February 1981 Report. No. STAN-G-8 I-838

Dynamic Program Building

by

Peter Brown

Department of Computer Science

Stanford University
Stanford, CA 94305

Dynamic Program Building

Peter Brown
Computer Systems Laboratory
Stanford University
Stanford, California 94305

December 1980

Abstract. This report argues that programs are better regarded as dynamic running objects rather than as static textual ones. The concept of ***dynamic*** building, whereby a program is constructed as it runs, is described. The report then describes the *Build* system, which is an implementation of dynamic building for an interactive algebraic programming language. Dynamic building aids the locating of run-time errors, and is especially valuable in environments where programs are relatively short but run-time errors are frequent and/or costly.

This research was supported in part by National Science Foundation grant MCS-77-23738 and by the IBM Corporation.

Introduction.

One of the greatest movements over the past decade has been the much increased interaction between user and computer. To take one example, the non-interactive editor was common a few years back, but is now almost extinct. Users prefer interactive editors, which point out their syntactic errors as they make them, and, much more important, show them the effects of their edits, so that run-time errors (i.e., making unintended changes) are immediately apparent. Surprisingly, however, we still prepare our programs in the same batch-oriented manner. In particular the activity of entering a program into the computer may be separate from the activity of compiling it (though the two are the same for languages such as BASIC), and running a program is normally separate from both of these. This report examines whether there is any merit in preparing programs in a more dynamic way.

Motivation.

The research has been motivated by a number of happenings and phenomena. Among them are the following.

(1) The profile. If we are lucky enough to have a good profiling system (and sensible enough to use it), we are often surprised by the run-time behavior of our programs. We may find that a "working" program spends a lot of time doing unnecessary operations, or that the part of the program we spent hours optimizing only takes one per cent of the execution time. Do we spend too much time thinking of our programs as static pieces of text, nicely block structured, rather than as dynamic objects? To quote from a paper on the Smalltalk language [Ingalls, 1978], which is more oriented to the dynamic approach: "[Programming in Smalltalk] feels like organizing trained animals rather than pushing boxes around."

(2) The silly question. It is valuable experience to teach programming to beginners, as they sometimes ask "silly" questions that make you think about the fundamentals of programming. One recent silly question was "Why do I have to type in my program before I run it?"

(3) The disastrous error. A common story runs like this. "The program contained a minus sign instead of a plus sign, and as a result the rocket crashed. All the current attention to security, strong typing and so on, is useless; it just patches a few holes in a leaky bucket." Though we may (and do) disagree with the conclusion, this anecdote does make the important point that errors must be attacked on a broad front rather than on one single front.

(4) Human interactions. When one human explains a set of instructions to another, it has been found valuable, even vital, for them to execute the instructions together first. Thus, the instructor does not tell the pupil how to prune a rose bush, and then go away and leave him to prune one hundred bushes. Instead instructor and pupil execute the instructions together on one bush, and, if all goes well, the pupil can then do the others.

Existing interactive languages.

There exist many interactive programming languages, such as BASIC, JOSS, APL, LISP, Smalltalk, etc. How well do these cater for dynamic program building? The languages differ in details, but by and large they provide the following facilities.

(1) Most syntax errors are detected as the program is entered and can be corrected immediately.

(2) Statements (or whatever is the appropriate program unit) can be treated in two alternative ways.

   (a) They may be treated as immediate statements; in this case they are executed immediately they are typed in, and then discarded.

   (b) They are treated as program statements; in this case they are not executed immediately but are incorporated as part of the current program (or function or subroutine), and only executed when the program is run (or the function/subroutine is called).

In summary, the typical interactive programming language treats immediate execution and incorporation as mutually exclusive activities. In other interactive systems, such as editors, this dichotomy does not arise, because there is normally no concept of stored program, and everything is executed immediately.

Purpose of this research.

The purpose of this research is to investigate whether there is merit in combining incorporation with execution. We are not interested in providing this as an "optional extra" in a traditional system. Indeed there already exist many systems which do this by allowing programs to be changed or augmented during execution; these systems include some implementations of BASIC as well as special languages such as LC^2, PL/CS and Smalltalk. Instead we wish to be more radical and explore a system where the ***only*** way to build a program is to build it as it is running. We call this a **dynamic** building system. The interest of dynamic building is that two new disciplines are forced on the programmer.

