AN AUTOMATIC ALGORITHM DESIGNER: AN INITIAL IMPLEMENTATION[1]

Elaine Kant and Allen Newell

Department of Computer Science

Carnegie-Mellon University

Pittsburgh, Pennsylvania 15213

ABSTRACT

This paper outlines a specification for an algorithm-design system (based on previous work involving protocol analysis) and describes an implementation of the specification that is a combination frame and production system. In the implementation, design occurs in two problem spaces -- one about algorithms and one about the task-domain. The partially worked out algorithms are represented as configurations of data-flow components. A small number of general-purpose operators construct and modify the representations. These operators are adapted to different situations by instantiation and means-ends analysis rules. The data-flow space also includes symbolic and test-case execution rules that drive the component-refinement process by exposing both problems and opportunities. A domain space about geometric images supports test-case execution, domain-specific problem solving, recognition and discovery.

I. APPROACHES TO DESIGN

We are interested in systems that automatically design algorithms and create programs for them. The process of algorithm design requires a significant degree of both knowledge and intelligence, and may even require additional structure to make creative discoveries. In this paper, we briefly present our specifications for a design system and describe an initial implementation. The main goal of the research is to produce a successful automatic design system, but a secondary goal is to show that careful examination of human behavior can suggest novel and worthwhile system organizations.

Related research on program synthesis and transformation, formal derivation, and automated discovery suggests some possible approaches to the automation problem. However, none provides a satisfactory paradigm for the whole task of automatic algorithm design.

Program synthesis systems [1, 5, 13] often successively refine programs by transformations. Most of these systems lack robustness because they require numerous transformation rules to specify all the details about programming and because they have no problem-solving abilities other than simple pattern matching on the rules. This may indicate the desirability of having a few general operators with more sophisticated techniques for adapting rules to situations.

Formal derivation systems typically do have a small number of general operators. However, they require complete, formal specifications, which may not be available initially [14], and they often apply rather rigid methods to choose among applicable transformation rules. People still must specify the interesting lemmas or auxiliary definitions and decide among alternative axiom sets. One system [2] does address the choice problem by guessing recursive solutions to logical equations and verifying the guesses with a theorem prover and a small model.

Discovery systems are quite rare, and those that exist are either open-ended exploration systems rather than problem-solving systems that focus on a specific task [3, 10] or work in very narrow domains such as algebraic models [9].

We propose an approach that includes relatively few basic operators, specific knowledge about instantiation rather than many transformation rules, and several problem-solving methods that work with domain knowledge. We also attempt a structure that permits discoveries.

II. A FUNCTIONAL SPECIFICATION FOR AN ALGORITHM-DESIGN SYSTEM

The specifications for an algorithm-design system described here are based closely on the results of protocol analysis [6, 7].[2] We take algorithms about computational geometry (for example, producing the convex hull of a set of points in the plane) as our first task domain because it forces us to consider the interplay between external geometric models, mental imagery, and computationally efficient serial algorithms.

A. Algorithm-design methods to be supported

Design begins with the hypothesis of a *kernel schema* or solution plan and proceeds by *successive refinement*. Two closely related methods of program execution guide the gradual refinement of the initial schema. *Symbolic execution* uncovers information by running a partial algorithm description with symbols as data. During execution, the algorithm and symbols are elaborated (new assertions and new components are added). If symbolic execution is not sufficient to permit progress, a specific example is constructed that permits detailed execution (*test-case execution*).

In addition to driving the successive refinement, the two execution methods generate consequences and expose problems

[1]This research is supported by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory Under Contract F33615-81-K-1539. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

[2]We analyzed protocols of human problem-solving and design strategies in algorithm-discovery tasks because a system organization that parallels human reasoning lets us draw on human experts for techniques, promises to be flexible and robust, and has potential for learning.

and complications.[3] The test-case execution method allows the problem solver to generate examples that contain the right level of detail to make progress (the relevant knowledge must be evoked by concrete retrieval cues) while avoiding complications (such as boundary conditions) that are not important until the basic outline of the algorithm has been developed. However, test-case execution is more time consuming than symbolic execution because loops are executed for individual items, whereas symbolic execution is linear in the size of the algorithm structure. Thus, if the system understands a step well enough, it will execute it symbolically. Simple inferences may also be made during the execution process.

