consist of time-system constants, program input variables, and applications of *convert-time*. The decision procedure automatically simplifies nested applications of *convert-time*.

The final step in DSDRAT's analysis of *coordinate-to-time* is to add the instantiated theories from TableGraph and Connected Groupoid to the cumulative set of axioms for instantiated decision procedures. DSDRAT then removes the axioms in the NAIF domain theory that are implied by this set. The removed axioms include the three listed in section 5. When these axioms are used by a general-purpose theorem-prover (e.g., SNARK) for deductive synthesis, they typically invoke search through branch points introduced by paramodulation. By replacing these axioms with efficient decision procedures, DSDRAT speeds up program synthesis without manual tuning. We expect that DSDRAT will automatically achieve the same results as were manually achieved with this methodology (section 5). This will provide a key enabling technology for domain experts, such as the JPL NAIF group, to maintain and extend their own specialized AMPHION systems.

7. Conclusion

This paper addresses one aspect of scaling-up KBSE: enabling domain experts to construct and maintain their own domain-specific KBPS systems. The META-AMPHION system, currently under development, is designed to provide the KBSE analogue of application-generator generator technology. A key component of META-AMPHION is a sub-system to automatically operationalize a declarative domain theory for efficient deductive program synthesis. This paper describes extensions of a previous system, DRAT, that speeds up a theorem-prover for analytical reasoning problems by substituting decision procedures for axioms in a theory. An experiment with AMPHION (a real-world KBPS system) demonstrated that extending DRAT to deductive synthesis would be successful. The design for these extensions is described, and are currently being implemented.

In parallel work we are also exploring suitable user interfaces for META-AMPHION that will guide domain experts in developing and maintaining domain theories. There appears to be a useful synergy with the component described in this paper: the same process described in section 6 for operationalizing axioms in an existing domain theory might be useful in eliciting axioms from a domain expert. Instead of using a theorem-prover to answer questions when searching the hierarchy, the domain expert would be used as an oracle to construct portions of a domain theory by traversing the same hierarchy.

In previous work on DRAT, the work reported in [7] was extended to show that DRAT's attachment of literal satisfiability procedures to a theorem-prover was sound and complete. These results must be extended to DSDRAT and we are considering the framework reported in [2]. The extension of DRAT's classification procedure to DSDRAT's classification procedure is related to the work in [8,9]. We are also exploring the use of SPECWARE™[4] as a tool for part of the implementation of META-AMPHION

Acknowledgments

Sincere thanks to the anonymous reviewers and our colleagues Arthur Reyes, Lise Geetoor, Thomas Pressburger and Steven Roach for their helpful comments.

References

[1] J.C. Cleaveland and C. Kintala, "Tools for Building Application Generators." *AT&T Technical Journal*, Vol. 67, No. 4, 1988, pp. 46-58.

[2] F. Giunchigliam, P. Pecchiari, C. Talcott, "Reasoning Theories: Towards an Architecture for Open Mechanized Reasoning Systems," Stanford CS Technical Report CS-TN-94-15, 1994.

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

[4] R. Jullig and Y.V. Srinivas, "Diagrams for Software Synthesis," *KBSE 1993*.

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

[6] Z. Manna and R. Waldinger, "Fundamentals of Deductive Program Synthesis," *IEEE Transactions on Software Engineering*,(18) 8, August 1992, pp. 674-704.

[7] G. Nelson, "Combining Satisfiability Procedures by Equality Sharing," in *Automated Theorem Proving after 25 Years*, Bledsoe and Loveland (Eds), American Mathematical Society, 1984.

[8] D.R. Smith and M.R. Lowry, "Algorithm Theories and Design Tactics," *Science of Computer Programming* Vol. 14, 1990, pp. 305-321.

[9] D.R. Smith, "Classification Approach to Design," 1993 Kestrel Institute Technical Report.

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

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

[12] E.H. Tyugu, *Knowledge-Based Programming*, Turing Institute Press, Glasgow, Scotland, 1988.

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

[14] J. Van Baalen, "Automated design of specialized representations," *Artificial Intelligence*, Vol. 54, 1992.

This step provides a signature morphism, labeled step 1 in Figure 2, between the language for the Conversions node and the NAIF domain theory:

$$\begin{array}{ll} sp & \mapsto time\text{-}system \\ a & \mapsto time \\ c & \mapsto time\text{-}coordinate \\ abstr & \mapsto coordinates\text{-}to\text{-}time \\ conversions & \mapsto \{convert\text{-}time\} \end{array}$$

In step 2 DSDRAT pushes further down the taxonomy by extending this theory morphism to the Representation Conversions node by proving the instantiated *abstract identity* property for time-system conversions (see the axioms in section 5).