Firstly, he must think of his program as a dynamic object and understand its behavior. This applies to program building, because statements are entered in the order in which they are executed, and also, as will be seen, to program editing.

The second discipline applies to errors. In traditional interactive systems, syntax errors are detected immediately a statement is typed, and the translator prevents such wrong statements from being incorporated into a program. In our dynamic system this mechanism is carried over into run-time errors. Now it cannot, of course, be claimed that we have a system where it is impossible to construct a wrong program, and thus all the problems of the world are solved. Dynamic building does however offer the much weaker, though still valuable, property that each statement in the program must have been executed correctly for at least one set of data values. A statement using an unassigned variable could not, for example, be incorporated into the program. A program may, of course, run "correctly" in the sense that no run-time errors are reported, but may still produce wrong answers because the program does not match the problem that is to be solved--remember the minus for the plus, which caused the rocket crash. However a dynamic building system does provide some help with this mis-match problem. If we assume the current values of variables are displayed while the program is being built and that changes in value are highlighted in some way, then there is at least a reasonable chance that the programmer will notice an unusual value that indicates a logical error in the program. (In one sense this does no more than a system that allows the programmer to single-step through his program, looking at the values of variables. However there is the advantage, in the dynamic building system, that the facility is available when the programmer types a statement and hence when the purpose of that statement is in the front of his mind.)

Terminal devices.

If the program is regarded as a dynamic object, it is desirable to have a terminal suitable for displaying such objects. We shall therefore assume, in this description, that the programmer is using a cursor-addressed screen. It is also an advantage if the screen can be divided into separate windows, so that different kinds of information can be displayed in separate places.

Introduction to dynamic building.

We shall now explain, mainly by example, how dynamic building appears to the programmer. Our sample programs use constructs familiar in algebraic programming languages-we say more about choice of language later.

The program is always in some run-time state; either it is actually executing or it is waiting for the programmer to provide some input. When execution reaches a point where no instructions have been specified-we call this a loose end-the programmer is prompted to supply the necessary statements. (We assume the unit of communication is a statement, but it could be something larger or smaller.) In the display of a program, a loose end could be represented as a question mark. A program may contain several loose ends; when the programmer is prompted to supply a particular loose end the appropriate question mark is highlighted (e.g. by flashing a cursor, or by reversing foreground and background). In this paper we point an arrow at the currently active loose end. The only way the programmer constructs the program is by supplying loose ends.

Whenever the programmer supplies a conditional statement, which means there are two possible execution paths, an extra loose end is added to the program. This applies to explicit conditions, such as an IF or WHILE, or to implied conditions such as a FOR loop. (Conditions such as CASE statements, which represent an N-way branch, would need N extra loose ends.) To take the example of a WHILE statement, if the programmer types

```
        WHILE X > 0 DO
```

then the system turns this into

```
        WHILE X > 0 DO
            ?
        ENDWHILE
        ?
```

If the current value of X exceeds zero the body of the WHILE is entered and the first loose end is highlighted; if X does not exceed zero the body is not entered and the second loose end is highlighted. An IF statement is treated in a similar way.

Let us now go through the construction of a specific program. This program computes

$$S = \sum_{k=1}^{n} k^2$$

It prints the value of S for each n that is divisible by four, and stops after S exceeds 10000. The building process is as follows.