Design includes problem solving and reasoning about the task domain. Examples are generating test cases, attempting to find counterexamples, and "proving" conjectures by demonstration (first on "typical" and later on boundary-condition examples). An extreme, but important, example of reasoning is *discovery,* the sudden viewing of a situation from a new perspective that presents an opportunity to solve an outstanding problem or make an improvement. What is perceived is not a solution to the main goal at the time the insight occurs; however, there is usually some preparation for the discovery -- some previously considered but unsolved problem. The new perspective often arises out of a novel and accidental alignment of objects or facts.[4] A general recognition-based control structure for all the methods seems to be the appropriate device to support the possibility of such juxtapositions.

B. A problem space for representing algorithms

Much of the design process occurs within a single problem space of schematic structures that represent algorithms. A small number of basic types of *components* are instantiated and configured in multiple ways to represent algorithms. Expert design concepts (for example, schemas for divide and conquer and for dynamic programming) are built up from and coexist with the simpler vocabulary.

Components may be augmented with *assertions* that give additional information such as an ordering for a generator or a predicate on a test. An assertion is not limited to concepts in the algorithm space. For example, it can describe a relationship in the task space that must be satisfied. The assertions drive the decisions about how to refine a component into a new configuration of components.

The problem space must include *operators* that refine partially specified structures (add new components, links between components, or assertions) and that carry out the execution methods. The refinement operators must not insist that configurations form complete or consistent algorithms. While most components will have a set of standard inputs and outputs to represent primary data relationships, operators can add new connections at any time without checking for type mismatches. Instead, problems noticed during test-case or symbolic execution will be handled as they arise. All the problem-space operators should be very general and will require instantiation before they are applied.

Since kernel ideas, instantiation, and indeed all design decisions can be wrong, design involves search in the problem space. Most *search-control knowledge* should be provided by rules that select instantiations of the operators that modify components and assertions and select parts of the algorithm to refine or execute. In the absence of specific knowledge for instantiation, other search-control rules must provide more general techniques, such as means-ends analysis, to help adapt an operator to a specific problem state.

C. A problem space for the application-domain

The application-domain space must support test-case execution of algorithms, problem solving about the domain, and discovery. Test-case execution requires operations that produce specific examples (and judge their suitability) and operations that evaluate assertions (such as predicates on tests) in the domain space.

Our initial task domain will be a problem space for manipulating geometric images. It includes objects, such as points, line segments, and polygons; specific geometric operators, such as create a point or construct a line segment; and general operators (perceive, find, partition) that are typically specialized in geometric ways (partition a point set). The operators help create test cases (such as sets of points) and check assertions (such as whether a point lies above the X-axis).

Working within a geometric task domain imposes important (but largely unknown) constraints, as indicated by the facility with which humans process visual scenes and reason spatially. Our design for a geometric problem space is modeled upon the imagery simulation of Kosslyn and Shwartz [8]. It consists of a short-term *image* buffer (a planar array), with the image centered around a focal point of attention. External (perceived) and internal (imagined) images are coalesced into a single image. Several operators (rotate, pan, zoom and scan) change the focus by regenerating the buffer around the focus point, rather than shifting the center in a fixed space. A variable amount of detail is present, depending on point of view created by the movement operators. The image is viewed by a recognizer centered on the focus point, with the image memory permitting recognition of combined scenes. However, there are no operators that construct the image directly (write new patches of image into the buffer). Rather, all parts of the image are supported by the underlying semantic representation, and modifications are made to this underlying structure which is then pictured in the image.

Discovery seems to occur through a recognition of some interesting configuration in the task space. Certain configurations of data items lead to recognition or automatic inferences (for example, completing polygons that are missing one edge, seeing polygons in regular patterns of points, being reminded of related algorithms). The image structure just described seems to be appropriate to facilitate this recognition.

---

[3] Our analysis of human protocols showed that half of the design time was spent on symbolic and test-case execution, and design time was proportional to the number of components in the algorithm description. Even when subjects had difficulties in designing the algorithms, their unsuccessful behavior had a similar character of hypothesizing new structures and repeatedly attempting to execute the partial algorithm on sample data. Symbolic and test-case execution also had major roles in verifying, describing and analyzing algorithms.