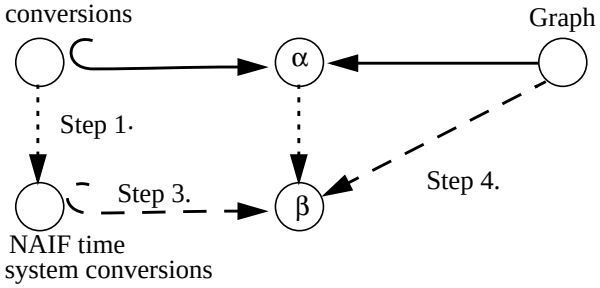

Figure 3. Following a link across the hierarchy.

In step 3 and 4 DSDRAT follows the link (shown as a dashed line in Figure 1) from the Conversions node to the Graph node, thereby constructing the definition of a graph whose edges are labelled by these conversions. An expanded view of this link is shown in Figure 3. (All the arrows in Figure 3 are theory morphisms.) This link consists of a predefined definitional extension (the solid arrow with a rounded tail) from the Conversions node to an intermediate node α (not shown in Figure 1), and a predefined theory morphism from the Graph node to α. Conceptually, the definitional extension 'expands out' the conversion functions to edges in a graph. The definitional extension is defined by the lambda expression schema below. The theory morphism maps the graph sort *node* to sort *sp* and maps the graph sort *edge* to the 'expanded out' set of conversion functions:

$$\{apply(\lambda p_1, p_2.\, convert(p_1, p_2, x), < a, b >) \mid a, b \in sp\}$$

where *convert* is a conversion function, p_1 is the source of the edge and p_2 is the target of the edge.

The theory morphisms DSDRAT constructs in following this link for the NAIF time-system conversions are shown as dashed lines in Figure 3. Step 1 was discussed previously. In step 3 the predefined definitional extension is applied to the result of step 1, yielding β. Specifically, the lambda expression schema is instantiated for all the collected NAIF time-system conversions (a singleton set), yielding

$$\{apply(\lambda ts_1, ts_2.\, convert\text{-}time(ts_1, ts_2, x), < a, b >) \mid a, b \in time\text{-}system\}$$

where ts_1 and ts_2 are time systems and *x* is a time coordinate. Note that because Figure 3 is a commutative diagram, this also defines the theory morphism from α to β. In step 4 the theory morphism is constructed from the Graph node to the extended NAIF time-system conversions (β). Once again, because Figure 3 is a commutative diagram, this is simply the composition of the predefined morphism from Graph to α and the morphism from α to β. This composition maps the graph sort *node* to the sort *time-system*, and maps the graph sort *edge* to the instantiated lambda expression. Note that while DSDRAT needs to do proofs when pushing down the hierarchy, in following the link from Conversions to Graph no proofs need to be done by DSDRAT—it is simply performing syntactic manipulations that are justified by proofs done off-line when the library of decision procedures was constructed.

The rest of the analysis consists of the same kinds of steps as those shown in Figures 2 and 3. Given the theory morphism constructed in step 4, DSDRAT pushes down the Graph taxonomy. However, to do so it needs to determine the properties of the associated path algebra. DSDRAT follows the link from Graph to Category, using the same steps as described for Figure 3. DSDRAT then pushes down the hierarchy under the Category (Path Algebra) node, incrementally constructing theory morphisms using the same steps as described for Figure 2. DSDRAT classifies the path algebra for *convert-time* as a totally connected groupoid (labelled Connected Groupoid in Figure 1). Totally connected means there is a path between any two nodes, and groupoid means that for any path there is an inverse path. In addition to determining the path algebra properties, this classification also produces an instantiation of the Connected Groupoid decision procedure schema.

After classifying the path algebra for the graph, DSDRAT returns to the Graph taxonomy and completes the classification of the graph. This classification yields the composite decision procedure schema TableGraph [Connected Groupoid]. Table Graph is a specialization of Graph that precomputes a table representation of paths when the schema is instantiated. For time systems this is possible because there are a finite number of time systems and paths representing conversions between time systems are uniquely determined by their endpoints. The instantiated TableGraph[ConnectedGroupoid] decision procedure is used to decide the satisfiability of conjunctions of equalities between terms of the form
(abs (coordinate-to-time x) y). This instantiated decision procedure generates witness ground terms for existentially quantified variables of sort *time-coordinate;* the terms

