Knowledge-Based Design of Generate-and-Patch Problem Solvers that Solve Global Resource Assignment Problems

Kerstin Voigt*
Computer Science Department, Rutgers University
New Brunswick, NJ 08903
voigt@cs.rutgers.edu

Abstract

We present MENDER, a knowledge based system that implements software design techniques that are specialized to automatically compile generate-and-patch problem solvers that satisfy global resource assignments problems. We provide empirical evidence of the superior performance of generate-and patch over generate-and-test, even with constrained generation, for a global constraint in the domain of "2D-floorplanning". For a second constraint in "2D-floorplanning" we show that even when it is possible to incorporate the constraint into a constrained generator, a generate-and-patch problem solver may satisfy the constraint more rapidly. We also briefly summarize how an extended version of our system applies to a constraint in the domain of "multiprocessor scheduling".

Introduction.

The MENDER project presented here is aimed at developing techniques to automatically compile generate-and-patch problem solvers; these problem solvers efficiently construct solutions that satisfy a conjunction of interacting constraints. MENDER is a part of the larger KBSDE effort towards automating knowledge-based design of algorithms [Tong, 1991]. MENDER builds on recent successes in automatically compiling conjunctions of constraints on composite structures into *"constrained" generate-and-test* problem solvers [Braudaway, 1991]. The constrained generator is the result of incorporating into the generator constraints that constrain local parts of the composite structure ("local constraints"). The conjunction of constraints may feature other constraints which restrict all or most parts of the artifact simultaneously ("global constraints"), a property that prevents their successful incorporation. These constraints could be satisfied by placing testers after the constrained generator. However, the resulting generate-and-test problem solver may still perform grossly inefficient.

The MENDER research capitalizes on the observation that once a complete composite artifact has been generated, frequently a small number of local, well-directed modifications ("patches") to the artifact suffice to produce a solution to a constraint. We have developed and implemented the MENDER compiler which automatically builds such *generate-and-patch* problem solvers from an inefficient generate-and-test problem solver. MENDER automatically compiles a *hillclimbing* search component, called a *"patcher"*, and interfaces it with the generator. Hillclimbing patchers are defined by the set of *patching operators* (i.e. parameter value changes) and an *evaluation function* that measures progress towards constraint satisfaction. We show that cost-effective generate-and-patch problem solvers are possible with patchers that have the following two properties. (1) Patching operators do not result in violations of constraints that have been satisfied through generation. (2) Patching searches the space of parameter value changes in a *constraint-oriented* fashion; i.e. when choosing the next move, changes that promise higher degrees of constraint satisfaction are prefered over those that yield no improvement, or worsen the state.

The MENDER research has focused on the automatic compilation of *hillclimbing patchers* that are specialized to satify global constraints that in some abstract sense involve the assignment of "resources" to "consumers". *Resource assignment problems* (RAPs) are constraints that constrain the nature of assignment between components of a composite "consumer structure" and a composite "resource structure". Our con-

*The research reported here was supported in part by the National Science Foundation (NSF) under Grant Number IRI-9017121, in part by the Defense Advanced Research Projects Agency (DARPA) under DARPA-funded NASA Grant NAG2-645, in part by DARPA under Contract Number N00014-85-K-0116, in part by 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 opinions expressed in this paper are those of the author and do not reflect any policies, either expressed or implied, of any granting agencies.

vention is that the consumer structure corresponds to the output of generate-and-test or generate-and-patch problem solvers. The resource structure is typically some other *constant* structure that is given as a part of the specification of a RAP. RAPs are of interest to us because they can successfully model significant portions of important and well-known classes of problems (floorplanning, scheduling, n-queens,...). Secondly, they share features that make them conducive to the automatic compilation of efficient generate-and-patch problem solvers.

In this paper, we report on MENDER's success in automatically compiling, and interfacing with a constrained generator, a patcher that satisfies two global constraints in the domain of "2D-floorplanning". The first global constraint which we will refer to as "fill house" constraint is informally defined as

> Constraint "fill house": The rooms in the floorplan must cover the house area completely.