Initially the program is null (i.e., it consists only of a single looose end) so the programmer is prompted for the first statement by the display

```
    →   ?
```

He types

```
        LET S = 0
```

This is executed and the programmer is prompted again. He types

```
        LET K = 1
```

and then, on the next prompt,

```
        WHILE S <= 10000 DO
```

Since the condition is true, the system prompts the programmer for the body of the WHILE by displaying

```
        LET S = 0
        LET K = 1
        WHILE S <= 10000 DO
    →       ?
        ENDWHILE
        ?
```

The programmer then types

```
        LET S = S+K↑2
```

followed by

```
IF MOD(K,4) = 0 THEN
```

This tests if *K* is divisible by 4. Since the condition is false with *K* having the value one, the system does not prompt for the body of the IF. Instead it displays the program as

```
        LET S = 0
        LET K = 1
        WHILE S <= 10000 DO
            LET S = S+K↑2
            I:‘ MOD(K,4)= 0 THEN

            ENDIF
    →       ?
        ENDWHILE
        ?
```

The programmer then continues with the body of the WHILE by typing

```
LET K = K+1
/
```

where we assume the ‘/’ is a terminator to indicate the end of a loose end. The system then goes back to execute the next iteration of the WHILE loop. After three more iterations the IF condition becomes true, so the programmer is prompted for the loose end that follows it. He types

```
PRINT S
```

which, given that it is executed on type-in, prints the value of S then and there. He then types the ‘/’ terminator to indicate the loose end is complete. The system resumes execution with the ENDIF and the statements beyond. It will perform several more iterations of the WHILE loop, printing some more values of S. When the WHILE loop is completed the programmer is prompted to supply the loose end beyond its ENDWHILE, and he can supply the final printing statements.

The above program works correctly the first time. Let us be more realistic and include an error. Assume that the programmer, when specifying the above IF statement, allows his attention to wander to happier things than IF statements, and inadvertently types

```
IF MOD(K,O) = 4 THEN
```

This causes a division by zero when executed, and hence is rejected. (More generally, an error in executing a statement may be caused by reasons outside that statement, e.g. by forgetting to initialize a variable. Therefore a wrong statement might better be held in abeyance rather than being rejected.)

Further points.

Three further points come out of this example.

Firstly, it should now be clear why a display divided into separate windows is desirable for dynamic building, as it is for debugging systems. An ideal is to have (at least) three windows, one containing the program, one containing its I/O with the terminal, and one containing a display of the values of the current variables and the recent program history.

Secondly, of course, we are concerned with program preparation, not production usage. When the end-user runs a program constructed by dynamic building, it should appear just as any other program. It would be a bug in the program if the end-user was asked to specify parts of the program! Indeed every production program should be passed through a simple checker that made sure it contained no loose ends.

Finally, we have displayed the program in a conventional indented layout. It would be more attractive, but much more ambitious to implement, if the program were displayed in a graphic way, such as by a Nassi-Schneiderman diagram.

Choicc of language.

An important aim of the research is to investigate dynamic execution within the ***context of a*** conventional interactive algebraic ***programming language.*** The temptation to build a new language round the concept of dynamic building was strong, but it was resisted. There were two reasons for this: firstly the chances of a new language being used by more than a handful of people are slim; secondly any concept can be made to appear viable by designing a whole language around it.

We have therefore taken the most popular interactive language, BASIC, as the main basis for our work. We have included structured IF and WIIILE statements, as indeed many extended BASICs do. As a detail, we have not included BASIC's requirement that all statements be numbered; since wc are assuming a cursor addressed screen, there is no need to use such line-numbers for editing. These line-numbers could, of course, be included if it was desired to match standard BASIC exactly. The example shown earlier corresponds closely to the actual language implemented.

In principle, most programming languages arc suitable for dynamic building techniques. The one significant constraint is declaration before use, as is required by conventional one-pass compilers.

Implementation.

A pilot implementation of the system has been undertaken on the Xerox Alto computers. It is called ***Build.*** The order of priority for implementation has been:

(1) developing a dynamic building system integrated with an editing system;

(2) designing a good user interface with separate windows on the screen, good error messages, etc.;

(3) implementing a rich subset of BASIC;

(4) making programs run fast.

Currently Build works and is usable. Area (1) above is complete, but the three lower priorities have received almost no at tention. The user interface and the employment of the screen is poor, and should be the next area of attention. A very small subset of BASIC has been implemented. For example, only integer data is supported, variables must consist of a single letter, and expressions must contain at most one operator. There are no subroutines, no functions and no declarations. There arc, however, reasonable control structures, since these are central to the research.