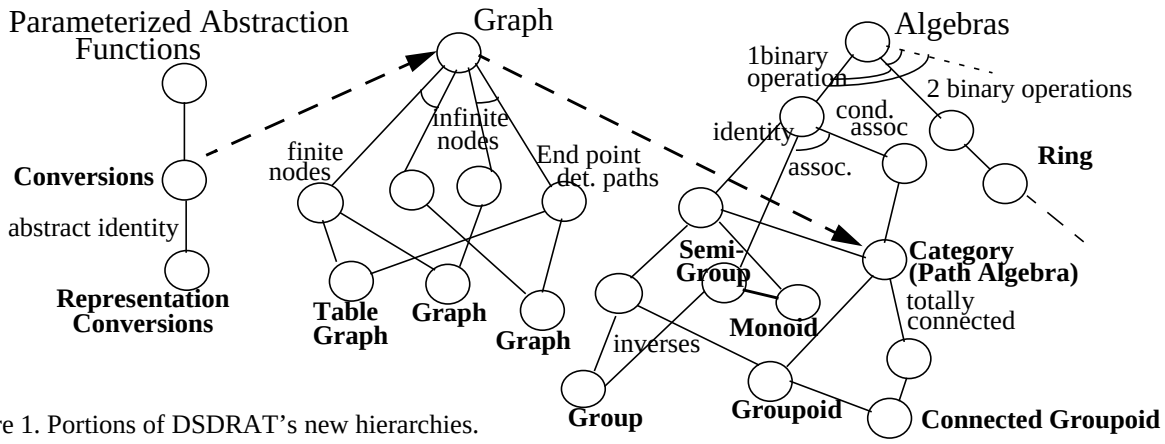

Figure 1. Portions of DSDRAT's new hierarchies.

unification in a type theory.

Given a theory morphism from a node at the top of the hierarchy, the classification is pushed down the hierarchy by incrementally proving additional properties in the domain theory. For example, to push down the hierarchy from Category (an algebra with nodes and composable arrows, where composition is partial and associative) to Groupoid requires proving the axiom that all arrows have inverses.

As classification is pushed down the hierarchy, DSDRAT also follows links for definitional extensions and reformulation rules across the hierarchy. For example, there is a definitional extension from the Graph taxonomy to Category. To classify a graph can require determining properties of its path algebra. Thus the path algebra is defined and classified. The classification begins at the Category node in the Algebra taxonomy (the paths of a graph are the arrows of the associated category). Furthermore, the instantiated decision procedure for a graph takes as parameter the instantiated decision procedure for the path algebra. Hence, for Graph there is a link to Category both for the purpose of determining further properties of a graph and also in order to instantiate the decision procedure parameter for the path algebra.

6.4 Example: Design of a decision procedure for an abstraction function

This subsection describes DSDRAT's design of the decision procedure for the *coordinates-to-time* abstraction function. This function was described in section 2 and its axioms were described in section 5. This example illustrates the design algorithm described in the previous subsection. It is typical of the analysis of the *ABS* component of an AMPHION domain theory, and similar to the analysis of the Σ*A* component. Although comparatively simple, the example is complex enough to illustrate the steps in DSDRAT's design algorithm as it traverses the hierarchy, and also to illustrate the design of a composite decision procedure.

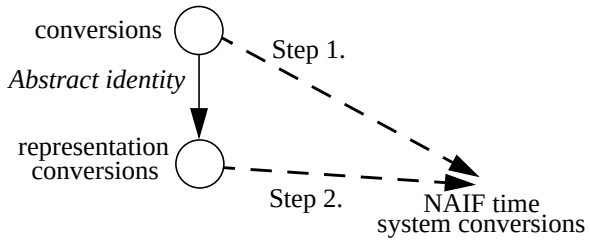

Figure 2. Pushing down the hierarchy.

DSDRAT's analyses of *coordinates-to-time* begins in the Parameterized Abstraction Function taxonomy (Figure 1). Parameterized abstraction functions have a signature of the form $s_1 \times \cdots \times s_k \rightarrow (c \rightarrow a)$, where the s_i are the sorts for the parameters of the abstraction function, *c* is a concrete sort and *a* is an abstract sort. Classification of a parameterized abstraction function results in DSDRAT collecting the set of that abstraction function's *conversions*. The conversions for an abstraction function are functions whose signature is $sp \times sp \times c \rightarrow c$ where $sp = s_1 \times \cdots \times s_k$. Conversions, given a source and target set of parameters (*sp*), map between elements of the concrete sort *c*. *Representation conversions* satisfy the additional property of preserving the identity of abstract objects (*Abstract identity* in Figure 2) with respect to the parameterized abstraction function *(absfn)*:

$$absfn(absid(p_1), x) = absfn(absid(p_2), rconvert(p_1, p_2, x))$$

where *rconvert* is a representation conversion function. Decision procedures for representation conversions are interfaced to the theorem-prover through *absfn*, and generate ground terms of the concrete sort *c*.

Step 1 of the classification of the abstraction function *coordinates-to-time*, whose signature is *time-system* → *time-coordinate* → *time*, is to collect the set of corresponding conversions, resulting in the singleton set whose element is *convert-time*, a function with signature *time-system x time-system x time-coordinate* → *time-coordinate*.

of Σ*A* and *ABS*, program fragments are returned in the language of Σ*C*. A decision procedure replaces deductive inference on the axioms in *DT*. As in DRAT, the library is organized hierarchically; a new portion of the library is shown in Figure 1. Each node in the hierarchy is a 6-tuple <*DT*,Σ*A*,Σ*C*,*ABS*,*I*,*DP*> where the first four elements are the index, *DP* is a decision procedure schema (implemented as a common lisp object class), and *I* is a procedure for instantiating a decision procedure schema given an instantiation of the 4-tuple index.

