

# Automated Synthesis of Constrained Generators

Wesley Braudaway\*

Department of Computer Science  
Rutgers University  
New Brunswick, NJ 08903  
ARPAnet:wes@aramis.rutgers.edu

Chris Tong\*

Department of Computer Science  
Rutgers University  
New Brunswick, NJ 08903  
ARPAnet:ctong@aramis.rutgers.edu

## Abstract

*Knowledge compilation* is an emerging research area that focuses on "compiling" a problem solver's inefficient, explicit knowledge representation into more efficient, implicit forms. This paper presents a technique that transforms a declarative problem description (specifying the problem but not how to solve it) into a reasonably efficient, generate-and-test problem solver. Our technique performs *constraint incorporation*, modifying the parameter generators so they only generate values that satisfy the problem constraints. Successful constraint incorporation depends upon choosing the right solution representation (i.e., the set of parameters). Having expressed a constraint in terms of a particular set of parameters, incorporation fails if the constraint is not factorable into constraints on the individual parameter generators. RICK, a Refinement-based constraint Incorporator for Compiling Knowledge, is a prototype program that compiles a problem specification into a problem solver using least commitment, *top-down refinement* to achieve constraint incorporation. RICK refines an *abstract* solution representation to avoid premature commitment to representations that hinder constraint incorporation. RICK is able to incorporate *local* constraints that constrain relatively small portions of the entire solution. We have tested these ideas by having RICK automatically construct a house floor planning problem solver.

\*The research reported here was supported in part by the Defense Advanced Research Projects Agency (DARPA) under Contract Number N00014-85-K-0116, in part by the National Science Foundation (NSF) under Grant Number DMC-8610507, and in part by the Center for Computer Aids to Industrial Productivity (CAIP), Rutgers University, with funds provided by the New Jersey Commission on Science and Technology and by CAIP's industrial members. The first author has also received support from IBM. The opinions expressed in this paper are those of the authors and do not reflect any policies, either expressed or implied, of any granting agency.

\* currently on leave at IBM Watson Research Center.

## 1 Introduction

Many AI systems perform run time evaluation of *explicitly* represented domain knowledge to find a solution to a problem. Such systems make a particular tradeoff between the "explicitness of the representation and the efficiency of the computation" [Dietterich (ed.), 1986]. Explicitly represented knowledge is easier to gather, but more costly to evaluate. This observation has spawned a research area called *knowledge compilation*, which includes methods for "compiling" an inefficient, explicit knowledge representation into more efficient, implicit forms. The work described in this paper is being conducted within the context of the KBSDE project [Tong, 1986]. The purpose of the project is to develop compilation techniques whose input is a declarative problem description that does not specify *how* to solve the problem. The problem description includes a set of *problem constraints* that must be satisfied by a solution. The compilation techniques we are developing produce a *generate-and-test* problem solver. Such a problem solver must *generate* a solution that passes a set of *tests*, one for each problem constraint. We have attempted to develop a compilation method that places as few requirements as possible on the domain, so that the method is broadly applicable. Thus, we have made the very weak requirement that the problem be solvable using a generate-and-test algorithm. In addition to insisting that our compiler be general, we have also required that it produce reasonably efficient algorithms.

Like computer language compilers, knowledge compilers can use *optimization* techniques to improve the performance of the resulting problem-solving system. One optimization technique for a generate-and-test architecture is *test incorporation* [Dietterich and Bennett, 1986]. Test incorporation involves test movement, constraint incorporation, or both. *Test movement* regresses tests back into the generator process to achieve early pruning without affecting the correctness of the problem solver. *Constraint incorporation* [Tappel, 1980] modifies the generator so that it enumerates *only* those values which satisfy a particular problem constraint; we will call such a generator a *constrained generator*. The test corresponding to the incorporated constraint can be removed from the generate-and-test problem solver. Constraint incorporation reduces the size of the problem solver's search space and results in a more efficient problem solver. Con-

straint incorporation makes use of explicitly represented domain knowledge (e.g., specifications for system components), thus distinguishing it from more conventional code optimization approaches which rely solely on syntactic knowledge (e.g., data dependencies among system components).

