Explicit Integration of Goals in Heuristic Algorithm Design

J. Mostow and K. Voigt*

Rutgers University Computer Science Department
New Brunswick, New Jersey 08903

Abstract

We describe a transformational derivation system that semi-automatically derives a simplified version of Mycin's therapy selection algorithm. It uses general transformation rules to explicitly integrate the multiple, sometimes conflicting goals that govern the design of heuristic algorithms. The generality of its transformations is demonstrated by using them to derive a variation based on formulating and integrating the same design goals differently.

I Introduction

Design of complex artifacts like programs, circuits, and buildings requires the integration of multiple, possibly conflicting design goals. Thus knowledge-based design systems ought to represent and reason about multiple design goals, since the better they "understand" the design process, the more intelligent the assistance they can provide. Current design systems tend to lack this capability. Either they fail to address multiple goals, or decisions that integrate particular goals and resolve specific conflicts are implicitly precompiled into their knowledge base by a manual knowledge engineering process [Mostow & Swartout 86]. To remedy this deficiency, we need better models for explicitly reasoning about multiple goals in design (and other knowledge-intensive problem-solving tasks) [Mostow 85].

One kind of intelligent design assistance is the ability to explain various features of the designed artifact. In the case of program design, [Swartout 83] points out that many questions about the designed artifact cannot be answered satisfactorily without referring to its development history, usually by asking its developer. For example, if a program feature depends on a particular manner of integrating certain design goals, explaining this feature involves referring to the design moves that carried out the integration, and the design rationale that motivated them [Neches et al 85]. Thus automating such explanations would require explicitly representing this design history information in machine-understandable form. Automated explanation is just one motivation for developing better models of goal integration in design; [Mostow 85] describes a host of others.

We have chosen to investigate this problem in the domain of heuristic algorithm design. To explore the kinds of knowledge and reasoning needed to integrate multiple design goals in this domain, we rationally reconstructed a simplified version of Mycin's therapy selection algorithm [Clancey 84], chosen because it tries to satisfy several conflicting design goals. We identified three kinds of goals:

- **Domain goals** (here: medical goals) prescribe the algorithm output.
- **Algorithm goals** govern the algorithm's performance.
- **Design process goals** dictate the resources available to the design process, such as the time within which the algorithm design must be completed.

In particular, we chose to address the following representative sample of the Mycin design goals listed in [Mostow & Swartout 86]:

- **Domain goals**:
  - Maximize drug effectiveness.
  - Minimize the number of drugs in the therapy.
  - Avoid contra-indicated therapies.
- **Algorithm goals**:
  - Maximize time efficiency.
  - Maximize space efficiency.

We encoded, in the form of domain-independent transformation rules, the general knowledge needed to integrate these goals and incorporate them into an algorithm. We call the resulting algorithm RMTSA, for Reduced Mycin Therapy Selection Algorithm. Given a set of drugs, the RMTSA is to generate the therapy (subset of drugs) that best satisfies the above medical goals, while not violating any of the algorithm goals. Notice the potential conflict among the medical goals in situations where they cannot all be satisfied perfectly. While design process goals influence the design, we have not modelled them explicitly in the system.

*This work was supported by NSF under Grant Number DMC-8610507, and by the Rutgers Center for Computer Aids to Industrial Productivity.

II The model of algorithm design

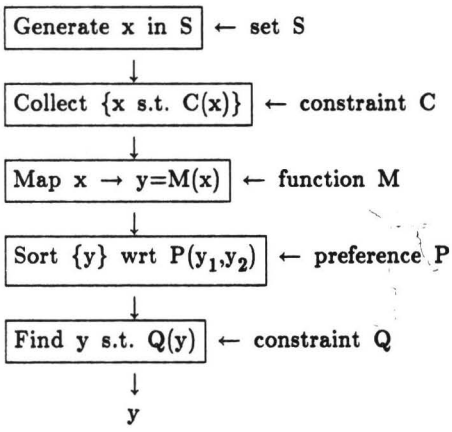
We model algorithm design as a transformational derivation process leading from an initial specification to an executable algorithm. Our algorithm design states consist of two parts, a dataflow graph and an agenda of design goals. The dataflow graph consists of algorithm components such as memories, mappings, filters, selectors, searches, and collects, whose input and output ports are connected through dataflow links. The agenda lists the design goals not yet incorporated in the algorithm being designed. Design operators

