***

Chapter 2
AUTOMATED REUSE OF DESIGN PLANS IN BOGART

Jack Mostow, Michael Barley, and Timothy Weinrich

Abstract

BOGART, part of the VEXED knowledge-based circuit design editor, partially automates circuit (re-)design by mechanically "replaying" the recorded history of design decisions made in a previous design. We illustrate how it designs part of a content-addressable memory by replaying the design plan for a comparator, and how it helps implement several specification changes for a simple arithmetic and logic unit (ALU). We evaluate how well BOGART addresses five general issues raised by this approach to intelligent design automation. BOGART has been used by students in a VLSI course to help design simple NMOS digital circuits, and its techniques have been applied to mechanical design and algorithm design. Experimental results indicate that automated replay can be significantly more effective than structure-copying in reducing user effort.

2.1. INTRODUCTION

Much of design consists of re-design, whether in the adaptation of a previous design to a new context, or in the design iteration cycle. Mechanical design databooks, software subroutine libraries, and hardware standard cell catalogs all testify to the usefulness of reusing previous designs. However, these approaches to reuse are somewhat inflexible: a databook entry, subroutine, or standard cell has limited value when it is not reusable "as is," even though many of the design *decisions* incorporated in it may still apply. If the stored design is 99% right for a given task, patching it to fit often takes much more than 1% of the effort

© 1989 Computational Mechanics Publications.
Adapted from 'Automated Reuse of Design Plans,'
by Jack Mostow, Michael Barley, and Timothy Weinrich,
from the *International Journal of Artificial Intelligence and Engineering*; Vol. 4, No. 4, pp. 181–96; October 1989.

needed to create the design in the first place, cancelling so much of the advantage of reusing it that it may be easier to re-design from scratch. To take a software example, consider the case where a procedure is needed to compute some mathematical function with more precision than the one in the subroutine library. Writing the procedure from scratch is likely to be easier than patching the library subroutine, even if they use the same basic algorithm. Although human designers sometimes modify the structure of an old design to fit a new application, this process tends to require considerable expertise; since we are interested in *automating* the reuse of designs, we want to avoid the difficulties associated with structure-patching.

A consensus has emerged among artificial intelligence researchers investigating design that the solution to this problem is to reuse the *process*, not the *product*, of design. In particular, if a designer's decisions can be recorded in machine-understandable form, it should be possible to "replay" the recorded design history [1, 2, 3, 7, 16, 20], perhaps after editing a few decisions to fit the new problem.

Replaying a design history is less straightforward than it sounds [8, 9]. Some of the issues include:

1. *Acquisition*: How can design decisions be captured at an appropriate level of description?
2. *Retrieval*: Given a design problem, how can relevant previous designs be identified?
3. *Flexibility*: How can a previous design be profitably reused even though it is only partially suitable?
4. *Appropriateness*: How is it possible to identify which decisions made in a previous design should be replayed?
5. *Correspondence*: How is it possible to decide which parts of a previous design correspond to which parts of a new problem?

This chapter presents BOGART, a facility that automatically replays "design plans" in the VEXED system for designing digital circuits (Volume I, Chapter 8). It should be emphasized that BOGART is only a proof of concept. Like VEXED, it is a research tool, not a design tool intended for realistic tasks. BOGART is an experimental vehicle for investigating the issues listed above. Its purpose is to test the hypothesis that *automated reuse of design plans is both feasible and useful.*

The rest of this chapter is organized as follows. Section 2.2 illustrates how VEXED assists and records the design of a simple comparator circuit. Section 2.3 shows how BOGART uses the recorded design plan to design part of a unit cell for a content-addressable memory. Section 2.4 describes the results of subjecting BOGART to a "trial by students." Section 2.5 reports on an experiment to compare the usefulness of copying and replay in designing several variants of a simple ALU. Section 2.6 analyzes the extent to which BOGART addresses the five issues raised above. Section 2.7 describes applications of the BOGART approach to mechanical design and algorithm design. Section 2.8 briefly surveys related work. Section 2.9 concludes, with a summary of what our experiments showed about our hypothesis.

2.2. USING VEXED TO CONSTRUCT A DESIGN PLAN

In this section we briefly introduce VEXED, a design aid for NMOS digital circuits, and show how it constructs the design plans replayed by BOGART; VEXED is described more fully in Volume I, Chapter 8. VEXED is implemented in Interlisp-D on Xerox D-series machines and uses the Strobe object-oriented programming system developed at Schlumberger-Doll Research.

2.2.1. VEXED's Model of Interactive Design

VEXED was developed to investigate the hypothesis that the design process can usefully be modelled as top-down refinement plus constraint propagation (see Volume I, Chapter 8). In this model, a circuit design problem is represented as a "black box" module with specifications on various features of its inputs and outputs, such as their datatypes, values, timing, and encoding. Each *top-down refinement* step decomposes a module into a few interconnected submodules, each of which is refined in turn until the entire circuit has been refined into known components of the target technology (e.g., transistors, gates, standard cells).

The purpose of top-down refinement is to factor the original design problem into independent subproblems, but in practice the connections between modules mean that decisions about how to implement one module can constrain the possible implementations of another. For instance, if the output of module A is connected to the input of module B, then the decision to implement module A with serial output precludes implementing module B with parallel input. *Constraint propagation* is the process of inferring how decisions made at one point in the design constrain the options elsewhere in the design.

This model of design is embodied in VEXED with a division of labor between user and machine intended to exploit the strengths of each party [6]; the user is responsible for strategic decisions, while VEXED takes care of the detailed manipulation and constraint propagation needed to carry out those decisions and deduce their effects.

2.2.2. Problem Specification

Design in VEXED starts with a top-level specification provided by the user in a somewhat unreadable syntax resembling the programming language LISP. Figure 2-1 paraphrases the specification for the output value of a one-bit comparator circuit. (The figure omits specifications for the inputs and for other features of the output, such as datatype, encoding, and timing.) No circuit schematic is shown yet because the specification does *not* say how it is to be implemented. In fact, several different circuits could be designed to realize this specification.

(input-1 AND input-2) OR ((NOT input-1) AND (NOT input-2))

Figure 2-1: Output Value Specification for COMPARATOR-CELL

2.2.3. Refinement Rules

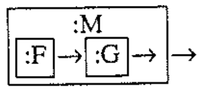
VEXED has a catalog of "if-then" refinement rules, one of which is shown in Figure 2-2. The "if" part of a rule tests whether the rule can be used to correctly refine a given module. The "then" part refines the module into one or more submodules and their interconnections. A rule states that any module satisfying the "if" part *can* be refined by the "then" part, not that it necessarily *should* be; different rules can represent alternative ways to refine the same module.

2.2.4. VEXED's Interactive Design Cycle

VEXED is based on an interactive cycle in which some parts are done by VEXED and others by the user. At the start of each cycle, VEXED displays a menu of the modules remaining to be refined, and the user selects which one to refine. VEXED finds all the rules that can refine the module, that is, those

---

If the specified output value of module :M has the form ":FN1 OR :FN2," then it can be refined into a circuit of the following form:



where :F outputs :FN2 and :G outputs (:G_{in} OR :FN1).

Figure 2-2: Paraphrase of the OR-DECOMP rule;
Rule variables, e.g. :M, are prefixed by a colon.

whose "if parts" match the module specifications, and displays them in a menu. The user chooses which one to apply, based on what kind of design it will eventually lead to. Thus the user is responsible for choosing what part of the design to work on next and what implementation strategy to try.

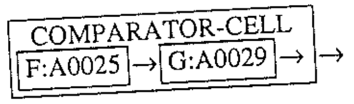
2.2.5. Rule Application

VEXED applies the selected rule to the selected module, refining it into submodules and automatically propagating the effects of this decision to other modules affected by it. VEXED's constraint propagator, named CRITTER, is described in Chapter 8, Volume I.

The OR-DECOMP rule refines COMPARATOR-CELL, whose output value specification is shown in Figure 2-1, into modules F:A0025 and G:A0029. Figure 2-3 shows the result, with some of the connections omitted for simplicity. F:A0025 computes the second part of the specified expression and outputs it to the input port of G:A0029, which does the rest of the work.

2.2.6. Manual Refinement

What if no rules apply to a given module, or the user doesn't like any of the ones that do? In this case, the user can use VEXED's graphical editor to decompose the module manually into submodules and their interconnections, and type in their specifications in VEXED's specification language. VEXED uses con-



F:A0025 outputs (NOT input-1) AND (NOT input-2)

G:A0029 outputs G:A0029$_{in}$ OR (input-1 AND input-2)

Figure 2-3: Result of Applying OR-DECOMP Rule to COMPARATOR-CELL

straint propagation to check that the decomposition correctly implements the module. In fact, VEXED has a learning facility named LEAP (for "Learning Apprentice") that generalizes the manual decomposition step into a new rule that correctly refines any module to which it applies [5].

After a manual refinement step, VEXED's normal interactive design cycle resumes, treating the new submodules just as if they had been created by a rule. Thus when VEXED lacks a rule capable of decomposing a module as desired, the user is not forced to design the module the rest of the way down by hand.

2.2.7. Resulting Design

VEXED's interactive cycle repeats until the entire design has been refined down to the level of primitive components. Figure 2-4 shows the final design for COMPARATOR-CELL, with the module structure omitted. The circuit contains four inverters, two transistors used as AND gates, and one NAND gate. It works as follows. The input signals are fed to one AND gate, and the inverted input signals are fed to another. The results of these two conjunctions are wired together to compute the desired output value. The additional inverters and NAND gate are used to amplify the signal, which is attenuated by the transistors.

Circuit design experts may notice that this design could be improved. In particular, the output signal could be amplified simply by passing it through two inverters in series, thereby eliminating a NAND gate. Although VEXED's refinement rules ensure that the design is functionally correct, VEXED leaves the user responsible for minimizing the consumption of resources like the number of cir-

cuit components. As this example illustrates, choosing the wrong decomposition can lead to a sub-optimal design. Reference [19] describes a system called SCALE that learns optimal decompositions.

2.2.8. The Design Plan

VEXED records the successive refinement steps in a tree-like *design plan* like the one shown in Figure 2-5, which shows the final design plan for the comparator. The plan has a node for each module in the circuit. The step that refined a module into its submodules is shown as a thin box labelled with the name of the rule. For instance, the top step used the OR-DECOMP rule to refine COMPARATOR-CELL into submodules F:A0025 and G:A0029. (Other aspects of Figure 2-5 will be explained as needed.) Although not shown, the design plan also includes the "bindings" for each refinement step, which indicate the module or expression corresponding to each rule variable. In the top step, variable :M was bound to COMPARATOR-CELL, :F to module F:A0025, and :G to module G:A0029.

Because the user cannot always tell the right rule to choose at each step, it is sometimes necessary to retract a design decision. VEXED provides a backtracking command that returns the design to a user-selected past state, retracting all the refinement steps made since then and erasing them from the design plan. Thus the final design plan is an *idealized* history that omits steps that were taken but later retracted [7].