For this constraint we will briefly sketch the steps of MENDER's compilation of a generate-and-patch algorithm, and present the results of a preliminary performance study on the resulting problem solver.

The second constraint – refered to as "no overlap" – is defined as

> Constraint "no overlap": Rooms in the floorplan do not overlap.

We will not sketch the compilation of a generate-and-patch problem solver for "no overlap", but towards the end of this paper we will present performance data on the algorithm that MENDER constructed. The "no overlap" constraint is interesting to us because it lies on the boundary between local and global constraints. The RICK compiler [Braudaway, 1991] has been able to partially incorporate "no overlap" into a generator of floorplans by compiling filters that forward propagate necessary conditions derived from generated rooms to rooms that are to be generated next. We compared the performances of the RICK-compiled constrained generator with the MENDER-compiled generate-and-patch algorithm for "no overlap". We will see that generate-and-patch consistently performed better than constrained generation.

To show generality of MENDER's methods we will also briefly describe how MENDER can automatically compile a generate-and-patch problem solver for a constraint in the domain of "multiprocessor scheduling".

Classification-based compilation of hillclimbing patchers.

In the past, we have already presented a *classification-based* technique to construct hillclimbing patchers [Voigt and Tong, 1989]. The here presented 5-step technique is an improvement over the previous one in that a lattice-like taxomomy of abstract RAP schemas can be exploited in ways that eliminate costly matching and theorem proving. The lattice formed by the 16 abstract RAPs is illustrated in Fig. 1. The least

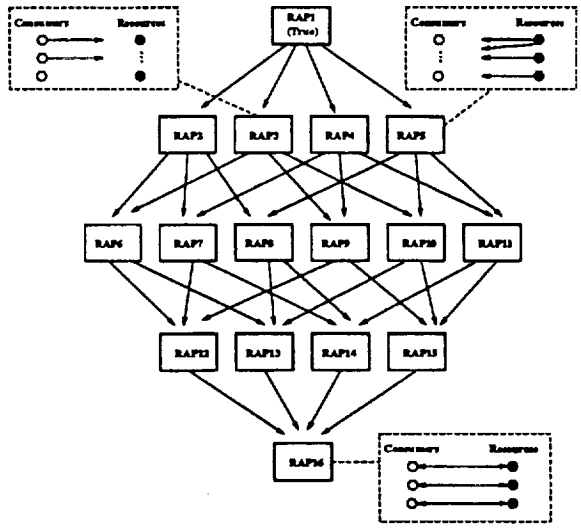

Figure 1: Taxonomy of abstract RAP schemas.

constrained RAP schema, RAP1 (unconstrained, or "True"), is found at the top of the lattice, the most constrained RAP schema, RAP16 (bijection from consumers to resources, and bijection from resources to consumers) forms the bottom of the lattice. RAP2, RAP3, RAP4, and RAP5 are the *basic* RAPs. All RAPs on the remaining layers of the lattice are constructed from the basic ones by conjunction, as indicated by downwards arrows (e.g. RAP6 ≡ RAP2 ∧RAP3). Thus, RAP16 is the conjunction of all four basic RAPs. We will see later how explicit knowledge of these relationships between the RAPs can be exploited in constructing efficient algorithms.

Step 1 – Classifying the constraint. In its first step, MENDER asks the user to define the global constraint, e.g. "fill house", in a guided sequence of questions. The answers to these questions are rephrased as a first-order logic representation of the concrete constraint. In parallel to formulating the concrete constraint MENDER parses a *taxomomy* of abstract RAPs; when all questions have been answered the abstract RAP class for the constraint has also been determined. For example, the "fill house" constraint may be formulated as

$$\forall p \{\text{point-of-rect}(p,Hs) \Rightarrow \exists rm [\text{member}(rm,Fp) \wedge \text{point-of-rect}(p,rm)]\}$$

> "Each point p in the rectangular house (Hs) is assigned to at least one room (rm) point in the floorplan (Fp)."

and is automatically classified as an instance of the abstract resource assignment problem RAP5, which is represented by the abstract first-order logic sentence

below:

$\forall r$ {*substruct(r,Resources)* $\Rightarrow$
$\exists c$ [*substruct(c,Consumers)* $\wedge$ *assign(r,c)*]}

"Each resource r in the resource structure (Resources) is assigned to at least one consumer c in the consumer structure (Consumers)."

