CHANGE-EPISODES IN CODING: WHEN AND HOW DO PROGRAMMERS CHANGE THEIR CODE?

WAYNE D. GRAY
Army Research Institute
5001 Eisenhower Ave.
Alexandria, VA 22333

JOHN R. ANDERSON
Psychology Department
Carnegie-Mellon University
Pittsburgh, PA 15213

ABSTRACT

Any change in a programmer's code or intentions while coding constitutes a change-episode. Change-episodes include error detection and correction (including false positives) as well as stylistic, strategic, and tactical changes. In this study we examine change-episodes to determine what they can add to the study of the cognition of programming. We argue that change-episodes occur most often for constructs that allow the most variability (with variability defined by the language, the task, and the programmer's history). We predict and find that those constructs that are involved in the most change-episodes are those for which much planning is needed during coding. Similarly, we discuss two ways in which a goal can be changed in a change-episode. One involves relatively minor editing of a goal's subgoals, suggesting that much planning is local to the current goal. The other entails major transformations in the goal's structure. Finally, we find that change-episodes are initiated in one of three very distinct circumstances: as an interrupt to coding, a tag-along to another change-episode, or a byproduct of symbolic execution. Our findings support the distinction between inherent and planning subgoals (2, 3) and the distinction between progressive and evaluative problem-solving activities (6).

INTRODUCTION

Although making changes and catching errors while coding seems a ubiquitous part of our own programming behavior we have never seen the topic referred to, let alone treated in a systematic way. It occurred to us that such changes without external feedback represent key junctures in the process of coding. At these junctures existing plans are modified, scrapped, or fixed, and new plans are developed. If these junctures could be studied systematically they might provide a wealth of information about the cognitive processes involved in coding.

In this paper we examine the changes that programmers make while writing, but before testing, their code. We present the analysis techniques that were developed to probe the nature of these junctures, or ***change-episodes***. The results of this analysis help illuminate the nature of programming as a complex cognitive skill.

In the next section we define change-episodes, provide a perspective on coding as a problem-solving activity, and propose hypotheses in the context of a tripartite analysis of change-episodes. This discussion is followed by the details of this empirical study. We then present the data and conclude by summarizing the findings, assessing their implications for understanding programming in particular and problem-solving in general.

CHANGES WHILE CODING

A change-episode occurs when either the programmer alters code that is already on the screen or modifies plans that have been articulated (based on the verbal protocol). There are three parts to a change-episode: the change-goal, the noticing event, and the fix. The ***change-goal*** is the goal that the programmer later decides to change. Are all goals equally likely candidates for change or can we identify systematic differences between those that are and those that are not change-goals? The ***noticing event*** is the first indication that the programmer wants to change the goal-structure of his/her solution. Do noticing events occur at random or at particular times or places during coding? The ***fix*** is the goal-structure after it has been changed. Are there as many types of changes as there are change-episodes or can most changes be placed in one of a small number of change categories?

The next section presents our theoretical perspective on coding. Then for each part of the change-episode, that is, the change-goal, noticing event, and fix, we present hypotheses that arise from this theoretical perspective and discuss how we count, measure, or otherwise quantify each part.

Coding as Problem Solving

We view coding as a type of problem-solving activity and problem solving as involving search in a problem space (1). The initial state for our programmers is defined by the problem statement in conjunction with the knowledge and skills that the programmers bring to the experiment. The problem statement defines as the goal state a LISP function that will search a hierarchy to determine if *=person1* has as a descendent *=person2*. It also provides certain design specifications and constraints, such as, the search must be depth-first and self-terminating, iteration must be used (not recursion), and so on. (See the THE STUDY section for more details.)

The knowledge and skills that our programmers possess (advanced novice LISPers) determine what parts of the code involve problem solving and what parts simply entail the retrieval of information from long-term memory.

The goal-structure of the problem solution can (and will) be presented hierarchically. However, because a superordinate goal is retrieved effortlessly from long-term memory does not mean that its subordinates also exist in long-term memory. For example, in coding the function, most of our programmers retrieve a subgoal-template for coding the let/loop body of the function (a let/loop construction is specified in the problem statement) from long-term memory:

right-paren predicate-let variable-declaration loop-body left-paren

We view each part of this template as a separate subgoal. Solutions to the first two subgoals "(" and "let" are available in long-term memory and are immediately typed. The third subgoal, the variable-declaration, constitutes a subproblem that has its own subgoals, some of which require lookup in long-term memory and others which require problem solving. The types of problem-solving activity varies between subgoals. For example labeling the variable may require finding an appropriately mnemonic name whereas deciding upon an initial value for the variable may require determining the best data-type (atom or list) based upon how the variable later is used. Similar considerations surround the coding of the loop-body subgoal.

The above subgoal-template lists only ***inherent goals***. A complete goal-structure hierarchy would include all inherent goals as well as all ***planning goals*** used in coding the function. The distinction between inherent goals and planning goals was made by Anderson, Farrell, and Sauers (2, 3). The important feature of inherent goals is that, in achieving them, one achieves part of the original task. On the other hand, planning goals produce results that are used to guide solution of the original problem but the results themselves are not part of the final solution. For example, if the programmer decides that the variable will initially have the value of the first parameter, s/he still must decide the data-type for the variable, whether to make it an atom or a list. Making that decision requires determining how the variable will be used in the function. The goals associated with this determination are clear-cut examples of planning goals in that they are used to guide the solution to the problem but are not part of the final solution. In contrast the goal of initializing is part of the final solution and is therefore an inherent goal. (Note that, as in the above example, an inherent goal can have both planning and inherent subgoals but, strictly speaking, planning goals can have only planning subgoals.)

As experience in dealing with a particular goal and its subgoals is gained, planning goals drop out of the goal-structure hierarchy and the goal-structure that remains increasingly mirrors that of LISP (as represented in the subgoal-template shown above). This evolution of knowledge structures is explained by the ACT* process of knowledge compilation (4, 5).

Following Allwood (6) we distinguish between two types of problem solving (and therefore, coding) activities: progressive and evaluative. Progressive activities work directly towards the goal state of the problem. Evaluative activities evaluate some already executed part of the problem solution. By this dichotomy, coding and planning what to code are progressive activities while checking and changing code (that is, change-episodes) are evaluative activities. However, while Allwood saw evaluative activities as merely affirming the correctness of the solution or flagging it as erroneous, we view them as also involved in issues of programming style and elegance. Hence, rather than talking as Allwood does about ***error detection and correction*** we prefer to talk about change-episodes. This difference in terminology arises because we study programming while Allwood studied statistics problems, a domain that is much less tolerant of individual choice. In our definition, error detection and correction is a subcategory of change-episodes.

Historically most accounts of problem-solving have emphasized progressive activities and tended to ignore evaluative activities. The study of programming has followed this trend. Even those areas that may be thought to involve evaluative activities, such as debugging or software maintenance and revision, really deal with the progressive side of problem solving. For example, debugging activities that involve working from a given symptom (initial state) to a working program (goal state) are progressive activities. Questions as to how programmers evaluate their progress while debugging and when they decide to test the code, that is, activities that involve the problem solver evaluating his/her problem solving efforts, are usually not addressed.