This paper focuses on constructing constrained generators of *hierarchical* solutions, i.e., solutions that have parts (e.g., a house with rooms). Hierarchical solutions can be generated using correspondingly hierarchical generators (e.g., Figure 3). That is, solution parts are generated by sub-generators, and primitive solution parameters are assigned values by parameter generators. A particular *solution representation* is defined by a set of primitive parameters. To incorporate a constraint into a hierarchical generator requires that it be localizable to and incorporated in one or more of the (primitive) parameter generators. Incorporating constraints is difficult when the solution representation is "inappropriate". A constraint expressed in terms of a particular solution representation can have a structure that does not "match" that of the generator; that is, the constraint may not be factorable into constraints on the individual parameter generators. We will call this difficulty the *structure mismatch problem*. This paper demonstrates a system which incorporates the constraints while avoiding this structure mismatch problem. RICK, a Refinement-based constraint Incorporator for Compiling Knowledge, is a prototype program that compiles a problem solver using least commitment, *top-down refinement* to achieve constraint incorporation. The least commitment approach helps to avoid a premature commitment to a representation that may lead to the structure mismatch problem.

Section 2 of this paper defines the class of domains for which our method applies and illustrates an example from this class: the house floor planning domain. Section 3 illustrates the structure mismatch problem in the house floor planning domain. The fourth section discusses the concepts behind our approach and illustrates their implementation in RICK. We compare our work with related research in Section 5. Finally, Section 6 summarizes the paper and some of the limitations of our approach.

## 2 The problem domain

Parameter instantiation design problems. The KBSDE project has focused on *design* domains where the problem solver constructs an artifact that satisfies a set of problem constraints. Many design problems can be viewed as *parameter instantiation* problems. The *hierarchical structure* of the artifact is usually pre-determined for a class of parameter instantiation problems; for instance, all house floor plans consist of rectangular rooms, which, in turn, consist of sides and corners. The design task remaining for the problem solver is to "fill in" and "interconnect" the structure by assigning values to the unspecified artifact parameters in a way that is consistent with the problem constraints. RICK constructs a problem solver for parameter instantiation problems whose solutions are composite objects (e.g., house floor plans). The problem constraints are presumed to be

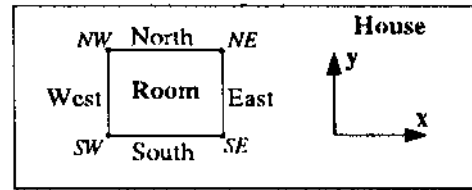


Figure 1: A House/Room Representation.

- SC1. **SIDE MAGNITUDE.** Each side of a room must have a minimum magnitude of 4 feet.
- SC2. **INSIDE HOUSE.** Each room must be inside the house.
- SC3. **SIDE ADJACENCY.** Each room must share a side with a house side.
- SC4. **NON-OVERLAPPING.** No two rooms may overlap.
- SC5. **FILL HOUSE.** The set of rooms must fill the house.

Figure 2: Constraints defining acceptable floor plan solutions.

hard constraints that define feasible solutions (rather than soft constraints that define the relative optimality of feasible solutions). We also presume that the design problem is not over-constrained: a solution that satisfies all the constraints can be found.

A house floor planning domain. To illustrate our ideas, we will use the parameter instantiation problem of constructing "house floor plans". A house floor plan is a two-dimensional, rectangular house placed at the origin of an x-y grid and having rectangular rooms as parts; such a floor plan is abstractly depicted in Figure 1. All lengths and coordinates are multiples of 1 foot. The solution is a floor plan that contains a problem-specific number of rooms, and satisfies all of the problem constraints shown in Figure 2. Constraints SC1, SC2 and SC3 are "local" constraints since they constrain each individual room. Constraints SC4 and SC5 are more "global," constraining *pairs* of rooms, and *all* rooms, respectively. Our examples will focus on the incorporation of the local constraints.

