Palo Alto Research Center

# Diagnosing Bugs in a Simple Procedural Skill

By Richard R. Burton

XEROX

COGNITIVE AND INSTRUCTIONAL SCIENCES SERIES
CIS-8 (SSL-80-10)

# Diagnosing bugs in a simple procedural skill

Richard R. Burton

March 1981

XEROX
PALO ALTO RESEARCH CENTER
Cognitive and Instructional Sciences Group
3333 Coyote Hill Road / Palo Alto / California 94304

Diagnosing bugs in a simple procedural skill

Richard R. Burton

**Abstract**

To account for student errors in simple procedural skills, Brown and Burton (1978), proposed "the Buggy model." In the model, a student's errors are seen as symptoms of a "bug," a discrete modification to the correct skills which effectively duplicates the student's behavior. This paper describes an operational diagnostic system based on the Buggy model that has been used with several thousand students over the last two years to find systematic errors in the domain of place-value subtraction. Subtraction is used as a paradigmatic case for studying diagnosis; simple enough that it is possible to diagnose real students and complex enough to provide an interesting case study of the subtleties of diagnosing students in a natural setting.

A brief overview of the Buggy model and a framework for diagnostic systems using the model is presented. This provides both a framework for understanding the final diagnostic system and an appreciation of the subtleties of diagnosis. The complicating factors in diagnosis are presented and solutions to some of the complications are discussed.

Two diagnostic systems are described; one for dealing with standard tests and one for interactive diagnosis. The problems particular to each type of diagnosis are examined.

A definition of "subskill" (what a student must know to perform a skill) is derived from the set of observed bugs. Some uses of subskills are described and all of the subskills of subtraction are given.

To appear in *Intelligent Tutoring Systems*, Sleeman, Derek and Brown, John Seely, eds., Academic Press, 1981.

Diagnosing bugs in a simple procedural skill

Richard R. Burton

## 1.0 Introduction

To account for student errors in simple procedural skills, Brown and Burton (1978), proposed "the Buggy model." In the model, a student's errors are seen as symptoms of a "bug," a discrete modification to the correct skills which effectively duplicate the student's behavior. As an example, consider the following two errors made by a student in the procedural task of subtraction:

| | |
|---|---|
| 500 | 312 |
| -65 | -243 |
| 565 | 149 |

Both errors are accounted for by the bug named **0-n=n**. (Footnote: Bug names appear in bold type.) The student is using a modification to the subtraction procedure which dictates that when the top digit in a column is 0, write the bottom digit as the answer for that column. Modifications may be the result of deleting part of the correct procedure, of adding incorrect sub-procedures, or of replacing correct subprocedures by incorrect ones.

This chapter describes an operational system based on the Buggy model that has been used with several thousand students over the last two years to diagnose systematic errors in the domain of place-value subtraction. For the purposes of this chapter, subtraction is used as a paradigmatic case for studying diagnosis. Subtraction is simple enough that it is possible to diagnose real students in a natural setting and complex enough to provide an interesting case study of the subtleties involved. Initially the diagnostic system had a fixed, prespecified set of bugs, and it simply chose those bugs that maximally accounted for the student's behavior. As the range of erroneous behaviors to be diagnosed expanded, the space of possibilities became too large for these exhaustive search techniques. Also, many students have subtle variations of the 'taught' methods, make careless mistakes while following their correct or faulty procedures, and switch back and forth between old and recently-learned procedures. All of these factors complicate the diagnostic process and set the stage for the system to be described in this paper.

The first section presents a brief overview of the Buggy model and characterizes what we mean by diagnosis. Then the framework for diagnostic systems is presented using a simple system as an example. This provides both a framework for understanding the final diagnostic system and an appreciation of the subtleties of diagnosis.

The second section describes the complicating factors in diagnosis and indicates why the simple system must be extended. This discussion also mentions solutions to some of the complications.

The third section describes the diagnostic system that is used when students are given a pre-specified test and the methods for dealing with the complications. The interactions between the heuristics of the system and the items on the test are examined. An interactive version of the diagnostic system is presented and the areas of problem generation and logical equivalence are considered.

The fourth section presents a definition of "subskill" (what a student must know to perform a skill) that is derived from the set of observed bugs. Some uses of subskills are described and an example of each of the subskills of subtraction is given.

### 1.1 Procedural Network Model

When we set out to explain the causes of observed student errors, we knew a representation was required, but we did not know how complex the representation would need to be. We did not know what primitives or control structures would be appropriate so, in an attempt to avoid having our language shape the way we saw the data, we chose a representation language which allowed us freedom to capture each newly observed set of student errors in the way that seemed most appropriate to those errors. Thus our initial representation scheme was *ad hoc by design.*

The representation scheme distinguishes between goals and methods for satisfying those goals. For each goal there is a correct method of achieving that goal and, optionally, alternative correct methods and alternative incorrect or "buggy" methods. Methods are written in a stylized subset of the programming language Lisp. The most frequent construct in the implementation of the methods is a call for other goals to be satisfied (i.e., breaking a goal down into subgoals). This led to the formation of "procedural networks" (Sacerdoti, 1977) which represent the goal-calling structure, the associated conditionals, the primitives and other information about the intention of the procedures along with diagnostic suggestions. All this information is represented in the form of a network. The network, however, is not meant to be a cognitive construct, but simply a framework for relevant pieces of information.

The observed data is represented in the network as procedures (methods and their variants) which mimic the behaviors of the students. To guarantee some generality (so that not every student could be hypothesized as having a different bug unrelated to any other bug or to the correct skill) the representation scheme was constrained in two ways. First, if a student was executing part of the procedural task correctly, the scheme's representation of that part of the student's behavior had to be the same as it was for a correct student. A bug, in other words, must be the least possible variant of the correct skill that mimics the behavior.

The second constraint was to distinguish between primitive and compound bugs. If we found a student who had behavior A and another who had behavior B and a third whose behavior was the combination of A and B, then this third behavior must be modelled in the representation as the composition of the bug A with the bug B. These two constraints led to numerous reworkings of the decomposition of the skill into subskills.