Build is implemented as a pure interpreter, i.e., it interprets the source program in the form in which it was typed. Conventional pure interpreters, as found on the cheaper home computers, suffer from the problem that syntax errors are not found until run-time. However in Build, where run-time is the same as type-in-time, there is no such problem.

Build is written in BCPL, as this is the most portable language available on the Alto. Little advantage has been taken of the graphics facilities of the Alto. Thus it should be easy to adapt Build to run on an ordinary cursor-addressed screen; however, the current implementation does use the mouse to position the cursor, and this would need changing.

Block structuring.

The first observation to be made about dynamic building is that, because the program is always running, it must always be properly block structured. Build currently supports two structuring constructs, the IF and WHILE blocks shown in our earlier example. We call the IF and WHILE lines LBs (for Left Bracket) and the terminating END lines RBs (for Right Bracket). At all times each LB must bc matched by an RB. Thus, when the programmer types an LB, the system automatically appends an extra loose end followed by an RB, as we have described already. The programmer never types an explicit RB. Statements within IF and WIIILE blocks are automatically indented so that the structure is clear.

The requirement for correct block structuring has an effect on editing. (We shall discuss editing in detail later, and shall concentrate here only on structuring.) If an LB or RB is deleted when the program is edited, the corresponding RB or LB is automatically deleted too. The statements in between remain intact-they just lose one position of indentation when the program is redisplayed; if, however, there is only a loose end between the LB and the RB this is deleted too.

The programmer may well want to edit a new LB/RB pair into an existing program. (A special case is a replacement of, say, an LB. This is treated as a deletion, performed as described above, followed by an insertion.) To cater for this, Build interrogates the user whenever he inserts an LB. The user is told to point where the RB is to be inserted.

Suspended mode.

A run is in ***suspended mode*** if it has completed, or if there has been an error or break-in. The ***program counter*** points at the next statement to be executed-the program counter is reset to the start of the program if a run was completed-and the user is asked what to do next. Suspended mode is just like command mode in those conventional systems that have a special command which resumes execution where it left off. The main options available in suspended mode are to resume execution, to edit the program, to save or restore the prograrn from the filing system (though this is not currently implemented in Build) or to terminate the session altogether.

Editing.

Dynamic building has an effect on editing. We shall claim it is a beneficial effect. Given that new statements are only entered into the program when they are about to be executed, the normal concept of editing becomes unsuitable and a more rigid discipline needs to be adopted. The user cannot point at an arbitrary point in his program and insert a new statement there. Instead what happens in Build is this.

(1) Build keeps, for each statement, a count of the number of times that statement has been executed.

(2) There is an overriding principle that if a statement that has been executed on the current run is changed (either by deleting the statement or inserting something immediately before it), then the program must restart from scratch. (Hence at the first edit the program will always restart since the very way programs are constructed ensures that each statement has been executed at least once. Subsequent edits that are made after a program has been re-run may not necessitate a restart, since the rc-run may have left some statements unexecuted.) An edit to a declaration should always cause a restart, though currently Build only has default declarations, so the case does not arise. When the program is restarted, the programmer is given a suitable message and all the execution counts are set back to zero.

(3) When the programmer wants to perform an edit, he points at the statement to be changed. Let us assume first that a deletion is required. Build deletes the statement and, if the statement has been executed, resets the program counter back to the start. The program remains in suspended mode in case the user wants to do some more edits.

(4) An insertion causes a more unconventional action. The programmer points where he wants the insertion to be, and Build then inserts an edit loose ***end*** into the program at that point. The program resumes execution (at the start if the insertion was before a previously executed statement). When the program reaches the edit loose end it prompts the programmer to supply the statement to be inserted. The programmer then types in his new statements, which are, as is always the case, executed as they are typed in. After the last new statement the programmer types a terminator. Build then goes into suspended mode so that the programmer can specify where any further edits are to be.

(5) A replacement is treated as a deletion followed by an insertion.

Example of editing.

The following is a very simple example of editing.