## 3 Constraint incorporation and representation shift

A simple but inefficient generate-and-test. A simple, hierarchical, generate-and-test problem solver for the floor planning domain is shown in Figure 3. The generator creates a candidate solution containing descriptions for each of the  $n$  rooms specified by a particular problem. Each room is generated by assigning values to its parameters using the initial representation:  $(x,y)$  (location of the SW room corner),  $l$  (length in the x direction), and  $w$  (width in the y direction). The generators are invoked in a particular sequence (e.g.,  $x \rightarrow y \rightarrow l \rightarrow w$ ). After all the generators have been invoked, the testers evaluate the candidate solution for

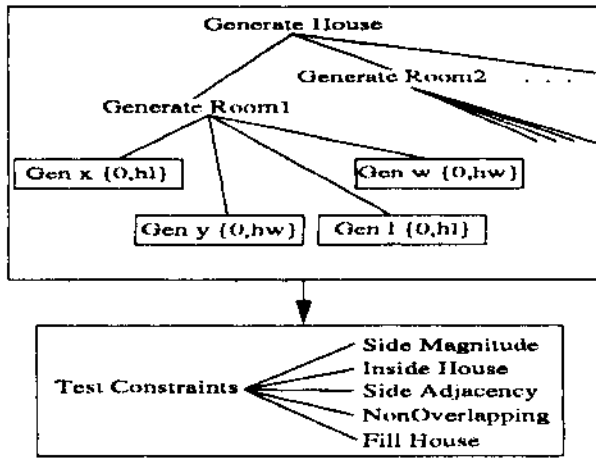


Figure 3: A simple generate-and-test floor planning algorithm. Brackets indicate the generator range for the primitive parameter generators (boxed)<sup>1</sup>.

satisfaction of the problem constraints. A failure of any test results in chronological backtracking through the generator sequence so a new candidate solution can be generated.

For a 9x9 foot house with 4 rooms, this problem solver (Figure 3) has a generation space containing  $10^{16}$  candidate solutions (4 rooms, each having 4 parameters that each range over 10 values). Only a small number of these are acceptable solutions (that satisfy constraints SC1-SC5). In a solution for this problem, each room is in a house corner; One room has 4 possible sizes ( $l = 4 \mid 5$  and  $w = 4 \mid 5$ ); its two adjacent rooms have only two possible sizes ( $w = 4 \mid 5$  for one,  $l = 4 \mid 5$  for the other); the final room fills the remaining area. Thus there are  $16 = 4 \times 2 \times 2 \times 1$  different room configurations. These rooms can be labelled  $R_1, \dots, R_4$  in  $4!$  different ways. Thus the total number of acceptable solutions is  $16 \times 4! = 384$ .

Assuming a uniform distribution of acceptable solutions in the space of candidate solutions, the generate-and-test problem solver will generate roughly  $10^{16} / 384 = 2.6 \times 10^{14}$  candidates before finding a solution. This problem solver is very inefficient.

The structure mismatch problem. We use constraint incorporation to improve the efficiency of the problem solver. For example, constraint SC1 is easily incorporated by changing the lower bounds of the length and width generators from 0 to 4. The candidate solutions that violate SC1 will not be generated; this leaves a search space of roughly size  $1.7 \times 10^{14}$  candidate solutions (4 rooms, each having 4 parameters, 2 ranging over 10 values, and 2 ranging over 6 values).

Unfortunately, incorporating a constraint can be difficult when the problem constraint on the entire solution cannot be partitioned into constraints on individual parameter generators, as we have just done. This case can occur when an inappropriate solution representation is chosen. For example, representing rooms by the four pa-

The  $hl$  (the house length value in the  $x$  direction) and  $hw$  (house width value in the  $y$  direction) are parameters whose values are problem-specific.

rameters  $\langle z, y, l, w \rangle$  hinders incorporation of the "Side Adjacency" constraint (SC3) that "all rooms share a side with a house side:"

$$\forall R \exists RS, HS [room(R) \wedge sideof(RS, R) \wedge sideof(HS, house) \rightarrow segmentof(HS, RS)].$$

Assuming the "Inside House" constraint (SC2) has already been incorporated, the "Side Adjacency" constraint simplifies to:

$$z(R) = 0 \vee y(R) + w(R) = hw \vee z(R) + l(R) = hl \vee y(R) = 0.$$

This constraint implies that a room must either have its west side on the west side of the house, or its north side on the north side of the house, etc. Because this constraint refers to all the room parameters, and is a disjunction (when expressed using this room representation), there is no easy way to factor it into constraints on the individual room parameters. Because of the structure mismatch problem in this case, incorporation fails.