- transform a partially designed algorithm so as to incorporate, pursue, or avoid violating a design goal,
- integrate multiple, possibly conflicting, design goals into a combined goal, or
- reformulate a design goal to enable the application of another design operator.

We adopted the ideas of using dataflow graphs and allowing non-correctness-preserving transformation rules from [Kant & Newell 83, Kant 85, Steier & Kant 85].

We have not attacked the problem of automatically controlling the selection of transformation rules. We avoided all issues of control by alternating user selection of a supposedly suitable transformation rule from the full set of rules, with automatic application of this rule to the current algorithm design state.

Our system models the design of algorithms like the one shown below, consisting of a cascade of generators, mappings, filters, and sorters.



III A note on representation and implementation

Our system is implemented in LOOPS, an object-oriented programming environment developed at Xerox [Bobrow 85]. Algorithm components, design goals, and data input to the algorithm are represented as objects. Links between the components of a dataflow graph are represented as slots that contain pointers to preceding and succeeding algorithm components.

The semantics of each component type (memory, mapping, test, filter, search, and collect) is operationally defined by a rule for compiling it into Interlisp code [Teitelman 78]. A dataflow graph is compiled into its corresponding code by compiling each of its components and appropriately nesting the resulting pieces of code. The dataflow graph representation of the algorithm is executable if all its components are executable. In order for a component (or rather its compilation) to be executable, the component must not contain any non-operational statements. Otherwise, more transformations are needed to operationalize the component. The $Search component is an example of a compound component, which can include other algorithm components within its own internal structure (see Figure III-1). The $Search schema has slots .Set for the choice set, .Map for the mapping function (identity as default), and .When for one or more filters (TRUE as default).

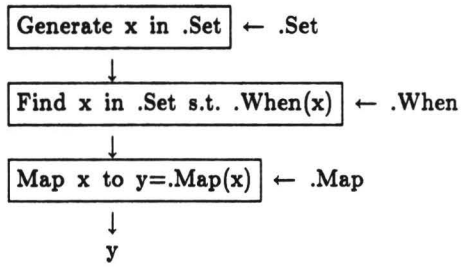


Figure III-1: Internal structure of $Search

IV The initial algorithm design state

We take the specification for the design of the RMTSA as the initial state of our derivation. Figure IV-1 shows the algorithm specification formulated in our representation.

Initial dataflow graph:

```
[$MycinTherapySelect <the algorithm>
   .Input        = <some set of drugs>
   ... ]

   .DomainGoals = ($more-drug-effective
                   $fewer-drugs
                   $fewer-contra-indications)
   .AlgrthmGoals = ($less-time-cost
                   $less-space-cost)
```

Figure IV-1: The initial state of design for the RMTSA

The initial specification of the RMTSA consists of an initial dataflow graph and an initial agenda of design goals. The dataflow graph has two components. The mapping component $SubSets1 enumerates all subsets of the input set. It is followed by the selecting component $NDSelect, which non-deterministically selects from its choice set of therapies (subsets of drugs) a therapy that best satisfies the domain goals (.DomainGoals) without the algorithm violating any algorithm goals (.AlgrthmGoals). The agenda of design goals lists three domain goals and two algorithm goals.

V Derivation of RMTSA1: first example

We now derive the RMTSA, introducing transformation rules as needed, and showing those portions of the resulting algorithm design state where changes have occurred.

