\ tc_1$) says "take tc_1 and interpret it as a time in UTC coordinates." This diagram says the time obtained by interpreting tc_1 as a time in UTC coordinates is the same as the time obtained by interpreting tc_2 as a time in *Ephemeris* coordinates. tc_2 is equal to the evaluation of ($UTC2Ephemeris\ tc_1$). The following axiom expresses the same relation.

$$\forall tc (= (absctt\ UTC\ tc)\\ \quad (absctt\ Ephemeris\ (UTC2Ephemeris\ tc)))$$

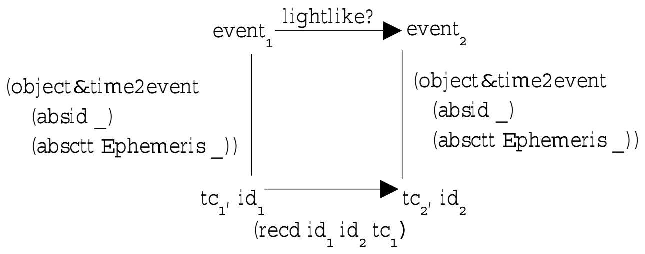
The following axiom expresses the inverse function.

$$\forall tc (= (absctt\ Ephemeris\ tc)\\ \quad (absctt\ UTC\ (Ephemeris2UTC\ tc)))$$

The presence of representation conversion axioms in a domain theory causes a combinatorial explosion in theorem prover search because these axioms imply an infinite number of different conversion sequences for getting from one coordinate system to another. As a result, when the theorem prover must construct a witness for an existentially quantified variable that involves converting between coordinate systems, there are many possible witnesses and adequate way to prefer one witness over another based on syntactic properties of formulas.

The NAIF domain theory for light time corrections provides an example. To compute the angle at which sunlight strikes the surface of Saturn at a given time t, it is necessary to know the position of the Sun relative to Saturn when the light at Saturn at time t left the Sun. The program must find the location of Saturn at time t and the location of the Sun at time $t - \Delta t$, where Δt is the time it takes light to travel from the Sun to Saturn.

To describe light-time corrections, the NAIF specification language uses an abstract *event*. An event is a location in space at a particular time, for example the location of a spacecraft at the time a photograph is taken. The abstract relation for describing the light-time constraint is *lightlike?*. Two events are *lightlike?* if a photon or ray of light leaving the first event arrives at the second event. At the concrete level, there are two functions that compute times based on the *lightlike?* constraint. These are *sent* and *recd*. Given the location and time of origin ($event_1$) and a destination location, *recd* computes the time at the destination ($event_2$) so that ($lightlike?\ event_1\ event_2$). Given a location and a time of a destination ($event_2$) and the location or the origin, *sent* computes the time at the origin ($event_1$) so that ($lightlike?\ event_1\ event_2$). The following diagram describes *recd*.



The relevant axioms in the NAIF domain theory are

$$\forall\ onid\ dnid\ ets\\ (lightlike?\\ \quad (obj\&time2event\\ \quad\quad (absid\ onid)\ (absctt\ Ephemeris\ ets))\\ \quad (obj\&time2event\\ \quad\quad (absid\ dnid)\\ \quad\quad (absctt\ Ephemeris (recd\ onid\ dnid\ ets))))$$

$$\forall\ onid\ dnid\ ets\\ (lightlike?\\ \quad (obj\&time2event\\ \quad\quad (absid\ onid)\\ \quad\quad (absctt\ Ephemeris\ (sent\ onid\ dnid\ ets)))\\ \quad (obj\&time2event\\ \quad\quad (absid\ dnid)\ (absctt\ Ephemeris\ ets)))$$

[1] In the NAIF domain, time is considered to be a total order. That is, for any two times t_1 and t_2, either $t_1 = t_2$, $t_1 > t_2$, or $t_1 < t_2$. Relativistic effects are ignored.

The presence of the *lightlike*? axioms in a domain theory causes combinatorial explosion problems similar to those causes by representation conversion axioms.