When the theory resolution interface gives an instantiated decision procedure a set of literals in the language of Σ*A* and *ABS*, the decision procedure returns terms in the language of Σ*C* as bindings for existential variables in the literals (universal variables when considered as an unsatisfiability problem). The decision procedure can also return a set of residual literals, if it is unable to completely resolve the literals given as input. More formally, given a set ϕ of literals in the language of Σ*A* and *ABS*, the decision procedure returns a set of literals ϕ' and set of terms *t* in the language of Σ*C*, such that (*outs* are variables, DT_I is the instantiated theory for the decision procedure):

$$DT_I \vdash [\phi \Rightarrow \phi']! \; outs \leftarrow t$$

As an example, consider the decision procedures indexed under the Graph taxonomy in Figure 1. These decision procedures generate terms representing paths in a graph. The specification language (Σ*A*) sort 'nodes' consist of the node labels of the graph, and the concrete language (Σ*C*) sort 'edges' consist of the edge labels of the graph. The properties of the graph determine which decision procedure in the taxonomy is used. A decision procedure is applicable if an instantiation of its theory (i.e., DT_I) is implied by the domain theory defining a graph; the decision procedure with the most specific such theory is best. Instantiated decision procedures from the Graph taxonomy take as input conjunctions of literals and build internal graph data structures representing those conjunctions. These decision procedures decide satisfiability of the conjunctions (with respect to the instantiated theory for the decision procedure) by manipulating the graphs. They also determine when variables in the conjunction are connected in the graph to constants (program input variables) and construct ground terms for those variables by traversing the graph.

Instantiated decision procedures can be composed horizontally or vertically (where the concrete language for one decision procedure is the same as the abstract language for the following decision procedure). When decision procedures are combined, they communicate by passing variable bindings back and forth [7]. In addition, decision procedures can be nested —one decision procedure can take another as a parameter in order to solve subproblems.

Each decision procedure in the Graph taxonomy is parameterized by a path algebra; this parameter is instantiated by a decision procedure in the hierarchy below Category (Path Algebra) in the Algebra taxonomy. Graph-based procedures invoke their procedure parameter to check the consistency of paths in graphs they are constructing and to determine if there are shorter equivalent paths. If an inconsistent path is found, the graph-based procedure signals unsatisfiability. If the path algebra procedure determines that there is a shorter equivalent path between two nodes than the existing path, the existing path is replaced by the shorter one. This ensures that the terms constructed by traversing a graph are always the simplest terms possible.

6.3 DSDRAT's design algorithm

DSDRAT's design algorithm is an extension of DRAT's design algorithm. The top-level control loop is similar to the pseudo-code description in Section 4. Given a structured AMPHION domain theory <*DT*,Σ*A*,Σ*C*,*ABS*>, DSDRAT begins by classifying the symbols in Σ*A* and *ABS*. Relation symbols and some function symbols (those whose semantics are not given by implementation equations that can be converted to rewrite rules) in Σ*A* are classified. In addition, in order to connect the decision procedures at the abstract level to procedures at the concrete level, DSDRAT classifies the parameterized abstraction functions in *ABS*. The left-most taxonomy in Figure 1 is used for this purpose.

In the hierarchy in Figure 1, there are three taxonomies labeled Parameterized Abstraction Function, Graph, and Algebra. Each taxonomy is an and-or tree with downward links labeled by properties in roman, such as associativity of an algebra with a binary relation. Links with incompatible properties have an arc drawn between them. Nodes are theories that accumulate their axioms (properties) along the paths leading to them. Nodes with a bold label have an associated decision procedure schema. The dotted lines are definitional extensions and reformulation links.

Parts of a domain theory are classified by constructing *theory morphisms* from the theories in the library hierarchy to parts of the domain theory. Theory morphisms are the generalization of DRAT's instantiation of individual function, relation, and sort symbols. A theory morphism is a map from the language of one theory to the language of another theory such that the axioms of the first theory are mapped to theorems in the second theory. DSDRAT invokes AMPHION to prove such theorems. Constructing theory morphisms from the nodes in the top of the decision procedure hierarchy is mainly syntactic, since there are relatively few axioms associated with such nodes. However, constructing these morphisms can involve simple syntactic reformulations, such as tupling together sorts and currying functions. These reformulations are handled through

decision procedures for several different kinds of axiom sets. To test the effectiveness of these procedures, we developed a test suite of forty specifications to compare total proof steps and run-times for three different configurations of the theorem-prover: without the strategy described in section 2, with that strategy, and with the combination of the strategy and decision procedures. Attempts to prove specifications without either the strategy or decision procedures ended in failure to find any proofs in under 100 minutes, so this configuration was abandoned. All of the forty problems were proved with the strategy-only configuration and the strategy/decision procedure configuration. On average, the strategy/decision procedure configuration found proofs in an order of magnitude less time than the strategy-only configuration. For example, in the proof of one specification the strategy-only configuration took 430 steps and 289 seconds. On the same specification, the strategy/decision procedure configuration took 58 steps in 15 seconds (Each step is a resolution or a paramodulation).