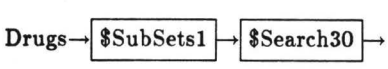
The initial algorithm design state appeared in Figure IV-1. It lists $fewer-contra-indications as one of its domain goals. The transformation rule **TreatGoalAsConstraint** treats this goal as a constraint. The cut-off value for thresholding -- here 0 -- is provided by the system developer to indicate that no contra-indications will be tolerated. The result of thresholding is a new predicate, $Pred31, that returns TRUE if the number of contra-indicated drugs in a given therapy is less than or equal to this cut-off value.

> **TreatGoalAsConstraint**: Treat goal G as a constraint. Do so by thresholding using a user-provided cut-off value.

Application of **RefineSelectToSearch** replaces the non-deterministically selecting component $NDSelect1 by a newly created search component $Search30.

> **RefineSelectToSearch**: Replace non-deterministic selection of an element according to constraints C_1, ..., C_n by a search for this element.

At this stage, $Search30 appears still in default format, and is operational. In fact, the entire algorithm is operational. However, since no goals have been incorporated yet, it would return the first generated subset of drugs, not the therapy that best satisfies the domain goals.



.DomainGoals = ($more-drug-effective
$fewer-drugs)
...
.Constraints = ($Pred31)

Next, we apply the transformation **IncorporateConstraints** to the contra-indications constraint (now predicate $Pred31) in order to incorporate it into $Search30.

> **IncorporateConstraints**: To incorporate constraints C_1, ..., C_n on a selected element, use them to filter the choice set.

Then transformation rule **ConstraintToFilter** creates a filter component($PFilt31) for $Pred31. The last step is needed since in our representation only components, not predicates, are compilable into executable code.

> **ConstraintToFilter**: Reformulate constraint C as a filter.

We obtain the algorithm design state shown.

with:$Search30.When = ($PFilt31)

The goals $more-drug-effectiveness and $fewer-drugs are converted into preferences by applying TreatGoalAsPreference twice, once to each goal.

> **TreatGoalAsPreference**: Treat goal G as a preference.

Only the agenda of design goals has changed.
.DomainGoals = NIL
...
.Preferences = ($more-drug-effective $fewer-drugs)

We can now use **SortByPreferences** to incorporate the preferences. The resulting dataflow graph will generate possible therapies, sort them according to the preferences, and search for the first one that satisfies the constraints.

> **SortByPreferences**: To incorporate one or more preferences P_1, ..., P_n on a selected element, use them to sort the choice set. If $n > 1$, mark the preferences for integration. Additional effect: reduces average runtime.

with: $Sort30.SortCrit =
(*COMBINE* $more-drug-effective $fewer-drugs)
.Preferences = NIL

In order to reduce the space cost of our algorithm, we now use **CondensePreference** to condense $more-drug-effectiveness into 3 categories; we pretend we have been given this number by a domain expert.

> **CondensePreference**: To help minimize space, condense a preference P into N categories.

We now notice a type clash: the input to $Sort30 is a set of drugs, whereas $more-drug-effective in the sort criterion compares individual drugs. Consequently, we extend the preference $more-drug-effective into one that allows the comparison of sets of drugs. Here the type clash is resolved by defining a new preference,

$more-therapy-effective, which distinguishes between *bags* of drugs [Mostow & Swartout 86]. Depending on the kind of type clash and its context, other measures might be needed.

The changes brought about by condensing and bag-extending $more-drug-effective are not reflected in the dataflow graph or the goal agenda. Apart from the creation of a new preference, the changes have taken place in the internal specification of $more-drug-effective. The internal specifications of the domain goals are stated in rather unreadable LISP code, so we spare the reader the details.

The transformation **ConjoinPreferences** integrates the preferences $more-therapy-effective and $fewer-drugs.

> **ConjoinPreferences:** To combine two preferences P and Q, form their logical conjunction P∧Q.

This step creates a preference $better-therapy, which becomes the new sort criterion in $Sort30.SortCrit.