4 TOPS

TOPS takes a domain theory and a library of procedure templates as input. The output from TOPS is a modified domain theory and a set of procedure instances. We refer to the execution of TOPS as *theory compilation*, a process that is similar to partial deduction [6], [4], and knowledge compilation [5], [13]. Amphion uses the modified theory and executes procedures to synthesize programs to solve problems in the NAIF domain. We refer to this as *program synthesis*, to differentiate the execution of TOPS from the execution of TOPS-created procedures.

Axioms captured (implied) by the TOPS-created procedures are removed from the domain theory. Remaining axioms that contain terms in the languages of the theories of the procedures are separated. At program synthesis time, only the TOPS-created procedures operate on these terms. The algorithm for creating these procedures is described here.

In the pseudo code below, `Procedures` is the set of procedures created by TOPS. This algorithm attempts to construct a procedure for each abstract relation symbol in the domain theory. `MakeProcedure` generates a procedure `P`. `Capture&Separate` removes axioms from the domain theory `T` if they are implied by `P` and separates other axioms. `Validate` tests the procedure to ensure it captures all necessary axioms. When a procedure is synthesized and validated, it is added to `Procedures`. If any step fails, no procedure is added to `Procedures`, and the domain theory is not modified.

```
$TOPS(Theory : T)$
    $Procedures \leftarrow \{\}$
    $For\ each\ R \in \Sigma A_T$
        $P \leftarrow MakeProcedure(T, R)$
        $T' \leftarrow Capture\&Separate(T, P, R)$
        $IF\ (Validate(P))THEN$
            $Procedures \leftarrow Procedures \cup P$
            $T \leftarrow T'$
    $return(Procedures, T)$
```

4.1 MakeProcedure Algorithm

`MakeProcedure` constructs a deductive synthesis procedure P for a relation symbol R that operates on a language L. Given a query formula, P must provide a set of equalities between existential variables in the query and ground terms in L such that the formula is valid. This set of equalities is called an *answer*. At program synthesis time, P looks up previously computed answers. At theory compilation time,`MakeProcedure` is responsible for creating the procedure P, compiling the list of results that P will use at program synthesis time, and ensuring that the results in the list are adequate to produce answers for any query. The algorithm is given here.

```
$MakeProcedure(Theory : T, Relsym : R)$
    $Index \leftarrow Classify(T, R)$
    $AbsFns \leftarrow ExtractAbsFns(T, Index, R)$
    $Slots \leftarrow InputOutput(AbsFns, T, R)$
    $termlist \leftarrow UnitAnswers(AbsFns, Slots, T, R)$
    $return(CreateProcedureInstance(Index, termlist))$
```

Procedure creation begins with the classification of R, which is completed by function `Classify`. This algorithm selects a template for a procedure that decides the satisfiability of conjunctions of abstract literals in L. The `Classify` function is described in detail in [16]. As with DRAT, the library of procedures is organized in a hierarchy. TOPS classifies each abstract relation symbol in this hierarcy. The root nodes of the hierarchy are defined syntactically (e.g., unary relation, binary relation); nodes lower down are semantically specialized (e.g., symmetric relation, 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 index returned identifies the deepest node reached in classification. For example, when `MakeProcedure` is called for the abstract sort *Time*, only one relation symbol, namely equality, is found. This is classified as transitive, reflexive, and symmetric. When `MakeProcedure` is called for the abstract sort *Event*, *lightlike*? is identified as a relation symbol and is classified as reflexive and antisymmetric but not transitive.

The procedure template identified by `Classify` contains code for determining satisfiability. The remaining effort in creating the procedure is to build the list of answers so that the procedure can be used to construct terms demonstrating validity. Ultimately, P, at program synthesis time, will need to complete literals of the form $(R\ (f_{absfn1}\ t_1...t_n)(f_{absfn2}\ t_{n+1}...t_m))$ where R is the abstract relation, f_{absfn1} and f_{absfn2} are abstraction functions, and $t_1...t_m$ are concrete terms. Some of the t_is may correspond to existential variables in the original query. To complete a literal of this form, P must generate bindings for these existential variables. The theory compilation process proceeds in three phases. First, all possible abstraction functions are identified. Second, the terms t_i which may contain existen tialvariables are iden tified. Third, a set of queries is generated, and proofs are constructed for each query. The answer terms from these proofs are collected and organized in the answer list. These steps are detailed below.