Footnote: From a psychological point of view, the procedural network representation is a *descriptive* tool. In particular, we are not proposing the procedural network as an alternative to, for example, the class of production rule models which try to model internal mental processes of the level of working memory. Our goal is to characterize the entire class of data we observe in a form that facilitates comparing descriptions of the data without worrying about underlying mechanisms. It seemed premature to focus on the possible mechanisms of a particularly well-formed subset of the data without first characterizing the entire range of phenomena.

## 1.2 Diagnosis

There are many possible meanings of the term diagnosis, many different methods of testing or determining a diagnosis, and many different purposes to which diagnoses are applied. (Diagnosis is often referred to in some work as developing a model of the student.) We shall discuss the range of possible meanings and choose the one most pertinent to our work. The *dimension* of this range is a measure of the depth of understanding of what the student is doing.

The simplest level of diagnosis is determining whether or not a student has mastered a skill, and possibly the degree of mastery, represented by a numeric value. This level of diagnosis is not particularly useful for remediation, since one does not have any idea what parts of the skill need more work. It is interesting to note that without a model of the skill being tested, even determining that the student possesses the skill is hard. A student may have an almost arbitrarily complex perversion of the skill and produce the right answers for most typical problems. But without a good model, it is difficult to determine how much the boundary conditions of the skill have been stressed. If a test includes only examples that the student gets right, it will always be concluded that s/he has mastered the skill. Footnote: Usually the determination of what problems to include (i.e., what possible mislearnings are detected by the test) is left to test-makers' intuitions of the kinds of mistakes they have seen other students make. A model of the procedural skill can provide insight into what the possible ways of mislearning a skill are and thus aid in the creation of diagnostic tests.

A more complicated level of diagnosis is determining what subskills the student has not mastered (Friend, 1980). This technique has useful applications for both remediation and coaching/tutoring. The WEST coaching system ("An investigation of computer coaching for informal learning activities," Burton and Brown, this volume) presents a system that determines which subskills a student does not know and uses that information to provide and control tutorial feedback. This level of diagnosis has its own set of problems that will not be touched upon until Section 4.3, in which a theoretical basis for formalizing the notion of subskills will be advanced.

The level of diagnosis appropriate for this paper is determining what internalized set of incorrect instructions or rules gives results equal to the student's results. **That is, we require of a diagnosis that it be able to predict, not only whether the answer will be incorrect, but the exact digits of the incorrect answer on a novel problem.** This means having a model of the student that replicates behavior on the problems observed so far and that *predicts behavior* on problems not yet done. It is clear that we require a model of the student's encoded skill to be sufficiently complete that it can be executed on new problems.

A usage of the term diagnosis that is still more specific than the above is to determine what has *caused* the student to develop his incorrect procedures. At present, diagnoses at this level are not very well understood and are beyond the scope of this paper. Brown and VanLehn (1980) have developed a principled theory that explains many of the observed bugs.

### 1.3 Framework for the Diagnostic Process

In the next section we describe some of the complexities that a diagnostic system faces with real students. We begin by introducing, and exploring the shortcomings of, a simple prototypical system for diagnosing idealized student behavior using an executable model of the possible bugs. We shall direct our discussion by way of examples in subtraction. Many of these complications occur in other domains, but subtraction provides a particularly clear illustration of the problems.

#### *1.3.1 A core but overly simple system: Naïve Diagnostic System:*

The leverage provided by the procedural network model is an efficient means for predicting the answer to any problem from any bug. The Naïve Diagnostic System compares the student's answers with the output of each bug run through the test and picks the *one* bug that gives the same answers as the student. (We will later discuss why there might not be exactly one.) Since there are only about a hundred and ten primitive bugs for subtraction (that we have identified so far), the computation is not overwhelming.

Figure 1 presents a sample "diagnostic table" that might arise from such a system for an idealized student. The diagnostic table provides a summary of how well a bug matches the student's responses on the given set of problems. The test problems with the correct answers appear at the top of the table. The student's answers appear on the next line using the convention that "+" indicates a correct student answer. Each of the remaining pairs of lines provide the name of a bug and the answers produced by it under the assumption that the student had this and only this bug.

For each of these lines a "***" means that the bug predicted the student's *incorrect* answer. A "*" means that the bug in that row would give the correct answer and also that the student got the correct answer. Thus "*" and "***" indicate agreement between the student's behavior and the predicted behavior of the bug, and positive evidence that the student has that bug. Negative

evidence is portrayed in two ways: A "!" means that the bug in that row predicts the correct answer but the student gave a wrong answer. The number predicted by the bug is shown if it disagrees with both the student's and the correct answers.

Figure 1 shows a student who has the bug mentioned in the introduction, **0-n=n**. Also shown is the entry for the bug "the smaller digit in each column is subtracted from the larger regardless of which is on top" (called **smaller from larger.**) This bug generates the same answer as the student on the fourth problem but is not an acceptable diagnosis because a diagnosis must account for the entire set of student behavior.

Figure 1

Student diagnostic table with bug **0-n=n**.

| Problems and Correct Answers | | | | |
|---|---|---|---|---|
| 45 | 40 | 139 | 500 | 312 |
| -23 | -30 | - 43 | - 65 | -243 |
| 22 | 10 | 96 | 435 | 69 |
| **Student Answers:** | | | | |
| + | + | + | 565 | 149 |
| **0-n=n:** | | | | |
| * | * | * | *** | *** |
| **smaller from larger:** | | | | |
| * | * | 116 | *** | 131 |

*** - bug predicts student's incorrect answer
 * - bug predicts student's correct answer
 ! - bug predicts the correct answer, student's answer incorrect answer
 + - student's answer is correct

This diagnostic system can go wrong in two ways: there may be more than one bug that exactly matches every answer given by the student; or there may be no bug at all that matches. If there is more than one bug and the system is interactive, it can attempt to generate a new problem for the student so that it can distinguish between the bugs. (Finding such a problem can often be extremely difficult, as will be discussed in section 3.3.2.) If the student is not on-line (as is the case with standardized tests), the test given the student must be redesigned. Footnote: There are two properties that the test must have in order to prevent such ambiguous diagnoses from arising. One is to have, for each of the hundred bugs, at least one problem which causes a symptom for each bug. In the case that the student has only one bug, this bug can thus be distinguished from the correct procedure. The second property the test must have is to contain problems that distinguish between any two of the hundred bugs. For example if the test in Figure 1 included neither the third problem nor the fifth problem, it could not be determined whether the student had **0-n=n** or **smaller from larger**. This does not mean that a diagnostic test need contain several hundred problem. Problems can be generated which test more than one bug at a time, producing distinct incorrect answers. For example, the tests we use in our data collection have these properties and contain only 20 problems. The problem of test construction will be addressed later.