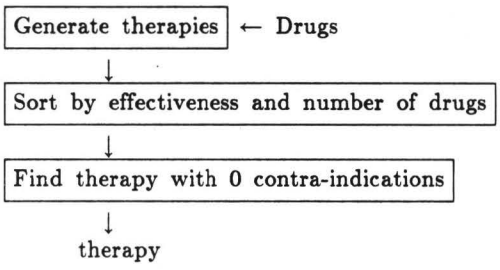
with:$Sort30.SortCrit = $better-therapy

A final transformation pursues the algorithm goal $less-time-cost.

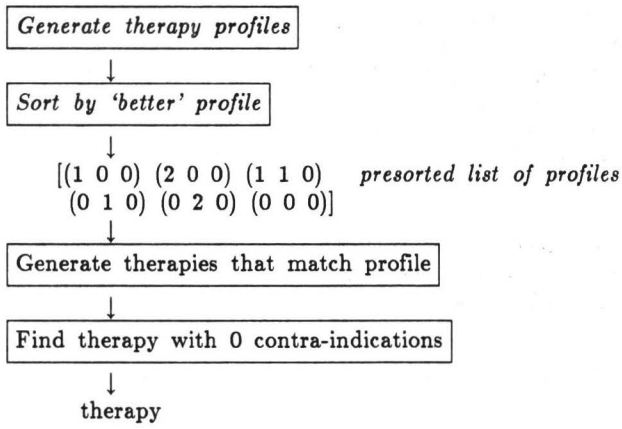
> **PreComputeData:** To help reduce run-time, precompute data known at design time.

Here it precomputes the data provided by the dataflow subgraph consisting of the components $SubSets1 and $Sort30, which depend only on information known at design time. Precomputation eliminates the costly sorting from the algorithm, by abstracting the elements of the choice set (therapies) to therapy profiles, sorting the profiles according to the sort criterion, and storing the sorted list of profiles in a table. At runtime, the algorithm will use this precomputed table to generate the equivalence classes of therapies matching each successive profile, thereby generating therapies sorted according to the original sort criterion. The next two diagrams summarize this transformation step.

Before **PreComputeData**:



After **PreComputeData**:



At run time, the algorithm uses the presorted list of profiles, or "instruction table" (as it is called in Mycin), to generate therapies in sorted order according to the sort criterion $better-therapy. An instruction, for example "(2 0 0)", means "Compose a therapy by selecting two drugs from drug effectiveness category 1, and no drugs from categories 2 and 3". Therapies matching the sorted profiles are tested for contra-indications, and the first therapy with none is returned.

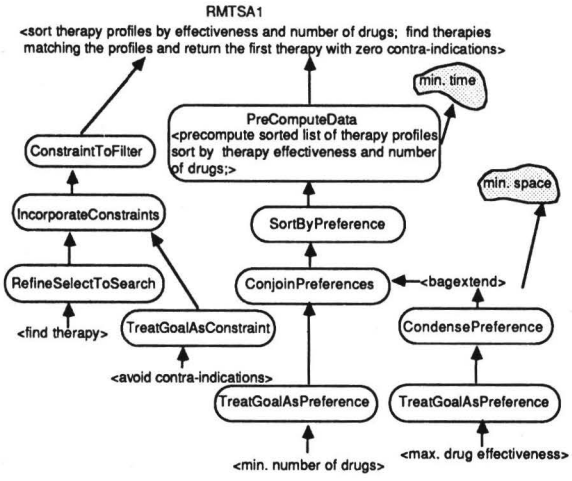
Notice that the instruction table is actually just a small portion of the entire table that would have been computed if we had not instructed our profile generator to produce only a subset of all possible profiles. In our implementation, we contented ourselves with generating 15 profiles, which we reduced further by selecting only profiles that we considered "realistic". For example, if acceptable therapies never contain more than 5 drugs, then the profile "(3 1 2)" prescribing 6 drugs is not realistic. To keep the set of profiles finite, we chose not to generate any profiles that use more than 5 drugs.