Perhaps more important than the above results is that, because these procedures take advantage of well-defined mathematical structures in a domain theory, as did DRAT, this approach is amenable to automation. The next section describes the design of our system DSDRAT, which extends DRAT from analytical reasoning problems to deductive synthesis problems. We are currently implementing these extensions of DRAT.

6. Automating domain theory operationalization

6.1 Extending DRAT to deductive synthesis

Scaling up from analytical reasoning problems to deductive synthesis problems requires extensions for: 1) the kinds of problems that are solved, 2) the outputs of the proof process, 3) the complexity of the domain theories. Technically, analytical reasoning problems are ground (un)satisfiability questions (i.e., is a ground formula (un)satisfiable in a given theory). In contrast, deductive synthesis problems for AMPHION are specifications given as pre- and post-conditions:

$$DT \vdash \forall ins \exists outs\ R(ins) \Rightarrow R(ins, outs)$$

where *DT* is the domain theory, *ins* is a vector of input variables and *outs* is a vector of output variables. In order to simplify the exposition, we will assume the precondition (*R(ins)*) is always true. There is a special subset of the concrete part of the domain theory language, called the *output* language, whose symbols name components of the target software library. Deductive synthesis proves a theorem by constructing substitutions for the output variables that are terms in the output language. The variables in these terms are input variables; hence these terms represent program fragments that compute the value of an output variable from the values of input variables. (Technically, deductive synthesis through resolution theorem-proving solves the non-ground unsatisfiability question: "is $DT \cup \{\exists ins \forall outs \neg R(ins, outs)\}$ unsatisfiable?" The decision procedures designed by DSDRAT also solve unsatisfiability questions and work in conjunction with the resolution proof process through theory resolution.)

In contrast to analytical reasoning problems, for deductive synthesis problems an important consideration is the algebraic structure of output terms, e.g., the equivalence classes of these terms. Inferences on this structure provide KBPS some of its advantage over the context-free, macro-expansion code-generation process used in application generators. Besides correctness we usually want to place additional requirements on these output terms for deductive synthesis, such as that they represent the best program in their equivalence class, by some measure. So far in this research we have made the assumption that the best programs satisfying a post-condition are represented by the ground terms with the smallest number of function applications.

In addition to these differences in the problem types and proof-process outputs, AMPHION domain theories also differ from analytical reasoning task 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.

DRAT for deductive synthesis (DSDRAT) extends DRAT according to these differences. DSDRAT takes a structured AMPHION domain theory as input and produces a specialized theorem-prover as output, with a reduced set of axioms. The specialized theorem-prover includes a set of decision procedures interfaced through theory resolution. These decision procedures construct complex terms for output variables in specifications; these terms represent program fragments in the output language. DSDRAT also mechanically generates a simple abstract-to-concrete strategy from the partitioning of the domain theory into an abstract and concrete part. (This mechanically-generated strategy is a simplification of the existing manually-tuned AMPHION strategy (see [5]) and is not described in this paper.). Formally, DSDRAT takes as input a 4-tuple <*DT*,Σ*A*,Σ*C*,*ABS*>, where *DT* is an AMPHION domain theory, Σ*A* is the abstract specification language, Σ*C* is the concrete target language, and *ABS* is the set of parameterized abstraction maps.

6.2 Decision procedures for deductive synthesis

Decision procedures in DSDRAT's library are also indexed by a 4-tuple <*DT*,Σ*A*,Σ*C*,*ABS*>. Conceptually, each decision procedure schema in DSDRAT's library solves a small, parameterized, program synthesis problem. Problems given to the decision procedure are specified in the language

query $\phi \in \Phi$ into a satisfiability question *SAT*(ϕ). A literal satisfiability procedure designed by DRAT is independent of the particular set of ground literals, including the ground axioms of *T*; it actually solves all problems with the same non-ground axioms as *T*.

The following pseudo-code is an abstract description of DRAT's design algorithm:

```
procedures←{}, T_I←{}, T'←T
UNTIL empty(T') DO
    instance←choose-procedure(T')
    IF null(instance) THEN EXIT
    procedures←procedures ∪ instance
    T_I←T_I ∪ theory(instance)
    FOR EACH axiom in T'DO
     WHEN T_I implies axiom DO
          T'←T'- axiom
    END FOR
END UNTIL
```

The set of decision procedure instances is built up incrementally while the set of axioms in `T'` is pared down incrementally. The process stops either when `T'` is empty or no more decision procedures are applicable. Each time **choose-procedure** is invoked, it chooses a sort, relation, or function symbol from `T'`, classifies that symbol in the hierarchy as far down as possible, and then returns the decision procedure (`instance`) and associated theory (`theory(instance)`) with the deepest node reached in the hierarchy. (For DRAT, the top nodes of the hierarchy were defined syntactically (e.g. unary function, binary relation), nodes lower down were semantically specialized (e.g. one-to-one function, total order)). Both the procedure and the theory are instantiated by the classified symbol. The new procedure is added to `procedures` and its theory is added to T_I. DRAT then determines which axioms in *T'* can be removed, i.e., those axioms that are implied by T_I. DRAT is sound and complete, see [13] for details.