MENDER pairs up the terms of the abstract RAP language (e.g. *substruct*, *assign*) with the concrete terms of "fill house" as follows:

p ..... *r*
rm ..... *c*
Hs ..... *Resources*
Fp ..... *Consumers*
point-of-rect(p,Hs) ... *substruct(r,Resources)*
member(rm,FP) ... *substruct(c,Consumers)*
point-of-rect(p,r) ..... *assign(r,c)*

Step 2 – Deriving an evaluation function. Associated with each abstract RAP is a *generic evaluation function* which indicates in abstract terms how to measure degrees of constraint satisfaction by quantifying assignments between consumers and resources. From RAP5 we retrieve the generic evaluation function shown in Fig.2. It computes the number of resources assigned to at least one consumer. MENDER specializes this abstract evaluation function into a concrete evaluation function for "fill house" by instantiating the abstract terms with the corresponding concrete terms. The evaluation function for "fill house" then computes the number of points p in house Hs that are also points in some room rm in floorplan Fp (Fig.2).

Typically MENDER faces a scenario in which a constrained generator exists which already guarantees the satisfaction of several constraints. MENDER's task consists in constructing and interfacing with this generator a patcher which will satisfy an additional global constraint. When some or all of the constraints that have been incorporated into the generator match several of the basic RAP schemas (or conjunctions thereof), then properties of the RAP lattice can be exploited to derive an evaluation function that is more efficient than the one in Fig. 2. For example, imagine that a constrained generator exists that guarantees that all rooms in the floorplan are located *inside* the house (RAP2) and do *not overlap* (RAP4), and and the house is *flat* (RAP3) (i.e. one room point cannot coincide with more than one house point). Then satisfying "fill house" (RAP5) actually implies satisfying the conjunction of these constraints. The conjunction in turn is equivalent to RAP16. We know that RAP16, that is, the conjunction of RAP2, RAP3, RAP4, and RAP5, implies that the number of consumer substructures must be equal to the number of resource substructures. I.e.

$$\text{RAP2} \wedge \text{RAP3} \wedge \text{RAP4} \wedge \text{RAP5} \Rightarrow \text{SIZE}(C) = \text{SIZE}(R),$$

```
Evaluation function for RAP5:

variables: r c out temp;
constants: Consumers Resources
begin-proc
  out <- 0
  forall SUBSTRUCTS r in Resources
    temp <- 0
    forall SUBSTRUCTS c in Consumers
      if ASSIGN(c,r) = true
         then temp <- 1
              exit-forall end-if
    end-forall
    out <- out + temp
   end-forall
   return out
end-proc
```

⇓ *INSTANTIATE* ⇓

```
Evaluation function for ''fill house'':

variables: p rm out temp;
constants: Fp Hs
begin-proc
    out <- 0
    forall POINT-OF-RECT p in Hs
       temp <- 0
       forall MEMBER rm in Fp
          if POINT-OF-RECT(p,rm) = true
             then temp <- 1
                  exit-forall end-if
        end-forall
     out <- out + temp
     end-forall
     return out
end-proc
```

Figure 2: Generic evaluation function for RAP5; instantiated for "fill house".

where C and R stand for *Consumers* and *Resources* respectively. We also know that the following implications and equivalences hold for the basic RAPs (we write C_a and R_a for "assigned consumers" and "assigned resources" respectively):

$$\text{RAP2} \Leftrightarrow \text{SIZE}(C) \geq \text{SIZE}(R_a) \wedge \text{SIZE}(C) = \text{SIZE}(C_a)$$
$$\text{RAP3} \Rightarrow \text{SIZE}(C_a) \geq \text{SIZE}(R_a)$$
$$\text{RAP4} \Rightarrow \text{SIZE}(C_a) \leq \text{SIZE}(R_a)$$
$$\text{RAP5} \Leftrightarrow \text{SIZE}(C) \leq \text{SIZE}(R_a) \wedge \text{SIZE}(R) = \text{SIZE}(R_a)$$