Abstraction functions are identified by the function `ExtractAbsFns`. This algorithm returns a set of abstraction functions, all of which hav e the same abstract range sort. It accomplishes this b y using SNARK to generate answers to queries of the form
$\exists(x_1, x_2) R(x_1, x_2)$ or
$\exists(x_1, x_2)(sort(x_1) \wedge sort(x_2) \wedge (x_1 = x_2))$.
The result of these queries is a set of terms of the given sort related by R. If any of these terms contain concrete arguments, the term represents an abstraction function. `ExtractAbsFns` collects all of the answers generated during a bounded searc h. If the list of answ ersis exhaustive, then the algorithm finds all possible abstraction functions. If searc his cut off, some abstraction functions may be missed. If P does not capture all of the necessary axioms, the `Validate` process will fail.

Once a set of answers has been acquired, `ExtractAbsFns` partitions the set into subsets of terms that each hav e the same head symbol. The most specific generalization (MSG) is computed for each subset [11]. The resulting set of MSGs is returned as the set of abstraction functions. In the *lightlike*? example, SNARK is given the query $\exists(x_1 x_2)(lightlike?\ x_1\ x_2)$. One result of this query is the pair of bindings
$x_1 = (obj\&time2event\ o_1\ t)$,
$x_2 = (obj\&time2event\ o_2\ (recd\ o_1\ o_2\ t))$.
The v ariables o_1, o_2, and t represent concrete sorts, so `ExtractAbsFns` iden tifies*obj&time2event* as an abstraction function.

`InputOutput` attempts to iden tifycombinations of parameters to the abstraction functions that, when ground, allow the construction of ground terms for the remaining parameters. The parameters in this combination are labeled *input*, and the parameters for which terms are constructed are labelled *output*.

T o partition the parameters of the abstraction functions, SNARK is called with a query of the form $\exists(\hat{y_1}, \hat{y_2})(R\ (absfn_1\ \hat{y_1})(absfn_2\ \hat{y_2}))$, where $\hat{y_1}$ and $\hat{y_2}$ are v ectors of variables, one variable for each parameter of the abstraction functions $absfn_i$. The parameters are labeled as input or output based on analysis of the results. F or the NAIF domain, it is only necessary for a parameter to be labeled as either input or output. If suc h a partition is not possible, no procedure is created for R.

For *lightlike*? the query is

$$\exists(y_1\ y_2\ y_3\ y_4)\ (lightlike?\ (obj\&time2event\ y_1\ y_2)\ (obj\&time2event\ y_3\ y_4)).$$

An answer to this query is a binding pair. When the query is applied to the NAIF domain theory, the first parameter slot always contains constants. The second slot contains v ariables and the concrete functions*sent* and *recd*. F rom this, it is determined that the first slot is input and the second slot is output. For time conv ersions the query is
$\exists(y_1\ y_2\ y_3\ y_4)(= (absctt\ y_1\ y_2)\ (absctt\ y_3\ y_4))$.
Again, the bindings for y_1 and y_3 are constants (suc h as *Ephemeris* or *UTC*), and y_2 and y_4 are bound to either v ariables or concrete functions. F rom this, it is determined that the first slot is input and the second slot is output.

Compilation of solutions to synthesis queries is completed b y function `UnitAnswers`, which attempts to assure coverage of the entire input parameter space b y examining the answer sets obtained during the input/output analysis. The coverage considers universally quantified v ariables in the answer set as well as constants and functions. The queries for this step are similar to the queries for `InputOutput` except that the existential v ariables for input slots are replaced with constants or universally quantified variables.

Given the set of abstraction functions, the input and output labeling for these functions, and the properties of the relation, `UnitAnswers` constructs a query for each pair of abstraction functions and each set of slot instances. These queries are submitted to SNARK, and the parameter sets and returned bindings are retained. SNARK is restricted to return terms in the concrete language. A list, *termlist*, of these answers is ordered so that the most specific solutions appear first, and more general solutions appear later. During program synthesis, *termlist* is searched sequentially .