Constraint incorporation as representation shift. Note, however, that the constraint refers to the containment (segmentof) of a room *side* within a house side. A more appropriate solution representation would have represented a room rectangle by two perpendicular "side objects" (each "side object", in turn, having a location and a length). Given this representation, we could incorporate the "Side Adjacency" constraint by simply modifying the generator of one room side so as to place it along one of the four house sides.

Our example suggests an obvious method for incorporating a constraint: shift the representation of either the constraint, the generator or both, until incorporation succeeds. Unfortunately, the space of alternative representations is rather large. Also, the need to incorporate multiple constraints into a single generator means that the method must find a *single* representation that enables incorporation of *all* the constraints. Problem reformulation is a difficult research problem with many unresolved issues [Korf, 1980]. In the next section, we present an alternative approach for knowledge compilation that reduces the combinatorial explosion when searching a space of representations.

## 4 Knowledge Compilation as Incremental Constraint Incorporation

### 4.1 Viewing Constraints as Generators of Partial Solutions

Construction of the inefficient, generate-and-test problem solver depicted in Figure 3 was based on viewing a problem constraint as a *test on complete solutions*. In contrast, our constraint incorporation approach (implemented in the RICK program) is based on viewing a constraint as a *constrained generator of partial solutions*. For example, the "Side Adjacency" constraint (SC3):

$$\forall R \exists RS, HS [room(R) \wedge sideof(RS, R) \wedge sideof(HS, house) \rightarrow segmentof(HS, RS)],$$

can be re-expressed as a *partial room generator*  $G_p(SC3)$ <sup>2</sup> that constructs only a single room side (on the house boundary):

**Select a House Side,  $HS$ , from  $\{W, N, E, S\}$   
 Generate a Room Side,  $RS$ , from  
 $\{W, N, E, S\} \cap SidesParallelTo(HS)$   
 with Corners  $\in Points(HS)$**

In contrast, a generator of *complete* rooms would construct complete 4-sided rooms. Generator  $G_p(SC3)$  is *constrained* since it only generates (partial) solutions that satisfy the Side Adjacency constraint.

Generator  $G_p(SC3)$  is expressed in terms of *abstract objects* such as room sides and house sides, instead of primitive parameters such as  $x$ ,  $y$ ,  $l$ , or  $w$ . This formulation of the generator is motivated by the constraint itself, which only specifies these abstract objects, and makes no commitment to their primitive parameter representation. Several representation alternatives are usually possible; to unnecessarily commit to a particular one might make incorporation of other constraints difficult or impossible (due to the structure mismatch problem).

In short, to facilitate a least commitment approach, the constraint should be re-expressed as a "partial solution generator" that is:

- *partial*. It only generates those parts of the (hierarchical) solution specified in the constraint.
- *abstract*. The generated parts are expressed in terms of abstract objects.

Which constraints can be re-expressed in this way? Almost by definition, in order for a constraint to be re-expressible as a generator of partial solutions, it must be a "local" constraint; that is, it must constrain only a part of the solution. This paper summarizes how RICK incorporates the three constraints SC1-SC3 (Figure 2) that are local to individual rooms.

#### 4.2 Extending this Idea into a Complete Method

The following observations will help motivate the extension of this idea into a complete method:

- (PARTIAL) By construction, the partial solution generator  $G_p(C)$  only generates solutions that satisfy the constraint  $C$ .
- (EXTEND) Any consistent extension of  $G_p(C)$  into a generator of *complete* solutions will also only generate solutions that satisfy  $C$ , thereby completely incorporating the constraint. In a similar manner, if a constrained generator of partial solutions is guaranteed to satisfy a *set* of constraints, then so will any consistent extension.

We will call such extensions *refinements* of the partial solution generator. For instance, the "single room side" generator  $G_p(5C3)$  generates room sides on the house boundary (thus creating partial rooms that satisfy SC3).

<sup>2</sup>The notation  $G_p(C)$  specifies a partial solution generator whose generated values always satisfy the constraint  $C$ .

$G_p(5C3)$  can be extended into a *complete* room generator that generates rooms with (at least) one side adjacent to the house boundary. Thus the "Side Adjacency" constraint SC3 is still guaranteed to be satisfied.