It follows that

$$\text{RAP2} \wedge \text{RAP3} \wedge \text{RAP4} \wedge \text{SIZE}(C) = \text{SIZE}(R) \Rightarrow \text{RAP5}$$

Therefore, knowing that all rooms are inside the house and do not overlap, and the house is flat,

$$\text{SIZE}(Fp)^{point} = \text{SIZE}(Hs)^{point} \Rightarrow \text{"fill-house"}$$

```
More EFFICIENT evaluation function
for ''fill house'':

variables: out,r
constants: Fp

begin-proc
    out <- 0
    forall r in Fp
      out <- out + rect-length(r) * rect-width(r)
    end-forall
    return out
end-proc
```

Figure 3: More efficient evaluation function for "fill house".

That is, the objective of "fill house" coincides with the *number of house points being equal to the number of all room points in the floorplan.* Progress towards satisfying "fill house" can be measured by the total number of room points in the floorplan. MENDER searches its knowledge base of data types and finds that the number of points of an object of type rectangle can be efficiently computed as the product of rectangle length and rectangle width. By adopting this measure as the revised evaluation function for "fill house" (see Fig. 3) MENDER constructs an evaluation function that computes significantly faster the original one (Fig. 2).

Step 3 – Characterizing "improving operators". Next MENDER continues by working with the less efficient evaluation function in Fig. 2. Two additional pieces of information are associated with a generic evaluation function: the direction of change towards greater satisfaction of the constraint("increase" or "decrease"), and a characterization of events that can cause the desired change. For RAP5 the direction of positive change is "increase" (i.e. higher value of the evaluation function indicates greater constraint satisfaction), and the event to cause such change is *"increase the frequency of 'ASSIGN(c,r) = true' "*. For "fill house" this translates into the information that the value of the evaluation function can be increased (improved) if the floorplan is modified such that more frequently 'point-of-rectangle(p,rm) = true'. MENDER regresses this event through the definitions of relevant datatypes and predicates in its knowledge base, and thereby determines that only increases of the "length" and "width" parameters of the rooms in the floorplan can – in one application – lead to greater "filling" of the house. These parameter changes are termed "improving operators" to distinguish them from parameter value changes (e.g. changes in room location, "shrinking" of rooms) that are guaranteed *not* to contribute towards greater constraint satisfaction.

Step 4 – Instantiating the "patcher shell". The *patcher shell* is a piece of code that realizes a basic *hillclimbing* strategy (with backtracking). It is rendered operational for a given global constraint by instantiating it with the concrete evaluation function and "improving operator" information. The patcher shell provides the choice of two types of "greedy" control strategies: "greedy", and "greedy and strictly ascending". These controls differ in how and to what extent the evaluation function and "improving operator" information are employed. Both controls use the evaluation function to order the applicable patching operators at each choice point in decreasing order of progress towards satisfying the constraint. The "greedy and strictly ascending" control restricts the set of patching operators to only those that are "improving". For example, to patch for "fill house" possible parameter value changes are limited to increased values in the "length" and "width" parameters of rooms. In "greedy" patching, which operates with the full set of applicable operators (e.g. *all* changes of room "length" ,"width", and changes of the "x-coordinate" and "y-coordinate" of room location points) knowledge of "improving operators" can help to significantly reduce the cost of evaluating the promise of legal "next" operators at each decision point. Among the set of options only "improving" operators are evaluated in detail, while the closer examination of the remaining operators (a priori known to make no or negative progress) is suspended until after all prefered operators have failed to provide a solution. We found empirically that for up to 80% of all applicable operations the evaluation function value was never computed, reducing the cost/node considerably.