2.3. HOW BOGART REUSES A DESIGN PLAN

In this section we try to indicate what it is like to use BOGART to help design circuits. Specifically, we will show how BOGART uses the COMPARATOR-CELL design plan shown in Figure 2-5 to automate much of the design of a one-bit key-test unit for a cell in a content-addressable memory. Each cell stores a key and a data item. To retrieve the data item associated with a given key, the key is broadcast to all the cells. Each cell tests the broadcast key against its stored key. If they match, the cell returns its data item on a common bus. The key is tested by combining the outputs of several one-bit key-test units, one for each bit in the key. Figure 2-6 specifies the output value for such a unit. The unit holds one bit of the key. The unit compares the stored bit against the current value of input-1 and outputs TRUE if they match. If the control input Load

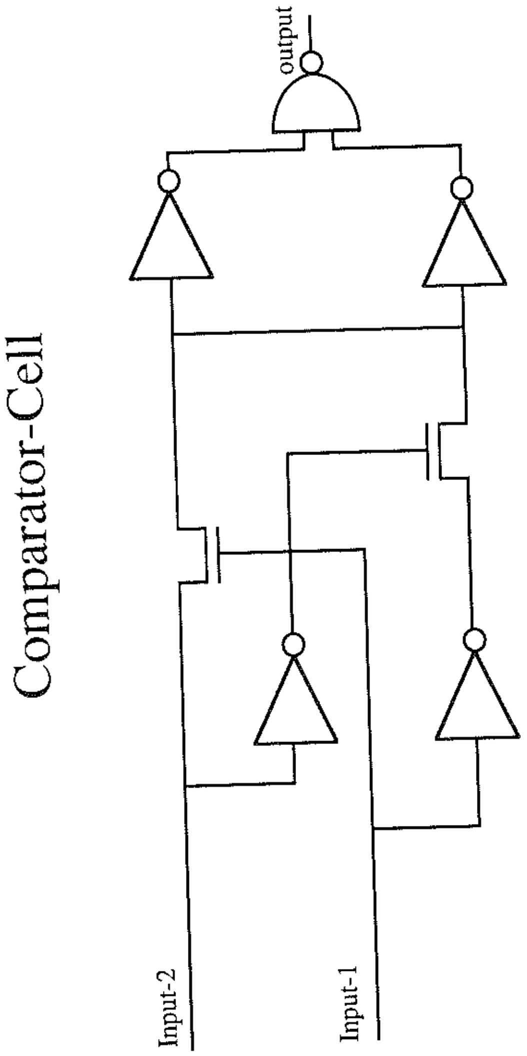
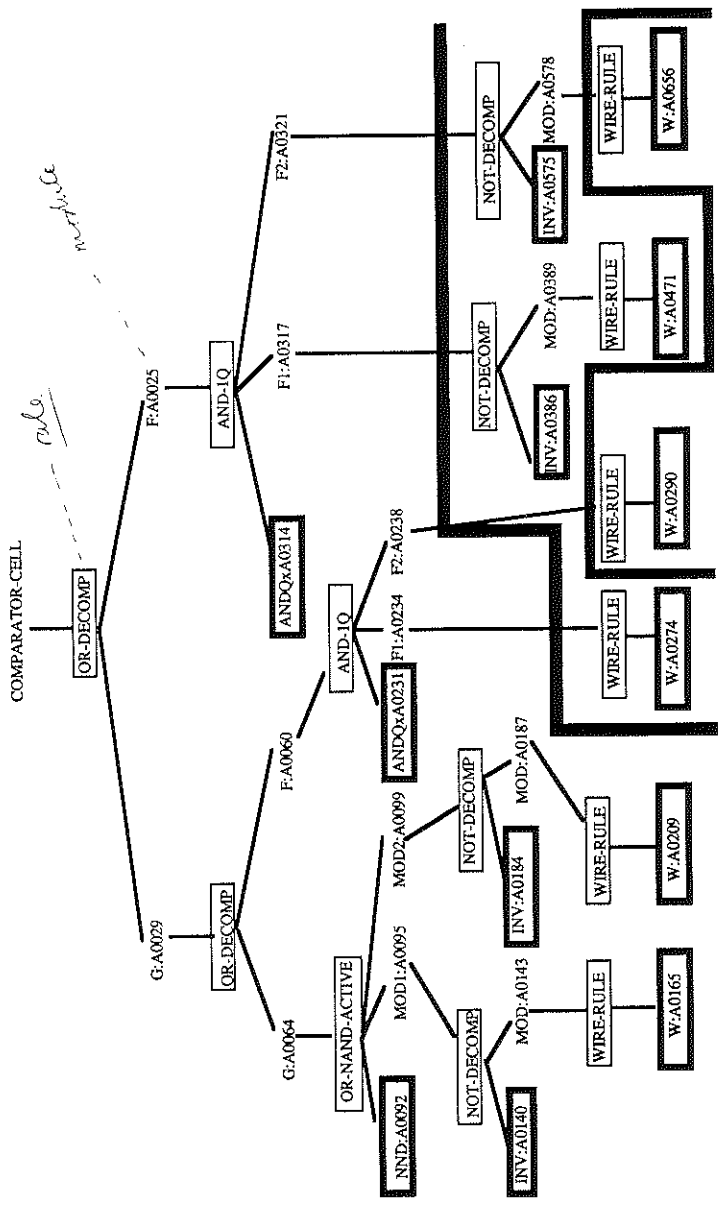


Figure 2-4: Final Circuit Design for COMPARATOR-CELL



Unboxed nodes represent modules, e.g., COMPARATOR-CELL. The rules used to refine them are boxed with thin lines, e.g., OR-DECOMP. The heavily boxed modules show how far down the user asked to replay the plan, in this case all the way to the leaves of the tree, e.g., NND:A0092. The lower boundary shows how far replay got for the KEY-TEST-UNIT problem. The upper boundary shows how far BOGART replayed when two clauses were transposed.

Figure 2-5: The Design Plan for COMPARATOR

is TRUE, the value of input-1 gets stored as the new key. The specification is expressed rather unusually because VEXED's specification language lacks a more conventional way to describe memory.

```
input-1 AND (input-1 when Load was last TRUE) OR
(NOT input-1) AND (NOT input-1 when Load was last TRUE)
```

Figure 2-6: Output Value Specification for KEY-TEST-UNIT

KEY-TEST-UNIT could be implemented in VEXED in the same interactive fashion as COMPARATOR-CELL. However, it is possible to automate most of its design by reusing the design plan for COMPARATOR-CELL. We now describe how a VEXED user would use BOGART to refine KEY-TEST-UNIT most of the way down into primitive components.

2.3.1. Deciding What to Replay

Our scenario begins with the user deciding to implement KEY-TEST-UNIT by replaying a previous design and invoking the Replay option instead of choosing a refinement rule to apply. BOGART displays a menu of known design plans, including the design currently in progress as well as previously saved designs. A design doesn't have to be complete for BOGART to replay its design plan. Designs are saved when completed, but can also be saved before backtracking, to allow retracted decisions to be replayed later.

The user selects a design plan that seems relevant, and chooses what subtree of it to replay. In our example, the user selects the design plan shown in Figure 2-5. The user can designate any subtree by selecting its root and leaves. To find the part of the previous design that resembles the module to be implemented, the user can use VEXED's display facilities to browse through various parts of a circuit and adjust the level of detail displayed. Since it is the top-level module in Figure 2-5, COMPARATOR-CELL, that most resembles the KEY-TEST-UNIT to be implemented, the user selects it as the root of the desired subtree.

In Figure 2-5, the heavily boxed modules represent how far down to replay. Here the user has accepted the default, which is to try to replay down to the leaves of the original design plan. However, the user could have moved the boundary, thereby instructing BOGART to stop short of the leaves. This feature allows reuse of an *abstracted* version of the plan, in that the overall strategy (for how to decompose the high-level modules) is retained while the low-level details are dropped, as in the ARGO system (Chapter 3).

2.3.2. BOGART's Automatic Design Cycle

Once the user has selected what part of the design plan to replay, BOGART starts the replay process, which consists of repeatedly selecting the next step to replay and refining the corresponding module. The replay cycle is entirely automatic, relieving the user of the responsibilities required in VEXED's interactive design cycle; this lessening of the user's burden is BOGART's raison-d'être.

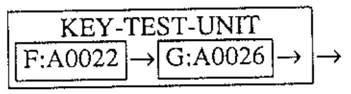
The first step of the replay cycle chooses which module to refine next. BOGART maintains an agenda of unrefined modules that correspond to nodes in the design plan, listed in the same order in which the originals were created. BOGART picks the first module in the agenda and refines it, using the rule recorded in the design plan. Thus the user is spared from choosing which module to refine next and which rule to apply. The rule is applied as described in Section 2.2, by matching the "if part" against the module specification, creating the submodules and interconnections prescribed by the "then part," and performing constraint propagation.

If the rule condition is not satisfied, BOGART skips over the agenda item and tries it again on the next cycle, since it is possible for the rule condition to become satisfied by information filled in later by the constraint propagator, based on subsequent decisions made elsewhere in the design.

2.3.3. Identification of Corresponding Modules

When BOGART replays a step, thereby creating new modules, it uses a simple *correspondence heuristic* to identify the original module corresponding to each new one. Consider the top step in Figure 2-5, which used the OR-DECOMP rule to refine COMPARATOR-CELL, binding the rule variables :F and :G to COMPARATOR-CELL's submodules F:A0025 and G:A0029, respectively, as shown earlier in Figure 2-3.

When this step is replayed to refine KEY-TEST-UNIT, the OR-DECOMP rule binds :F and :G to KEY-TEST-UNIT's new submodules F:A0022 and G:A0026, respectively. KEY-TEST-UNIT's output value specification is shown in Figure 2-6; F:A0022 computes the second half of this expression, and outputs its value to G:A0026, as sketched in Figure 2-7.



F:A0022 outputs
(NOT input-1) AND (NOT input-1 when Load was last TRUE)

G:A0026 outputs
G:A0026$_{in}$ OR (input-1 AND input-2)

Figure 2-7: Result of Replaying the OR-DECOMP step for KEY-TEST-UNIT

As Figure 2-1 shows, BOGART's correspondence heuristic assumes that a new module corresponds to an old module if they are created by the same step in the design plan and bound to the same rule variable. According to this heuristic, F:A0022 corresponds to F:A0025 and G:A0026 corresponds to G:A0029. Since F:A0022's specification, shown in Figure 2-7, resembles F:A0025's, shown in Figure 2-3, the design plan for F:A0025 can be replayed to help implement F:A0022. Similarly, the design plan for G:A0029 can be replayed to help implement G:A0026. Thus the heuristic works well in this example.

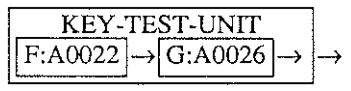
Table 2-1: BOGART's Bound-to-same-variable Correspondence Heuristic

Design:	:FN1	:FN2	:F,:G
COMPARATOR	input-1 AND input-2	(NOT input-1) AND (NOT input-2)	F:A0025, G:A0029
KEY-TEST-UNIT	input-1 AND (input-1 when Load was last TRUE)	(NOT input-1) AND (NOT input-1 when Load was last TRUE)	F:A0022, G:A0026
TRANSPOSED	(NOT input-1) AND (NOT input-1 when Load was last TRUE)	input-1 AND (input-1 when Load was last TRUE)	F:A0022, G:A0026

However, the "bound-to-same-variable" correspondence heuristic is not infallible. For example, suppose the two AND clauses in Figure 2-6 are transposed as follows:

(NOT input-1) AND (NOT input-1 when Load was last TRUE)
OR (input-1 AND (input-1 when Load was last TRUE))

Then the result of replaying the OR-DECOMP step is as indicated in Figure 2-8. F:A0022's specification no longer resembles F:A0025's as closely, so F:A0025's design plan gives less help in implementing F:A0022. The AND-1Q step used to decompose F:A0025 (see Figure 2-5) still works, but the rest of the subplan for F:A0025 cannot be replayed: since F:A0022's output specification has no NOTs, the NOT-DECOMP steps are no longer applicable. The subplan for G:A0029 cannot be replayed as far as before, either.



F:A0022 outputs input-1 AND (input-1 when Load was last TRUE)

G:A0026 outputs G:A0026$_{in}$ OR
(NOT input-1) AND (NOT input-1 when Load was last TRUE)

Figure 2-8: When the Correspondence Heuristic Fails: Result of OR-DECOMP for Transposed KEY-TEST-UNIT Specification

2.3.4. Replay of Non-refinement Steps

While BOGART's correspondence heuristic depends on the top-down decomposition nature of VEXED's rules, there are some exceptions. In particular, the Refinement-Get-Signal rule implements a specified data value by "stealing" it from a point where it is already computed somewhere else in the circuit. To understand how much VEXED's top-down model really restricted BOGART, we decided to see if we could find a way to replay Get-Signal steps.