To represent the hierarchical structure of the artifact (e.g., a house with rooms), RICK uses an *object hierarchy* to organize the knowledge needed for constructing partial solution generators and their refinements. The hierarchy contains two types of relations between objects: "subclass" ("isa") and "part-of". The hierarchy economically (and modularly) represents an entire set of alternative solution representations, at differing levels of abstraction. This hierarchy also contains the relevant structural constraints common to rectangular objects (e.g., the West room side is perpendicular to the North room side.)

We have described the first and third steps of a complete method for incorporating constraints by refinement. The three steps are:

- STEP1. Re-express each constraint  $C$  as a partial solution generator  $G_p(C)$  (see PARTIAL).
- STEP2. Merge each partial solution generator into a single solution generator  $G_m(\{C_i\})$  which is guaranteed to satisfy the set of constraints  $\{C_{ij}\}$
- STEP3. Refine generator  $G_m(\{C_{ij}\})$  into a generator of complete solutions (expressed in terms of primitive parameters) (see EXTEND).

For a more complete description of the method, refer to [Braudaway, 1988].

By representing partial solution generators, we have fulfilled the major requirement of a *least commitment* approach for designing constrained generators. That is, when RICK incrementally constructs the generator, it only adds more detail when necessary. Since RICK constructs a partial solution generator *from* the problem constraint (achieving incorporation during construction), it avoids the structure mismatch problem described in Section 3. We have replaced the structure mismatch problem with the new problem of merging several partial solution generators into a single consistent solution generator.

#### 4.3 Merging Partial Solution Generators

In our floor planning example, RICK creates three partial solution generators, one for each of the local constraints SC1-SC3 (Figure 2). From constraints SC1 and SC3, RICK produces two "room side" generators, and from constraint SC2 it produces a generator pair for the diagonally opposed room corners, SW and NE. Merging these partial solution generators is complicated by the *interactions* that occur between them. For example, each room must both be inside the house (SC2) and have at least one side on the house boundary (SC3).

When "simulated", a partial solution generator creates "values" that are either abstract objects from the object hierarchy (e.g., corners SW and NE for a room  $R$ ), computable functions on such abstract objects (e.g., corner SW  $\perp$  *Points(lcft* house boundary)), or both. To

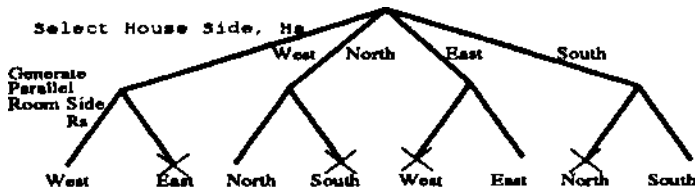


Figure 4: Paths formed during simulation.

say that generators *interact* means that at least one combination of generatable values is *inconsistent*.

This suggests a simple approach for solving the interaction problem:

- Systematically generate all combinations of values of partial solution generators, testing each combination (an abstract, partial solution) for consistency.
- Construct the merged generator by caching *only* the consistent solutions.

Figure 4 illustrates RICK's application of this method to the floor planning domain. Each combination of values is depicted as a different path in a search tree. For example, the left most path in the tree is constructed when the partial solution generators produce values: [HS=House West Side, RS=Room West Side]; the next left most path corresponds to [HS=House West Side, RS=Room East Side]; etc.

After generating a particular path in the search tree (a particular value combination), a corresponding set of constraints, implied by the value combination is collected. For example, the value combination [HS=House West Side, RS = Room East Side] implies that the East room side must be placed on the West house side:  $X_{\text{coord}}(\text{SEcorner}(\text{RS})) = X_{\text{coord}}(\text{SWcorner}(\text{HS}))$ .

Because all the constraints in this domain are linear, algebraic constraints, RICK uses a linear program (the Simplex method) to determine their consistency (i.e., the existence of a feasible solution to these constraints).<sup>3</sup>

The linear program discovers that the above constraint is inconsistent with (SC2), the "Inside House" constraint:

$$X_{\text{coord}}(\text{SWcorner}(\text{HS})) < X_{\text{coord}}(\text{SWcorner}(\text{RS})),$$

and a constraint associated with rectangles in general:

$$X_{\text{coord}}(\text{SWcorner}(\text{RS})) < X_{\text{coord}}(\text{SEcorner}(\text{RS})).$$

The inconsistency of these constraints follows by transitivity. Intuitively, this indicates that the abstract solution has forced the SW room corner outside the house.