Step 5 – Building "generate-and-patch". MENDER interfaces a constrained generator with a patcher such that constraints satisfied by generation will not be violated during patching. This is accomplished by making the patcher adhere to the same restrictions that are forced upon generation when incorporating the local constraints. While these restrictions limit the set of patching operators, chances are that sufficient options remain to produce solutions. It is also exactly because of these restrictions that the patcher search space is typically smaller than the original generation space, allowing faster problem solving by patching than backtracking and regenerating.

Experimental results.

We present the results of our preliminary empirical studies of the performances of the generate-and-patch problem solvers that MENDER constructed for the "fill house" and "no overlap" constraints.

"Fill house". In the generate-and-patch problem solver for "fill house" the (RICK-compiled) constrained generator guarantees that all generator outputs are floorplans with nonoverlapping rectangular rooms no smaller than 5× 5 units, and are lo-

cated inside a given house area and adjacent to at least one house wall. We compare performances of constrained generate-and-test (g&t), generate-and-patch with "greedy" control of patching (g&p:greedy), generate-and-patch with "greedy and strictly ascending" (g&p:gr-asc) control. As added control condition, we also test generate-and-patch without any informed control strategy (g&p:default). Our performance measure is the "repair effort" expended by each problem solver after the first candidate has been generated.[1] "Repair effort" for generation is measured in number of nodes (alternative selections of parameter values) expanded through backtracking and regeneration. Repair by patching is measured in number of nodes (modifications of parameter values) expanded within the patcher space.

We ran each problem solver 20 times for floorplans with 1, 2, 3, and 4 rooms. The constrained generator produced floorplans in random order. The house dimensions were chosen relative to the number of rooms, such that the smallest legal floorplan would cover ~ 30% of the house area. Our results, averaged over the 20 runs, are plotted in Fig.4. Overall we find

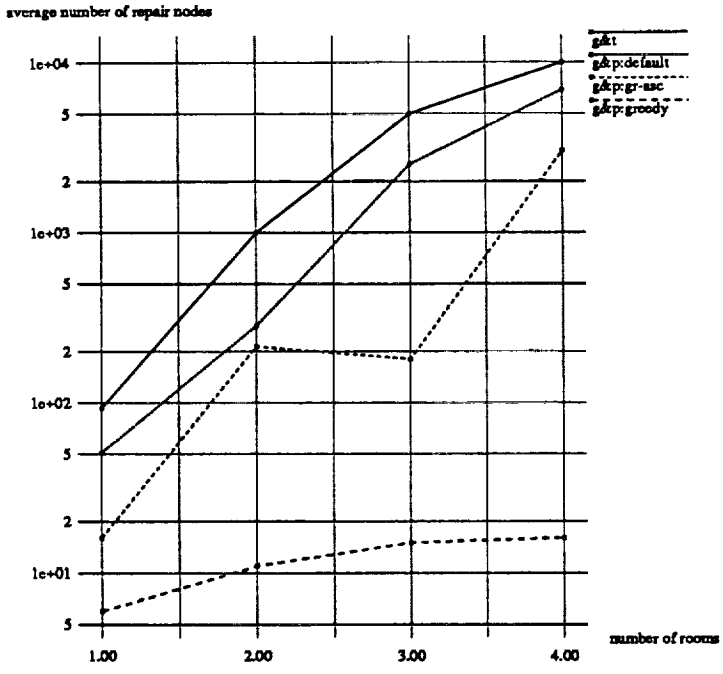

Figure 4: "Repair effort" of generate-and-test and generate-and-patch for "fill house" (y-axis scaled logarithmically).

that all types of generate-and-patch were consistently better than constrained generate-and-test which confirms our intuitions and observations. The plots show that generate-and-patch with "greedy" control very significantly outperformed constrained generate-and-test. E.g., for 4-room floorplan 10,000 generations[2] did not suffice to satisfy "fill house". In contrast, patching achieved repair in only 16 patcher nodes. Not surprising is the inefficient performance of generate-and-patch without control (g&p:def). Generate-and-patch with "greedy and strictly ascending" control was second best but notably less efficient than "greedy". In the past, we had shown that this type of problem solver can perform much better, when patching is interleaved with "block-preventing" moves that circumnavigate dead-ends [Voigt and Tong, 1989]. Naturally, this facility involves some additional cost which judging from the good performance of "greedy" was not warranted by our examples. At this point, we withhold judgment on the relative merits of "greedy" versus "greedy and strictly ascending patching". Which problem solver is likely to be more cost-effective seems to depend on the domain and the constraint. We need to study MENDER-compiled generate-and-patch problem solvers for larger numbers of more diverse constraints to obtain more conclusive results.