It turned out to be easy, but in a restricted sense. When Refinement-Get-Signal is invoked, it asks the user to identify which signal to "steal." It then performs a Get-Signal operation that inserts the actual connection in the circuit design. More precisely, it refines the module into a wire whose source is the signal. When BOGART replays a Refinement-Get-Signal step, it simply asks the same question, and lets the user identify the appropriate signal in the new circuit. A more sophisticated replay mechanism would automatically guess which new signal corresponded to the old one.

2.3.5. Results of Replay

The replay cycle repeats until the agenda is empty or none of the items left can be replayed, because their rule conditions are not satisfied. If a step in the design plan remains inapplicable, the subplan that starts with that step is not replayed, but this does not stop BOGART from replaying other branches of the design plan.

For the KEY-TEST-UNIT specification, BOGART replays 13 steps of the 15-step design plan shown in Figure 2-5. The two steps that were not replayable, shown below the lower boundary in the figure, implemented the two instances of input-2 in COMPARATOR's specification simply as wires. The expression corresponding to input-2 in KEY-TEST-UNIT's specification is

(input-1 when Load was last TRUE)

This expression does not satisfy the "if" part of WIRE-RULE; it must be implemented as a memory rather than a simple wire. Even though the user asked BOGART to replay the entire COMPARATOR-CELL design plan, BOGART refrains from replaying steps that are no longer applicable. If the specification of KEY-TEST-UNIT is transposed as discussed above, BOGART replays only 9 of the 15 steps, down to the upper boundary in Figure 2-5.

After BOGART replays as much of the requested portion of the design plan as it can, the automatic mode ends and VEXED's interactive design cycle resumes. At this point, the design of KEY-TEST-UNIT is not quite completed; in particular, the storage of input-1 remains to be implemented. The user continues by choosing another design plan for BOGART to replay, selecting an individual rule for VEXED to apply, refining a module by hand, or backtracking to retract decisions whose results turned out to be unsatisfactory. A completed version of the partial KEY-TEST-UNIT design produced by replay is shown in Figure 2-9.

2.4. HOW BOGART IS USED

So much for theory; how well can BOGART assist the design process in practice? To get a preliminary answer to this question, we subjected BOGART to a "trial by students." In Fall 1986, a graduate-level class on "Introduction to VLSI Systems," taught by Dr. Don Smith in the Computer Science Department at Rutgers, used VEXED and BOGART to design simple circuits. Each team of two students designed one circuit, with roughly the complexity of a one bit full adder. Without previous designs to replay, the students were restricted to replay-

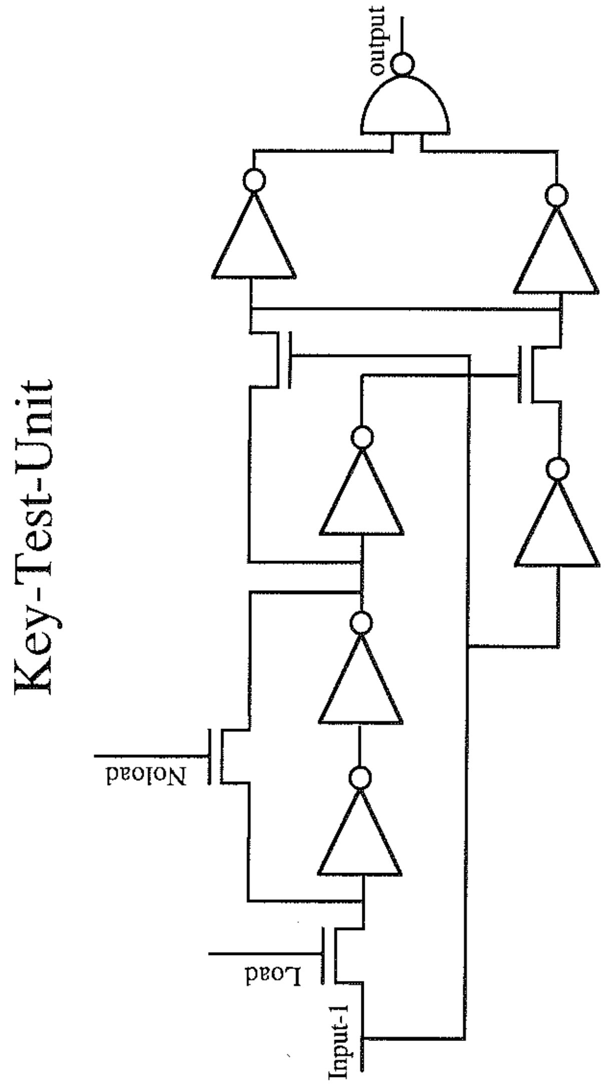


Figure 2-9: Final Circuit Design for KEY-TEST-UNIT

ing portions of their own design. Even with this restriction, they reported finding several uses for replay.

2.4.1. Replicated Designs

The simplest use was for designing replicated structures. Given two modules with identical specifications, students could implement one of them automatically by using BOGART to replay the design plan for the other. Although a simple structure-copying mechanism could achieve much the same effect, it wouldn't solve the problem of interfacing the copied structure to its surrounding environment, or verifying that the structure satisfies its specification (especially timing constraints) in that environment.

2.4.2. Design by Analogy

Design by analogy is a generalized form of replication. Although the students lacked previous design plans to reuse, we thought they might use BOGART to design part of a circuit by analogy with a similar part of the same circuit. Apparently this kind of "internal analogy" did not arise very often, partly because the design plans were so small. We did hear of one unintentional case where the student tried to use BOGART to replicate a module, but it didn't work quite as expected, because the specification for the new module was a transposed version of the old one. BOGART replayed as much of the module's design plan as it could, and the student used VEXED to complete the rest. If the student had intentionally used BOGART in this manner, we would certainly call it design by analogy.

2.4.3. Design Iteration

The most common use of BOGART occurred in conjunction with backtracking. We added a feature to VEXED's backtrack mechanism to allow the user to save (and name) the current design plan before backtracking so that it would be available for BOGART to replay. Students found two uses for this facility, consistent with the following *derivational* model of design iteration [8]:

1. Successively refine the design.
2. Decide to modify a previous decision.
3. Save the current design plan.
4. Backtrack to the decision point and make the modification.
5. Replay the relevant portion of the saved plan.
6. Go back to 1 and repeat until satisfied.

One kind of design iteration occurred when the specification was found to be flawed. The student would save the current design plan, backtrack to the beginning, fix the specification, and then replay the design plan to salvage whatever design decisions were still relevant.

Another kind occurred when the student decided to retract a refinement step in order to try something else, either because it had led to a dead end, or because the student wanted a better solution. VEXED's backtracking mechanism cannot retract a refinement step without retracting all subsequent steps, which might represent a considerable number of decisions on the part of the user. By saving the design plan before modifying the decision, and then replaying as much as possible of it afterwards, students avoided having to make the same decisions over again. A more sophisticated *dependency-directed* backtracking mechanism [15, 17, 18] might achieve the same effect more efficiently by eliminating the expense of re-executing each step, but constraint propagation complicates the problem of adding such a mechanism to VEXED.

2.4.4. Design Exploration

In theory, BOGART also supports a more systematic form of design exploration. The user could compare alternative designs by developing and evaluating a (perhaps partial) design, saving its design plan, backtracking to a "crossroads" in the plan where an alternative strategy seems worth investigating, and trying it. If the second design proves inferior, the user could restore the first design by backtracking to the crossroads and replaying the saved plan. Although this scenario is possible in principle, it is not feasible in practice, because VEXED and BOGART are, like many experimental AI systems, painfully slow: a single refinement step and the ensuing constraint propagation can take several minutes. Design exploration might be performed without backtracking if VEXED had a version management facility for representing more than one design at a time. Even with such a facility, replay could still help the user develop one design by borrowing useful ideas from another.

2.5. EXPERIMENTAL MEASUREMENTS

While our experience with student use of BOGART indicated that they found it useful, we wanted to test it more systematically. The students' experience raised the question of whether BOGART was providing much more than a simple module-copying facility. This capability is found in standard circuit editors but was lacking in VEXED at the time. It is not surprising if replay is easier than designing from scratch, but is it significantly better than copying? Also, how does the work performed by BOGART compare with the effort required to invoke it? How do the relative benefits of copying and replay depend on the relationship between the old and new designs?

2.5.1. Experimental Methodology

To address these questions, we performed an experiment in which we compared the amount of user effort required using both approaches. To design a meaningful experiment, we had to define suitable experimental conditions and measures. This task turned out to be less straightforward than it might at first appear. It required some simplifying assumptions which we shall now attempt to make explicit.

2.5.1.1. What to compare

An ideal experiment would measure the "natural" use of copying and replay, including dead ends and backtracking. However, the results would depend on the individual user's familiarity with the problem and the tool. To obtain statistically significant results, one would have to test many different users on different tasks -- an experimental luxury we did not have. Instead, we factored out these confounding effects by finding the number of decisions required to produce a correct design without backtracking. Thus the applicability of our results to predicting the performance of copying and replay in practice depends on the assumption that their relative usefulness would stay the same when normal trial and error are added in.

We used the same target design in both cases, to factor out any differences between copying and replay that might affect the correctness or quality of the design. Thus our experiment avoided any errors caused by copying a module for a function it does not correctly perform. Replay prevents such errors by retesting rule preconditions. Relying on this feature, we let BOGART decide how far to replay the plan, rather than choosing leaves ourselves. This mode is the default, and reflects typical use of BOGART.

Next, we had to decide exactly what to compare. That is, what design operations would the user be allowed to perform under the two conditions? For example, what kind of copying would we provide? To test the usefulness of replay conservatively, we decided to provide a copy command capable of copying any module. To clarify further the differences between copying and replay, and to avoid the complications of trying to replay Copy-Module steps, we excluded the use of copying when we allowed replay. Thus our experiments compare copying against replay without copying.

2.5.1.2. How to measure user effort

To measure user effort, we assumed that it is the sum of the effort required to make each decision leading to a design. Since BOGART has a well-developed menu-based graphic interface, each decision corresponds to selecting an item from a menu or graphical display. We ignored any other decisions that the user might be making, such as what to think about next, or where to look on the screen.

Since some user decisions are harder than others, one would ideally want to measure the effort required for each one. However, this effort depends on several irreproducible factors, including differences among individual users, so we did not try to measure it directly. Instead, we estimated the difficulty of a decision according to the number of options available. If the user spends some time thinking about each option before choosing one, we would expect the amount of effort involved in a decision to increase with the number of options. On the other hand, decisions where all the options are familiar can require less effort, especially if one of them is the normal choice. Also, every decision requires a certain mental effort to process, even if the right choice is obvious. We therefore classified the various types of decisions as follows, in decreasing order of difficulty.

Many options: Decision made when many (over 20) options existed. Examples:

- Choosing one port out of all those in the circuit, as when choosing a port from which to obtain a Get-Signal.
- Choosing from a large menu.

Few options: Decision made when few (20 or fewer) options existed. Examples:

- Selecting a module to refine.
- Selecting a rule to fire.
- Choosing to refine a module by Replay or Copy-Module.
- Selecting a plan to replay.

Notice that selecting a module, rule, or plan would become more difficult in a larger design or knowledge base, but in our experiment there were relatively few options to choose from.

One sensible option: Decision made which any user familiar with VEXED could recognize as the sensible, workable choice. Examples:

- Selecting a rule to fire when the only alternatives are poor, unnecessarily strange, or equivalent to the one chosen.
- Connecting inputs to the ports of a module that computes a function like AND which is symmetric in its inputs, so that it doesn't matter which input signal is connected to which input port. The need to hand-wire signals to ports arises when Copy-Module is invoked.