Change-Goals

A change-goal is the problem-solving goal that a programmer either sets or completes and, at a later time, changes; it is the goal that is the target of the change-episode.

Hypotheses

We predict that the probability of a goal being the target of a change-episode will be correlated with the amount of planning involved in its execution. Hence, an analysis of change-episodes will identify many of the places where planning occurs.

Rationale. Based upon the analysis in the preceding section, we expect goals to have a lot of planning subgoals if either (a) the goal is not yet well learned and has not been subjected to knowledge compilation, or (b) if there are so many variations as to how the goal can be used that it is unlikely that a specific rule for the required variation has yet been compiled. In the first case, poorly learned goals can easily be coded wrong and become the target of an error detection type of change-episode. In the second case, choosing the correct instantiation of a goal may depend upon

having a fairly complete idea of how the rest of the function will be coded. To the extent that the exact plan depends upon the instantiation of a number of different highly variable goals, we would expect the exact instantiation of any one variable goal to affect the programmer's plan and through that plan, to affect the specifications of other goals (both coded and not yet coded).

In the present study, our programmers' instructional and programming histories were known. Therefore a first estimate of the relative ratio of inherent to planning subgoals in a particular goal was based upon a task analysis of LISP plus knowledge of this history. Analyzing change-episodes to identify which goals were the most frequent and least frequent change-goals was expected to provide a second, and converging, estimate of the amount of planning required to execute a given goal. If so, it would provide additional support for the distinction between inherent and planning goals and the role of knowledge compilation. Additionally, if this measure could be validated it could then be applied to situations in which the programmer's history was not well known. This would provide a tool for identifying weaknesses in computer language curricula as well as a means of predicting likely places for bugs and problems in code comprehension.

The next section provides a detailed exposition of the syntax and semantics of the LISP goal-structure hierarchies upon which much of our analysis rests. Since our hypotheses and conclusions are independent of these hierarchies, readers can skip this section and still be able to appreciate our main findings and follow our conclusions regarding change-episodes in coding.

What are Change-Goals?

A change-goal is identified as a goal-structure in a goal-structure hierarchy. The goal-structure for a given change-goal consists of a node#, a goal-label, and a subgoal-template. In this section we discuss the parts of the goal-structure and how they are derived. In a later section we will discuss how a given goal-structure is identified as *the* change-goal for a change-episode.

As an example, consider the following piece of code whose goal-structure is shown in Figure 1:

```
(cond  ((null queue) (return nil))
       ((equal (car queue) ancestor) (return t))
)
```

This is a two clause, LISP conditional (hence the **cond**) that occurs inside the loop in an iterative *let/loop* construct. The first clause tests whether the list contained by variable **queue** is **null** (that is, empty) and if so, exits the loop and returns **nil**. The second clause tests whether the first item in queue is **equal** to the target **ancestor**, if so it exits the loop and returns the value **t** for true.

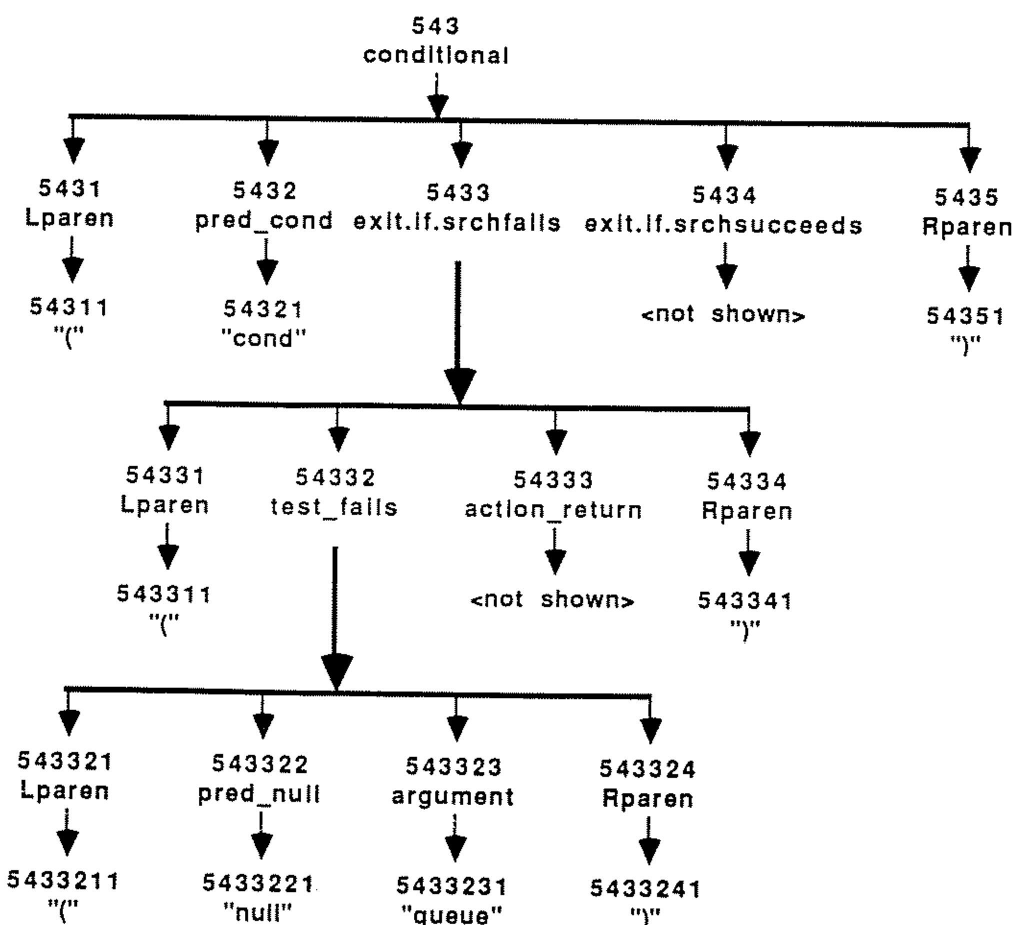

Figure 1
Goal-Structure Hierarchy

The node#, *543*, (see Figure 1) uniquely places the goal-structure in the goal-structure hierarchy. The goal-label, ***conditional***, contains semantic information regarding the purpose of the goal. The subgoal-template, ***Lparen pred_cond exit.if.searchfails, exit.if.searchsucceeds, Rparen***, contains both semantic information regarding the nature of the subnodes, and syntactic information regarding their left-right order in the goal. Note that each slot in the subgoal-template is itself a goal-label that has a subgoal-template. Hence the goal-structure for a given goal comprises two levels in the hierarchy: the goal and its immediate subgoals.

The goal-structure hierarchy distinguishes between higher goals and leaf goals. The leaf goals in Figure 1 are the terminal nodes of the goal-structure to which they are attached. They are what actually gets typed. For example, node# *54321*, goal-label: ***cond*** is what gets typed as the predicate, or operator, for its superordinate (node# *5432*, goal-label: ***pred_cond***). Leaf goals do not have subgoal-templates; they are the keystrokes that get executed. Higher goals are all of the non-terminal nodes in the hierarchy; they are cognitive constructs.