The second way the simple diagnostic algorithm can go wrong is by having the student's behavior not emulated by any bug. In these cases, no bugs match all of the student's answers. With real students, there are many ways that this can happen. We will continue by examining some of them.

## 2.0 Complicating Factors in Diagnosis

### *2.1 Performance lapses:*

The diagnostic situation is complicated by the fact that students sometimes really do make mistakes from fatigue, laziness, boredom, or inattention. This happens while they are following a buggy procedure as well as while they are following a correct one. A closely related problem is that students sometimes do things for reasons that are beyond the limits of the theory being used to determine consistency. For example, they may copy some answers from their neighbors or get digits for their answers from nearby problems. This brings into question the notion of a consistent error: A student who always makes the rightmost digit in the answer be the same as the date is being completely consistent in a sense we have not included in our database. Consistency in the sense that we use it implies that the student will always give the same answer to the same problem and that the answer is completely determined by the problem alone.

To be of practical use, a system must find a student's systematic errors even in those cases where some non-systematic errors are present. This means that the diagnostic system cannot insist on exact matches between its proposed bugs and the student's behavior. The system must find the best match and then decide if it is really good enough to be a diagnosis. One major ramification of this fact is that hypotheses about a student's bug can not be rejected with the first piece of evidence to the contrary. This rules out a major class of diagnostic techniques which use problems that very few hypotheses explain to quickly reduce the space of possibilities.

### *2.2 Errors in primitive subskills:*

Another source of inconsistency, or "noise" as we shall refer to it, is that specialized subskills that are assumed to be primitive in the model may, in fact, have errors. This can be seen in subtraction in the form of errors in the standard subtraction facts table such as 13-7=8. One "solution" to this problem is to consider each fact to be a separate subskill and to consider all of the possible values of, for example, 13-7 except 6 as being bugs. Footnote: Alternatively, one could postulate a model for doing "facts" and include this in the model of subtraction. One has then just pushed the problem down a level by assuming a different level of primitives which might themselves not be completely correct. This approach has several problems. It is not applicable in the common case where facts table errors are not consistent. It is unconfirmable in practice because a typical test of 20 items requires about 60 subtraction facts to be used. Since there are 100 subtraction facts, not every fact can be tried once, much less enough times to determine consistency. This solution also has the drawback that almost any answer can be "explained" as a compounding of facts table errors which could swamp the diagnostic system with complex alternative theories.

### *2.3 Subskill variants:*

Another source of noise is that repeated drilling on parts of the skill can lead to variants of the "standard" algorithm that cause the student to do correctly parts of problems that the bug predicts he would miss. For example, ten is learned as one of the subtraction facts by repeatedly solving exercises of the form 10-y. Based on this experience, students sometimes perceive the leftmost one and zero digits of a number (e.g., 1093) as being the single "digit" ten rather than as two digits, hence hiding cases where a bug such as 0-n=0 might otherwise arise. Another example arises because students are drilled on two-column problems requiring borrowing—they will often know how to borrow correctly in the units column of a problem but have a bug in the borrowing procedure in the remaining columns.

### *2.4 Compound Bugs:*

Students often have mislearned more than part of the skill, and thus have a combination, or compound, of two or more primitive bugs. In a recent experiment with approximately one thousand third, fourth and fifth grade students, thirty-seven percent of the diagnoses were compound bugs (VanLehn, 1981). A straightforward way for the simple system to diagnose these students is for it to consider subsets of primitive bugs up to some cardinality. We have observed students with as many as four bugs in a single skill (see figure 2). With 110 primitive bugs, this leads quickly to more hypotheses than can be feasibly examined (approximately $10^8$) by exhaustive search. For the remainder of this section we explore heuristics to reduce the space of bugs, and describe the heuristics for searching the reduced space efficiently.

Figure 2

Student diagnostic table with a compound bug

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 31 | 53 | 23 | 2003 | 203 | 1043 | 520 | 10214 | 909 | 10300 | 70001 |
| -11 | -20 | -8 | - 24 | -109 | -505 | -467 | -8375 | -458 | - 546 | - 391 |
| 20 | 33 | 15 | 1979 | 94 | 538 | 53 | 1839 | 451 | 9754 | 69610 |

Student Answers:

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| + | 30 | 5 | 1 | 106 | 2 | 140 | 161 | 501 | 200 | 0 |

**smaller from larger & O-n=O & n-O=O & stop working when the bottom number runs out:**

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| * | *** | *** | *** | *** | *** | *** | *** | *** | *** | *** |

**n-O=O:**

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| * | *** | ! | ! | 4 | 508 | ! | ! | ! | ! | ! |

**stop working when the bottom digit becomes a blank:**

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| * | ! | *** | 79 | ! | ! | ! | ! | ! | 754 | 610 |

**smaller from larger:**

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| * | ! | 25 | 2021 | *** | 1542 | 147 | 18161 | 551 | 10246 | 70390 |

**O-n=O:**

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| * | ! | ! | ! | ! | 1038 | 60 | 1909 | *** | 9800 | 70000 |

Diagnosis: a consistent bug = {**smaller from larger & O-n=O & n-O=O & stop working when the bottom number runs out**}.