If an abstract solution is consistent, RICK *caches* it into a unified generator description. As illustrated in Figure 4, four out of eight abstract solutions are found to be consistent (the ones in the figure that have not been crossed out), and are cached into a single, unified generator description:

<sup>3</sup>We realize in retrospect that we should use an integer programming method to guarantee the soundness of the answer. We can also extend this approach by using routines that check the consistency of other classes of (nonlinear) constraints. Note that this floor planner *cannot* be implemented as a linear program, primarily because some of the constraints SC1-SC5 are expressed as disjunctions.

Select a House Side *HS*, from {W, N, E, S}

Generate a Room Side, *RS*, from

{W, N, E, S}  $\cup$  SidesCorrespondingTo(ffS)<sup>4</sup>  
with Corners E *Points*(HS)

The merge process has thus resolved the interaction between the "Side Adjacency" (SC3) and the "Inside House" (SC2) constraints.

By using a "good" abstraction hierarchy, only relatively abstract value combinations need to be tested. Furthermore, there are a relatively small number of these abstract solutions. The pruning of inconsistent *abstract* solution candidates removes entire equivalence classes of inconsistent solution candidates produced by a *complete* solution generator.

#### 4.4 Refining into Primitive Representations

After merging the partial solution generators (STEP2), the resulting unified generator must be refined into a *complete* generator, which consists of a set of primitive parameter generators. Constraint incorporation has been completed in the merging step. Thus all complete generators refined from the merged generator (STEP3) will have the same generation space (even though it will be represented differently). Since RICK only distinguishes between alternative algorithms based on the size of their constrained generation spaces, it views all of the refinements as equally "good", and constructs all of them. In the floor planning example, these alternative refinements correspond to the various alternatives for representing a rectangle in terms of primitive parameters (e.g.,  $\langle x, y, l, w \rangle$ ).

In actuality, the alternative problem solvers are not equally good; differences in data and program structures cause differences in space and time complexity. Ideally, selection should be based on criteria for data and program structure optimization. In the current prototype system, the user selects the "best" alternative, which is then translated into LISP.

When applied to the floor planning example, our method incorporates the three local floor planning constraints SC1-SC3 (Figure 2). The resulting problem solver (Figure 5) generates each room by picking a room location (X,Y) from a point on a house side. The room specification is completed by generating length and width values (starting from the required minimums) that place the room inside the house with respect to the room's location (X,Y).

This problem solver generates roughly  $6 \times 10^{10}$  candidate solutions for a 9x9 house with 4 rooms (see Figure 5). For each of the 4 rooms, there are 4 ways to pick a house side, HS; for each of these cases, there are 21 ways to pick a room side that is a subsegment of that house side; also, there are 6 ways to pick the perpendicular magnitude. Thus the total number of candidate solutions is  $(21 \times 6 \times 4)^4$ .

This is a reduction in complexity of 6 orders of magnitude from the inefficient, generate-and-test problem

<sup>4</sup>A mapping that associates West room side with West house side, North room side with North house side, etc.

```

(GENERATE-OBJECT ROOM NUMBER-OF-ROOMS ;basic loop.
(ENUMERATE HS ;Enumerate house side range from:
(W N E S))
((GENERATE X
;Generator room X location range from:
(CASE HS (W '0) ;place room on house west side.
(N [0 (- hl 4)])
(E hl)
(S [0 (- hl 4)])))
(GENERATE Y
(CASE HS (W [0 (- hw 4)])
(N hw)
(E [4 hw])
(S '0)))
(GENERATE L ;Length in X direction
(CASE HS (W [4 hl])
(N [4 (- hl X)])
(E [(- hl) -4])
(S [4 (- hl X)])))
(GENERATE W ;Width in Y direction
(CASE HS (W [4 (- hw Y)])
(N [(- hw) -4])
(E [(- Y) -4])
(S [4 hw]))))

```

Figure 5: Generator created by constraint incorporation.<sup>5</sup>

solver previously shown. Further research will attempt to extend RICK, allowing it to also incorporate necessary constraints. A "necessary constraint" that is not satisfied by a partial solution will not be satisfied by any extension of the partial solution. For example, the constraint that "no rooms overlap" is a necessary constraint in the floor planning domain, since, if rooms overlap in a partial solution, they continue to overlap in all extensions of that partial solution. The problem solver resulting from incorporation of this constraint will have a generation space of 1008 candidates (384 of which are solutions) for this 9x9 example.