We can further define higher goals by distinguishing between two types: penultimate goals and higher-order goals. Penultimate goals have one subgoal which is always a leaf. For example:

> Node#: *5432*
> Goal-label: ***pred_cond***
> Subgoal-template: ***"cond"***

In contrast, a completed higher-order goal always has more than one subgoal none of which are leaf goals. In Figure 1 five higher-order goals are shown: Node# *543*, goal-label: ***conditional***; node# *5433*, goal-label: ***exit.if.searchfails***; node# *5434*, goal-label: ***exit.if.searchsucceeds***; node# *54332*, goal-label: ***test_fails***, and node# *54333*, goal-label: ***action_return***.

There are two points to emphasize regarding goal-structures and the goal-structure hierarchy. First, the hierarchy captures inherent goals only, not planning goals. When planning goals are stripped away, what remains, the inherent goals, closely mirrors the structure of LISP.

Second, our estimation of what goals are compiled and what are not is built into the semantics of the hierarchy. For higher-order and penultimate goals, all labels other than ***predicate*** or ***argument*** reflect estimations of the compiled knowledge that our programmers have acquired. For example, in Figure 1 the first conditional clause, node# *5433*, is actually the first argument of the predicate **cond**. If our programmers had had no or very little experience with conditionals, we would have given this node the label ***argument*** with the prediction that its subgoal-template would contain a large number of planning subgoals. If our programmers had been introduced to conditionals and had used them enough to have learned their structure, then we would have labeled this node (node# *5433*) ***clause1***. Such a label would presume that our programmers knew that a conditional clause required a left-paren, a test part, an action part, and a right-paren; quite a few planning subgoals would be required to turn ***clause1*** into an exit test. Instead of calling this node ***argument*** or ***clause1***, we have labeled it ***exit.if.searchfails***. This label reflects the knowledge and skill that our advanced novice programmers had in using a conditional clause to exit a loop when a guard variable became null. As shown in Figure 1, we believe that our programmers had a subgoal-template for this goal that consisted of a left-paren, a standardization of a clause test (labeled ***test_fails***), a standardization of a clause action (labeled ***action_return***), and a right-paren. In this case we believe that knowledge compiled at the ***exit.if.searchfails*** goal largely dictates the type of subgoals (as reflected in the subgoal-labels) and the syntax of the subgoals. Hence, the subgoal-template for ***exit.if.searchfails*** should have very few planning subgoals.

The Fix

The *fix* is the goal-structure after it has been changed. Comparing the goal-structure before the change-episode to the goal-structure after the change-episode enables us to pinpoint the change-goal and, in so doing, to determine the relationship and distance between the change-goal and the noticing event. Additionally, comparing the before and after goal-structures provides information as to ***how*** the goal changed; this is the main topic of this section.

Issues & Hypotheses

There are two issues. First, are there as many types of changes as there are change-episodes or can most changes be placed in one of a small number of change categories? For this issue we did not start with a hypothesis but with the hope that most types of fixes could be classified into one of a small number of categories. As discussed in the next section (How are Fixes Identified?), in encoding the fix types we were able to induce two categories that accounted for the vast majority of change-episodes.

Second, what can the type of changes tell us about the role of planning in coding? For this issue our perspective on coding as problem solving led us to hypothesize that most changes would be limited to minor alterations of the change-goal's subgoal-template.[1]

Examples of minor changes would be inserting missing subgoals or transposing the order of subgoals already coded. More major changes would involve the deletion or addition of entire subgoal levels, as when a missing intermediate level goal is inserted between its already partially coded superordinate and its completely coded subordinates. If most planning involves planning subgoals directly attached to the goal being coded then we would expect that most changes would involve modifications to the current goal and would therefore be relatively minor. In contrast, if the

---

[1]Note that minor changes to the subgoal-template could reflect major additions or deletions of lines of code. For example, the **conditional** shown in Figure 1 has two clauses. If the programmer had decided to delete the second clause (node# *5434*; goal-label: ***exit.if.searchsucceeds***) this would have been a substantial deletion of code but only a minor change to the **conditional**'s subgoal-template.

planning subgoals encountered while coding had little to do with the currently active goal then changes might involve modifications to distant cousins or siblings, or involve major deletions or additions of intermediate level goal-structures.

How are Fixes Identified?[2]

Our meta-rule is to compare the before and after goal-structure hierarchy to locate the change-goal as the highest goal that changes. For each change-episode we derived two goal-structure hierarchies (such as shown in Figures 1 and 2). The first was the goal-structure hierarchy at the time that the noticing event occurred (discussed in more detail in the Noticing Event section). The second was the goal-structure hierarchy after the fix had been made. By comparing the before and after goal-structure hierarchies, two rules were induced.

Rule A: **Change in subgoal-template.** In locating the change-goal, the single most important rule is to find the goal whose subgoal-template changes. There are four basic changes to the subgoal-template: replacements, transpositions, insertions, or deletions. Replacements and transpositions are prototypical of two different kinds of changes. Transpositions signal a syntactic change in the subgoal-template, while replacements involve the semantics of subgoal-labels. Deletions and insertions involve both types of change.

Changes identified by Rule A are relatively minor changes that are local to the goal. As such, Rule A suggests the operation of planning subgoals located at the level of the change-goal's subgoal-template. In the course of coding a function in a general depth-first, left-to-right fashion (3), the programmers sometimes encounter a planning subgoal that causes them to transpose, replace, insert, or delete a sibling subgoal (where both subgoals share the same superordinate).

Rule B: **A goal-label by any other subgoal-template.** The identifying characteristic of a goal is its label, not its subgoal-template. As an example, consider the case in which a penultimate goal is changed to a higher-order goal. In Figure 2 we have diagrammed a portion of both the before and after goal-structure hierarchy for the body of a local variable update: **(cons (expand (car storer)) storer)**.