*** - bug predicts student's incorrect answer
 * - bug predicts student's correct answer
 ! - bug predicts the correct answer, student's answer incorrect answer
 + - student's answer is correct

The diagnosis is a compound of the bugs **smaller from larger, O-n=O, n-O=O** and **stop working on a problem when the bottom number runs out.** The evidence for and against the individual primitive bugs is also included.

*2.4.1 Reducing the space of compound bugs:*

Many of the arbitrary compound bugs, or compounds, will be logically equivalent in the sense of producing the same behavior. If only one member of each class of logically equivalent compounds is examined, the search space is reduced.

One grouping that suggests itself is to ignore the order of the bugs within the compound; so that the compound of bug A with bug B, denoted as {A B}, is the same as the compound of bug B with bug A, {B A}. This can easily be accomplished by canonicalizing the compounds, for example, by alphabetizing the component letters. There is, however, a useful interpretation that suggests itself for the difference between {A B} and {B A}; that being whether the correct model was perturbed by bug B then the result perturbed by bug A or vice-versa. Under this interpretation, the ordering indicates the order of perturbation in the model. (This is analogous to the decision in a production

system of the order in which rules A and B are tried.) In many cases the bugs are independent so that the order of perturbation makes no difference. But independence is not always the case. In some cases, the preconditions of one of the bugs can subsume the preconditions of the other, preventing it from ever being tried. For these cases, the standardization must put bugs in order of decreasing specificity. For example, the bug **0-n=0** would be ordered before smaller from larger.

Another observation about grouping compound bugs into equivalence classes is that some bugs "hide" other bugs in the sense of preventing their preconditions from ever being met. In these cases, {A} is logically equivalent to {A B}. For example, the bug **smaller from larger** will hide any bug in the borrowing procedures so that their compound will be equivalent to **smaller from larger** alone. This is because **smaller from larger** redefines the column subtracting procedure so that it never needs to borrow; any bug that has **trying to borrow** as one of its preconditions will never be tried.

Canonicalization fails entirely in a few cases where bugs have overlapping preconditions which prevents there from being a single "right" order. Two orderings may be different from each other and different from any of their subsets. For example, the bug **when borrowing from a column in which the top and bottom digits are the same, borrow from the next column to the left instead** and the bug **when borrowing from a column with a zero as the top digit, change it to a 9 and return** have preconditions that overlap in problems requiring a borrow in a column which has zero over zero (e.g., 305-108). Thus {A B} is actually different from the bug {B A}; the first gets 107 for the answer while the second gets 297. Our representation makes the distinction between these two bugs so that both are representable. Footnote: The diagnostic compounding mechanisms use a canonical ordering routine so that this difference would not be "discovered" by the system itself. (This problem presents itself in production system formalisms in the form of ordering of rules in the rule set. One consequence of this for diagnostic systems based on such production rules is that their diagnosis must, in some cases, include not only what the buggy rules are but also how they are ordered.)

There are also cases in which different combinations of bugs are the same (generate the same results on all problems) even though they are made from different primitives and can not be grouped for any principled reason. These cases are rare but do necessitate that the system be able to deal with multiple diagnoses. This also creates the additional annoyance that after determining that two bugs are the same, the system must then decide which is a more appropriate label for the diagnosis.

### *2.4.2 Heuristics to limit the search:*

While the number of hypotheses to be searched is greatly reduced by embedding knowledge about hiding and canonical ordering, the space is still too large to search exhaustively. One standard approach to limiting combinatorial search is to search only those compound theories for which there exists support in the data (a variant of "bottom-up" searching). The system limits its search to only

combining primitive bugs for which evidence has been found. For this approach to succeed in finding a compound bug {A B}, there must be evidence for A and B separately. We will address the ramifcations of this limitation and methods for circumventing them in a later section.

Care must be taken when searching the space of compounds because primitive bugs do not always combine in simple ways. There are situations in which a bug will interfere with another bug in such a way as to cause the student to get the right answer on problems where one of the bugs would predict the occurrence of a wrong answer. For example, on the problem 313-208, a student who has both the **smaller from larger** bug and the **n-0=0** bug will get the right answer, 105. A student with the **smaller from larger** bug only will get 115. This has the important ramification for the diagnostic system that *it cannot eliminate a hypothesis just because it predicts that the student would miss a problem that he gets right.*

Another complexity of bug combination is that two bugs that generate the same answers on all problems can each combine with the same third bug and generate different behavior. Two bugs generating the same behavior occur in the situation of needing to borrow from a column which has a zero on top. One bug **increments the zero to one instead of decrementing it** {A} and the other bug **decrements the bottom digit instead of the top** {B}. When each of these bugs is compounded with the bug **0-n=n** {C}, however, we get different behavior. In the problem 304-159, the first bug compound {A C} will get 165 for an answer (because incrementing the zero has disabled the **0-n** condition), while the second bug compound {B C} will get 245. Footnote: The situation here is actually slightly more complicated. The two bugs actually can be distinguished in the case where the digit below the zero is also zero or blank. In this case, the bottom zero can not be decremented. However, if each of the bugs is compounded with the bug **ignore columns in which there is a zero on top and a zero or a blank on the bottom** {D}, the compounds {A D} and {B D} are equivalent. When these compounds are compounded with **0-n=n**, the resulting triples {A C D} and {B C D} are distinguishable again.

Allowing bugs in an executable model to be arbitrarily combined may result in the implementation problem of loops. For example, if the primitive bug of **moving from left to right instead of right to left** is not implemented very carefully, it can loop when combined with a bug in the recursive borrowing procedure that has built into it the notion of moving from right to left. These problems, once discovered, can be overcome by reimplementing the offending bugs.

# 3.0 The DEBUGGY Diagnostic System

## 3.1 Method of diagnosis

In this section we will describe an extension of the Naive Diagnostic System. First we consider the off-line version, called DEBUGGY, then the interactive version, called IDEBUGGY. From each student the system has the answers for every problem on the test (and the test problems themselves). At a high level, DEBUGGY compares the student's answers with results of the primitive bugs, looks for combinations of the bugs that compare favorably, tries to explain the noise in the resulting set of compound bugs, and then chooses the best and decides if it is good enough to be a diagnosis. The details of these steps are discussed below.