## 5 Related Work

Knowledge compilation as a design process. To facilitate the comparison of our work with related work in knowledge compilation, it is useful to view knowledge compilation as a *design process* that designs a problem solver from the specification for a class of problems. We contrast three approaches according to their *model of the design process* and the manner in which domain knowledge is used to produce reasonably efficient algorithms.

Knowledge compilation as iterative re-design. Tappel [1980] defines test and constraint incorporation as modifications to a data-flow graph representing the components of the algorithm to be synthesized. A test refers to certain solution components; hence these solution components must be generated before the test can be run. Tappeler's approach moves a test backwards in the algorithm's data-flow graph until it is placed just after the generators of these referenced solution components. An attempt is also made to modify the generator so only values that satisfy the test will be enumerated.

Tappel's approach takes a generate-and-test algorithm as input and iteratively re-designs it by incorporating constraints.

The strategy used by DIOGENES [Mostow, 1988] is similar to Tappel's but uses a transformational approach to modify an initial generate-and-test algorithm (for possibly over-constrained problems) into a more efficient heuristic search algorithm. This approach uses domain knowledge that has been procedurally embedded in some or all of the transformations. However, the design knowledge for controlling the compilation process is supplied by the user of the DIOGENES system (who selects among applicable transformations). The model of design used by both of these strategies is essentially hill-climbing that iteratively re-designs a problem solver to improve its efficiency (though some steps do not directly improve efficiency but have the purpose of enabling other optimization steps).

Knowledge compilation as schema instantiation. KIDS [Smith, 1988] focuses on mapping declarative knowledge into a *global search* algorithm. A global search algorithm splits a set defining all candidate solutions of a problem into subsets and extracts solutions from the subsets when possible. This approach uses declarative domain knowledge in the form of procedural schemas. For example, one global search schema enumerates all bounded sequences over a finite set. The method finds the domain-specific schema that best matches the problem domain, and then instantiates it into an efficient algorithm (using information drawn from the problem specification). Constraints are formulated and incorporated during instantiation of the procedural schema. KIDS is able to reason about problem specifications that include functional constraints (constraints on I/O behavior) and optimality criteria.

Knowledge compilation as top-down refinement. RICK constructs constrained generators of solutions whose feasibility is defined by problem constraints. In contrast with the iterative re-design approaches, which incorporate constraints *after* a complete problem solver has been created, RICK refines the problem constraints into partial solution generators (which are then merged), thus simultaneously designing the problem solver and incorporating constraints. In contrast with KIDS, RICK designs the problem solver by refining the problem description into a correct *data* representation (i.e., a representation of the generated solution) rather than choosing the proper *procedural* representation for the problem. By using a least commitment approach for representation selection, RICK decouples the issues of incorporating constraints and representing generated solutions.

Difficulties in designing a representation. Expressed in terms of a *particular* solution representation, some constraints are *local* constraining relatively small portions of the entire solution, while others may be *global* constraining the entire solution or interrelating several parts of the solution. Therefore, whether a con-

The CASE statement defines a conditional generator range that depends on the value of HS.  $[A B]$  is the generation sequence from A to B, incremented by 1.

straint is local or global is a function of the solution representation. To incorporate as many constraints as possible into a more constrained problem solver, RICK must use a representation which maximizes the number of local constraints (the type of constraints RICK can incorporate during compilation). However, the odds are that no single representation language exists in which *all* the constraints are local. In other words, not all problem constraints can be incorporated using the method described in this paper. Another research effort within the KBSDE project [Voigt, 1989] focuses on the problem of procedurally embedding the "left over" (unincorporated) global constraints.

Other work. Other related work includes the discussion by Steier and Kant [1985] on "developmental evaluation", a framework which could be said to include our abstract, symbolic simulation approach; Cohen's program [1986], which translates a declarative specification into a generate-and-test algorithm; and Van Baalen's program [1988], which iteratively *designs* useful, analogical problem representations (for solving verbal reasoning problems such as those found on college admissions tests) built from a library of standard data structures.