When the partitioning of the parameters for *lightlike*? is complete, `UnitAnswers` evaluates the sort of the first parameter slot. Since there are many constants in this slot (i.e., names for the different coordinate systems), `UnitAnswers` attempts to generalize b y replacing concrete constants with universally quantified v ariables. The following queries are given to SNARK as a result of the analysis of *lightlike*?:

$$\forall(n_1\ n_2\ t_1)\exists(t_2)\ (lightlike?\ (obj\&time2event\ (absid\ n_1)\ (absctt\ Ephemeris\ t_1))\ (obj\&time2event\ (absid\ n_2)\ (absctt\ Ephemeris\ t_2)),$$

and

$$\forall(n_1\ n_2\ t_2)\exists(t_1)\ (lightlike?\ (obj\&time2event\ (absid\ n_1)\ (absctt\ Ephemeris\ t_1))\ (obj\&time2event\ (absid\ n_2)\ (absctt\ Ephemeris\ t_2)).$$

The results from these queries are the pairs $t_2 = recd(n_1\ n_2\ t_1)$ and $t_1 = sent(n_1\ n_2\ t_2)$. These results are written into the term list for the generated procedure. This completes the creation of the procedure. The remaining steps validate the procedure and modify the domain theory.

At program synthesis time, a procedure must evaluate the satisfiability of a conjunction of literals and generate bindings for variables. Suppose that the separated clause

$$\begin{array}{l}(lightlike?\\ \ (obj\&time2event\ (absid\ Sun)\ (absctt\ Ephemeris\ t_1))\\ \ (obj\&time2event\ (absid\ Mars)\ (absctt\ Ephemeris\ t_2))\\ \rightarrow \phi\end{array}$$

(where t_1 and t_2 are variables and Sun and $Mars$ are constants) is resolved against a second clause of the form $\square \rightarrow \psi$ with the unifier t_2/c where c is a ground term. First, the procedure determines that the antecedents are satisfiable. Then, since c is ground, it is possible to generate a ground binding for t_1. The literal matches the second entry in the procedure's answer list:

$$\begin{array}{l}(lightlike?\\ \ (obj\&time2event\ (absid\ n_1)\ (absctt\ Ephemeris\\ \qquad\qquad\qquad\qquad\qquad (sent\ n_1\ n_2\ t_1)))\\ \ (obj\&time2event\ (absid\ n_2)\ (absctt\ Ephemeris\ t_2))\end{array}$$

Thus the binding $t1/(sent\ Sun\ Mars\ c)$ is applied to the consequent. Note that had the procedure been given a pair of literals in the antecedent such as $(lightlike?\ E_1\ E_2)$ and $(lightlike?\ E_2\ E_1)$, the procedure would have detected unsatisfiability in the conjunction of the antecedents, since $lightlike?$ is antisymmetric. In this case, the resolvant is discarded.

4.2 Capture&Separate

Once a procedure has been created, TOPS modifies the domain theory by first removing all captured axioms, then separating any remaining axiom that has terms or literals in the language of the theory of the procedure. When this process is complete, there should be no axioms remaining in the domain theory that contain concrete terms in the language of the procedure. If any such axioms are present, the synthesized procedure has failed to capture them, and TOPS does not continue procedure generation.

Next, each axiom in the domain theory that contains the abstract relation symbol R is temporarily removed from the domain theory. Then the axiom (with fresh constants substituted for universally quantified variables) is given as a query to the theorem prover augmented with the created procedure. If the theorem prover is able to prove the query, the axiom is removed from the domain theory.

If the axiom cannot be proved, then the procedure does not capture it, and procedure construction fails. This happens when the theory of the procedure is a subtheory of the theory of the language L, i.e., the classification does not push deeply enough into the hierarchy or no procedure template exists to cover the theory. Axiom testing continues until either all axioms containing R have been removed or procedure construction fails. `Validate` searches the domain theory for axioms whose consequent contains subterms in the concrete language of the procedure. If any such axiom is found, it was not captured by the procedure, and `Validate` fails.