Although this algorithm is not sensitive to the way axioms are formulated, it is sensitive to the choice of language in formulating a problem, for example, a function formulated as a relation. DRAT's algorithm was augmented with *isomorphic reformulation* [13], implemented by placing reformulation rules on nodes in the hierarchy. These rules change a problem's formulation and also often restart classification at different nodes in the hierarchy.

DRAT has been successfully applied to analytical reasoning problems like those on the GREs. 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 was applied to twenty analytical reasoning problems and averaged two orders of magnitude speed-up over the theorem-prover alone.

5. Empirical results of manual domain theory operationalization

We manually applied the DRAT methodology to the NAIF domain theory. The objective was twofold: First, to experimentally determine whether analogous speed-ups could be obtained for deductive synthesis as were obtained for analytical reasoning problems. Second, to understand the technical issues in extending DRAT to deductive program synthesis. This section describes the experimental results for the first objective. The following section discusses the second objective. Our experiment targeted parts of the NAIF domain theory that we had previously needed to manually tune either through reformulation or through adjusting the theorem-proving strategy. If the DRAT methodology could successfully replace such parts of the domain theory with decision procedures, then automation of this methodology could likely replace manual tuning by deductive synthesis-experts.

One pattern of axioms that requires deductive-synthesis tuning is typified by those for conversions between time systems (see section 2 for notation):

$$\forall\, ts_1, ts_2, n:[\, a\&t(\mathit{coordinate}\text{-}\mathit{in}\text{-}\mathit{time}(ts_1), n) = a\&t(\mathit{coordinate}\text{-}\mathit{in}\text{-}\mathit{time}(ts_2), \mathit{convert}\text{-}\mathit{time}(ts_1, ts_2, n))]$$

$$\forall\, ts, n[\mathit{convert}\text{-}\mathit{time}(ts, ts, n) = n]$$

$$\forall\, ts_1, ts_2, ts_3, n:[\mathit{convert}\text{-}\mathit{time}(ts_2, ts_3, \mathit{convert}\text{-}\mathit{time}(ts_1, ts_2, n)) = \mathit{convert}\text{-}\mathit{time}(ts_1, ts_3, n)]$$

These axioms describe conversions between time systems, and will be used as an example in section 6. The first axiom states that a time co-ordinate in any time system can be converted to another time system by an application of the function *convert-time*. The second axiom describes the identity conversion, while the third states that nested applications of *convert-time*s with matching time-system arguments can be replaced by a single application. The second and third equations can be oriented from left to right and consequently are treated as simplification rewrite rules. However, equations like the first axiom need to be used in both directions and can not be oriented. As a result, SNARK uses paramodulation for inferences with the first axiom. Each paramodulation represents a multi-branched choice point in the search space. We developed a decision procedure to reason efficiently about representation conversions and replaced the axioms above with an instantiation of the procedure. We also replaced several other sets of similar axioms (e.g., co-ordinate frame conversion, co-ordinate system conversion) by different instantiations of this same procedure. These instantiated decision procedures are very similar to modules that might be developed by a KBSE expert when hand-crafting a KBPS system.

We performed this manual replacement of axioms by

Up to now, no tools have existed to facilitate domain theory development and extension for AMPHION, particularly the task of tuning a domain theory and theorem-proving strategies for efficient program synthesis. Definitional extension and specification of new software components is fairly routine. However, adding axioms that interact with existing axioms often leads to a combinatorial explosion during deductive synthesis. To date, we have addressed these problems as is usually done in applications of automated theorem-proving: manually reformulating the domain theory and tuning the theorem-proving strategy. However, continuing this methodology will make it impossible to transition AMPHION from a few application domains in the research laboratory to a plethora of application domains in the field.

3. META-AMPHION

META-AMPHION is a system, currently under development, whose objective is to empower domain experts (without substantial training in KBSE) to specialize AMPHION to an application domain and maintain it themselves. META-AMPHION includes a user interface to guide domain experts in creating and extending a domain theory, a subsystem to check that software components are specified correctly in a domain theory through verification against the component code, and a subsystem to check for missing axioms. The research described in this paper is for a subsystem—DSDRAT—to automatically operationalize a domain theory for efficient deductive synthesis.

Our approach to operationalizing a declarative domain theory is analogous to applying AMPHION at the meta-level: given a meta-theory of program synthesis and a domain theory for a particular application domain, META-AMPHION constructs an efficient domain-specific KBPS system by composing components from a library and then instantiating the generic AMPHION architecture.

The components are decision procedures that perform specialized inference tasks much more efficiently than applying general-purpose theorem-proving to axioms. In the usual approach to decision procedures, the specialized inference tasks are specified syntactically, e.g., 'decide the equality relation'. In contrast, in our approach the specialized inference tasks are smaller-grained and specified semantically, e.g., 'decide whether a set of ordered pairs (represented as ground literals) defines a partial order'. In our approach, the decision procedures decide theories related to standard mathematical structures, such as various algebras (e.g., groups, path algebras). The decision procedures incorporate algorithms, such as graph-search algorithms, that would also be found in hand-crafted KBPS systems that did not use automated theorem-proving.