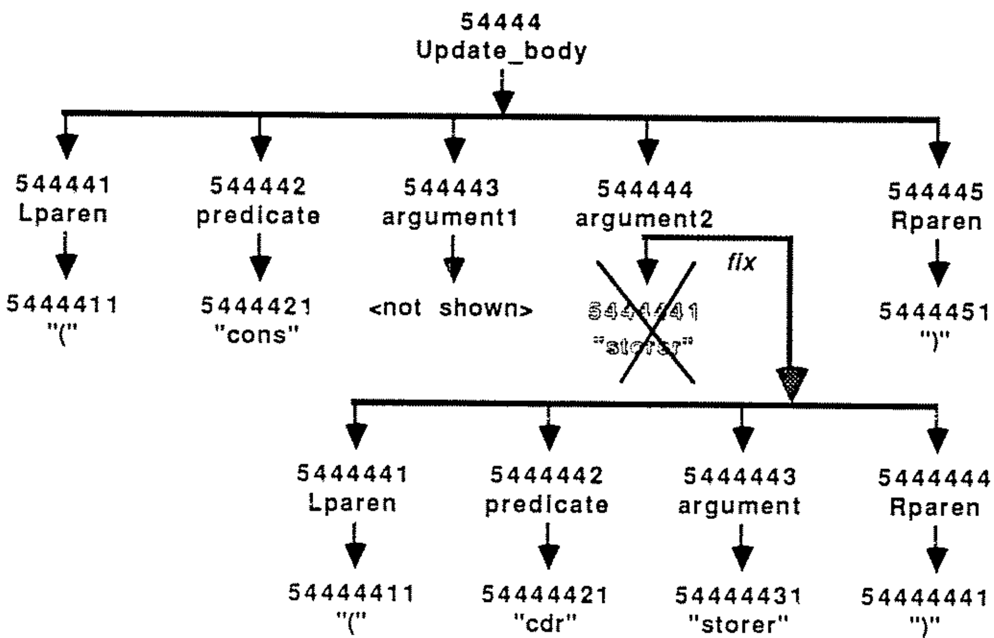

Figure 2
Goal-Structure Hierarchy for Body of a Local Variable Update

In this case the predicate, **cons**, will take whatever is returned by argument1, **(expand (car storer))**, and add it to the front of argument2, **storer**. Later the programmer changes her mind and decides that she wants to add whatever argument1 returns to what is left of the list **storer** after the first item is removed, **(cdr storer)**. From the point of view of node# *54444*, goal-label: *Update_body*, nothing has changed. The subgoal-template is still:

Lparen predicate argument1 argument2 Rparen.

In contrast, for node# *544444*, goal-label: ***argument2***, much has changed. Where before it was a penultimate goal with a subgoal-template of *storer*, now it is a higher-order goal with a subgoal-template:

Lparen predicate argument Rparen.

Despite these drastic changes, the goal has not changed. It is still ***argument2*** for the ***Update_body*** goal. However, the goal-structure has been transformed (see Figure 2). There is now an intermediate goal between ***argument2*** and ***storer***.

[2]A detailed understanding of this section is not needed to follow our conclusions regarding change-episodes. Readers may want to skim this section now, refering back as needed.

Noticing Events

A noticing event signals the beginning of a change-episode. It is the first indication that the programmer wants to change the goal-structure of his/her solution. Noticing events are indicated by either keystrokes or verbal protocol. For example, if the programmer decides to interrupt his/her coding to change something that was coded three lines above, the noticing event might be indicated by the use of the *upline* command. Verbal indications might be something as simple as an *oops* followed by revision, or may include a complete statement of the change-goal.

How are Noticing Events Classified?

Noticing events are classified by the activity in which they occur. Our analysis yields four categories: interrupts, tag-alongs, symbolic execution, and miscellaneous.

Interrupts are interruptions in the progressive activity of coding. To count as an interrupt, a noticing event has to occur shortly after some keyboard event with no intervening activity. For a ***tag-along*** classification, the noticing event has to occur during another change-episode (an evaluative activity).

Symbolic execution (7) is really a category of programmer activity not just a type of change-episode. By symbolic execution, Kant and Newell refer to that programming activity in which the programmer is symbolically (or mentally) executing the program (or a portion of the program) with either general symbols or specific values as inputs. As a category of noticing events, we use symbolic execution to apply to those change-episodes that are initiated during symbolic execution (in contrast to interrupts and tag-alongs). The final category, ***miscellaneous***, includes noticing events that occur when the programmer is straightening up the screen (prettyprinting), reading the text, or whatever.

Hypotheses & Rationale

Are change-episodes begun as an abrupt interruption to the progressive activity of coding or are they a more natural outcome of an activity such as symbolic execution that was begun for some other reason? Do change-goals differ for change-episodes initiated by different types of noticing events? Is there a relationship between the activity that immediately precedes the noticing event and the change-goal? These are the three broad questions regarding noticing events that we had expected this study to address. In some cases we had hypotheses based upon our theoretical perspective; in other cases we did not and were unable to think of any theory that would make predictions.

First, while we would have been surprised if all change-episodes were noticed during, for example, interrupts and none were noticed during symbolic execution, we did not have any hypotheses regarding the proportion of change-episodes in each noticing event category. Second, we had expected (as discussed above under **Change-Goals**) that for each noticing event category the probability of a goal being the target of a change-episode would vary with the amount of planning involved in its execution. Within this constraint we had no hypotheses regarding whether different change-goals had different probabilities of being noticed, for example, by an interrupt versus by symbolic execution.

It was only for the interrupt category of noticing events that we had a specific hypothesis regarding what the programmer was doing when the noticing event occurred. This hypothesis is discussed next.

Issues for Interrupts.

When programmers interrupt their code to initiate a change-episode, what is the relationship between the goal that they interrupt and the change-goal? To ask the same question from a different perspective -- if a given change-goal is noticed with an interrupt, when is that interrupt most likely to occur? In the context of the goal-structure hierarchy, will the interrupted goal be a superordinate, subordinate, sibling, uncle, or so on, of the change-goal?

We predict that the interrupted goal is more likely to be either the change-goal itself or a subordinate of the change-goal than either a superordinate or sibling. This is because the programmer will only maintain a representation of a piece of code while it is being worked on. Thus, if the interrupted goal is subordinate to the change-goal, the change-goal will be in the state of being worked on. On the other hand, if the interrupted goal is a sibling or parent of the change-goal, the change-goal will not be under consideration.

Pinpointing the Interrupted Goal.[3] For each change-episode, we derived a goal-structure hierarchy of the programmer's code at the instant that the noticing event occurred. Because of our hypothesis concerning the relation between the interrupted goal and the change-goal, we were interested in (1) whether the interrupted goal was a leaf, penultimate, or higher-order goal, and (2) its relationship to the change-goal. In the preceding section, The Fix, we discussed how we pinpointed the change-goal. Here we discuss the criteria used to pinpoint the interrupted goal.

Consider, as an example, the change-episode (discussed earlier) whose before and after goal-structure hierarchies are diagrammed in Figure 2. The interrupted goal would be a leaf goal if the noticing event occurred during the typing of **storer** (node# *5444441*). To be precise, if the programmer had either partially typed **storer**, for example, **sto** or had just finished typing the final **r**, then the noticing event would be said to have interrupted a leaf goal that was the *child* of the change-goal ***argument2*** (node# *544444*). In contrast, if the noticing event occurred after the programmer typed **storer** *plus* a space (**storer_**) then the interrupted goal would be the penultimate goal ***argument2*** (node# *544444*) and the interrupted goal would be the change-goal itself (*self*).

Finally, if the programmer had completed the ***update_body*** goal (node# *54444*) by typing the closing right parenthesis (node# *544445*) and (perhaps) a space but had not yet started the next goal, then the interrupted goal would be the higher-order goal ***update_body*** goal (node# *54444*) and the interrupted goal would be the *parent* of the change-goal. (Note that in our analysis scheme, parentheses play the role of delimiting the beginning and end of higher-order goals.)

---

[3] Those not interested in methodological details may wish to skip this section.