Usual option chosen: Decision made that is "almost always" made in this context, or could have been made by the machine, with a small cost to correct it if the machine chose poorly. Examples:

- Decision to refine the already-chosen module with rules (instead of by replay).
- Decision to execute the already-chosen rule.
- Decision to connect signals to ports after invoking Copy-Module.
- Decision to wire the only two remaining ports to each other.
- Confirmations (saying "Y" to "Do you really want to do that?")
- Decision to replay the design plan all the way down to the leaves (the default) instead of specifying a termination point.

No real choice: Decision that could have been made by the machine in this context if there had been no user, and definitely would have been the right thing to do. Examples:

- Refining the only unrefined module.

- Invoking the command for selecting a signal to connect to the input port of a wire created by the Refinement-Get-Signal rule.

Rather than attempt to assign arbitrary numerical weights to these different types of decisions, we report the number of decisions of each type separately. The total number of mouse clicks constitutes an unweighted measure of effort.

It should be noted that this measure is somewhat sensitive to certain decisions incorporated in the design of VEXED's user interface. In particular, many decisions in the last category could in principle be automated. However, an objective estimate of user effort required a systematic analysis of all the input the user actually had to provide, even if we wished that some of it could be eliminated.

As a coarser-grained but less implementation-dependent measure, we also counted the number of steps in the old and new design plans, and how many of the latter were performed by replaying the former. Since Refinement-Get-Signal steps require some user effort to replay, we kept a separate count of them.

2.5.1.3. Choice of benchmark task

To compare replay and copying, we measured the effort required to implement several variants of a two-bit-wide ALU (arithmetic and logic unit). The ALU's actual output value specification in VEXED is shown in Figure 2-10, the design plan we developed in VEXED is shown in Figure 2-11, and the schematic for the resulting circuit is shown in Figure 2-12. It computes PLUS, AND, or OR of its two data inputs DATA-1 and DATA-2, depending on the value of the 2-bit control input INSTR to the case statement represented by the (SELECTQ ...) expression. For example, if the value of INSTR is 3, the ALU computes the logical OR of its two data inputs. VEXED's specification language uses a prefix notation similar to the language LISP for easier parsing, but the details of the notation do not matter here. Suffice it to say that the expression (DATA-VALUE DATA-1 I) denotes the value of the signal DATA-1 at time I.

We chose this problem for several reasons. First, it is about as large as VEXED can handle -- 76 transistors. The exact number depends on the implementation of certain components like half-adders and selectors, which are marked as primitive in VEXED's knowledge base. By treating these modules as primitive components whose implementations can be retrieved from a library, VEXED is able to represent the 2-bit ALU design using only about 25 modules.

Besides the complexity of the ALU, its *n*-bit-wide character tests the usefulness of copying and replay in designing iterated structures. Finally, data-sharing among the operations tests the ability to replay Get-Signal steps.

A more thorough empirical evaluation might use several diverse design tasks.

```
(VALUE
 FEATURES
 (DATA-VALUE
    NIL
    (I (ALL I))
    (EQUAL (DATA-VALUE OUT I)
           (SELECTQ
              (DATA-VALUE INSTR I)
              (1 (+ (DATA-VALUE DATA-1 I) (DATA-VALUE DATA-2 I)))
              (2 (AND (DATA-VALUE DATA-1 I) (DATA-VALUE DATA-2 I)))
              (3 (OR (DATA-VALUE DATA-1 I) (DATA-VALUE DATA-2 I)))
              NIL))
    NIL)
 (ENCODING
    NIL
    (I (ALL I))
    (EQUAL (ENCODING OUT I)
           (INTEGER (WIRES 2)
                    (BITS 2)
                    (FIRST-BIT LSB)
                    (BIT-ENCODING (BIT (0 LOW) (1 HIGH)))))
    NIL)
 (TYPE
    NIL
    (I (ALL I))
    (EQUAL (TYPE OUT I) INTEGER) NIL))
```

Figure 2-10: Specification of a Two-Bit ALU

However, VEXED can only handle small designs, so it is not possible to test BOGART across any realistic distribution of design tasks. Nonetheless, we consider the ALU task indicative of some of the ways in which replay might be used. Moreover, we expect that the usefulness of replay would increase with the size of the design.

2.5.2. Experimental Results

The results of our experiments are summarized in Figures 2-16 and 2-17. Figure 2-16 compares the total user effort required, with copying versus with replay, to implement the two-bit ALU from scratch and then to implement four variations of it, ranging from simply renaming the inputs to doubling the data width. This figure is intended to facilitate several comparisons of user effort: designing from scratch versus implementing a specification change; specification changes of different kinds; and the usefulness of replay compared to copying. The fewer decisions required, the less effort required. Figure 2-17 breaks down user effort into the number of decisions of each type, ranging from the

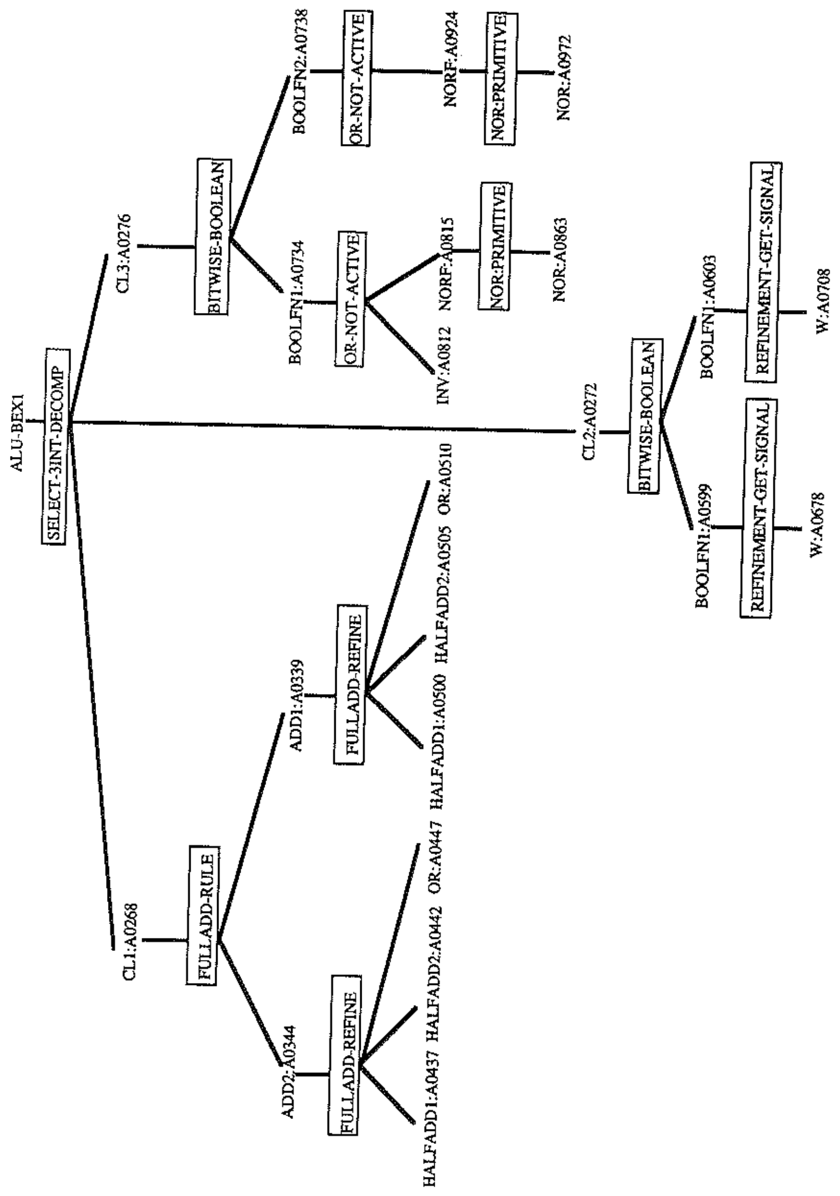
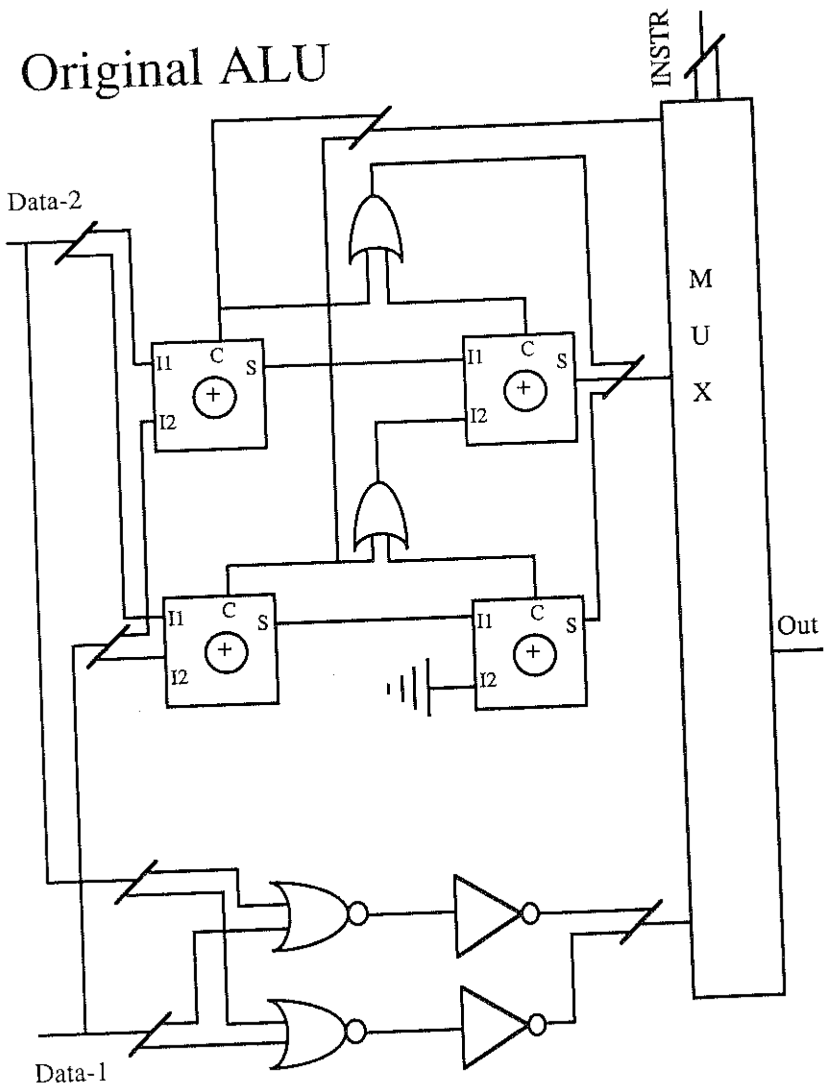


Figure 2-11: Design Plan for a 2-bit ALU



The multiplexer decodes a two-bit control input to select the PLUS, AND, or OR of Data-1 and Data-2.
PLUS is computed by four half-adders, each with outputs for Sum and Carry.
AND is taken from the Carry output, which is 1 iff both inputs are 1.
OR is computed by inverting the NOR of the two inputs.