"No overlap". In a further study, we compare the performance of the MENDER-compiled greedy generate-and-patch problem solver (g&p:greedy) to satisfy "no overlap" with the RICK-compiled constrained generator (constr-g) which outputs nonoverlapping floorplans. We measure and compare the performances of both types of problem solvers in number of nodes expanded in the respective generation and patching spaces. We ran generate-and-patch and constrained generation 20 times for 2 to 5 rooms and a 15×15 house. Note that constraint incorporation by RICK does not prescribe a particular (constraint-dependent) generation order. Therefore, performance data were collected with a randomized generation order in the constrained generator. The results, averaged over 20 runs, are plotted in Fig. 5. We find that generate-and-patch finds solutions considerably faster than the constrained generator. These data show that the utility of generate-and-patch algorithms is not restricted to constraints whose extremely high degree of globality renders the compilation of a constrained generator infeasible or undesirable. Even when it is possible to compile a constrained generator, a generate-and-patch algorithm may be a viable, and potentially preferable alternative.

Applying MENDER to other domains.

At the present time all problem solvers that MENDER has automatically constructed solve problems in the domain of 2D-floorplanning. However, we have worked

[1] Generating the first candidate has a fixed cost shared by all problem solvers, and is therefore ignored in our study of comparative cost.

[2] Although the plot shows data point 10,000, patching was cut off at 10,000 nodes without solutions.

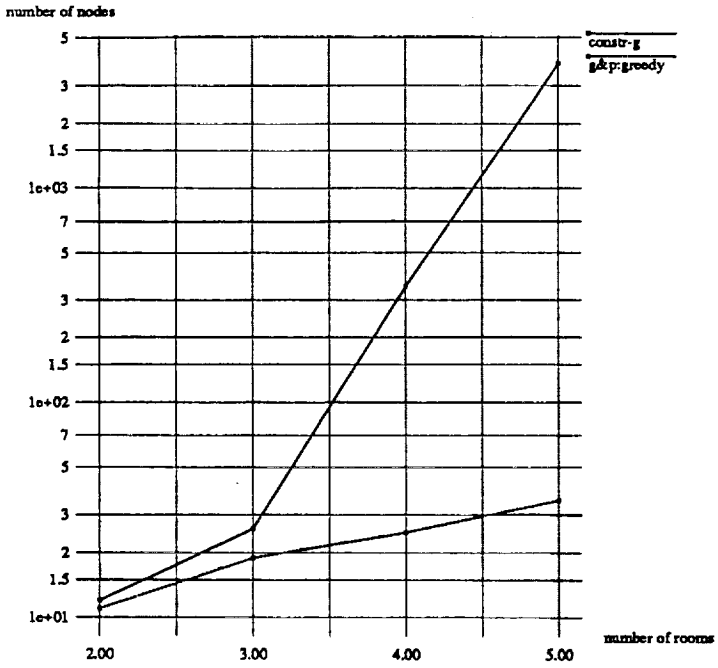

Figure 5: Satisfying "no overlap" by constrained generation vs. generate-and-greedy-patch (15x15 house) (y-axis scaled logarithmically).

out paper traces on how an extended version of our system will be able to compile generate-and-patch problem solvers for constraints in a variety of other domains (e.g. n-queens, graph-coloring, scheduling, VLSI-design) [Voigt, 1991]. Here we will briefly summarize how MENDER could be applied to a *multiprocessor scheduling* problem taken from the listing of NP-complete problems in [Garey and Johnston, 1979]:

> Multiprocessor Scheduling:
>
> "INSTANCE: Set T of tasks, number $m \in Z^+$ of processors, length $l(t) \in Z^+$ for each $t \in T$, and a deadline $D \in Z^+$.
>
> QUESTION: Is there a m-processor schedule for T that meets the overall deadline D, i.e., a function $\sigma : T \rightarrow Z_0^+$ such that, for all $u \geq 0$, the number of tasks $t \in T$ for which $\sigma(t) \leq u < \sigma(t) + l(t)$ is no more than m and such that, for all $t \in T, \sigma(t) + l(t) \leq D$?"

Suppose that this problem is presented to MENDER in terms of a consumer structure *Schedule* which is a set of *tasks* with known lengths and a resource structure *TimeTable* which is a list of consecutive *time slots*. The latest time slot corresponds to the deadline D. Assume that a generator produces schedules by assigning starting times to each task. To satisfy scheduling constraint a schedule must satisfy the following conjunction of subconstraints: (1) a task in *Schedule* must be assigned a starting time that corresponds to a time slot in *TimeTable*, and (2) a time slot in *TimeTable* must not be assigned to more than m tasks. (Note that we model the m processors as a *capacity limitation* on each time slot.)

MENDER would recognize this constraint as an instance of RAP7 which is a conjunction of RAP2 and RAP4$^+$ where RAP4$^+$ is a generalization of the original RAP4. RAP4 requires that each resource is assigned to at most 1 consumer. RAP4$^+$ requires that each resource be assigned to at most m consumers for $m > 2$. Based on this classification of the constraint, MENDER would derive as an evaluation function a measure that *combines* the number of all tasks in *Schedule* assigned to time slots *within* TimeTable with the sum of how many times exceeding m each time slot has been assigned to some task. Given this evaluation function, MENDER determines that assigning *earlier starting times* is most likely to improve a schedule with respect to the evaluation function. Therefore, generate-and-patch with *greedy* patching would prefer scheduling tasks earlier over rescheduling tasks to later starting times.

Related research.

KIDS [Smith, 1991] and STRATA [Lowry, 1991] are two algorithm design systems that are closely related to MENDER. Both systems design search algorithms but differ from MENDER in the assumptions made about the initial problem specification, and in the way domain knowledge and algorithm knowledge are used to construct search operators and control facilities.

KIDS automatically constructs search algorithms, e.g. a *global search algorithm*, by retrieving from a library of abstract global search theories a theory that applies to the datatypes mentioned by the constraint. The abstract theory is then specialized into a global search algorithm through a series of program-transformations. Selection of global search theories and transformation steps is done in interaction with the user. KIDS enables the search algorithm to make use of problem-specific information by deriving *necessary filters* that prune those parts of the search space that are void of solutions. The derivation of necessary filters is accomplished by a *deductive inference* component.

KIDS is a much larger and more general algorithm design system than MENDER. KIDS works with a larger and more varied library of abstract search theories, enabling it to not only construct global search algorithms, but local search and divide-and-conquer problem solvers as well. MENDER is restricted to compiling local search algorithms, and does so only for constraints that fall into one of 16 abstract RAP categories. However, precisely because of its restrictions, MENDER has several advantages over KIDS. MENDER is fully automatic whereas KIDS requires

that the user make important design decisions. Because MENDER's constraint knowledge is restricted to RAPs for which generic evaluation functions are known, the cost of compiling search control facilities, i.e. the cost of retrieving and instantiating an evaluation function schema, is relatively cheap in MENDER. The derivation of necessary filters by KIDS's deductive inference component can be very costly. For constraints that are as global as "fill house", we expect KIDS to have great difficulty in deriving a filter. MENDER, however, can easily and cheaply provide an evaluation function to guide the search.