Summary

Any change in a programmer's code or intentions constitutes a change-episode. Change-episodes include error detection and correction (including false positives) as well as stylistic, strategic, and tactical changes. In this study we examine change-episodes from the general perspective of coding as problem solving and from the more specific perspective offered by Anderson et al.'s (2, 3) theory of inherent and planning goals and Allwood's (6) distinction between progressive and evaluative problem-solving activity.

We argued that change-episodes occur most often for constructs that allow the most variability (with variability defined by the language, the task, and the programmer's history). Therefore we predicted that the more planning a goal needs during coding, the more likely it is to be the target of a change-episode. Similarly, we discussed two ways in which a goal-structure can be fixed. One way, Rule A, involves relatively minor editing of the subgoal-template, the other, Rule B, entails major transformations in the goal-structure. Finally, we proposed that change-episodes may be initiated (the noticing event) by interrupts to coding, as tag-alongs to other change-episodes, or during symbolic monitoring. We predicted that for interrupts, the interrupted goal was more likely to be either the change-goal itself or a subordinate, than a superordinate or sibling.

THE STUDY

The tripartite analysis of change-episodes and the techniques for analyzing the goal-structure hierarchy were based upon the analysis of ten advanced-novice LISP programmers writing one LISP function. Because data for the current study was collected in the course of a larger, non-related study, five of the programmers received special training prior to coding the function. Consequently, although the analyses developed were based upon an examination of data from all ten programmers, only protocols from the five programmers in the control group were encoded. These are reported below.

Programmers (their prior experience)

The five programmers were paid volunteers from the introductory LISP course offered by the Psychology Department at Carnegie-Mellon University. None of the five had any prior experience with LISP; all had had some programming experience ranging from a failed PASCAL course to self-rated expertise in PASCAL and BASIC. All programmers were in the 12th to 13th week of the 15-week course and all had completed the eleven modules offered on the LISP Tutor (8).

The Function

The function that the programmers coded was similar to those they would find in the ***Search Techniques*** chapter of their textbook (9). The specifications for the function were fairly complete. The programmers were told to write an iterative function, using the let/loop construction, and not to make any recursive function calls. The function was to take two parameters and perform a depth-first search of a tree, returning **t** if the second parameter was a descendent of the first, and returning **nil** otherwise. Additionally, they were told to use an expansion function, called ***expand***, that takes a node in the tree and returns a list of all of its immediate descendents. Finally, they were told to create a local variable to store nodes returned by the expansion function until they could be individually checked. The experimenters' code for this problem is given below:

```
(defun descendent (ancestor progeny)
  (let ((queue (expand ancestor)))
    (loop
      (cond ((null queue)(return nil))
            ((equal (car queue) progeny)(return t)))
      (setq queue (append (expand (car queue))(cdr queue)))
    )))
```

Procedures

Programmers were run individually. Upon arriving for a session they were given an overview and asked about their programming background. They then practiced talking aloud while solving mental arithmetic and anagram problems (10).

Programmers next read an abridged version of the search chapter from their textbook (9). They were then given a description of the problem along with a sample tree diagram and told to talk aloud as they read. After they indicated they understood the problem, they were asked to list the nodes from the tree diagram in the order in which they would be visited. This task provided a check to see if they understood what a depth-first search entailed. None of the programmers had any difficulties with this task.

Programmers then talked aloud as they coded the problem on a computer terminal using a version of the EMACS (11) editor. All programmers were familiar with EMACS, though some occasionally tried to use commands that were not implemented in our version. Programmers worked on the problem until they were satisfied that it was complete. While coding, programmers were free to consult their textbook or the experimenter regarding any LISP questions they might have. (The experimenter did not tell the programmers how to code the function, but did, when asked, provide *textbook* answers to how a particular function worked or the syntax of a particular construction.) Programmers were also free to consult the problem description and diagram. The experimenter prompted the programmers to talk aloud as needed.

Data Collected and Protocol Transcriptions

Both keystroke and videotape data were collected. The keystroke data was time-stamped at the one second level. The videotape was time-stamped to the millisec although in practice all times were rounded to the nearest second.

The video camera was focused on the screen. This permitted us to correlate what programmers were typing and seeing with what they were saying. It also greatly facilitated the interweaving of the time-stamped keystroke data with the verbal protocol.

RESULTS

Overall

As Table 1 indicates, there were large differences among our programmers in time to code the function. Time in minutes ranged from 52.60 to 7.92 with a mean of 20.83. Likewise the number of change-episodes per programmer varied from 8 to 37. Looking at the relationship between the number of minutes to code the function and the number of change-episodes yields an average of one change-episode every 1.01min. To examine the distribution of change-episodes throughout the coding period, we divided each programmer's coding time into ten equal intervals. The distribution of change-episodes by interval did not differ from what would be expected by chance (χ^2 [9, n=105] = 9.56). (Note that .05 is the level of significance for all statistics reported in this article and all tests reported are two-tail.) Change-episodes occur both frequently and evenly while the programmer is coding.

Table 2 shows the type of change-goal by programmer. There were large differences overall and between programmers in the distribution of type of change-goal. Overall 27% of the changes involved leaf goals, 14% penultimate goals, and 59% higher-order goals. These differences contrast with the proportion of leaf, penultimate, and higher-order goals in the experimenters' solution shown above. There are 156 goals in the experimenters' solution: 44% (68) leaf goals, 44% (68) penultimate goals, and 13% (20) higher-order goals. A χ^2 test indicates that the number of changes for a goal type is not proportional to the number of goals of that type in the experimenters' solution (χ^2 [2, n=105] = 202.63). This difference in found versus expected is also significant when parentheses are excluded from the count of leaf and penultimate goals (χ^2 [2, n=105] = 60.35). For both tests, the proportion of change-episodes involving higher-order goals is much greater than would be expected by chance. Concomitantly, the proportion of change-episodes involving leaf and penultimate goals is less than expected.

Table 1

Time to code and number of change-episodes by programmer

Programmer	Time(min)	Change-Episode	Changes per Min
MK05	52.60	37	.70
NS03	16.43	14	.85
WB12	16.23	33	2.03
AP02	10.98	13	1.18
MK02	7.92	8	1.01
Mean	20.83	21	1.01

Table 2

Frequency of change-goal type by programmer (with percentages in parentheses).

Programmer	Leaf Goal	Penultimate Goal	Higher-order Goal	Total
MK05	9 (24)	3 (8)	25 (68)	37 (100)
NS03	7 (50)	5 (36)	2 (14)	14 (100)
WB12	6 (18)	4 (12)	23 (70)	33 (100)
AP02	4 (31)	3 (23)	6 (46)	13 (100)
MK02	2 (25)	0 (0)	6 (75)	8 (100)
Sum	28 (27)	15 (14)	62 (59)	105

Detailed Analyses

Which Goals are the Change-Goals?

Rules A and B were used to identify the change-goal for each of the 105 change-episodes. Since the goal for leaf goals was always the word being typed, the goal-label was more or less unique to a particular programmer coding a particular function. Because of this we did not tabulate changes in leaf goals. In contrast, since goal-labels for penultimate goals and higher-order goals were common across programmers, it made sense to ask which of these goals changed the most and the least often.