Note that the theory of the specialized procedure and the theory of the axioms removed from the domain theory do not need to be logically equivalent. The procedures created by TOPS cannot produce every concrete answer that can be produced without the procedures. For example, it is possible for SNARK to produce a term of the form $(absctt\ UTC\ (Ephemeris2UTC\ (UTC2Ephemeris\ tc)))$. The TOPS-created procedure will only produce the equivalent term $(absctt\ UTC\ tc)$. Any term produced by a TOPS-created procedure can be produced by SNARK using the original domain theory.

5 Experimental Results

The TOPS compilation algorithm is highly effective. This section describes experimental results from applying TOPS 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. The TOPS procedure library contained two entries needed for the NAIF domain: one for representation conversions and one for acyclic, reflexive relations.

To test TOPS, we compared the performance of three deductive synthesis systems: a TOPS-created system, an untuned system, and a system manually tuned to the NAIF domain by program synthesis experts. The benchmarks against which these comparisons were made were the accuracy of the generated programs, the number of deduction steps in a proof, the number of deduction steps in the search to derive each proof, and the time required for each system to

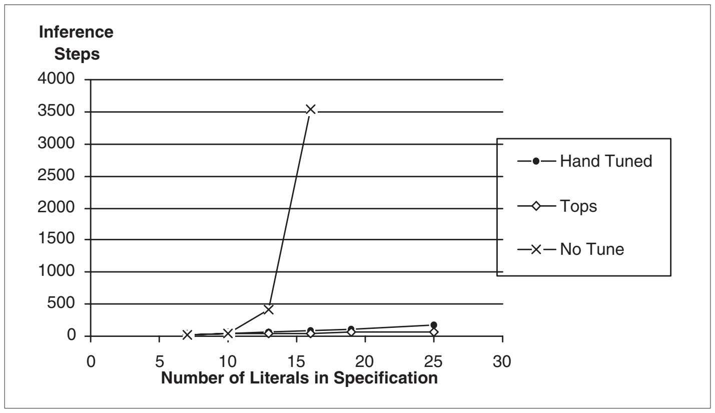


Figure 1. Search Tree Nodes

derive an answer.

A series of 27 specifications were used to test the configurations described above. These specifications ranged from trivial with only a few literals to fairly complex with dozens of literals. Half of the specifications were obtained as solutions to problem specifications given by domain experts, thus this set contains representative problems encountered during real world use. For each specification, the size of the specification (number of literals), the number of inference steps required to prove the specification, and the number of steps required to search for the proof were recorded for the un-tuned domain theory, the hand-tuned domain theory, and the TOPS-tuned domain theory. The NAIF domain theory consisted of 330 first-order axioms. The TOPS library contained two entries, one for representation conversions and one for acyclic, reflexive relations. Five procedures were synthesized and used to prove each of the specifications.

Figure 1 compares the number of inference steps that each configuration required to find a proof of specifications (the y axis) with varying numbers of literals (the x axis). This graph clearly shows the exponential nature of deductive synthesis when using only universal tactics. Compared to this, the manually-tuned and TOPS-created configurations scale relatively well. A closer comparison of the search space for the hand-tuned and TOPS-created systems reveals 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.

To understand why TOPS does better than the human experts, recall that there are three approaches to tuning a deductive synthesis system: (1) adding heuristics to guide search; (2) reformulating the domain theory to guide the theorem prover; and (3) adding specialized inference procedures to the theorem prover. The human experts who have tuned Amphion domain theories utilized the first two methods. Instructions were written to order the set of formulas that the theorem prover generates in searching for a proof so that those formulas most likely to lead to a proof are selected for further processing first. These instructions are based on heuristics discovered by theorem proving experts. The human experts also considered the behavior of the theorem prover, then reformulated axioms in the domain theory to force the theorem prover to apply particular inference rules.

In contrast to the human efforts, TOPS creates and integrates specialized inference procedures. This approach removes some axioms from the domain theory and separates others, resulting in a smaller search space for the theorem prover. TOPS is able to do this because of the uniform interface provided by separated inference rules. As a domain theory is modified, TOPS recompiles (from scratch) the theory, simplifying the tuning of the system.