### *3.1.1 Forming the initial hypothesis set:*

DEBUGGY begins by considering a fixed set of hypotheses that includes all of the primitive bugs and roughly 20 common compound bugs. (See Friend and Burton (1981) for a description of the 110 primitive bugs.) The results of all 130 bugs are compared with the student's answers. This comparison is used to determine the subset of the known bugs which will become the *initial hypothesis set.* The bugs in this subset form the basis for the compounding heuristics. The initial hypothesis set contains any bug that explains *at least one of the student's wrong answers,* and has what we refer to as full-problem evidence, meaning each has correctly predicted the student's wrong answer on at least one problem.

An alternative to full-problem evidence for inclusion in the initial hypothesis set is single column evidence which explains only a single incorrect column of the student's answer. For example, **0-n=0** has single-column evidence in the problem 303-218=105 because it explains the ten's column. A full-problem evidence scheme is used because it is efficient to obtain and is very constraining on the set of bugs. The latter is crucial since the search space grows rapidly with the number of primitive bugs considered for compounding.

There are two possible advantages to a single-column evidence scheme: theories can be ruled out quickly (by having to look at only the first column rather than the whole problem to find out that a hypothesis is not correct); and better candidates for compounding can be found (because the candidate only has to exhibit itself in a single column rather than in the whole problem). The first advantage is offset by the efficiency of the procedural network representation; complete answers can be provided fast enough that time is not an issue. Even if a single-column evidence scheme were used, full-problem information would still be needed to check for interactions and determine the consistency of local column theories across the whole problem. Thus, DEBUGGY uses full-problem evidence to constrain its initial hypothesis set. The second advantage is counteracted by the fact that, while single-column evidence does find more candidates for compounding, many of them are spurious. We will later discuss the use of single-column evidence to extend the initial set of hypotheses as needed.

### *3.1.2 Reducing the initial hypothesis set:*

The elements of the initial hypothesis set (bugs that completely explain at least one erroneous student answer) are combined to generate additional hypotheses particular to the student. Thus, the space of hypotheses considered for a student is limited to the powerset (set of all subsets) of the initial hypothesis set. Even this limited space is often too large to search exhaustively (not infrequently the full-problem evidence set contains around 20 bugs, leading to $20^4$ hypotheses); so attempts are made to reduce the initial hypothesis set before looking for compounds.

Some bugs appear in the full-problem evidence set because they agree coincidentally with a student answer actually caused by a different bug. For example on the problem 700-5, the bug **adding instead of subtracting** will give the same answer as the bug **smaller from larger**. If this problem is the only piece of evidence for **adding instead of subtracting**, and if there is other evidence for **smaller from larger**, **adding** will be removed at this stage, saving the system the necessity of exploring all compounds containing **adding** as one of the constituents. Another reason primitive bugs in the initial hypothesis set may not be good candidates for compounding is that some are specializations of other bugs; if a student has a general bug, many of their answers will agree in places with its specializations. For example, **smaller from larger** is more general than **0-n=n**. If a student is following **smaller from larger**, some answers will be explainable by **0-n=n**. The subsumption heuristic will discard **0-n=n** in this situation.

The initial hypothesis set is reduced by finding and removing primitive bugs that are completely subsumed by other primitive bugs. A strict definition of subsumption is used because it is possible to remove a bug which should be one constituent of the final compound theory. The subsumption process considers all of the problems on the test before removing a bug. For this process, there are three classes of predictions by the bug. Ordered in terms of value, or "goodness," they are: (1) those that agree with the student's answer, (2) those that predict the correct answer when the student's answer is incorrect, and (3) those that predict a wrong answer different from the answer the student gave. For a bug A to be removed, there must exist another bug B such that A is the same or worse on every answer, and worse on at least one, than B. In particular, a bug will be kept if that bug predicts the right answer when the possible subsuming bug predicts a wrong answer. These retained bugs are more likely to compound well because the problems for which they predict the correct answer represent situations wherein they do not apply. Hence they would not interfere with other bugs that do explain the behavior.

### *3.1.3 Compounding the hypothesis set:*

Pairs of bugs in the reduced full-problem evidence set are formed and the resulting compound hypotheses are compared with the student's answers. If the compound explains more of the student's behavior than either of its constituent hypotheses, the compound is added to the set of hypotheses.

Any new compounds are paired with the existing hypotheses so that all subsets of bugs will be considered. There is a limit (currently 4) to how many primitives are allowed in a compound; compounds with more than this number of constituents are rejected. As an efficiency consideration, information stored in the network indicates which bugs hide which other bugs or subprocedures. For example, a link from the bug smaller from larger to the procedures for borrowing is used to store the fact that this bug will prevent borrowing from ever occurring. Such links are examined when a compound is first considered so that any attempt to compound smaller from larger with any bug of the borrowing procedures will be rejected immediately.

This "hill-climbing" algorithm which needs independent reasons for belief in each of the pieces is very efficient, but occasionally misses a bug because it is not one of the considered hypotheses. In some cases, one of the bugs in the compound will completely cover the symptomatic cases for the other, making it impossible to get independent evidence for the covered bug. For example, consider the case of a student who has the bug 0-n=0 {A} compounded with the bug when borrowing from a column with a 0 as the top digit, decrement the bottom digit of the column instead {B}. There will never be any evidence for {B} (problems for which {B} alone predicts the students' wrong answer) because every problem causing {B} to generate a symptom will be further disturbed by {A}. Special cases like this are handled (once we discover them) by adding them to the starting list of bugs to be tried on every student. A related, more frequent problem occurs when bugs that are not logically covered appear to be so because the particular test being given did not happen to contain the right type of problems. This difficulty can be overcome in large part by the careful test-item selection that we will describe in Section 3.2. (Also we will discuss a general structure called a skill lattice that provides a means of identifying those bugs potentially covered by other bugs, both in general and for a specific set of test problems.)