Penultimate goals. We counted 15 types of penultimate goals represented in our 5 programmers' final code. Since there were only 15 change-episodes involving penultimate goals (see Table 2), we reduced the penultimate change-goals to two categories: predicates (any LISP operator or function name) and arguments. In their final code our programmers had 86 predicates and 75 arguments. One change-episode involved a predicate (with 8 expected by chance) while the other 14 involved arguments (7 expected by chance). This difference is significant (χ^2 [1, n=15)]= 13.10).

As judged by our task analysis of LISP and our programmers' LISP histories, the predicates were more standardized (and therefore had fewer associated planning goals) than the arguments. To support this assertion we looked in the programmers' final code for the labels that each had given to the same penultimate goal. For predicates there were ten cases where, if a penultimate goal was used, it was given the same label by all programmers (for 5/10 cases the label was explicitly mentioned in the problem statement). In contrast, for arguments, there were only three cases where the same label was used for the same penultimate goal (and 3/3 of these labels were explicitly mentioned in the problem statement).

This difference in specification of penultimate predicates and arguments is directly represented in the goal-structure hierarchy analyses. For example, in Figure 1, node# ***5432***, goal-label: ***pred_cond*** is a penultimate goal whose subgoal-template consists of the leaf goal ***cond***. The leaf goal is completely determined by the penultimate goal-label. In contrast, the penultimate goal at node# *543323*, goal-label: ***argument*** does not demand that its subgoal-template be ***queue***; presumably there are planning subgoals associated with this node that help select the label. This lack of specificity at the subgoal level exists despite the fact that the higher-order goal ***test_fails*** (node# ***54332***) is standardized to take exactly one argument that must occupy the third slot in its subgoal-template. Therefore the analysis of penultimate goals involved in change-episodes supports the notion that the more standardized a goal is (that is, the fewer planning subgoals associated with it), the less likely it is to be involved in change-episodes.

Higher-order goals. For higher-order goals we looked at the found versus expected frequencies of being involved in a change-episode for 15 higher-order goals that appeared at least four times in the programmers' final code. A χ^2 was significant (χ^2 [14, n= 61] = 118.42); hence, compared to what would be expected by chance, some higher-order goals were more likely and others less likely to be involved in change-episodes. (Note that as for the χ^2 reported above, this χ^2 was weighted by how often a particular goal appeared in the final code for all five programmers.)

Let's take a closer look at one high, one average, and one low probability higher-order change-goal. First is the high. The goal ***conditional*** was involved in 16 change-episodes with all five subjects contributing at least one episode. All except one of these change-episodes entailed either the deletion (6), insertion (3), transposition (3), or replacement (3) of conditional clauses. From our perspective, this is delightful. The conditional statement is an example of a LISP construct that has a semi-fixed internal structure. All conditionals have a ***Lparen***, a ***pred_cond***, one or more clauses, and a ***Rparen*** (see Figure 1). Within this rigid structure the number, type, and order of clauses can vary and it was these attributes that were involved in the 15 change-episodes. (For example, in the experimenters' solution to the problem, shown above, we used one conditional which included two conditional clauses. In contrast, none of our programmers coded their conditional(s) as we did. Two used two conditionals with two or three clauses per conditional. Two used one conditional with three clauses. And one used one conditional with the same two clauses we used, but in a different order. Although some are inelegant, none of these approaches are wrong.)

Because of the variability in how it can be used in LISP, the goal-structure for the conditional cannot contain just inherent goals, but must contain a large proportion of planning goals. The presence of planning goals is reflected by the frequency with which the conditional is involved in change-episodes.[4]

The conditional contrasts nicely with its main component, the **conditional clause** which was involved in an average (expected) number of change-episodes. As far as our programmers knew, all conditional clauses, regardless of their standardization, were composed of exactly four elements: **Lparen test action Rparen** (see Figure 1, node# *5433*). Presumably, fewer planning subgoals are attached as subgoals for conditional clauses than for conditionals.

An even better contrast is to compare the conditional with two standardized subgoals for the test component of a conditional clause: **test_fails** (see Figure 1) and **test_succeed**. The subgoal-template for test_fails consists of exactly four elements in an exactly specified order: **Lparen pred_null argument Rparen.** The subgoal-template for test_succeed consists of exactly five elements in an exactly specified order: **Lparen pred_found argument1 argument2 Rparen.** (Note whereas **pred_null** exactly specifies its subgoal-template, **null**, for our programmers **pred_found** could take one of a limited number of functions.) Test_fails and test_found occurred ten times in our programmers' final code; however, neither was ever involved in a change-episode.

Summary. The analysis of which goals are most likely to be change-goals corresponds well with our expectations about which goals have the highest ratio of planning to inherent subgoals. This finding holds for both penultimate and higher-order goals.

---

[4]This is not to say that more specific versions of **conditional** would never emerge. For example, with lots of practice in using the let/loop construction in coding search functions our programmers would probably develop a standardization of **conditional**, ***conditional_search.exits***.

How are Change-Goals Fixed?

As discussed earlier, the information contributed by the *fix* plays a large role in deciding which goal is the change-goal. Our two rules are ***Rule A**, changes in subgoal-template*, and ***Rule B**, a goal-label by any other subgoal-template.*

Table 3 summarizes which rule was used to identify the change-goal for each of our three goal types. (Because one change-episode was begun but never *fixed*, the sums add up to 104 change-episodes rather than 105.) For 94 out of 104 change-episodes the change-goal was identified using Rule A. This predominance of Rule A identifications holds for each goal type. For leaf goals it is true by definition. Leaf change-episodes are limited to typographical errors. Any correction that left the name the same would be an insertion, deletion, transposition or replacement. Any change in name would not be considered a leaf goal change. However, for both penultimate goals and higher-order goals Rule B can apply but usually does not. Fourteen out of 15 penultimate goals and 53 out of 62 higher-order goals are identified by Rule A.

Table 3

Rule used to identify change-goal for each type of goal.

	Leaf Goals	Penultimate Goals	Higher-order Goals	Totals
Rule A	27	14	53	94
Rule B	0	1	9	10
Totals	27	15	62	104

Rule A is the primary way of locating the change-goal. In these cases the fixed-goal is basically the change-goal with one or more of its subgoals replaced, deleted, transposed, or inserted. For example, correcting a misspelled word, or changing the order of two clauses in a **conditional** both result in the fixed-goal being essentially the same as the change-goal.

For Rule B, while the goal-label remains the same an entire subordinate level is either inserted or deleted. When a level is inserted then the fixed-goal is the parent (+1) of the change-goal. For example, earlier (see Figure 2) we discussed the case in which the programmer changed the argument from a list **storer** to the list with the first element removed (**cdr storer**). In this case the argument went from a penultimate goal (node# *544444*) with subgoal-template *storer* to a higher-order goal with subgoal-template: ***Lparen predicate argument Rparen.*** One subgoal of this new subgoal-template, ***argument*** (node# *5444443*) had as its subgoal-template *storer*. If the fix had gone the other way, from (**car storer**) to **storer**, then the fixed-goal would have been the child (-1) of the change-goal.