So, in effect, our experimental results compare operationalizing by designing specialized inference procedures to doing so by methods (1) and (2). These results provide evidence that operationalizing by designing procedures is more effective. An important additional benefit of our technique is that we have shown that it can be automated. Experience with maintaining the Amphion NAIF domain theory shows that operationalization is brittle: each time the domain theory is modified, the system must be re-tuned. Hence, an automatic process is preferable.

6 Summary

This paper describes TOPS, an automated technique for operationalizing a domain theory to do deductive synthesis. Given a domain theory, TOPS identifies decision procedure templates from a library that can be used for subsets of the theory. It instantiates these templates generating procedures that find witnesses for variables in deductive synthesis queries. These witnesses represent code fragments.

The resulting procedures are interfaced to a resolution theorem prover that uses separated inference rules. These rules operate on separated clauses of the form $\overline{C_1} \Rightarrow C_2$, where $\overline{C_1}$ is a conjunction of literals in $L_1 \subseteq L$, where L is the language of the domain theory, and C_2 is a disjunction of literals in $L - L_1$. Separated inference rules employ standard theorem proving techniques to manipulate the consequents of separated clauses but use the decision procedures to decide satisfiability and validity of the antecedents.

The contribution of TOPS is to extend the DRAT technique by using partial deduction to instantiate decision procedure templates. The result of the partial deduction is to compile tables that are used by the decision procedures at program synthesis time. The procedures use the tables to lookup witnesses for variables in a deductive synthesis query.

A domain theory operationalized by TOPS has axioms that cause combinatory explosion removed and in their place decision procedures are interfaced to create a theorem prover specialized to a particular domain. Experimental results show that a deductive synthesis system specialized by TOPS can be dramatically more efficient than a declarative domain theory and can be at least as efficient as a theory operationalized by manually reformulating axioms and adding heuristics to guide search.

Of course, while TOPS works well for the NAIF domain, its ability to achieve dramatic speedups is limited to domain theories that admit simple table lookups for witness generation. In the future, we would like to extend TOPS to more sophisticated witness generation techniques.

6.1 Acknowledgements

The authors would like to thank Mike Lowry, Tom Pressburger, and John Cowles for their insights during discussions that led to this work. We would also like to thank the anonymous reviewers for their comments on the presentation.

References

[1] R. Boyer and J. Moore, *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, 1985.

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

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

[4] N. Jones, C. Gomard, P. Sestoft, *Partial Evaluation, and Automatic Program Generation*, Prentice Hall, New York, 1993.

[5] R. Keller, "Applying Knowledge Compilation Techniques to Model-Based Reasoning," *IEEE Expert*, April, 1991, pp. 82-87.

[6] 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.

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

[8] Madden, P. and Bundy, A., "General Techniques for Automatic Program Optimization and Synthesis Through Theorem Proving," *Proceedings of EWAIC'93*, 1993.

[9] Z. Manna and R. Waldinger, "A Deductive Approach To Program Synthesis," *ACM Transactions on Programming Languages and Systems*, Vol 2, No 1, Jan 1980, pp. 90-121.

[10] Navigation and Ancillary Information Facility (NAIF), "SPICE," http://pds.jpl.nasa.gov/naif.html.

[11] G. Plotkin, "A Note on Inductive Generalisation," *Machine Intelligence 5*, M. Meltzer and D. Michie (eds.), Elsevier North-Holland, New York, 1970, pp:153-163.

[12] C. Rich and R. Waters, "Automatic Programming: Myths and Prospects," *IEEE Computer*, Vol. 21, No. 8, Aug. 1988, pp. 40-51.

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

[14] M. Stickel, R. Waldinger, M. Lowry, T. Pressburger, and I. Underwood, "Deductive Composition of Astronomical Software from Subroutine Libraries," *CADE-12*, 1994.

[15] M. Stickel, "SNARK – SRI's New Automated Reasoning Kit," http://www.ai.sri.com/ stickel/snark.html, 2000.

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

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