Figure 2-12: Circuit Schematic for the Original 2-bit ALU

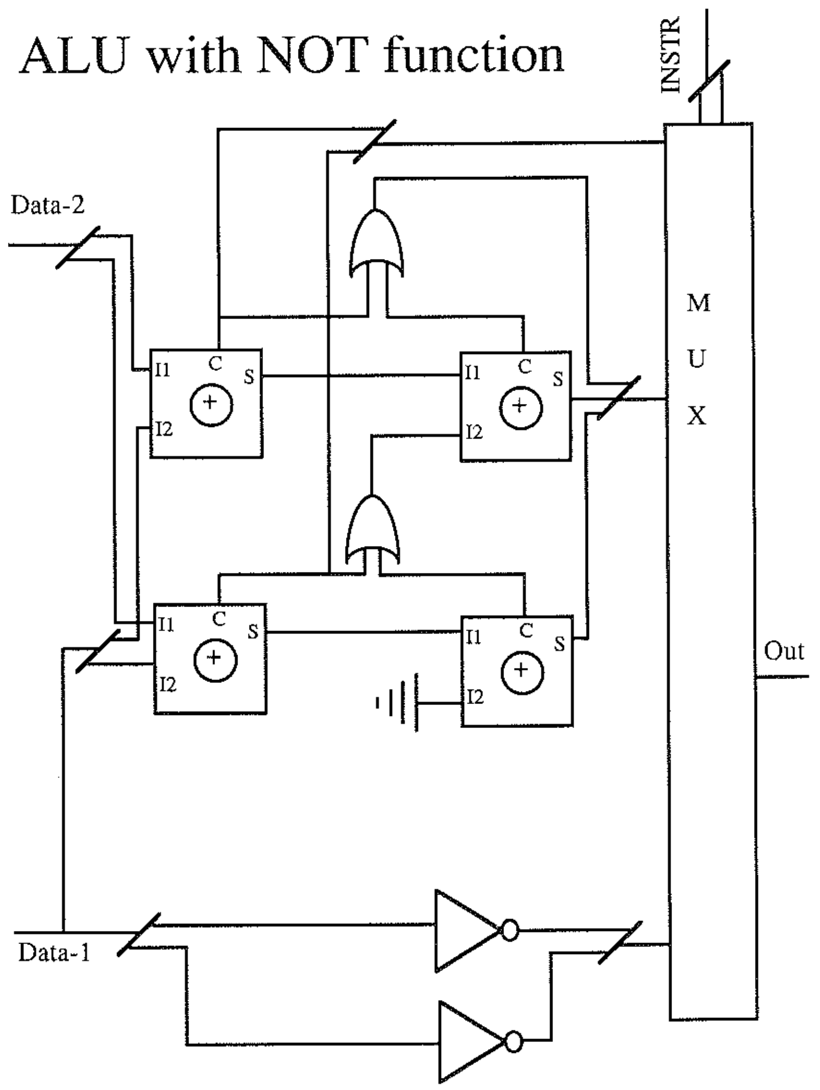


Figure 2-13: Circuit Schematic for ALU Variant Computing NOT Instead of OR

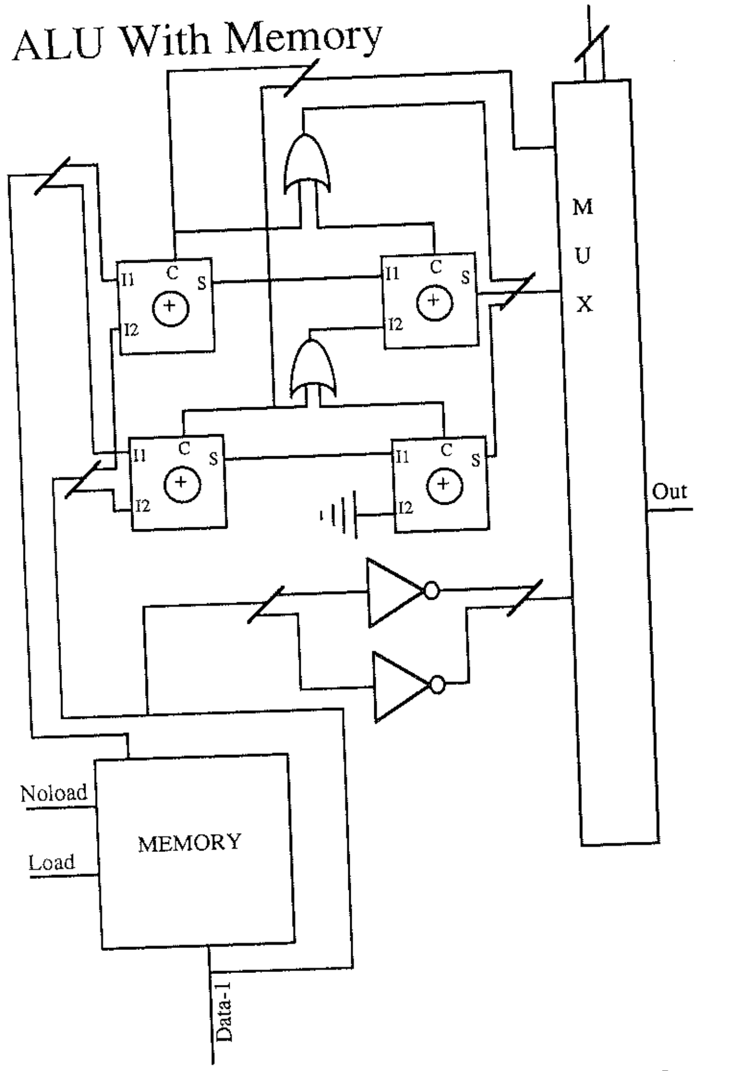


Figure 2-14: Circuit Schematic for ALU Variant that Uses Memory Instead of Data-2 Input

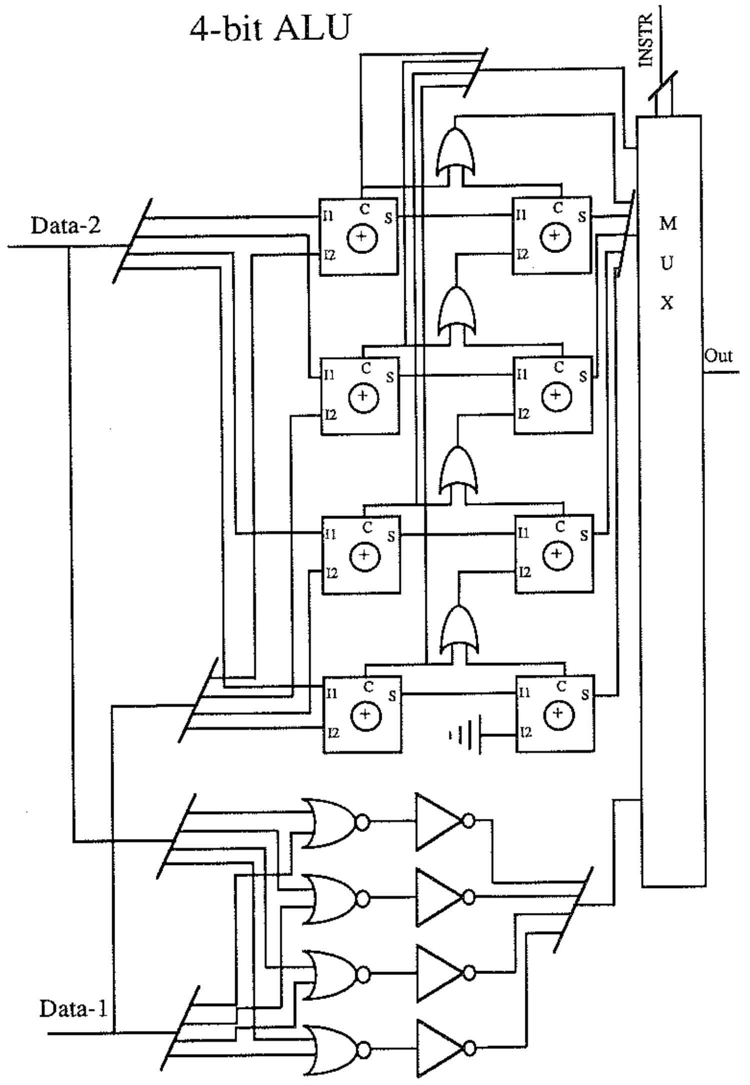


Figure 2-15: Circuit Schematic for 4-bit Variant of ALU

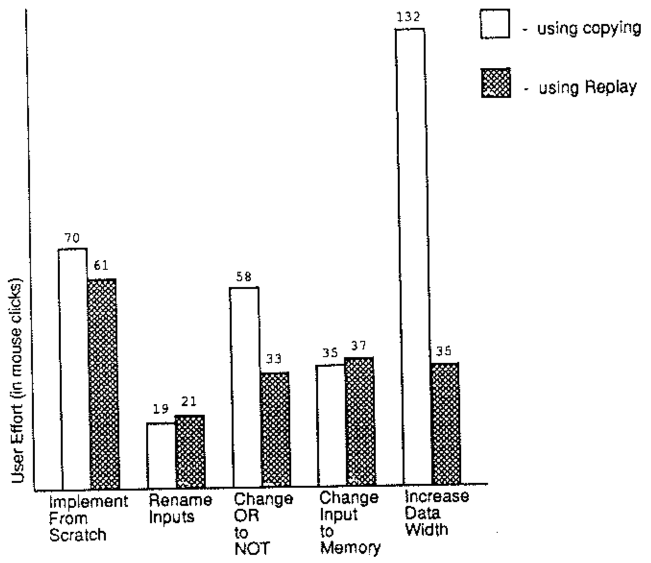


Figure 2-16: Relative User Effort with Copying versus Replay: Total Decisions Required

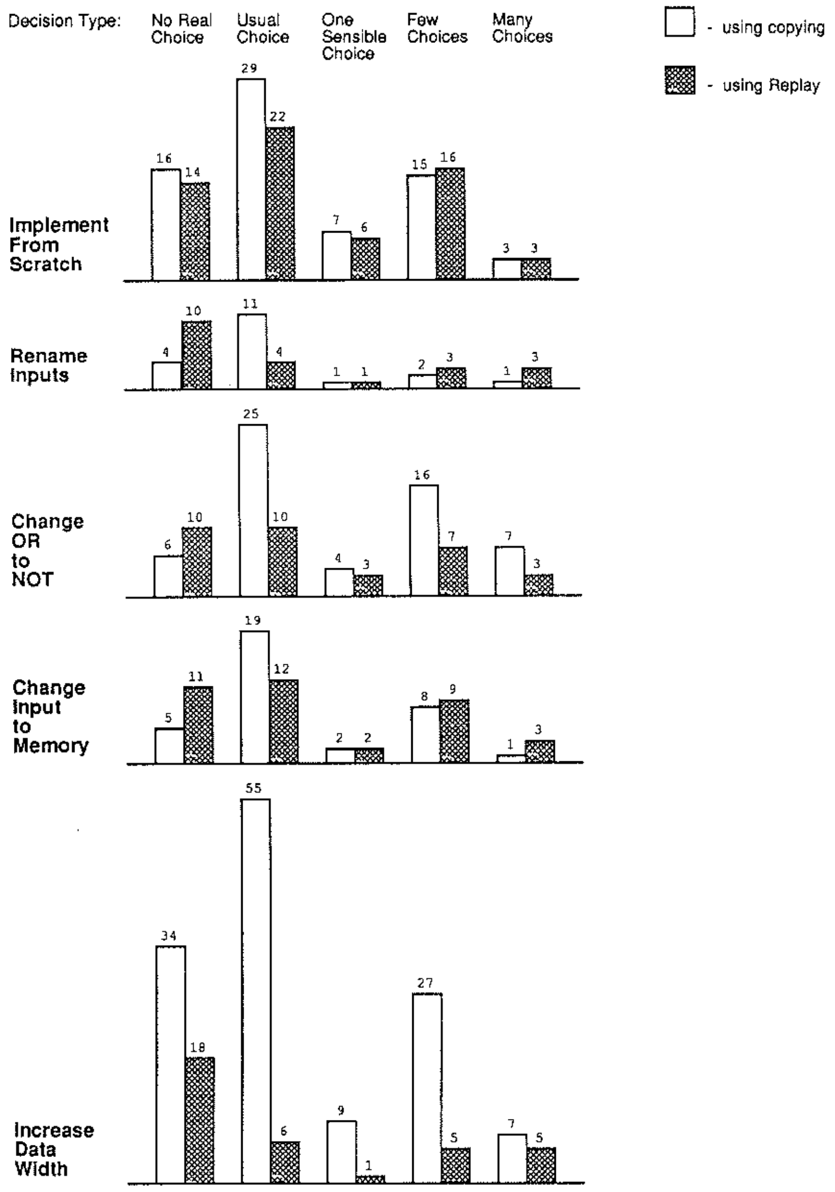


Figure 2-17: Relative User Effort with Copying versus Replay: Decisions of Each Type

easiest ("no real choice") to the hardest ("many choices"). Reduced effort is reflected not only by a reduction in the number of decisions, but also by a shift from harder to easier types of decisions. We now explain each experiment in turn.