Presorting the profiles requires some design-time interaction. Since $more-therapy-effective defines a partial order, $better-therapy is partial too. For instance, it does not determine whether the profile "(2 0 0)" (two first choice drugs) is better than "(0 1 0)" (one second choice drug). In situations like this, the rule prompts the system developer to make the decision based on domain knowledge. Ideally, the expert should also be required to explicitly formulate the reasons for the decision, or say if it is arbitrary. These reasons could then be recorded and used to eventually linearize partial orders.

The design of RMTSA terminates after the application of **PreComputeData**. The criteria for successful termination of the design process are

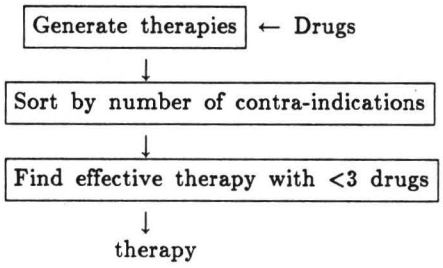
fullfilled. All domain goals have been incorporated into the algorithm and have been reformulated and integrated so that the algorithm is fully operational. Further, the algorithm satisfies the posted algorithm goals. A design would terminate in failure if one or more of the domain goals or algorithm performance goals could not be satisfied by applicable transformation rules.

To summarize, we show the rule tree underlying the derivation of RMTSA1. A longer version of this paper [Mostow & Voigt 87] presents the LISP code corresponding to the final algorithm, and a demonstration of its executability.



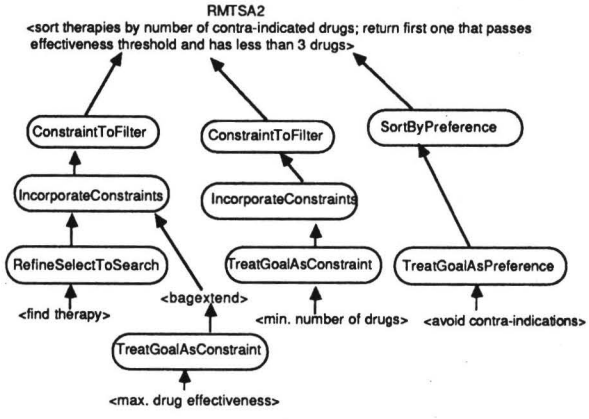
VI RMTSA2: a variation

To test the generality of the transformation rules, we applied them to a variation of the RMTSA problem. The variation starts out with the same initial algorithm design state as for RMTSA1, except for the algorithm goals $less-time-cost and $less-space-cost. The design diverges from the previous derivation when we formulate the goal $fewer-contra-indications as a preference and all the other goals as constraints. The resulting algorithm, shown below, returns the therapy with fewest contra-indications, subject to constraints on its effectiveness and number of drugs. It generates possible therapies, sorts them by the number of contra-indicated drugs they contain, and outputs the first one that has at most two drugs and passes a minimum effectiveness threshold.



[Mostow & Voigt 87] details the derivation, the final state, and the result of compiling and executing it. The variation produces substantially different behavior from the original version: different questions are asked at design and runtime, and different therapies are output. Nonetheless, the nature of the design task remains very similar to the first example. The same rules apply; only the thresholding transformation had to be slightly modified. Some additional coding was necessary to enhance the design goal representations. The need to add information about the goals is not surprising, and can be viewed as feedback from implementation to specification [Swartout 83].

A summary of the derivation is shown below.



VII Conclusion

We have demonstrated how transformation rules can convert an initially non-operational algorithm specification into an operational algorithm so as to satisfy multiple, interdependent, and sometimes conflicting design goals. We hope to have given an impression of the various reformulation steps needed to incorporate domain goals into an algorithm, and of how performance goals shape the algorithm.