Assume that part of the program reads

```
         LET K = 3
  1 →  IF W = 0 THEN
  2 →       PRINT "FIRST TIME"
         ENDIF
         LET N = P/(6+Y)
  3 → WHILE N < 9 DO
```

(where the numbered arrows are for the purposes of our explanation). If the programmer points at arrow 1, and performs a delete, then the IF and ENDIF disappear but the PRINT lives on. Pointing at arrow 2 and deleting causes the PRINT statement to go. If an insertion is required at arrow 3, the program runs until it reaches that point. The following is then displayed

```
         LET K = 3
         LET N = P/(6+Y)
  →      ?
         WHILE N < 9 DO
```

The programmer supplies the statements to be inserted, and, if correct, they are duly incorporated into the program.

Commentary on editing.

The first point to make about editing is that, although the internal machinations are unconventional, it will often appear to the programmer like normal editing, at least in the case of programs that run fast. If the program is a slow one there will be a delay before the programmer is prompted for his new statements. The program may also produce some output before it reaches the point of insertion. (We shall discuss input later.) It may be that the program will not, on the current run, reach the point of insertion (e.g. it is within an IF block that is not executed). If so, the edit loose end may only be encountered much later, perhaps after other edits have been done. We think this is an advantage rather than a defect of the system. The whole aim is to focus attention on the run-time behavior of the program. If the programmer makes an edit that shows he has misunderstood the run-time behavior, it is good that the system behaves in a way that indicates something is wrong-the chances are that the edit is wrong. The reader may justifiably say, at this point: "What if the program uses a random-number generator, so that the program's behavior cannot be predicted?" This is a valid point, and we admit our system, based as it is on run-time behavior, is unsuitable for programs with random run-time behavior.

If the programmer wishes to make several simultaneous edits, it is desirable that he makes them in the correct execution order to avoid successive restarts. It may even be vital as in the example

<u>Before</u>	<u>After</u>
	LET D = 4
LET K = K+4	LET K = K+D

We assume that D is a newly introduced variable. If the programmer replaces the LET K statement before he gives a value to ***D*** he will get a run-time error, because ***D*** will be an unassigned variable. Again we would claim that this behavior is an advantage of the system.

A final point about editing is that it can, of course, turn a correct program into an incorrect one. For example, Build prevents statements containing unassigned variables from being incorporated into a program, but it does not prevent the programmer deleting al1 his initializing code and thus making existing statements wrong.

Rewinding interactive input.

All interactive input is remembered. If a program is automatically restarted because of an edit, its input is "rewound" and the rerun of the program will use the same values. This is a minor convenience which helps provide a controlled environment for testing. The programmer can, of course, demand a true restart, without rewinding, if he wishes to supply new input values.

Context-dependent errors.

We have discussed syntax errors and run-time errors, but there is an intermediate error called a *context-dependent error*--it is sometimes called a semantic error. An example of a context-dependent error is a missing declaration, or, in an interactive language that is analyzed a line at a time, an error in structuring such as a BEGIN not matched by an END. We have already seen that Build eliminates the latter kind of error by automatically supplying an RB for each LB. The missing declaration can be solved by prompting the programmer for the required declaration.

Further language points.

In this section we discuss a few specialized language issues.

Firstly, it was decided to provide traditional immediate statements in Build, i.e., statements that are executed but not incorporated. Such statements are useful for testing.

Secondly, if the source language supports an IF-THEN-ELSE construct, it is useful for the system to be warned, at the IF, whether an ELSE is expected. The BCPL language, for instance, does this by using TEST instead of IF when there is an ELSE clause. This helps the system insert the right number of loose ends.

Thirdly, GOTOs are nasty. Specifically, the problem comes from forward GOTOs (i.e., jumps to statements not previously executed), because they stop the program being constructed in a structured manner. Disguised GOTOs, like the RETURN, STOP or BREAK statements found in some programming languages, are not a problem, because they define a structuring level for the destination. Any GOT0 statement should act as a terminator of a loose end, since it marks the end of a block of code.