[4] Our human subjects sometimes made discoveries while feeling lost and staring "blankly" at a figure during test-case execution.

III. AN IMPLEMENTATION OF AN ALGORITHM-DESIGN SYSTEM

We are implementing an algorithm-design system that is both a protocol analysis/simulation system and a fully automated design system. The basic structure is a combination frame and production system. It is implemented in Lisp and can run with or without the OPS5 production system [4] as the automated control structure.

In the simulation mode, the user (rather than the production rules) makes the decisions about which design commands to invoke and how to instantiate them. Design commands include component, assertion, and item construction, symbolic and test-case execution, and goal creation and modification. The implementation also includes objects such as protocol phrases, episode groupings, and comments (all represented by frames), and it has operators that create and edit the objects and insert them into a history of design commands. The total history describes an actual protocol and its interpretation. The user interface includes facilities for graphic display of component configurations, for saving, restoring, and undoing history sequences, and for command completion (with prompts for arguments and access to help files).

The automatic mode is an augmentation of the simulation mode. Normally, the production rules make instantiation and search-control decisions and a history of design steps only (no protocols) is maintained. However, the modes can be mixed and the user can take over or relinquish control in midstream.

A. Representing algorithms with data-flow configurations

Partially specified algorithms are represented as states in a *data-flow problem space*, DFS. Each state is a *data-flow configuration*.[5] The algorithm steps are represented by *process components*. There are a small number of generic process components (a *memory*, and the flow-control components *generate*, *test*, and *select*) and a general *apply* component that can be specialized to domain operations such as make-line-segment. The inputs and outputs of the process components are represented by *ports* connected by *links*. Process components can be further specified by *assertions*. The components and assertions together modify and control the flow of *items* that represent data objects such as points and line segments. Items can fill several roles -- generic descriptions, symbols (for symbolic execution), and specific domain-space data. Assertions can be attached to any object, including an item, a process component, a whole configuration of components, and a problem space.

Figure 1 shows an example of a simple DFS configuration that, given a set of points, finds the subset that lies above (on the left side of) the X-axis. The input, **{i}**, is a memory containing a set of points, {A B C D}, which are enumerated by **Generate**$_1$. Those points that satisfy the predicate associated with **Test**$_1$ are added to the output set memory **{o}**. In the figure, points A, B and C have been enumerated. A was discarded, B has been added to the output, and C has not yet been tested.

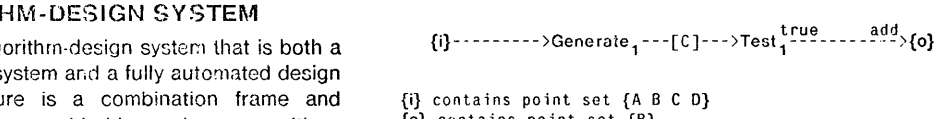

Figure 1: A subset test.

Components, links, ports, assertions, and items are all implemented by frame data structures chained in a tangled hierarchy (a directed acyclic graph). A refinement of a component is represented by a slot containing the names of the components in the configuration that implements it. For example, the TOP-LEVEL component frame has a slot called Components-list whose value is (MEMORY-1 GENERATOR-1 TEST-1 MEMORY-2). Two frames that define the Test$_1$ component of Figure 1 are shown in Figure 2. Backpointers between related frames (such as the Component-of slot in TEST-1) are automatically maintained. In addition to the value and default facets shown in the figure, if-needed servant functions and if-added/if-removed demons are provided via facets. Furthermore, all objects have a life time (creation and deletion dates) so that different DFS states can be represented concisely, and the history of events is tree-structured to enable alternate design paths to be tried. Different inheritance mechanisms allow objects to be viewed from different historical perspectives -- such as the current configuration of "visible" objects, or a "remembered" version of past states.

The current knowledge base of the system (before an algorithm is developed) includes the basic components described above, three types of ports (input, output, and signal-interrupt) and several named port instances for each component type. This algorithm-component knowledge base is fairly stable, although it is missing specialized expert schemata, such as divide and conquer. We do not expect it to grow to more than several times the current size.