The STRATA system by Lowry has been integrated into KIDS as the component which derives *local search* problem solvers. STRATA and MENDER are similar in that they derive search operators ("patching operators" in MENDER; "neighbourhood structures" in STRATA) from datatypes mentioned in the constraint formulation and a *cost function* whose value local search strives to optimize. A major difference between both systems lies in the nature of the initial problem formulation. STRATA accepts optimization problems that list output conditions and a cost function as two separate and independent components of the problem formulation. In principle, any set of output conditions could be paired with any cost function. MENDER's style of problem formulation offers less flexibility, in that the the output conditions and cost function ("evaluation function") are interdependent. In MENDER, a constraint is presented only in the form of output conditions. A suitable cost function is then *derived* from the output conditions. As in comparison with KIDS, MENDER trades flexibility and variety of problem classes for a more direct and *low cost* algorithm design process. Since MENDER's cost functions are instances of generic cost functions, the type of cost function is known to the system. Knowing the nature of the cost function a priori allows us to equip MENDER with a regression mechanism that is specialized – and therefore cost-effective– in tracing desirable cost function changes back to local modifications in the solution structure. For the similar task, STRATA needs to use the much more general and costly deductive inference component of KIDS.

MENDER-compiled patchers adopt a repair strategy that is similar to the one recently examined by Minton [Minton *et al.*, 1990]. Minton demonstrates how a local search problem solver controlled by a simple "minimize conflicts" heuristic can solve large-scale scheduling and very large n-queens problems in approximately linear time with respect to problem size. Socic and Gu [Sosic and Gu, 1990] reported comparable performances for similar local search problem solvers for very large n-queens problems. However, to automate the construction of efficient problem solvers that can take advantage of the "minimize conflicts" heuristic, a constraint has to lend itself to an easy quantification in terms of number of "conflicts". The notion of "conflict" associated with a given constraint may or may not be obvious from the formulation of the constraint. MENDER solves both these problems for constraints of type RAP. MENDER reexpresses RAP constraints in terms that allow the conceptualization of a notion of "conflict" that captures the specifics of the constraint and is amenable to easy quantification.

Future research.

In the near future, we plan to extend MENDER to handle global constraints in a variety of other domains, e.g. scheduling, VLSI-design, n-queens, graph-coloring, satisfiability. We also intend to explore possibilities of applying MENDER's classification-based approach to automatically compiling "look-ahead" facilities which detect and circumnavigate unpatchable states early on.

Acknowledgements.

I am grateful to Chris Tong for his insights and guidance. For valuable comments and suggestions I also thank Lou Steinberg, Don Smith and Tom Ellman. Further thanks to Wes Braudaway for the constrained generators constructed with his RICK compiler.

References

Braudaway, W.K. 1991. *Knowledge Compilation for Incorporating Constraints.* Ph.D. Dissertation, Rutgers University.

Garey, M.R. and Johnston, D.S. 1979. *Computers and Intractability. A Guide to the Theory of NP-Completeness.* Freeman.

Lowry, M.R. 1991. Automating the Design of Local Search Algorithms. In Lowry, M.R. and McCartney, R.D., editors 1991, *Automating Software Design.* Menlo Park: AAAI Press.

Minton, S.; Johnston, M.D.; Philips, A.B.; and Laird, P. 1990. Solving Large-Scale Constraint Satisfaction and Scheduling Problems Using a Heuristic Repair Method. In *Proceedings of AAAI-90.*

Smith, D.R. 1991. KIDS – A Knowledge-Based Software Development System. In Lowry, M.R. and McCartney, R.D., editors 1991, *Automating Software Design.* Menlo Park: AAAI Press.

Sosic, R. and Gu, J. 1990. A Polynomial-Time Algorithm for the N-Queens Problem. *SIGART Bulletin* 1(3).

Tong, C. 1991. A Divide-and-Conquer Approach to Knowledge Compilation. In Lowry, M.R. and McCartney, R.D., editors 1991, *Automating Software Design.* Menlo Park: AAAI Press.

Voigt, K. and Tong, C. 1989. Automating the Construction of Patchers that Satisfy Global Constraints. In *Proceedings of IJCAI-89*, Detroit.

Voigt, K. 1991. Working Notes. Computer Science Department, Rutgers University.