2.5.2.1. Implement from scratch

In the first part of the experiment, we measured the effort required to design from scratch the ALU shown in Figure 2-12. It takes only 12 steps (applications of VEXED rules) to completely refine the ALU specification into a circuit. Even so, it does provide some opportunity for copying or replay, since the two bits are computed by similar circuitry, so that one bit (BOOLFN2:A0738 in Figure 2-11) can be implemented by replaying the plan used for the other (BOOLFN1:A0734) -- a small example of "internal analogy" within one design. Also, the number of refinement steps does not include the many additional types of decisions listed in Section 2.5.1.2 that are made in the course of producing a design.

Results: Even for designing this simple circuit from scratch, replay proved slightly more useful than copying. As the table shows, it required only 61 total mouse clicks, instead of 70, because 2 of the 12 steps were replayed. Replay reduced the number of decisions in three categories, though it did increase one category by a single click. Of course this difference is neither substantial nor conclusive, since it is based on a single rather small design.

2.5.2.2. Rename input signals

As a baseline, it makes sense to compare the effort required to reimplement the identical circuit. In this case, the specification may be unchanged, or the input signals may be renamed. Renaming does not affect Copy-Module, since the user is responsible for deciding whether the old circuit fits the new specification. It does not affect BOGART either, because the correspondence heuristic relies on positional information rather than on the names of the input signals. However, a replay mechanism that relied on such information would have trouble reimplementing a specification with renamed inputs.

Results: As before, the circuit takes 12 steps to refine, but this time with much less user effort, since all of them are replayed. However, two of the replayed steps invoke the Refinement-Get-Signal rule, which requires the user to specify the signal source. These two many-option decisions are not required by Copy-Module. On the other hand, Copy-Module requires the user to specify the correspondence between the ports of the copied circuit and those of the new

module. Overall, copying and replay come out about even in this case. However, it is important to point out that this comparison omits the effort of deciding whether the old circuit is an appropriate implementation of the new module. In the case of Copy-Module, the user is entirely responsible for this decision, while in the case of replay, BOGART checks that each refinement step is still applicable. This difference could make a substantial difference in the actual relative effort required.

2.5.2.3. Change functionality to compute NOT instead of OR

To test the effort required to implement a change in specified functionality, we perturbed the specification so that when the INSTR input is 3, the ALU computes the value of (NOT DATA-1) instead of (DATA-1 OR DATA-2). The resulting design, shown in Figure 2-13, requires a total of 10 refinement steps.

Results: Replacing one of the ALU operations is nearly twice as easy (33 clicks) with replay as implementing the same change using copy (58 clicks), or as implementing the original circuit from scratch (61 clicks). The number of all but the easiest type of decision is roughly halved, because 7 of the 10 design steps involved are done by replay (although two of them invoke Refinement-Get-Signal, which requires some many-option decisions).

2.5.2.4. Introduce memory

To test the effect of further perturbation, we replaced one of the inputs with an accumulator, still computing (NOT DATA-1) when INSTR is 3. In VEXED's specification language, changing the second input signal to the stored value of the first input is expressed by replacing "DATA-2" with "previous value of DATA-1." This specification takes 14 refinement steps to implement, since implementing the memory to store the value of DATA-1 takes 4 additional steps. The resulting circuit is shown in Figure 2-14.

Results: The combined specification changes cancelled the advantage of replay relative to copying. Either way, it took about half as many decisions as implementing the original circuit from scratch. While replay took care of 10 of the 14 steps needed here (counting two Refinement-Get-Signal steps), copying did just as well. This is somewhat surprising, since we expected the advantage of replay over copying to increase with the magnitude of the specification change. We attribute the absence of such an advantage to the user effort required for replaying the Refinement-Get-Signal steps.

2.5.2.5. Increase data width

To test BOGART's ability to implement specification changes in the size of iterated structures, we changed the data width of the original ALU specification from 2 bits to 4 bits. In VEXED, this change is expressed by changing only the ENCODING feature of the output value specification:

```
(ENCODING
  NIL
  (I (ALL I))
  (EQUAL (ENCODING OUT I)
         (INTEGER (WIRES 4)
                  (BITS 4)
                  (FIRST-BIT LSB)
                  (BIT-ENCODING (BIT (0 LOW) (1 HIGH)))))
  NIL)
```

The resulting larger circuit, shown in Figure 2-15, incorporates 20 refinement steps instead of 12.

Results: Using BOGART to implement the increase in data width took only about half the effort (35 clicks) as implementing the original specification, and barely a quarter of the effort (132 clicks) needed to increase the data width using Copy-Module, because all 20 steps were performed by replay, though 4 of them invoked the Refinement-Get-Signal rule. The resulting circuit is nearly twice as large as the original 2-bit version; it uses 144 transistors, and takes about 45 modules to represent in VEXED.

2.5.3. Analysis of Results

In each case we tested, using replay to implement a specification change was significantly easier than implementing the original specification from scratch. On average, the number of every type of decision halved; in no case did it increase.

This observation did not hold for copying. Replacing OR with NOT using the copy command took about as much effort (58 clicks) as the original implementation, replacing an input with a memory took about half as much (35 clicks), and doubling the data width took about twice as much (132 clicks). Consequently, using replay required about half the effort as copying, ranging anywhere from a quarter as many total decisions to about the same number.

The power of replay appears to stem from at least two factors. First, replaying a step is more robust than copying a module, in the sense that the old rule may still be appropriate for implementing the new module, even if the circuit it created for the old module is not. If we think of a rule as a function F from a specification s to a (partial) implementation $F(s)$, we see that when an old implementation $F(s)$ is inappropriate for a new specification s', replay may be able to create an acceptable new implementation $F(s') \neq F(s)$.

Second, it is sometimes useful to replay the same portion of a design plan more than once in a single design. This effect is clearest in the datawidth variation experiment, where all 20 design steps were performed by using BOGART to replay a 12-step plan. When BOGART replays a rule that contains a loop for creating a sequence of n submodules, it replays the same subplan for each new submodule. If n is greater in the new problem than in the old one, this subplan will be replayed more than once. While such iterated structures are the clearest case of repeated replay, even the construction of the initial design illustrates the point that a design subplan can be useful more than once in the same design: although the design plan for the initial ALU has 12 steps, replay performs two of them by repeating steps used earlier in the plan. Of course, copying can also reuse the same part of a design more than once. But the opportunity for repeated reuse is magnified by the ability of replay to reuse the design plan of one module to help implement another even when their specifications differ.

Figure 2-17 indicates the few instances where using BOGART required more decisions of the same type than using Copy-Module. In every case where the number of decisions stayed the same or increased, it was traceable to one or both of the following causes:

- *BOGART's inability to replay Get-Signal steps completely automatically.* Since Get-Signal does not fit VEXED's top-down design paradigm, BOGART relies on the user to choose which port to steal the signal from. This type of decision was the most difficult one in our experiments, since it requires choosing from among all the ports in the circuit. Its effect was most pronounced in the case where DATA-2 was eliminated and the ALU was modified to use the previous value of DATA-1 instead. This specification change was implemented by storing DATA-1 in an accumulator and using Get-Signal to share its value with all three operations.
- *Inconsequential differences in the user interface.* Both copying and replaying require the user to decide what portion of the old design to reuse. This decision is of similar difficulty whether the user is copying or replaying. When copying, the user selects a module with a single mouse-click from a menu of the names of all modules in VEXED's knowledge base, from previous designs as well as the current one. But the Replay command first asks the user to specify the circuit name, displays the selected circuit, and then asks the user to select a module within it. Our experimental user (Weinrich) found that the latter method actually made the decision easier, since it was easier to remember which circuit the desired module was in than to recognize its name in a long list. However, this difference was reflected in our summary as requiring an extra decision to choose which node in the design plan to start replaying from. This

discrepancy illustrates the fact that our estimate of decision difficulty is only approximate. A more precise model of user effort would need to take into account not only the number of options for a decision, but the kind of thought required to understand them and select among them.

In summary, the results of our experiment support two claims. First, using replay can be much easier than reimplementing from scratch. Second, and less obviously, replay can be much more powerful than copying.

In particular, copying an old module is most helpful in implementing a new module whose specification is identical (except possibly for renamed signals), especially when the module's implementation exploits signal-sharing between different parts. Most of the user effort then consists of connecting the proper signals to the ports of the copied module.

In contrast, replay is most helpful when the specification of the new module is similar but not identical to that of an old module. The user effort then consists of performing steps that are not replayed automatically, namely invocations of Get-Signal, and performing the additional steps needed to complete the design.

2.6. DISCUSSION

The ability of a replay mechanism to reduce the burden on the designer depends on how well it addresses the five issues introduced in Section 1. We now examine how BOGART addresses each of these issues, and the degree to which its solution is facilitated by VEXED and its underlying model of design as top-down refinement plus constraint propagation.

2.6.1. Design Acquisition

VEXED's underlying model of design leaves the user two kinds of decisions: which module to refine next, and which refinement rule to use. VEXED automatically captures both kinds of decisions in the design plan it records. Each step in the plan includes the rule used to refine a module, and VEXED records the order in which the steps were performed.

Recording a design plan is of little use unless it can be replayed in a reasonable range of situations. VEXED meets this requirement by recording decisions in terms of general rules that can easily be replayed in a new context, rather than

low-level operations like "connect F:A0025$_{out}$ to G:A0029$_{in}$" or "draw a wire from point <489,563> to point <723,894>." Such low-level operations become meaningless or inappropriate in a new context where modules named F:A0025 and G:A0029 do not exist, or where there are no module ports at <489,563> and <723,894>. Thus capturing the design plan at an *appropriate level of description* makes it replayable in more situations.

To replay refinement steps performed by hand, we use LEAP [5], VEXED's "learning apprentice," to handle the problem of re-expressing manual steps at the level of general rules. Although LEAP works, the rules it synthesizes are not expressed in VEXED's rule language, which was designed to facilitate writing rules by hand, but in a simpler language designed to facilitate learning rules automatically. It was therefore necessary to implement a LEAP-to-VEXED translator. When the user refines a module by hand, LEAP generalizes the refinement step into a new rule, and the translator converts it into VEXED's rule language. The manual step is recorded just as if it had been performed using the new rule; consequently, it can be replayed just like any other rule. However, there are two caveats to this account.

First, the manual step is actually recorded by retracting it and then applying the new rule. This procedure serves to confirm that the new rule has the same effect as the manual step, but the real reason for it is simply that LEAP is not fully integrated with VEXED and BOGART, and cannot record the step itself.

Second, we discovered to our surprise that the new rule does not always fully reproduce the refinement step from which it was learned. The reason is that in order to learn as general a rule as possible, LEAP drops any details that appear specific to the example, i.e., are not needed to prove why the refinement step was correct. These details represent commitments like labelling a module as a primitive component, or choosing an arbitrary data encoding. Since such details are needed to create a complete design, we implemented some "residual" rules that can be used to fill them in. This issue did not arise during BOGART's trial-by-students or our comparison experiment, because LEAP is not widely used, partly because manual refinements require specifying the submodules in a rather unwieldy language.

While the extra rules enable the user to fill in the details omitted by replay, it appears conceptually straightforward to fill in these details automatically by extending LEAP to produce a design plan instead of a single rule. This plan would combine the general rule with the residual rules needed to fill in the details. It would then be spliced into the overall design plan so it could be replayed. In cases where some of the original details (e.g. data encoding) were incompatible with the new design, only the applicable portion of the learned plan would be replayed, and any inappropriate residual rules would be skipped. However, this extension has not been pursued, both because it appears less scientifically important than other extensions to LEAP, and because its absence has not been a practical limitation in using BOGART.