### *3.1.4 Coercion - explaining the "noise":*

During the compounding phase the system creates a structure that includes, for each of the primitive and compound bugs, the knowledge of where it agrees with the student and where it disagrees. After the structure is completed, the problem of noise is considered. The general approach is to try to find *rationalizations* for the discrepancies between the bugs' predictions and the student's actual answers. First, the set of hypotheses that have any chance of being selected as a diagnosis is chosen. The current selection criterion is whether the hypothesis explains a given

percentage (currently 40 percent) of the student's symptoms. This cutoff saves time and prevents coercion from turning a theory for which there is very little evidence into a viable diagnosis. For any problem in which a selected hypothesis disagrees with the student's answer, an attempt is made to force the hypothesis into generating the student's answer by varying the hypothesis slightly with one or more coercions. In the example in Figure 3, the best hypothesis is coerced in two problems: in problem 5 by a variant that recognized one and zero as the "digit" ten; and in problem 11 by the facts table error 15-8=8.

The coercions are designed to explain local variants of the bug and common performance lapses which would otherwise be counted against the bug. There are three types of coercions: facts table errors, variants and predicted inconsistencies. Facts table errors are limited to one fact error per problem which is off by 1 or 2. (For example, the facts error 9-3=3 will not be considered because it is off by 3.) These limitations on the use of facts errors as coercions are imposed because any possible answer could be explained with enough facts errors. This procedure gets a large number of the cases that the human diagnosticians believed were facts errors from looking at the student's scratch marks. Variants are local modifications which allow bugs to be applied differently at boundary conditions. The set of looked-for variants include considering the units column separately from the rest of the problem and treating the leftmost digits specially.

In certain cases, a bug will cause conditions that do not normally occur. In these cases, students may not have developed a consistent way of handling the unusual condition even though it is clear they are consistently following the original bug. For example, the bug **always borrowing whether or not the top digit is larger than the bottom** sometimes causes the result in a column to be larger than 9. Some students will truncate and use the units digit. Some students will write both digits in the answer. Some students will "carry" the extra ten back into the column it was borrowed from (thereby getting the correct answer). Some students will do a mixture of these. These three options are handled by predicted inconsistencies. Another example is that the bug **when borrowing from a column which has a zero on top, forget about the decrement operation** introduces the condition of having a zero on top that has been "touched" by a previous operation. Some students treat this condition as they would any other 0-n column. However other students will use **0-n=0** or **0-n=n** in this column even though they will solve "untouched" 0-n columns correctly. (These are not considered to be bugs in their own right because they never appear in isolation; they apply only in conditions which don't arise unless there is another bug.) The set of inconsistencies that are predicted by a bug are stored in the network.

The coercions are applied to the buggy procedure. This means that a student can make a facts error while doing smaller from larger and the system will recognize it (see figure 3). After coercions are done on each problem upon which the bug and the student disagree, an attempt is made to incorporate the coercions into the bug as pieces of the compound. This is done because in the evaluation phase a hypothesis is weighted down by the number of coercions it requires *unless the coercion is consistent throughout the test.*

The correct subtraction procedure is also coerced. This provides a reference plane as to how much of the student's incorrect behavior can be due to coercions. If a hypothesis with coercions does not explain the answers better than the coercions alone, it is not a very useful diagnosis. As a side-effect, coercing the correct subtraction procedure also diagnoses students with just facts table errors.

### *3.1.5 Choosing a diagnosis:*

After the coercion procedure, each bug is classified according to how well it explains the answers. The classification procedure is a refinement of the one described in Brown and Burton (1978). Briefly, it takes into account the number of predicted correct and incorrect answers as well as the number and type of mispredictions. Mispredictions which predicted a wrong answer when the student gave a correct answer are counted more negatively than mispredictions where the student got a wrong answer. (It is more likely that "noise" caused a student to miss an extraneous problem than that it caused him to get one right.) Coercions are counted negatively but not as negatively as mispredictions. Each type of coercion has its own weight so that different coercions can have differing effects on the final evaluation. For example, the variant of recognizing a one followed by a zero as 10 counts very little off. The goal of the classification procedure is to put each of the hypotheses into one of the classes: consistent bug (i.e., almost all of the student's errors are explained by the hypothesis); consistent bug but with other symptoms; some buggy behavior but not consistent; and unsystematic behavior.

The members of the highest non-empty class are then compared with each other and the best one is taken as the diagnosis. This final comparison allows simple theories that explain slightly less to be chosen over more complex theories. In actual use, the results of the entire diagnosis process are printed in order to improve the diagnostic process. Figure 3 provides excerpts of a diagnosis.

Figure 3

Diagnosis of compound bug with coercions and noise

```
Problems and correct answers:
43   80   127   183   106   800   411   654   5391   2487   3005   854   700   608   3014
-7   -24  -83   -95   -38   -168  -215  -204  -2697   -5     -28   -247  -5    -209  -206
36   56    44    88    68   632   196   450   2694   2482   2977   607   695   399   2808
Student answers:
+    64    +     +     +    542   106    +    2604     +    1088    +    605   309     +

Borrow Across Zero & Borrow Skip Equal:
*    !     *     *    *&    ***   ***    *     ***     *    ***&    *    ***   ***     *

O-n=n:
*    ***   *     *     *    768   216    *     !       *      *     *    705    !    3208

Smaller From Larger:
44   ***   164   112   132   768   204   450   3306     *    3023   613   705   401   3212
```

*** - bug predicts student's incorrect answer
 * - bug predicts student's correct answer
 ! - bug predicts the correct answer, student's answer incorrect answer
 + - student's answer is correct
 & - coersion applied

Diagnosis: a consistent bug = {Borrow Across Zero & Borrow Skip Equal}

The diagnosis is a compound of the bug when borrowing from a column with zero on top, leave that column alone and borrow from the next column to the left instead and the bug when borrowing from a column in which the top and bottom digits are the same, leave that column alone and borrow from the next column to the left. Problem 5 was coerced by the variant that recognizes one and zero as the "digit" ten thus blocking the first bug. Problem 11 was coerced by including the facts table error 15-8=8 with the diagnosis for this problem. The best hypothesis did not predict problem 2 correctly, however it still had enough evidence to be chosen as the diagnosis. The bugs smaller from larger and O-n=n are included because they explain the answer that is not explained by the first diagnosis. Note that even though there is positive evidence for them, there is substantial negative evidence against them as well.