## 6 Conclusions

Complete incorporation of a constraint into a generator is a difficult task, especially when the constraint (expressed in terms of the generated parameters) has a structure that is not factorable into constraints on the individual parameter generators. This paper has described and illustrated the results of the RICK program which avoids this structure mismatch problem by viewing problem constraints as *generators of partial solutions*. RICK incorporates a problem constraint by refining it into a generator of partial solutions that are guaranteed to satisfy the constraint. These partial solution generators are then merged into a single abstract generator. During merging, interactions between partial solution generators are resolved by "simulating" the generators and caching only those generated combinations of values that are proven to be consistent. Since the constraints are fully incorporated into the merged generator, further refinement of the merged generator into a complete generator of primitive parameters will not create a structure mismatch problem. RICK produces problem solvers that solve parameter instantiation problems for design domains. Thus, RICK incorporates constraints defining structural relationships, but does not currently incorporate constraints that restrict I/O behavior, or specify optimization criteria. In particular, RICK currently incorporates local constraints that are linear, algebraic constraints on composite objects.

Further work will attempt to extend the type of constraints that RICK can incorporate, and to extend the structural representation of the hierarchical solution. This includes generalizing RICK to handle a hierarchical solution containing multiple objects and necessary constraints on multiple objects (i.e., some global constraints). Experiments are being conducted to test whether RICK is insensitive to changes in the object hierarchy representation and constraint predicate defini-

tions. The goal of this study is to determine the power and generality of this approach to algorithm synthesis.

## Acknowledgements

We would like to thank the members of the KBSDE group and the Knowledge Compilation Seminar. We are also grateful to the members of the Rutgers AI/Design Project for the stimulating environment they provide.

## References

- [Braudaway, 1988] W. Braudaway. Constraint incorporation using constrained reformulation. Technical report LCSR-TR-100, Dept. of Computer Science, Rutgers Univ., April 1988. thesis proposal.
- [Cohen, 1986] D. Cohen. Automatic compilation of logical specifications into efficient programs. In *Proc. of AAAI-86*, pages 20-25, Philadelphia, PA., August 1986. AAAI.
- [Dietterich and Bennett, 1986] T. Dietterich and J. Bennett. The test incorporation theory of problem solving (preliminary report). In *Proceedings of the Workshop on Knowledge Compilation*, pages 145-161, Oregon State Univ., September 1986. AAAI, Oregon State Univ.
- [Dietterich (ed.), 1986] T. Dietterich (ed.). In *Proceedings of the Workshop on Knowledge Compilation*. AAAI, Oregon State Univ., September 1986.
- [Korf, 1980] R. Korf. Toward a model of representation changes. *Artificial Intelligence*, 14(1):41-78, 1980.
- [Mostow, 1988] J. Mostow. A preliminary report on diogenes: Progress towards semi-automatic design of specialized heuristic search algorithms. In *Proceedings of the Workshop on Automated Software Design*, St. Paul, MN., August 1988. AAAI.
- [Smith, 1988] D. Smith. Kids: a knowledge-based software development system. In *Proceedings of the Workshop on Automated Software Design*, St. Paul, MN., August 1988. AAAI.
- [Steier and Kant, 1985] D. Steier and E. Kant. The roles of execution and analysis in algorithm design. *IEEE Transactions on Software Engineering*, SE-11(11):1375-1386, November 1985.
- [Tappel, 1980] S. Tappel. Some algorithm design methods. In *Proc. of AAAI-80*, pages 64-67, Stanford Univ., August 1980. AAAI.
- [Tong, 1986] C. Tong. KBSDE: An environment for developing knowledge-based design tools. In T. Dietterich, editor, *Proceedings of the Workshop on Knowledge Compilation*, pages 127-138, Oregon State Univ., September 1986.
- [Van Baalen, 1988] J. Van Baalen. Overview of an approach to representation design. In *Proc. of the AAAI-88*, pages 392-397, St. Paul, MN., August 1988. AAAI.
- [Voigt, 1989] K. Voigt. Automating the construction of patchers that satisfy global constraints. In *Proc. of IJCAI-89*, Detroit, MI, August 1989.