2.6.2. Design Retrieval

BOGART relies on the user to decide when replay is relevant and to retrieve a suitable design plan, although VEXED's circuit-browsing facilities do provide some help in finding the relevant part of a given design to replay.

The difficulty of the design retrieval problem varies with how replay is used. For design iteration, the retrieval problem goes away -- the current design plan is the relevant one. For design by analogy, finding a suitable design to retrieve from a repository of previous designs requires knowing where to look. How can designers avoid a time-consuming search through such a repository when they've never seen the relevant entry or can't remember where to find it? If we developed a larger database of design plans, we would expect the process of finding relevant ones to become a bottleneck in using BOGART.

Precedent-finding is an active area of investigation in AI research on analogy [4]. It is not clear how to automate the retrieval of the designs -- or *parts* of designs -- most relevant to refining a given module. We expect a good index to help a great deal, just as it does in a databook. VEXED's taxonomy of module types (memory, computational, primitive, etc.) suggests the beginnings of a helpful indexing scheme. A database query mechanism that retrieves all modules with a given set of features could help the user browse for relevant designs -- assuming the user can articulate the relevant features and the system can efficiently find the modules with those features.

2.6.3. Flexible Reuse

Rarely can a previous design be used in its entirety. BOGART makes reuse more flexible in several ways:

- BOGART reuses the design *process* (i.e., a design plan), not the *product* of design (i.e., a specific circuit schematic). Design plans are more general than their resulting designs: the same design plan can produce different circuits when applied to different specifications.
- BOGART allows *partial reuse* of a design plan. Instead of replaying the top-level design plan for a circuit, the designer can choose to replay the subplan associated with any module.
- BOGART allows reuse of an *abstracted* design plan. The designer can reuse a design plan without replaying it all the way down, and choose other means to fill out the details. The abstracted plan may apply to a wider class of problems. Abstracted plans are also used in the ARGO system (Chapter 3).
- BOGART represents design plans as *trees*, rather than linear sequences, exploiting the hierarchical design structure imposed by the top-down refinement paradigm. Since different branches can be replayed to different depths, an additional degree of flexibility is achieved, in that the user can select any subtree to replay.

The number of replayable subplans is one way to measure the flexibility of partial reuse. While an n-step linear plan has only $n(n+1)/2$ contiguous, non-empty subsequences, a VEXED design plan consisting of n steps has up to $2^{n-1}+n-1$ non-empty subtrees, depending on its shape. Complete flexibility would allow any subset of the steps in a design plan to be replayed. An n-step plan has 2^n-1 non-empty subsets. Of course, not every subset makes sense to replay: skipping a step that creates a module makes it difficult to replay a step that refines the module!

BOGART can already replay disjoint subtrees of a design plan, but the user must invoke BOGART separately on each one, and indicate which module to apply it to. Giving BOGART a list of subtrees would let it replay (or try to replay) an arbitrary subset of the steps in a design plan. Specifying the correspondence between modules to be refined and subtrees to be replayed would be tricky in cases where a module refined by one subtree must first be created by replaying another, and hence is non-existent when the list of subtrees is given.

In principle, it might appear that the flexibility of being able to replay any of a combinatorially large number of subtrees of a design plan would impose a very difficult decision on the user who had to choose among them. However, in practice we have found that the user chooses the root of the subplan to replay, but accepts the default of replaying as much of it as possible. This choice may simply reflect the difficulty of deciding *a priori* which low-level design steps to omit, but it is justified if all replayable steps are actually appropriate to replay.

2.6.4. Appropriateness of Reuse

When is it appropriate to replay a design step? We distinguish different degrees of appropriateness [8]:

1. The design step is syntactically *executable*. For example, "connect F:A0025$_{out}$ to G:A0029$_{in}$" is executable if there exist ports named F:A0025$_{out}$ and G:A0029$_{in}$.

2. Replaying the step produces a *correct* result. For example, applying the OR-DECOMP rule correctly refines any module that matches its "if" part.

3. Replaying the step leads to a *good* design. Whether a design should be considered good is a function of the designer's goals, and depends on such factors as layout area, power requirements, and even the time available to do the design (see [7] and Chapter 9).

BOGART relies on VEXED to guarantee the correctness of replayed steps. A VEXED rule is written to preserve correctness when applied to any module that satisfies its "if" part. As we saw earlier, BOGART retests rules when replaying a design plan, and refrains from replaying steps that are no longer correct in the new context. VEXED's constraint propagator, CRITTER (Volume I, Chapter 8), provides an additional check on correctness by detecting constraint violations.

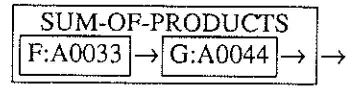
BOGART cannot tell whether replaying a step will lead to a good design, because without knowing the designer's *goals* [7], it has no idea what "good" is. The only goals represented in VEXED's model of the design process are of the form "refine this module." VEXED does not capture the designer's rationale for a design plan in terms of the goals it is intended to achieve, relative to speed, area, power, heat, cost, yield, etc. When replaying the plan, BOGART does not even know whether these goals are still in effect. Even if BOGART were given this knowledge, it would be difficult to ensure that replay would lead to a good design. Predicting the eventual consequences of a design decision is a hard task to automate, although some progress has been made recently (this volume, Chapter 9).

2.6.5. Correspondence Problem

Section 2.3.3 described BOGART's heuristic for finding the correspondence between modules in the old and new designs, and showed how it can fail. This heuristic constrains the correspondence problem in two ways. First, based on the top-down refinement model of design, it assumes that the submodules of two corresponding modules should be put in correspondence with each other, not with modules elsewhere in the hierarchy. Second, it assumes correspondence between modules bound to the same rule variable. In practice, this heuristic seems to work most of the time, but it fails often enough to be worth improving

on. As we saw, it can be fooled merely by transposing the order of clauses in a module specification.

The purpose of deciding which module in the old design corresponds to a module in the new design is to find a suitable subplan for refining the new module. This decision should be sensitive to how far the subplan can be replayed; the "bound-to-same-variable" heuristic ignores this factor.



F:A0033 outputs input-1 AND input-2

G:A0044 outputs G:A0044$_{in}$ OR (input-3 AND input-4)

Figure 2-18: Result of Applying OR-DECOMP to SUM-OF-PRODUCTS

In fact, we can view the correspondence problem as a local version of the retrieval problem: given a module to refine, find a suitable subplan to replay. The correspondence need not be one-to-one. For example, consider Figure 2-18, which shows the result of applying the first step of the COMPARATOR-CELL design plan to the specification for a boolean sum-of-products. BOGART's correspondence heuristic identifies F:A0033 with F:A0025 and G:A0044 with G:A0029. However, F:A0033 is less similar to F:A0025 than it is to G:A0029's submodule F:A0060, whose specified output value is the same as F:A0033's. It would make more sense to refine F:A0033 by replaying the subplan for F:A0060 (see Figure 2-5), even though the same subplan would also be used to refine one of G:A0044's submodules, and the subplan for F:A0025 would not be replayed at all. While this broader approach to correspondence-finding may exploit replay more fully in solving a given design problem, it may be very expensive to consider every subplan in a design plan as a candidate for refining each module.

2.7. APPLICATIONS TO OTHER DOMAINS

Since the general idea of replaying design decisions does not appear specific to circuit design, it makes sense to ask whether BOGART's approach to replay could be applied to other design domains as well. So far we have tried two such domains: mechanical design and algorithm design. The answer seems to be a qualified "yes."

2.7.1. Application of BOGART to Mechanical Design

The mechanical design system we investigated was MEET (Chapter 8, Volume I), which designs rotation transmitters composed of gears, shafts, and belts. MEET is based on the same top-down refinement model of design as VEXED. In fact, MEET is implemented on top of EVEXED, the domain-independent kernel of VEXED. Thus MEET uses the same module-like representation language and constraint propagator as VEXED. However, instead of circuit features like data value, its representation uses mechanical features like rotational speed. Similarly, in place of VEXED's knowledge base of circuit decomposition rules, MEET uses a knowledge base of decomposition rules for mechanical design. Thus it was both enticing and convenient to find out whether the BOGART approach would work for MEET. In fact, since MEET was based on the same EVEXED kernel, would the BOGART code itself work, or did it somehow implicitly depend on properties of the circuit design domain?

It turned out that BOGART was indeed able to replay design plans in MEET. Nonetheless, this capability was of little use. The reason is that VEXED's interactive top-down refinement model of design, on which BOGART is based, covers only a small part of mechanical design. For example, it is not appropriate for parametric design tasks, like designing individual gears. MEET was therefore augmented with other programs to perform parts of the design process where this model broke down. For example, DPMED solves parameter design problems by hillclimbing [14]. DPMED selects parameter values for primitive components (like gears) in designs constructed by MEET. Similarly, SPIKE helps design geartrains by finding a sequence of standard gear ratios to achieve a desired rotational speedup or reduction [13]. While BOGART can replay a MEET design plan, including calls to DPMED, this capability is simply irrelevant to the bulk of the design problem.

In fact, one of the main lessons of MEET is the inefficiency of using a generic expert "shell" like EVEXED to interpret a domain-specific knowledge base. While this approach made it possible to implement MEET quickly, the very

generality of the shell precludes taking full advantage of domain-specific representations and control strategies. For example, SPIKE designs geartrains much faster than MEET in part because instead of using a general-purpose module-based representation, it represents geartrains as simple sequences of gear ratios, and instead of using a general-purpose constraint propagator, it simply multiplies the ratios.

2.7.2. Extending BOGART to Heuristic Algorithm Design

The efficiency of specialized synthesis algorithms like SPIKE's motivated the creation of the DIOGENES project, whose goal is to help automate the development of heuristic search algorithms like SPIKE from explicit representations of domain knowledge, thereby combining the efficiency of the "specialized algorithm" approach with the generality of the "generic shell" approach [11]. To convert a problem class specification, expressed as a straightforward but inefficient generate-and-test algorithm, into an efficient heuristic algorithm for solving instances of that class, DIOGENES applies a sequence of speedup transformations selected by the user from a catalog of general rules. The problem of reimplementing the algorithm to incorporate a change in the specification is therefore analogous to reimplementing a circuit in VEXED when its specification is changed. This similarity raised the natural question of whether BOGART's replay techniques could be extended to work in DIOGENES.

While BOGART is based on a model of design by top-down refinement, DIOGENES is based on a less constrained transformational model. The key difference has to do with the form of rules in VEXED and DIOGENES for transforming specifications into circuits and algorithms, respectively. VEXED is based on the assumption that a rule refines a single unimplemented module into one or more interconnected submodules. BOGART exploits this assumption in its tree-structured representation of design plans, and in its correspondence heuristic. Each node in the tree corresponds to a module in the circuit. Each design step expands a leaf of the tree. While a few rules (e.g., Refinement-Get-Signal) violate this assumption, BOGART is unable to replay them, at least fully automatically.

In contrast to VEXED, DIOGENES allows transformation rules that rearrange previously created structure. DIOGENES represents search algorithms as parse trees in an object-oriented language of algorithm components like generators and tests [10]. However, instead of expanding a leaf of this tree, a transformation in DIOGENES can replace or rearrange any subtree. Therefore, some of BOGART's solutions to the replay issues discussed here did not work for DIOGENES.

Fortunately, we were able to implement a replay mechanism for DIOGENES by extending the approach used in BOGART. This mechanism, called XANA (Greek for "again"), is reported in [12]. While a detailed description of XANA is outside the scope of this article, it is worth describing in brief how XANA addresses each of the issues raised here, in order to compare it with BOGART.