Lastly, procedures and functions, though not yet in Build, should lend themselves well to dynamic building. The body of a routine (our collective name for a procedure or function) would be a loose end that is filled in by the programmer when the routine is called. Dynamic building may, indeed, be more suitable for constructing individual routines than for whole programs. For testing a routine, the main program could just be a series of calls to test all desired combinations of argument values. For example to test the function F to make sure that F(0) is 1, $F(1)$ is 16 and F'(2) is -9, the following sequence can be used

```
    PRINT F(O), " SIIOULD EQUAL 1"
```

The system now prompts for the body of F. On return from F, the above PRINT statement is completed and thus the programmer sees the result of F(0). He is then prompted to supply more main program, and so he types

```
    PRINT F(l), " SHOULD EQUAL 16"
```

At this stage there may be prompts for parts of F used by F(1) but not F(0). After the above is complete the programmer can complete the test by typing

```
    PRINT F(2), " SIIOULD EQUAL -9"
```

If a previously executed statement of ***F*** is edited the main program is automatically re-run, thus ensuring that ***F*** is properly tested.

On top of these facilities there should be a library mechanism whereby tested routines could be stored away. These could then be incorporated into other programs without building them from scratch again.

Related work.

The task of a debugging system is to illuminate the run-time behavior of a program, a similar aim to the present research. The classic reference for such systems is the work of Satterthwaite [1975]. One idea of Satterthwaite's is the "stream of consciousness" whereby the run-time system prints each statement as it goes through the program, together with values of the relevant variables and expressions. To avoid gigantic piles of output, he only traces the first and second iterations of each loop. Satterthwaite's system deals with batch compilers rather than interactive ones. Perhaps the most important relevance of this work to dynamic building is that it serves as an excellent model to show what information should be provided when tracing a program. With dynamic building most of this information should be made available as the program is typed in, since the sequence of typing follows the program trace.

Another good debugging system is BAIL [Reiser, 1975]. BAIL provides execution of immediate statements during debugging, and single-step execution of the source program. These are facilities that automatically come out of dynamic building. Some more recent debugging systems have augmented these facilities [Teitelman, 1977; Xerox, 1979], and in particular have exploited the use of a display screen with several windows to show different kinds of information simultaneously. The Build system would benefit from being enhanced in a similar way.

Two research projects emanating from Cornell [Teitelbaum and Reps, 1980; Archer et al, 1980] have been concerned with the interactive synthesis of programs. The system of Teitelbaum and Reps, which is reported to have an enthusiastic user response, builds programs in a dialect of PL/I by using templates. (The automatic supplying of RBs in the Build system is akin to these templates.) The ternplates constrain the user to produce programs that are syntactically correct and properly constructed, and are thus especially useful to beginners. Interestingly, the user does not have to fill in the "holes" in templates in a prescribed order, and therefore can construct a program in the order most natural to him. The other Cornell project has similar aims, but does not use the template mechanism; it has a novel system whereby immediate statements are automatically remembered, thus getting away from the normal dichotomy between immediate execution and incorporation into the program. The thrust of these two Cornell projects is different from that of dynamic building; dynamic building is aimed at making the user understand how his program runs, whereas the Cornell work helps the user construct statements and other program units.

L^* [Newell, McCracken and Robertson, 1977] is an extremely interesting, though sadly little known, programming language which takes a dynamic view of program construction. The language is innovative and unusual in many other ways too, and certainly does not share the aim of the present research to be based on conventional algebraic languages.

LC^2 [Mitchell, Perlis and Van Zoeren, 1968] is an early interactive language which allowed program statements to be specified at run-time as an alternative to compile-time.

Finally, the work on programming by example is certainly concerned with building programs in a dynamic manner. In this field, Tinker [Lieberman and Hewitt, 1980] is of most relevance to dynamic building. The Tinker system, which is used to construct LISP programs, aims to be practical and usable rather than super-clever at guessing the user's intentions. The user guides the system through the function being defined, and gives it snippets of code to put together. Although Build aims at simplicity and discipline, whereas Tinker aims to be an intelligent and non-constraining aid to program construction, the two systems have several similarities. Many of the points about error detection in this paper are made in the paper on Tinker, which was published earlier.

The chip simulator.