In retrospect, we identify four types of knowledge that enter the design process: *domain knowledge*, *algorithm knowledge*, *goal knowledge*, and *control knowledge*. In our example, domain knowledge describes the medical domain in which therapy selection takes place. Algorithm knowledge describes various algorithm components, prescribes how to combine these components into an algorithm, and tells how to judge the performance of an algorithm. Goal knowledge describes goals and the operators for reformulating and integrating them. We feel that this "goal space" requires further investigation.

In our implementation, we have represented in the computer enough medical, algorithm, and goal knowledge to design the RMTSA. Interaction with a human expert compensates for the system's lack of the experiential knowledge needed to set thresholds, determine the number of categories for condensing, etc. Manual selection of transformation rules simulates the considerable control knowledge needed to guide the algorithm design process to an acceptable solution.

An interesting step towards explicit representation of such control knowledge would be to automate the reasoning that influences the formulation of a goal as a preference or constraint. Comparing the two derivations, we can see the strong effect of such choices on the resulting algorithm. One heuristic is to formulate conflicting goals as preferences. This heuristic assumes that goal conflicts can be detected early in the design process.

A design aid that explicitly deals with design goals and their conflicts should be superior to the current practice of engineering implicit solutions to such conflicts by hand, leading to behavior that cannot be adequately explained. Although we have not built an explanation component, our example derivations contain the information needed to answer such questions as why the two algorithms produce different therapies as solutions, and whether the generation of one therapy before another is based on genuine domain knowledge or is just an artifact of the implementation.

Acknowledgements

We would like to thank Bill Swartout for suggesting the original problem and contributing to the initial research, Bill Clancey and Ted Shortliffe for unearthing answers to questions about Mycin, and Chris Tong and Lou Steinberg for helpful comments on an earlier presentation of this work.

References

[Bobrow 85] D. Bobrow.
What it takes to support AI programming paradigms, or: If Prolog is the answer, what is the question?
IEEE Transactions on Software Engineering SE-11(11):1401-1408, November, 1985.

[Clancey 84] Clancey, W.J.
Details of the Revised Therapy Algorithm.
Rule-Based Expert Systems.
Addison Wesley, 1984, pages 133-146.

[Kant 85] E. Kant.
Understanding and automating algorithm design.
IEEE Transactions on Software Engineering SE-11(11):1361-1374, November, 1985.

[Kant & Newell 83]
Kant, E., Newell, A.
An Automatic Algorithm Designer: An Initial Implementation.
In *AAAI83.* 1983.

[Mostow 85] J. Mostow.
Toward better models of the design process.
AI Magazine 6(1):44-57, Spring, 1985.

[Mostow & Swartout 86]
J. Mostow and W. Swartout.
Towards Explicit Integration of Knowledge in Expert Systems: An Analysis of MYCIN's Therapy Selection Algorithm.
In *Proceedings AAAI86*, pages 928-935. Philadelphia, PA, June, 1986.
Available from Rutgers University Laboratory for Computer Science as LCSR-TR-81 or AI/Design Working Paper #35.

[Mostow & Voigt 87]
J. Mostow and K. Voigt.
Explicit Incorporation and Integration of Multiple Design Goals in a Transformational Derivation of the MYCIN Therapy Selection Algorithm.
Technical Report AI/Design Working Paper 43, Rutgers University Computer Science Department, January, 1987.

[Neches et al 85]
R. Neches, W. Swartout, and J. Moore.
Enhanced maintenance and explanation of expert systems through explicit models of their development.
IEEE Transactions on Software Engineering SE-11(11):1337-1351, November, 1985.

[Steier & Kant 85]
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.

[Swartout 83] Swartout, W.
XPLAIN: A system for creating and explaining expert consulting systems.
Artificial Intelligence 21(3):285-325, September, 1983.
Also available from USC Information Sciences Institute as ISI/RS-83-4.

[Teitelman 78] Teitelman, W.
Interlisp Reference Manual
Xerox Palo Alto Research Center, 1978.