For the 10 change-episodes that required Rule B, a closer analysis revealed that the cases in which the fixed-goal was the parent (superordinate) of the change-goal were qualitatively different from the cases in which the fixed-goal was a child (subordinate) of the change-goal. Deletion of a subordinate level was the case for three of the 10 Rule B change-episodes. In two of these, previously correct code was changed into an obvious (to us anyway) error. In the third, an obvious error was corrected.

The situation for the seven *parent* change-episodes was very different. In all seven cases, correct code existed that represented one subgoal in an otherwise missing super. During the change-episode the rest of the super was built up around the one subgoal. An example was the goal for updating the local variable. The subgoal-template for the update was: ***Lparen pred_setq var_name update_body Rparen.*** In a parent change-episode the change-goal, ***variable_update***, was initially coded as just the ***update_body***. During the change-episode the mistake was noticed and the rest of the subgoals for *variable_update* were added.

These are examples of the local goal-structure being *transformed* rather than *unpacked*. (Such transformations have been noticed during debugging by Katz and Anderson [12].) Our interpretation is that these represent relatively uncompiled, unstandardized, goal-structures in which the left-to-right order information is overshadowed by the planning subgoals. Hence depth-first coding of certain subgoals occurs before the coding of more leftward subgoals. The planning goals result in a different order of coding than the hierarchical structure of LISP would dictate, but probably eases the working memory load in that a key part of the goal is coded and the rest can be built around it.

Summary. Our hope for a small number of fix categories was fulfilled. All 104 fixed change-episodes could be placed into one of two fix categories. Perhaps more astonishing was that 90% (94/104) of the fixes were accounted for by just one category, Rule A. Rule A accounts for minor changes (additions, deletions, insertions, or replacements) in the fixed goal's subgoal-template. These findings support the hypothesis that most planning occurs at planning subgoals that are directly attached to the change-goal's subgoal-template.

When are Changes Made? A Taxonomy of Noticing Events.

What is the programmer doing immediately before s/he decides, or ***notices***, that something already coded needs to be changed? The protocols suggest that most of these noticing events (102/105) occur either as an interruption to coding (53), during or immediately after another change-episode (a tag-along) (24), or as an outcome of symbolic execution (25). While we intended these categories to be exhaustive, it is interesting that the majority of change-episodes occur as interrupts and not as the result of the more deliberate processes involved in symbolic execution. (These data increase the importance of determining whether interrupts occur randomly during coding or whether there exists a systematic relationship between the goal that was interrupted and the goal that was changed.)

We looked to see if the three goal types were equally likely to be noticed during the three main types of noticing events. First we compared the expected versus found distribution for leaf goals versus all higher goals (higher-order plus

penultimate goals). The χ^2 was significant ($\chi^2[2, n=28] = 9.59$) indicating that leaf goals were more likely to be noticed by interrupts (22 cases found versus 14 expected) than during tag-alongs (5 found, 7 expected) or symbolic monitoring (1 found, 7 expected). There were no significant differences in the found versus expected distribution for penultimate versus higher-order goals.

Table 4
Distance & relationship of the interrupted goal to leaf, penultimate, or higher-order change-goal.

Distance	Relation	Change-Goal Type			Totals
		Leaf Goal	Penultimate Goal	Higher-Order Goal	
+3	uncle	---	---	1	1
+3	nephew	---	---	1	1
+2	grandparent	---	---	1	1
+2	brother	---	---	---	0
+1	parent	---	---	---	0
0	self	22	1	7	30
-1	child		3	6	9
-2	grandchild			8	8
-3	greatgrchild			3	3
	Totals	22	4	27	53

Interrupts: Stopping coding to make changes. In support of our hypothesis, the data show that the interrupted goal was more likely to be either the change-goal itself or a subordinate of the change-goal than a superordinate or sibling (that is, at the time of the noticing event the programmers were coding part of the structure that they ended up changing). These data are shown in Table 4 where negative numbers indicate the number of subordinate links from the interrupted goal to the change-goal and positive numbers indicate the number of superordinate links. For example, from Figure 1, if typing ***null*** (node# *5433221*) was interrupted to change ***test_fails*** (node# *54332*), the interrupted goal would be a grandchild (-2 links distance) of the change-goal. In contrast, if the ***conditional*** goal (node# *543*) was interrupted to change ***test_fails*** (node# *54332*) the interrupted goal would be a grandparent (+2 links distance) of the change-goal. Note that regardless of the direction of the super- or sub-ordinate relationship, most of the interrupted goals were very close to the change-goals; that is, there were very few links between the two. Of the 53 interrupts 30 occurred at distance 0, 9 at distance 1, 9 at 2, 5 at 3, and none at a distance greater than 3 links from the change-goal.

Table 4 indicates that a piece of code was changed only immediately after it was coded (Table 4, distance: 0, relationship: self) or while one of its subgoals was being coded (Table 4, distance: -1 to -3, relationship: child, grandchild, or greatgrandchild). Programmers seldom interrupted their coding to change an already completed goal. This was true despite the drastically restricted range of descendents for leaf and penultimate goals. For example, while the penultimate goals lack grandchildren, and so on, they do have a parent, grandparent, possibly brothers, uncles, and more.

This finding is both very interesting and disappointing for the human intellect. There must be a close proximity between the change-goal and the current activity for change to occur. Programmers will seldom interrupt their coding to change an already completed portion of code. When coding is interrupted to make a change it is either immediately after the code was completed (self) or, if before completion, during work on one of its subgoals. Changes made to completed code result from symbolic execution (discussed below).

Changes initiated by interrupts appear to be part of the planning process. Programmers apparently begin work on a goal before it is completely planned. The early work may serve to help programmers remember the goal-structure of the plan (2) without implying a commitment to particular details. As programmers begin to implement subgoals of the plan, implications are realized that cause the higher-order goal to be revised. Once the details of a plan are worked out in code and the programmer moves on either to finishing work on a superordinate or to working on a sibling goal, spontaneous interrupts seldom result in code revisions.

Tag-alongs: The change-episode as noticing event. Twenty-five change-episodes were initiated during another change-episode. Of these 16 involved the same change-goal as the immediately preceding change-episode with no other events intervening. Five of these 16 involved the repeated misspelling and correcting of the word (leaf goal) that was the change-goal in the prior change-episode. Five involved the programmer either renaming a parameter or variable, or apparently slipping and first typing (or saying) one argument name, then another, and so on (penultimate goals). The other 6 involved problems that two of our programmers had in deciding how many parentheses were required in a variable declaration statement when a single variable is being initialized (a higher-order goal). Examination of the textbook (9) revealed that out of 27 examples of such a statement, 3 different methods were used with no method dominant. We conclude that our programmers' methods of initializing variables were in an intermediate stage of compilation in which they knew that different methods existed but did not know when to apply one rather than another.