The advantage of this semantic and small-grained approach to decision procedures is that automated analysis of their applicability is far more tractable. The decision procedures are simply specified through the theories they decide. Given a theory of a decision procedure, DSDRAT analyzes a domain theory to identify sets of axioms that are equivalent to the theory. These sets of axioms are then removed from the domain theory and, in their place, instances of the decision procedure are interfaced to the theorem-prover. This specialized theorem-prover, with the remaining axioms, performs the same inferences as the general-purpose theorem-prover, with all the axioms, but much more efficiently. Conceptually, this process is repeated for all the decision procedures in the library.

The actual operationalization process is made efficient by arranging the theories for the decision procedures into a hierarchy; more specialized theories are lower in the hierarchy (e.g., 'equivalence relation' is below 'symmetric binary relation'). The nodes in the hierarchy are cross-indexed through reformulation rules to accommodate different ways of formulating a domain theory. The next section describes this process as it has already been applied to analytical reasoning problems, while subsequent sections describe extensions for program synthesis problems.

4. DRAT

DRAT (Designing Representations for Analytical Tasks) [14] was inspired by human problem-solving performance on analytical reasoning problems, such as those found on graduate-level standardized admission tests (GREs). When these problems are encoded in FOL, they are surprisingly difficult for a general-purpose theorem-prover, taking hours for problems solved in minutes by college students. Students use specialized representations and procedures to reason about these problems; in fact, GRE guidebooks show how diagrams can be used to assist in problem-solving. DRAT parallels human performance in designing specialized representations and procedures to solve analytical reasoning problems.

A problem is a pair $<T,\Phi>$, where T is a first-order theory. Φ is a set of queries. Possibility queries ask whether $T \cup \phi$ is satisfiable and necessity queries ask whether T implies ϕ or, equivalently, whether $T \cup \{\neg\phi\}$ is unsatisfiable. (In both cases ϕ is a ground formula.) Given a problem, DRAT automatically performs a semantic analysis of the non-ground axioms (i.e., axioms with quantified variables) in T, and replaces as many as possible with instances of specialized decision procedures. These decision procedures are interfaced to a theorem-prover through theory resolution [10]. Technically, DRAT designs a *literal satisfiability* procedure, which 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

program transformations, ad-hoc procedures, or the intricate workings of automated theorem-provers. We believe that the knowledge-acquisition bottleneck for KBSE is surmountable by domain experts only if domain descriptions can be developed, validated, and maintained with an underlying declarative semantics. Declarative semantics are a necessary but not sufficient requirement.

This paper describes research to automatically operationalize a declarative domain theory for the purpose of efficient, domain-specific KBPS. It is an essential component for enabling domain experts to develop and maintain their own KBPS systems. This research builds upon the AMPHION system, reviewed in section 2, which is the KBPS analogue of a generic application-generator architecture. The META-AMPHION system (section 3), which is currently under development, is the KBPS analogue of application-generator generator technology. The research described in this paper is for one component of the META-AMPHION system.

The technology for automatically operationalizing a declarative domain theory is an extension of the DRAT system (section 4). DRAT has previously been successfully applied to automatically operationalizing FOL theories for analytical reasoning problems. The techniques used in DRAT were manually applied to one of the existing AMPHION applications. Section 5 describes the resulting order-of-magnitude speed-ups in deductive program synthesis. Section 6 describes extensions to DRAT, called DSDRAT, for automatically operationalizing domain theories for deductive program synthesis. Section 6 also presents a detailed example of the DSDRAT analysis applied to an existing AMPHION domain theory. Section 7 concludes the paper.

2. AMPHION overview

AMPHION is a KBSE system that first guides a user in developing a formal specification of a problem. A user is guided in creating a diagram that represents the formal specification through a structured editor and a visual-programming interface. (The tables that drive the GUI are compiled from a domain theory.) Then AMPHION implements this specification, through deductive synthesis [6] using the same domain theory, as a program consisting of calls to components from a domain-oriented software library. AMPHION is described in detail in [5].

AMPHION is a generic system that enjoys the advantage of domain independence, being specialized to different domains via different domain theories. Maintenance, modification, and extension of a specialized AMPHION system is done by editing the domain theory. This is far easier than maintaining and extending ad-hoc, hand-crafted KBSE systems.

AMPHION has been applied so far to three application domains at NASA: AMPHION/NAIF (solar system geometry), AMPHION/CFD (computational fluid dynamics), and AMPHION/TOT (space shuttle navigation). Each of the AMPHION domain theories has been developed by an expert in deductive program synthesis, in consultation with domain experts. The NAIF domain theory is the most mature. It now consists of over three hundred axioms and has undergone over a dozen major extensions and revisions. AMPHION/NAIF has already generated programs that are in frequent use by space scientists. For example, AMPHION/NAIF has generated programs that perform geometry calculations, and animations, for science planning for the upcoming Cassini mission to Saturn. META-AMPHION will enable NASA domain experts, such as the group at JPL which developed and maintains the NAIF software component library, to maintain and extend specialized AMPHION systems themselves. The experiments and examples in this paper are from the AMPHION/NAIF system.