As well as Build, a second system which encompasses the dynamic building principle has been constructed. This is Hal, a functional simulator of a new string processing chip. Unlike Build, Hal programs have no block structure, but they do support GOTOs (which are conditional on signals output by the chip). Each instruction of a Hal program consists of a set of low-level signals to the chip, and the chance of making mistakes is high. Hence dynamic building, where the effect of each instruction can be monitored as it is typed in, is a great aid to preparing correct programs.

Getting user iced-back.

The amount of user experience of our dynamic building systems is currently small, and it is not yet possible to evaluate how users take to the discipline of dynamic building, and whether it really helps them build programs better. Indeed objective measurement of such factors is notoriously difficult even when there is a lot of user experience.

The best way to test user reaction would be to construct a complete dynamic building system for BASIC, say, that would compete with existing products. We believe that this is worth doing, but it represents about two years of work. A more limited approach is to apply dynamic building to specialized interactive systems, like the Hal simulator. This approach is the preferred one because

(a) Some of these systems are small and can be implemented in months rather than years. Thus user feed-back is more easily obtained.

(b) Users are less constrained by previous programming habits than, for example, with a BASIC system.

(c) In many cases programs are short but exceptionally subject to run-time errors, and this is just the application that dynamic building is good for.

Conclusions.

The research has now demonstrated that it is technically feasible to construct a dynamic building system and to coordinate it with interactive editing. Since user experience is still minimal the conclusions of this paper must be speculative, but we would like to make the following points.

With programming languages, amateurs such as scientists, engineers and some business men, prefer interactive languages, whereas professionals go for batch languages such as C or extended PASCAL. We believe that dynamic building will likewise attract amateurs. We also believe that there is a bigger need for simple program construction tools for amateurs than there is for sophisticated tools for professionals.

Secondly, dynamic building encourages program construction at a terminal. It is not suitable in an environment where there is always someone breathing down your neck wanting your terminal. It is suitable for a personal computer.

Thirdly, there is not a big learning hump. A dynamic building system is actually simpler than a traditional system as the difference between compile-time and run-time has been eliminated. The **only place** where there is any extra complication is in editing.

Finally, dynamic building, although necessarily imperfect in this respect, does provide a simple **and** practical way of finding more run-time errors than traditional systems.

To summarise, we do not expect the world to change its programming habits overnight and adopt dynamic building. However we do think that there is a large class of users who have trouble in producing even short programs that work, and who have equal trouble mastering sophisticated tools. Dynamic building is a simple and natural way of helping **such people.**

Acknowledgments.

I would like to thank Donald Knuth for a useful discussion on the relation between dynamic building and debugging systems. Even more, I would like to thank him and his research sponsors for giving me the opportunity to come to Stanford and work on these ideas.

References

Archer, J. R., R. Conway, A. Shore and L. Silver (1980). "The CORE user interface", Report TR 80-437, Cornell University.

Ingalls, D. H. H. (1978). "The Smalltalk- programming system: design and implementation," *Fifth **annual*** ACM ***symposium on principles*** of programming ***languages,*** 9-16.

Lieberman, H. and C. Hewitt (1980). "A session with TINKER: interleaving program testing with program design," Con*ference rccord* of the *1980* LISP conference, 90-99.

Mitchell, J. G., A. J. Perlis and H. R. Van Zoerner (1968). "LC^2 : a language for conversational computing," in M. Klerer and J. Reinfeld (Eds.): I*nt*eractive Systems ***for*** Experimental ***Applied*** Mathematics, Academic Press.

Newell, A., D. McCracken and G. Robertson (1977). "L^* : an interactive, symbolic implementation system," Dept. of Computer Science, Carnegie-Mellon University.

Reiscr, J. F. (1975). "BAIL -- a debugger for SAIL," Stanford University Computer Science Report STAN-CS-75-523.

Satterthwaite, E. H. (1975)."Source language debugging tools," Ph.D. thesis, Stanford University.

Teitelbaum, T. and T. Reps (1980). "The Cornell Program synthesizer: a syntax-directed programming environment ," Report TR 80-421, Cornell University.

Teitelman, W. (1977). "A display oriented programmer's assistant," Report CSL 77-3, Xerox PARC.

Xerox Corporation (1979). "MESA debugger documentation," Xerox PARC.