Component: TEST

slot	facet	value
Creation-date	Value	HISTORY-0
AKO	Value	COMPONENT
Input-ports	Default-filler	TEST-IN
Output-ports	Default-filler	TRUE-EXIT
Instances	Value	TEST-1

Component: TEST-1

slot	facet	value
Creation-date	Value	HISTORY-3
AKO	Value	TEST
Component-of	Value	TOP-LEVEL
Input-ports	Value	TEST-IN-1
Output-ports	Value	TRUE-EXIT-1
Assertions	Value	ASSERTION-2

Figure 2: Frames for the test components.

B. Implementing component operators

DFS operators that construct and refine the structures in an algorithm description accomplish simple but general tasks: adding, deleting, or changing the slot values of components and links. Other operators move items across links and execute components. A reasonably complete set of approximately fifty Lisp functions implement these operators. They are quite straightforward, and not many more appear to be needed.

[5] Data-flow representations have been studied in computer architecture research and have been used to describe artificial intelligence programs [11, 12] and to express algorithm transformations [15].

The operators to construct the subset-test configuration of Figure 1 are given in Figure 3. The *new-component* operator adds a new instance of the specified component to the current configuration, but does not link it up in any way. The *add-component* operator adds the new component specified in its first argument and links it to an existing output port, described by the second argument, using the input port on the new component described by the third argument. If an argument is missing (or is nil), defaults are used. In this case, the new components are linked to the most recently added component, using Default-filler ports for each component type. The process of instantiating these operators is discussed in section D.

```
(new-component 'memory)
(add-component 'generator)
(add-component 'test)
(add-component 'memory nil 'add-elem)
```

Figure 3: Constructing the subset test.

C. Representing and testing assertions

Assertions, which are also represented as frame objects (see Figure 4), can be attached to any type of object. Assertions can take values from the component inputs as parameters (for example, the TEST-IN port of the test above is a parameter in the assertion show below). There are currently about twenty-five assertions in several categories (for example: Booleans connectives, test-predicates, assertions about generator orderings, and references to geometry space operations), but assertions can denote anything (for example, that a configuration has not yet been refined to handle initialization). This set will grow as new algorithms are developed.

DFS operators have been written to add, remove, search for, test the truth of, and find the consequences of assertions. For example, to make the subset algorithm in Figure 1 be a test for the points above the x-axis, we apply the following operator which results in a new assertion frame.

(*add-assertion* '(test-predicate (on-side test-in x-axis left)) 'test-1)

```
       Component:  ASSERTION-2
slot              facet   value
-------------------------------------------------------
Creation-date     Value   HISTORY-4
Body              Value   (TEST-PREDICATE
                            (ON-SIDE TEST-IN X-AXIS LEFT))
AKO               Value   TEST-PREDICATE
About             Value   TEST-1
-------------------------------------------------------
```

Figure 4: An assertion frame.

Consider the evaluation of this assertion for point C during the test-case execution of Figure 1. The system looks for the Truth slot of the assertion. Nothing is recorded there, but the assertion is an instance of a TEST-PREDICATE, which is in turn an instance of a BOOLEAN-ASSERTION, and there is an if-needed function that computes the truth value. During this computation, the value of the ON-SIDE subassertion is needed. If it has not been previously stored, it is calculated by calling a geometry space function that uses the coordinates of the point C. The result is then stored for future reference.

D. Controlling search with a production system

Search is controlled primarily by production rules, with major goals explicitly represented in frame structures and with default-fillers for slots to represent typical port or assertion types (e.g., to indicate that tests have test-predicate assertions). The goal structure is simple -- goals have types (e.g., to refine a component or to execute a component) and can be in a number of states -- active, suspended (waiting for subgoals), sidelined (for lack of immediate relevance or a way to proceed), succeeded, failed, or cancelled. A hierarchy of subgoal relationships is maintained, but all goals are in working memory so the productions can notice whenever any goal is achieved, even if it is not active in the current state.

Operators are instantiated by production rules. For frequently used generic components, specific instantiation rules describe which process components to add to the algorithm description under different circumstances, defaults for how to link components together, and so on. Default-fillers are used by instantiation rules in absence of more specific rules. The current implementation has about 25 instantiation rules, and we anticipate that several orders of magnitude more will be needed.