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 specifications, and an implementation relation between the abstract and concrete theory. The implementation relation is axiomatized in the style of Hoare [3] through *abstraction maps* from concrete sorts to abstract sorts. Abstraction maps are often parameterized. To facilitate posting constraints during deductive synthesis, abstraction maps are reified, i.e., treated as first-order objects. The *abs* function is used to apply a reified abstraction map to a concrete object. An example used in this paper involves an abstraction map in the NAIF domain theory that maps from time co-ordinates in various representations (e.g., ephemeris time or universal time) to abstract time (independent of representation). The term *abs(coordinates-to-time(T_s), c)* denotes an abstract time, obtained by applying the abstraction map *coordinates-to-time*, parameterized on a time system, T_s, to a time co-ordinate, *c* (time co-ordinates are translated to reals or strings in FORTRAN).

To synthesize programs, AMPHION invokes SRI's FOL resolution theorem-prover SNARK [11]. Deductive synthesis generates a term (representing a program) in the concrete language from a specification in the abstract language. AMPHION includes a strategy to guide SNARK from abstract, specification-level constructs towards concrete, implementation-level constructs. This strategy is highly effective, reducing program synthesis times from hours (sometimes days) to minutes (see [5]). This strategy is also robust and does not need to be tuned for individual problems, but often does require manual tuning when a domain theory is modified.

META-AMPHION: Synthesis of Efficient Domain-Specific Program Synthesis Systems

Michael R. Lowry
Recom Technologies, NASA Ames
Code IC, M. S. 269-2
Moffett Field, CA 94035 USA
lowry@ptolemy.arc.nasa.gov

Jeffrey Van Baalen
Computer Science Department
University of Wyoming, P.O. Box 3682
Laramie, WY 82071 USA
jvb@uwyo.edu

Abstract

AMPHION is a real-world knowledge-based software engineering (KBSE) system whose program synthesis subsystem is based on deductive synthesis. AMPHION has a domain-independent generic architecture that is specialized to a domain through a declarative theory. Program synthesis has been made efficient and automatic through manual tuning of theorem-proving strategies and tactics, and careful formulation of domain theories.

The META-AMPHION system is being developed to empower domain experts to develop, maintain, and evolve their own AMPHION applications. META-AMPHION is intended to be the knowledge-based analogue of application-generator generator technology. This paper describes technology for automatically transforming declarative domain theories into efficient domain-specific program synthesis systems.

1. Introduction

Application generator technology is a successful, mainstream, domain-oriented program synthesis technology. For example, as early as the mid nineteen-eighties, over half of the COBOL code generated annually was synthesized by application generators (such as screen generators). In comparison, knowledge-based program synthesis (KBPS) technology has had little real-world impact to date. One reason for this difference is technological maturity. Application generators rely upon well-understood and mature compiler technology. KBPS technology is far less mature.

Although relatively mature, current compiler technology is limited in the kinds of transformations it can effect from source language to target language. This limits the distance that can be spanned from source-language constructs to target-language constructs. Consequently, source languages for application generators are typically more programming-oriented than specification-oriented. Furthermore, the generated code is often inefficient since the code-generation process usually relies upon context-free macro expansion and template instantiation. KBPS research aims to overcome both these limitations.

A more important reason for the success of application-generator technology is the ease with which application generators can be generated themselves. In other words, application-generator generator technology [1] has greatly lowered the expertise required to construct an application generator. Tools such as parser generators (e.g., YACC) and GUI-builders facilitate the development of application-generator user interfaces. Semantic-attribute grammar technology and macro-definition languages facilitate the development of the code-generation subsystem of an application generator. Application-generator generator technology enables bachelor-level computer scientists to develop their own application generators.

In order for KBPS to achieve comparable success, technology is needed that greatly lowers the expertise required to develop and maintain automatic domain-oriented KBPS systems. The goal of the research described in this paper is to develop the knowledge-based analogue of application-generator generator technology. Specifically, starting with a declarative description of an application domain, we aim to automatically generate an efficient, automatic, domain-specific KBPS; roughly comparable to a system that would be hand-crafted by an expert in KBSE.

A major premise of our approach is that it is essential for the application-domain description to be declarative, in order to enable domain experts to generate, validate, and maintain such domain descriptions. We have found that our domain experts (mainly physical scientists and engineers at NASA) are familiar, through their training, with the declarative semantics of first-order logic (FOL). They are able to interact with us to validate FOL domain theories and suggest corrections and refinements. (This is not to say that they find the current notation of our domain theories to be congenial, but rather that they understand the idea of model-theoretic interpretations.) In contrast, the operational semantics of KBPS systems are exceedingly difficult for them to understand, whether such a system is based on