Design acquisition: Like BOGART, XANA represents a design step as an application of a transformation rule chosen from a catalog. For each step, XANA records the name of the rule, the subtree to which it was applied, and the new subtree that replaced it. This subtree may contain newly created nodes as well as copies of nodes in the old subtree. A design plan in DIOGENES encodes the dependency structure among steps. Step 2 depends on step 1 if step 1 creates a node accessed by the rule applied in step 2. The directed graph of dependencies is acylic but not necessarily a tree, and is not isomorphic to the tree representation of the algorithm itself. Thus DIOGENES lacks VEXED's one-to-one mapping between modules and design steps.

Design retrieval: Unlike BOGART, which can apply a previous design plan to a completely new module, XANA assumes that it is being applied to a modified version of the previous initial specification. That is, XANA assumes it is being used for design iteration, rather than to solve a completely novel problem by analogy. Removing this limitation would require providing some means to identify the correspondence between the old and new specifications.

Flexible reuse: XANA can replay any coherent subset of recorded steps, where "coherent" means that if step 2 depends on step 1, step 1 must be replayed before step 2. However, it is possible to replay step 1 without step 2. Thus XANA can replay truncated versions of the design plan, but (unlike BOGART) must replay starting from one or more (though not necessarily all) of the roots in the dependency graph. In practice, we try to replay as many steps as possible.

Appropriateness: Like BOGART, XANA's appropriateness criterion for replaying a design step is whether the preconditions of the transformation rule are still satisfied. However, XANA imposes an additional requirement: it will not replay a step unless it can find new objects corresponding to all of the old objects on which the step depends.

Correspondence: Extending BOGART's "bound-to-same-variable" correspondence heuristic to work in XANA was probably the trickiest aspect of developing XANA. BOGART uses a rule variable name to identify a submodule -- the immediate offspring of a node in the tree of modules that represents the circuit. XANA extends this idea by using an access path to identify an algorithm component -- a descendant of a node in the tree of objects that represents the algorithm. The access path consists of the sequence of labels on the arcs leading from the node to its descendant. XANA's correspondence heuristic is defined recursively, as follows. If the old object was part of the initial specification, the corresponding new object is found by starting at the root that represents the new specification, and traversing the same access path. If the old object was created by a design step, the corresponding new step is identified,

and the corresponding new object is found by following the same access path starting from the root of the subtree created by the step.

A clearer description of XANA involves the details of how it represents design plans; see [12]. The main lesson relevant here is that the approach used in BOGART was successfully extended to replay a much broader class of transformation rules. While XANA is still at an early stage, a preliminary experiment similar to the one described in Section 2.5 showed that XANA replayed most of the steps that DIOGENES' user would otherwise have had to repeat. Thus it appears that XANA will be more useful in DIOGENES than BOGART was in MEET.

2.8. RELATED WORK

Few design-replay systems have been implemented. *Derivational analogy* [2] was proposed as a general method for solving problems by patching and replaying the solution plans for similar previous problems. The POPART system [20] replays program derivations, but records the design plan at the level of structure-editing, relies on the user to solve the correspondence problem by hand-crafting descriptions of the program objects manipulated by each step in the plan, and tests appropriateness only to the extent of checking syntactic executability. The REDESIGN system [16] used a design plan built by hand, but embodied many of the ideas later used in VEXED and BOGART, and included some structure-patching heuristics for modifying a design to fit a modified specification or repair a constraint violation. The ARGO system for refining VHDL behavioral specifications into digital circuits (Chapter 3) represented design plans as partially ordered sets of rules, allowed design plans to be abstracted by omitting lower-level steps, and was used to synthesize circuits containing up to a few hundred transistors. It was later extended to handle more automatically several of the issues discussed above (Chapter 3). For a detailed comparative analysis of these and other systems, see [9].

2.9. CONCLUSION

What conclusions can we draw from our experience with BOGART? First, although BOGART is an experimental system, it works well enough to be used by people other than its creator. Students were able to create complete, correct (though suboptimal) small designs in a few hours the first time they used VEXED and BOGART, even though they were unfamiliar with VEXED's specification language and inexperienced in VLSI design.

Despite the limitations of both systems, students found BOGART useful; in fact, they used BOGART to compensate for VEXED's lack of facilities for structure-copying and dependency-directed backtracking. We found that a replay facility can be useful even when simply using it to apply a set of rules that was already applied elsewhere in the circuit. When the designer is allowed to save and name design plans at different points during the design process, the replay facility can help salvage portions of a design that are still applicable after a change in the specification or the implementation strategy.

Although the lack of a repository of previous designs, the limited time of the experiment, and the simple nature of their designs left little scope for design-by-analogy or systematic design exploration, students used BOGART to support a semi-automatic design iteration cycle based on backtracking, patching the specification or revising a design decision, and replaying.

To compare the usefulness of copying and replay, we counted the number of user decisions required to design a small ALU and to implement several specification changes, ranging from renaming the inputs to modifying the functionality or the data width. Replay outperformed copying in all but one case, by factors ranging to almost 4.

To test the generality of BOGART's approach, we applied it to two other design domains. The approach -- in fact, the code itself -- carried over to a mechanical design domain, but failed to help most of the design process because it didn't fit BOGART's underlying model of design as top-down refinement. We had more success with an algorithm design domain, where we were able to extend BOGART's methods to handle a less constrained transformational model of design, so as to replay a much broader class of design plans.

Our experience to date supports the hypothesis that even a simple replay mechanism can automate many of the repetitious aspects of design, freeing designers to concentrate more on the design problem itself. However, the development of practical replay tools for realistic applications will require additional work on the issues of design acquisition, retrieval, flexibility, appropriateness, and correspondence.

2.10. ACKNOWLEDGMENTS

This work was supported by the National Science Foundation under Grant Number DMC-8610507, by the Rutgers Center for Computer Aids to Industrial Productivity, and by the Defense Advanced Research Projects Agency under Contract Numbers N00014-81-K-0394 and N00014-85-K-0116. The opinions expressed here are those of the authors and do not reflect any policies either expressed or implied of any granting agency.

This chapter is a slightly revised version of a paper that originally appeared in *International Journal for Artificial Intelligence and Engineering*, Volume 4, Number 4, October 1989, pages 181-196, published by Computational Mechanics Publications. A shorter, earlier version appeared in W. E. Eder (Editor), *Proceedings of the 1987 International Conference on Engineering Design (ICED87)*, Boston, MA, August 1987, Volume 2, pages 632-647, published by the American Society of Mechanical Engineers.

We thank Lou Steinberg for patiently explaining VEXED and REDESIGN and helping supervise the development of BOGART; Chris Tong for many illuminating conversations on design; Greg Fisher and Rich Cooperman for their work on XANA and DIOGENES; Rich Cooperman for helping prepare several figures; Chun Liew and Pat Riddle for their comments on several earlier drafts; and the entire Rutgers AI/Design group for providing the supportive and stimulating environment in which this research was performed. We are also grateful for encouragement and careful reading by Ken McCallum, the Editor of *International Journal for Artificial Intelligence and Engineering*, where this paper first appeared in substantially its present form, and for helpful suggestions from the anonymous reviewers of the journal version.

2.11. REFERENCES

[1] Balzer, R., Green, C., and Cheatham, T., ''Software Technology in the 1990's Using a New Paradigm,'' *IEEE Computer*, Vol. 16, pp. 39-45, November 1983.

[2] Carbonell, J., ''Derivational analogy: a theory of reconstructive problem solving and expertise acquisition,'' in *Machine Learning, Volume II*, R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, Eds., Los Altos, CA: Morgan Kaufman Publishers, Inc., pp. 371-392, 1986.

[3] Dietzen, S. and Scherlis, W., ''Analogy in program development,'' *Second Symposium on the Role of Language in Problem Solving*, North-Holland, April 1986.

[4] Defense Advanced Research Projects Agency (DARPA) Information Science and Technology Office, Morgan Kaufmann, Pensacola, FL, May 1989.

[5] Mitchell, T., Mahadevan, S., and Steinberg, L., ''LEAP: A Learning Apprentice for VLSI Design,'' *Proceedings of the Ninth International Joint Conference on Artificial Intelligence (IJCAI85)*, Los Angeles, CA., pp. 573-580, August 1985.

[6] Mostow, J., ''A decision-based framework for comparing hardware compilers,'' *Journal of Systems and Software*, No. 4, pp. 39-50, 1984, [Reviewed in *ACM Computing Reviews*, November 1984, pages 509-510.].

[7] Mostow, J., ''Toward better models of the design process,'' *AI Magazine*, Vol. 6, No. 1, pp. 44-57, Spring 1985.

[8] Mostow, J., ''Why are design derivations hard to replay?,'' in *Machine Learning: A Guide to Current Research*, T. Mitchell, J. Carbonell, and R. Michalski, Eds., Kluwer Academic Publishers, Hingham, MA, pp. 213-218, 1986, [Revised and condensed version of paper in *Proceedings of the 3rd International Machine Learning Workshop*].

[9] Mostow, J., ''Design by Derivational Analogy: Issues in the Automated Replay of Design Plans,'' *Artificial Intelligence*, Elsevier Science Publishers (North-Holland), Vol. 40, No. 1-3, pp. 119-184, September 1989, [In J. Carbonell (editor), Special Volume on Machine Learning, reprinted by MIT Press as *Machine Learning: Paradigms and Methods*, 1990.].

[10] Mostow, J., ''An object-oriented representation for search algorithms,'' *Proceedings of the Sixth International Workshop on Machine Learning*, Morgan Kaufmann, Cornell University, Ithaca, NY, pp. 489-491, June 1989.

[11] Mostow, J., ''Towards automated development of specialized algorithms for design synthesis: knowledge compilation as an approach to computer-aided design,'' *Research in Engineering Design*, Amherst, MA, Vol. 1, No. 3, pp. 167-186, 1990.

[12] Mostow, J. and Fisher, G., ''Replaying Transformational Derivations of Heuristic Search Algorithms in DIOGENES,'' *Proceedings of the DARPA Workshop on Case-Based Reasoning*, Pensacola, FL, pp. 94-99, May 1989.

[13] Prieditis, A., Steinberg, L., and Langrana, N., *SPIKE: A State-Space Search Design System*, unpublished Technical report, Rutgers University Computer Science Department, Fall 1987, [Rutgers University Technical Report CAIP-TR-051 and Rutgers AI/Design Project Working Paper Number 77.].

[14] Ramachandran, N., Shah, A., and Langrana, N., ''Expert systems approach in design of mechanical components,'' *Journal of Engineering With Computers*, Vol. 4, pp. 185-195, 1988.

[15] Stallman, R., and Sussman, G., ''Forward Reasoning and Dependency-Directed Backtracking in a System for Computer-Aided Circuit Analysis,'' *Artificial Intelligence*, Vol. 9, pp. 135-196, 1977.

[16] Steinberg, L. and Mitchell, T., ''The Redesign system: a knowledge-based approach to VLSI CAD,'' *IEEE Design & Test*, Vol. 2, No. 1, pp. 45-54, February 1985, [An earlier version of this article received the DAC84 Best Paper Award, appeared in the *ACM/IEEE 21st Design Automation Conference Proceedings*, June 1984, and is available as AI/VLSI Project Working Paper No. 11 / Rutgers Computer Science Department Technical Report LCSR-TR-47.].

[17] Tong, C., ''Goal-Directed Planning of the Design Process,'' *Proceedings of the Third IEEE Conference on AI Applications*, February 1987, [Available as Rutgers AI/VLSI Project Working Paper No. 41].

[18] Tong, C., *Knowledge-based circuit design*, unpublished Ph.D. Dissertation, Stanford University Computer Science Department, 1988.

[19] Tong, C. and Franklin, P., ''Tuning a knowledge base of refinement rules to create good circuit designs,'' *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, Detroit, MI, pp. 1439-1445, August 1989.

[20] Wile, D., ''Program developments: formal explanations of implementations,'' *Communications of the Association for Computing Machinery (CACM)*, Vol. 26, No. 11, pp. 902-911, 1983, [Available from USC Information Sciences Institute as RR-82-99.].