In the absence of specific knowledge, means-ends analysis rules find differences and select operators to reduce the differences. Production rules also describe how to execute components. The current implementation has approximately 20 rules that control means-ends analysis, voting on alternatives, and subgoaling, and another 10 that provide the control for symbolic and test-case execution and assertion checking. Specific Lisp functions associated with objects determine how to execute individual components and test assertions. As new specializations of components and assertions are added, corresponding functions and instantiation rules may need to be added, but the control should not change.

The design of the simple algorithm in Figure 1 is sketched in Figure 5. The first two production-rule applications decide to add a new component because the difference between the existing configuration and the desired one is that nothing is present. The operator is instantiated to add a memory, because the givens of the problem include a set of points. A new difference is selected (application 3), which is that the algorithm contains no active process component. The *add-component* operator can solve this (4), and it is instantiated to a generator after a voting process (applications 9-17). Both generators and selectors can follow memories, but a generator is more likely because it creates a set rather than a single item as output, and the desired output of the algorithm is a subset of the input. The components are linked according to the default rules. The system tries to refine the new component, but other than deciding to use the default ordering assertion, no refinement is necessary at this level (applications 41-59). Since an active process component is present, symbolic execution is applied (applications 62-89). This proceeds successfully until there is nothing to do with the output of the generator. A test is added (90-102), and attempts are made to refine it (103-146). The production rules find a test predicate by looking for assertions about the types of items that are inputs to the test. Symbolic execution then continues (147-172) until the output of some component (the test) matches the description of elements in the algorithm output (both are points, both have matching assertions about position relative to the X-axis), and the results are collected in a final memory.

The full derivation of the subset test includes 190 production-rule applications. The summary trace here omits most of the applications related to housekeeping tasks, voting procedures, false paths, and means-ends analysis steps. Operator applications are shown in square brackets, and comments are in braces.

```
{the top level goal is to refine algorithm Subset}
1. notice-difference::empty-configuration
2. reduce-difference::empty-configuration
       [new-component memory]  {a memory component is added}
3. notice-difference::no-active-process-component
4. reduce-difference::no-active-process-component
   {it has been decided to apply the operator add-component}
5. instantiate::add-component:from-port
   {the from-port is instantiated according to default rules}
9. instantiate::add-component:generator-since-need-set
10. instantiate::add-component:generator-follows-memory
11. instantiate::add-component:select-follows-memory
13. vote  {decide on component to instantiate add-component}
14. instantiate::add-component:to-port
18. instantiation-complete {a generator component is added}
       [add-component generator mem-out-1 gen-in]
41. reduce-difference::no-assertions
    {the current goal is to refine the generator}
42. instantiate::add-assertion:defaults
    {assertion about default generator ordering is added}
59. goal::sideline-goal-with-no-reduce-difference
    {cannot further refine generator, so sideline goal}
60. goal::resume-if-suspended-with-sidelined-subgoals
    {resume goal of refining whole algorithm}
62. reduce-difference::symbolic-execution
74. symbolic-execution::instantiate-inputs-succeeds
78. symbolic-execution::set-up-for-refinements
    {find most refined components}
83. symbolic-execution::executable
    {symbolically execute memory component}
88. symbolic-execution::executable
    {symbolically execute generator component}
99. symbolic-execution::unconnected-link
    {notice generator output not connected to a component}
    {decide to apply add-component operator}
95. instantiate::add-component:compare-follows-component
96. instantiate::remove-if-not-enough-inputs
97. instantiate::add-component:test-follows-active-component
98. instantiate::add-component:apply-follows-active-component
99. instantiate::remove-if-not-enough-inputs
    {test is only component with the proper inputs}
102. instantiate::add-component:to-port
       [add-component test gen-out-1 test-in]  {test added}
127. reduce-difference::no-assertions
     {new goal is to refine test}
128. instantiate::add-assertion:test-relevants
     {a relevant test predicate is found}
       [add-assertion (on-side test-in x-axis left) test]
146. goal::sideline-goal-with-no-reduce-difference
     {no way to refine test further}
147. goal::resume-if-suspended-with-sidelined-subgoals
     {resume goal of refining whole algorithm}
166. symbolic-execution::set-up-for-refinements
     {haven't executed test yet}
170. symbolic-execution::executable  {so execute test}
172. symbolic-execution::unconnected-link-alg-complete
     {notice that output on link from test satisfies assertion
      on algorithm output, so add final memory component}
       [add-component memory true-exit add-elem]
{algorithm is now complete}
```

Figure 5: Sketch of system's design of algorithm in Figure 1.