### 3.2 Interaction between test and diagnosis

An integral part of an off-line diagnostic program is the set of test problems that are given to the students. DEBUGGY includes automatic programs to measure various diagnostic properties of tests and symbiotic programs that aid test creation. One essential property for a test is that of distinguishing all of the bugs. (One test analysis program determines the equivalence classes of bugs imposed by a test. That is, it determines which bugs give the exact same answers on every problem on the test.) Using this program, we were able to design a test capable of distinguishing among 1200 compound bugs with only 12 problems! A second important property of a test is that it cause each bug to be invoked often enough to determine that it is consistent. The current tests

that we are using are designed to cause each primitive bug to generate at least three errors. To accomplish this it was necessary to have 20 problems on the test.

Another important property of the test is that it generate answers that will serve as useful input for the diagnostic program. For DEBUGGY's compounding heuristics to be maximally effective, the test should be designed to test the subskills on different problems so as to maximize the chances of finding evidence for a bug in one subskill without the interference of a bug in another subskill. This can be difficult. For example, consider the test given the student whose diagnosis appears in Figure 2. Each of the four primitive bugs that are constituents in the final diagnosis have independent evidence in only one problem. If any of these problems had not been on the test, the diagnostic program would not have had the correct diagnosis in its search space of hypotheses. One possible alleviation is to use partial evidence (such as column evidence) to expand the number of primitive bugs considered for compounding. The interactive diagnosis system uses this solution. For off-line diagnosis, however, the test can be designed in advance to minimize this problem.

DEBUGGY includes a program based on the skill lattice (described in a later section) to find those subskills that are not being tested independently. The independence constraint opposes the goal of producing small tests because a small test must exercise as many skills as possible in each problem. The 12-problem test mentioned above requires several different skills in each problem. For isolating bugs in students who have primitive bugs, the test works very well. On the other hand, when a student with a compound bug misses a problem, it is sometimes difficult to recognize that fact from the primitive bugs separately. By adding more problems that use the subskills independently, the test increases the effectiveness of the diagnostic algorithm.

### 3.3 Interactive Diagnosis

The DEBUGGY system described above is an off-line system. An interactive version of DEBUGGY (called IDEBUGGY) has also been developed. Interactive analysis has the potential for allowing a much faster, better confirmed diagnosis because the problem sequence can be tailored to the student. IDEBUGGY uses many of the same internal routines as DEBUGGY with a quite different control structure. In addition, interactive use requires the techniques of problem generation and recognition of logical equivalence. Whereas DEBUGGY has been tested on thousands of students, IDEBUGGY has seen little use and hence the techniques are correspondingly less refined. The discussion of IDEBUGGY is included here in order to lay out the space of diagnostic techniques by pointing out differences between on-line and off-line diagnosis.

IDEBUGGY presents the student with problems and using his answers, maintains (generates and evaluates) a set of hypothetical diagnoses from the set of primitive bugs and the space of compound bugs. After getting each student answer, a choice is made whether to give another problem or to stop and report the diagnosis. Thus, each new problem is determined by the state of possible

hypotheses up to that point. When enough evidence for one hypothesis is collected, and there are no closely competing hypotheses, that hypothesis is returned as the diagnosis.

### *3.3.1 Organization of IDEBUGGY:*

IDEBUGGY is organized as a collection of tasks together with a heuristic strategy for deciding which task to do next. These tasks include such things as generating a good successive problem, reconsidering previously suspended hypotheses or deciding to produce a diagnosis. Before describing the operation of the tasks, we present a brief description of the context within which the tasks operate and of how the context originates. Each task is small in terms of the amount of computation it requires. A task changes the global context so that after it has run a different task will be choosen. This organization allows IDEBUGGY to either be directed by its own strategy heuristic or be guided in a symbiotic manner by a teacher or diagnostician. The global context consists of all hypotheses (primitive and compound bugs) that have been considered. With each hypothesis is noted how well it predicts a subset of the problems that have been given the student so far. An assumption made during design of IDEBUGGY (proved true in use) is that there are too many hypotheses to try all of them on every problem in interactive use. Thus, the set of hypotheses to compare with each student answer is carefully chosen to control the proliferation of hypothesis/problem pairs.

To start, the student's first problem is tried on a preselected list of common bugs. This provides an initial set of hypotheses. Before each task choice, the set of hypotheses that explain the observed behavior well enough to be considered reasonable diagnoses are selected as the "current set." The "current set" is an important factor for the decision of what task to do next. (The non-current hypotheses are saved for later consideration; no hypothesis is ever thrown away.)

The tasks available at each choice are:

(1) generate a simple problem;
(2) generate a problem that splits between the bugs in the current set of hypotheses;
(3) allow more obscure bugs as hypotheses;
(4) create more hypotheses by extending (compounding) the current ones;
(5) reconsider previously suspended hypotheses;
(6) propose a diagnosis;
(7) or: give up.

The choice of task is determined primarily by the size of the current set of hypotheses. If there is more than one bug in the current set, a problem is generated which splits between them. That is, a problem is found that will generate positive evidence for at most a subset. (The problems of doing this will be addressed in the next section.) If there is only one hypothesis in the current set, there are several alternatives: gain more support for this hypothesis by generating a problem that splits

between it and the next best hypothesis (even though the second best one is not good enough to be current); extend the current hypothesis by compounding it with other hypotheses and see if any of these do an equally good job of explaining the evidence; generate a problem which tests a subskill not yet tested so that other bugs are not overlooked; or propose the current hypothesis as the diagnosis.