Each of the other 9 tag-alongs was noticed during another change-episode but involved a different change-goal. Within this group no patterns were noticed.

Symbolic execution. In 24 change-episodes there was clear evidence that the programmers had been symbolically executing their code immediately prior to *noticing* the need to change a previously coded (or stated) goal. In all cases the evidence for symbolic execution came directly from the verbal protocols.

Almost by definition, there should be a longer interval between the noticing event and the immediately prior keystroke for symbolic execution than for either interrupts or tag-alongs. In the three categories, the mean time in seconds between last keystroke and noticing event was 52.5 (symbolic execution), 2.2 (interrupts), and 4.0 (tag-alongs). An ANOVA (unweighted for unequal numbers of entries) finds these differences significant (F [2, 99] = 41.48). Orthogonal comparisons show that all of this difference is due to the symbolic execution versus rest comparison (F [1, 99] = 82.89) and none due to the interrupt versus tag-along ($F < 1$).

Summary. Change-episodes are initiated by noticing events that occur either as an interruption to coding, as a tag-along to another change-episode, or as an outcome of symbolic execution. Interrupts occur at close proximity to the change-goal and either immediately after the change-goal was completed (self) or during work on one of its subgoals. Changes made to completed goals when other goals intervene between completion and change are the result of symbolic execution.

SUMMARY & CONCLUSIONS

Change-episodes, a prominent feature of our programmers' behavior (averaging one per min), seem to represent key junctures in the process of coding. As such they cannot be ignored by theories of coding. Here we summarize our major findings and discuss possible directions and applications of our work on change-episodes.

Importance of Change-Episodes to the Cognition of Programming

Overall the most important finding was the systematicity in which goals were change-goals, when the need for a change was noticed, and how the change occurred (the fix). Such systematicity argues for the role and importance of the heretofore ignored issue of evaluative activities (6) in programming. In addition, the particular pattern of findings supports the distinction between inherent and planning goals (3) and the placement of planning goals at the level of the goal-structure for which planning occurs.

For both higher-order and penultimate goals, those least likely to be the target of a change-episode were those which, for a priori reasons, we believed had a low ratio of planning to inherent subgoals (3). Conversely, those most likely to be targeted, such as ***code a conditional*** were those for which many different (and legal) instantiations exist. It seems reasonable that our programmers required many planning subgoals to code these goals. Our inferenc[illegible] is that the more planning involved, the more likely a change-episode will occur.

One of the surprising findings was how easy it was to identify most change-goals by looking for relative[illegible] minor changes in a goal's subgoal-template (Rule A). This finding supports the hypothesis that most planning [illegible]curs at planning subgoals that are attached directly to the change-goal. The exceptions not captured by this fix categ[illegible] were few. Ten out of 104 change-episodes involved either deleting or inserting an entire hierarchical level (Rule B). [illegible]or this minority of cases, left-to-right syntactic information seemed lacking and programmers performed a depth-first expansion of rightward subgoals before coding the more leftward ones. The change-episode transformed the goal-structure rather than just rearranging or editing it (see also [12]).

Another surprise was that most (102/105) change-episodes occurred in one of three very distinct circumstances: as an interrupt to coding, as a tag-along to another change-episode, or as a byproduct of symbolic execution. Out of 53 interrupts, 50 occurred either right after the change-goal had been coded or after work on a direct descendent of the change-goal. We characterized this finding as disappointing for the human intellect. There must be a close proximity between the change-goal and the current activity for change to occur. Programmers almost never interrupt their coding to change an already completed portion of code. Changes made to completed code are the result of symbolic execution.

Goals which are unstandardized, either because they have been recently acquired or because of the many different ways in which they can be used, contain a high ratio of planning to inherent subgoals. These high ratio goals are most likely to be the target of change-episodes. Change-episodes are a type of evaluative activity that takes the programmer away from the progressive activity of coding. Understanding the role of such evaluative activities in programming, how they are used and how they are acquired, is a necessary but neglected aspect of the study of programming.

In conclusion, we believe that the issue of change-episodes in coding is a good problem to work on. It is good because change-episodes are very common in coding and seem to represent key junctures in the coding process. Understanding the cognitive processes involved in change-episodes has direct relevance to any theory of coding, implications for debugging, comprehending, and designing programs, as well as for progressive and evaluative activities in other cognitive tasks.

REFERENCES

1. Newell, A. (1980). Reasoning, problem-solving, and decision processes: The problem space as a fundamental category. In R. Nickerson (Ed.), **Attention and performance VIII.** Hillsdale, NJ: Erlbaum.
2. Anderson, J. R., Farrell, R., & Sauers, R. (1982). **Learning to plan in LISP** (Tech. Rep. No. ONR-82-2). Pittsburgh, PA: Carnegie-Mellon University.
3. Anderson, J. R., Farrell, R., & Sauers, R. (1984). Learning to program in LISP. **Cognitive Science, 8,** 87-129.
4. Anderson, J. R. (1982). Acquisition of cognitive skill. **Psychological Review, 89**, 369-406.
5. Anderson, J. R. (1983). **The architecture of cognition.** Cambridge, MA: Harvard University Press.
6. Allwood, C. M. (1984). Error detection processes in statistical problem solving. **Cognitive Science, 8,** 413-437.

7. Kant, E., & Newell, A. (1984). Problem solving techniques for the design of algorithms. **Information Processing & Management, 20**, 97-118.
8. Anderson, J. R., & Reiser, B. J. (1985, April). The LISP tutor. **Byte**, pp.159-175.
9. Anderson, J. R., Corbett, A. T., & Reiser, B. J. (1987). **Essential LISP**. Reading, MA: Addison-Wesley Publishing Company, Inc.
10. Ericsson, K. A., & Simon, H. A. (1985). **Protocol analysis: Verbal reports as data.** Cambridge, MA: MIT Press.
11. Gosling, J. (1981). **Unix EMACS user manual.** Pittsburgh, PA: CMU Computer Science Department.
12. Katz, I. R., & Anderson, J. R. (1986). **An exploratory study of novice programmers' bugs and debugging behavior.** Presented at the First Annual Empirical Studies of Programmers Conference (July). Washington, DC.

AUTHOR NOTES

The research was supported by a Secretary of the Army Science and Engineering Fellowship, a Sloan Foundation Fellowship, and the gracious hospitality of the Psychology Department at Carnegie-Mellon University to the first author. Support for the second author was provided by ONR contract N00014-87-k-0103. Nothing in this paper should be construed as representing official Army policy. Request for reprints should be sent to: Wayne D. Gray, Army Research Institute, PERI-IC, 5001 Eisenhower Ave, Alexandria, VA 22333 or gray@ari-hq1.arpa.

We wish to thank A. Corbett and M. Lewicki for their cooperation and assistance in collecting the protocol data. Also B. John and A. Newell for the use of the Users Studies Laboratory, and L. Reder for space and equipment. Thanks also to F. Conrad, A. Corbett, B. John, I. Katz, and C. Kessler for their comments on an earlier draft. Thanks to H. Simon for advice on encoding verbal protocols and to C. Fisher for assistance in formatting and printing.