E. Implementing the domain space

We have implemented a simple version of the geometry space, with frame objects representing geometric objects (including points, line segments and polygons). Lisp functions implement operators (to construct and modify the objects) and determine the relationships between objects (such as above, between, inside, and convex) that are used in assertions. General operators such as *perceive* or *find an item with a given property* are not yet implemented. Currently the system uses analytic geometry computations, and the imagery model is not yet implemented.

IV. DISCUSSION

Our initial implementation is currently operational. In addition to constructing small examples such as the subset test described here, we are using the convex hull algorithm (whose discovery has been worked out by hand in our protocol-analysis research [6, 7]) as a large test case. This example contains instances of a variety of critical issues for an algorithm-discovery system. There are both generate-and-test and divide-and-conquer algorithms. The former was found initially and was used in part in obtaining the latter, so interesting issues of knowledge transfer arise. Some genuine discoveries occur, providing good tests of whether our system can recognize and capitalize on adventitious situations. The coupling between the algorithm space and the geometry space is continuous and intimate, and this will test whether our proposed imagery scheme will have the desired properties. On the other hand, we have still to address the task of passing from the DFS representation of an algorithm to a program in some standard language.

ACKNOWLEDGEMENTS

We thank David Steier for helpful comments on this paper. Steier, Brigham Bell, and Edward Pervin are helping implement the system.

REFERENCES

[1] Balzer, R. "Transformational implementation: an example." *IEEE Transactions on Software Engineering SE-7*, 1 (January 1981).

[2] Bibel, W. and Horning, K. M. LOPS - A System Based on a Strategical Approach to Program Synthesis. Proceedings of the International Workshop on Program Construction, France, September, 1980.

[3] Davis, R. and Lenat, D. B.. *Knowledge-based Systems in Artificial Intelligence.* McGraw-Hill, 1981.

[4] Forgy, C. L. OPS5 User's Manual. Tech. Rept. CMU-CS-81-135, Carnegie-Mellon University, July, 1981.

[5] Kant, E. and Barstow, D. R "The Refinement Paradigm: The Interaction of Coding and Efficiency Knowledge in Program Synthesis." *IEEE Transactions on Software Engineering SE-7*, 5 (September 1981), 458-471.

[6] Kant, E. and Newell, A. Naive algorithm design techniques: a case study. Proceedings of the European Conference on Artificial Intelligence, Orsay, France, July, 1982.

[7] Kant, E. and Newell, A. Problem Solving for the Design of Algorithms. Tech. Rept. CMU-CS-82-145, Carnegie-Mellon University, November, 1982.

[8] Kosslyn, S. M.. *Image and Mind.* Harvard University Press, Cambridge, Massachusetts, 1980.

[9] Langley, P. A., Bradshaw, G. L., and Simon, H. A. Bacon.5: the Discovery of Conservation Laws. Proceedings of IJCAI-81, 1981, pp. 121-126.

[10] Lenat, D. B. "The Nature of Heuristics." *Artificial Intelligence 19*, 2 (1982), 189-249.

[11] Moore, J. A. *The Design and Evaluation of a Knowledge Net for MERLIN.* Ph.D. Th., Carnegie-Mellon University, 1971.

[12] Newell, A. Heuristic programming: Ill structured problems. In *Progress in Operations Research*, Aronofsky, J., Ed.,Wiley, 1969, pp. 360-414.

[13] Rich, C. *Inspection Methods in Programming.* Ph.D. Th., Massachusetts Institute of Technology, June 1980.

[14] Swartout, W., and Balzer, R. "On the Inevitable Intertwining of Specification and Implementation." *CACM 25*, 7 (July 1982), 438-440.

[15] Tappel, S. Some Algorithm Design Methods. Proceedings of the First Annual National Conference on Artificial Intelligence, August 18-21, 1980, pp. 64-67.