If the current set is empty, IDEBUGGY can either: consider more obscure bugs for the hypotheses, combine existing hypotheses into compounds, generate simple problems which isolate single subskills, or give up if the student has already answered many problems. Generating problems which test subskills in isolation is done because the student's bug may be a compound. A problem which tests a single subskill may turn up one of the primitive constituents which can then be extended into a complete hypothesis.

When the decision is made to extend a hypothesis, it is combined with every hypothesis that has full-problem evidence, with every hypothesis that has single-column evidence and with a set of common primitive bugs. This allows many more possibilities than the heuristics used off-line, but since the number of problems given the student is less, the chances of getting extraneous evidence for inappropriate hypotheses is reduced. When a hypothesis has accumulated sufficient positive evidence, an attempt is made to coerce its negative evidence.

To reduce the amount of computation performed when a new student answer is received, a large number of the hypotheses are not tried on new data. Hypotheses that are in this state are referred to as "suspended" because they have been suspended from consideration by the program. Hypotheses are suspended when they have mispredicted more than a certain percentage of the student answers (currently 30 percent). Because a student may make performance errors in the first problems he does, the correct hypothesis can be suspended. To guarantee that the best hypothesis is found, suspended hypotheses are reconsidered whenever a current hypothesis has mispredicted more problems than the suspended hypotheses have. During reconsideration, each hypothesis is tried on problems it has not yet done, i.e., those the student answered while the hypothesis was suspended. If the hypothesis mispredicts one, it is suspended again. If it correctly predicts all of them, it is added to the set of current hypotheses.

### *3.3.2 Problem generation:*

The diagnostic system needs to be able to generate (or have prestored) problems with many different features. This need arises when the system has more than one hypothesis and requires a problem that differentiates between them. For any domain with a large number of hypotheses to consider, prestoring problems for all possible combinations of bugs is not practical. The system is therefore faced with the problem of generating problems to fill this need. A piece of leverage that the system does have is an executable model of the bugs that can determine whether or not any

particular problem will differentiate between them (by running them and seeing if the answers are different), so the system only needs a generator that is likely to have the right features.

IDEBUGGY uses problem generators designed to produce problems specific to a small number of conditions (listed in Table 1). Each generator produces a problem sequence that is rich in problems containing their target condition, though not all the problems contain it. The resulting problems are filtered by actually running them through the models of the bugs to find the symptom-producing problems in the sequence. This freedom allows more randomness in the generation process which in turn avoids over constraining the generated problems. For example, the generator for problems with the same digit in both the top and the bottom can be random about whether or not the columns on either side of that column require borrowing. This allows the single condition generator to be used effectively to generate symptoms for compound hypotheses. The generators are associated in the network with each primitive bug. For compound bugs, the generators for each of the primitives are cycled through the generators stored off the primitive bugs.

Table 1
Problem Generators

| |
|---|
| arbitrary problems |
| problems that do not require borrowing |
| problems that require borrowing |
| problems that have zero on the top |
| problems that have zero on the bottom |
| problems that require borrowing from zero |
| problems that borrow from two zeros in a row |
| problems that have the same digit on top and bottom in a column |
| problems that borrow from a column which has a blank on the bottom |
| problems that have a zero in the top and bottom of a column |
| problems that have one and zero as the leftmost digits |
| problems that have a one in the top number |
| problems that require a borrow from a column which has zero over blank |
| problems that require a particular subtraction fact |

In some cases two hypotheses will be very similar (two compounds, for example, that share common elements) and the above process will not find a problem that splits them. In these cases, an additional heuristic is used provided that the two hypotheses give different answers to one of the problems the student has already solved. Variants of the existing, splitting problem are found by incrementing and decrementing some of its digits, adding extra columns on the left or right, or by deleting columns.

The "varying-an-existing-problem" heuristic is also used when attempting to gain confirming evidence for a hypothesis that explains most but not all of the student's answers. Problems similar to the ones that the hypothesis mispredicted are generated. If the mispredicted problems were careless errors, the new problems will increase the positive evidence for the hypothesis. If they were symptoms of a part of the student's bug that is not included in the current hypothesis, the new problems may provide some evidence as to what the missing piece is and will, at least, provide negative evidence against the current hypothesis so that it is not proposed (incorrectly) as a diagnosis.

### *3.3.3 Logical equivalence:*

In IDEBUGGY, logical equivalence is a significant problem because the program can waste a lot of time attempting to split logically equivalent hypotheses. The problem is differentiating when a bug {A B C} is logically equivalent to another bug {D E F} from when the problem that separates the two has not yet been found. Our heuristic solution is to use the problem generation techniques mentioned earlier to try to find a problem on which the two bugs differ (hence proving that they are not equivalent). If, after some suitably large number of attempts (500), they are not shown to be different, then the hypotheses are treated as if they were the same. (Footnote: This heuristic is useful because it uses "smart" problem generators and it gets better as the problem generators get better.) This is done by selecting the hypothesis with the simplest bug (or one of the simplest in case of ties) to remain in the current set and having the selected hypothesis point at the others. Whenever a new problem is tried on the selected member, however, it is also tried against all of the equivalent ones and any that are found to differ are split out.

## 4.0 Definition of Subskills

One interesting benefit of the database of bugs is that it is possible to define an intuitively appealing notion of subskill in a way that has many uses in the diagnostic process. The term "subskill" has been used in the diagnostic literature to informally refer to any part of the skill being talked about. Additionally, there have been attempts to characterize subskills in terms of groupings of test items or as pieces of a representation of the correct skill (Durnin and Scandura, 1973). These notions of subskill are too coarse to account for the range of behaviors captured by the database of bugs, in the sense that there is no direct correspondence between the bugs and any other notion of subskill. The primary limitation of existing characterizations of subskill is that they are driven from the correct skill and nothing else. *We have observed bugs, however, which have no vestiges in the correct skill.* For example, some students when borrowing will skip columns that have the same number on the top and the bottom. We would like to say that these students are lacking a subskill that a student who subtracts correctly has. This subskill is knowing that the bottom number doesn't matter during a borrow operation. This subskill is the *absence* of a procedure in the correct skill. (There is an alternate view that the buggy student is making

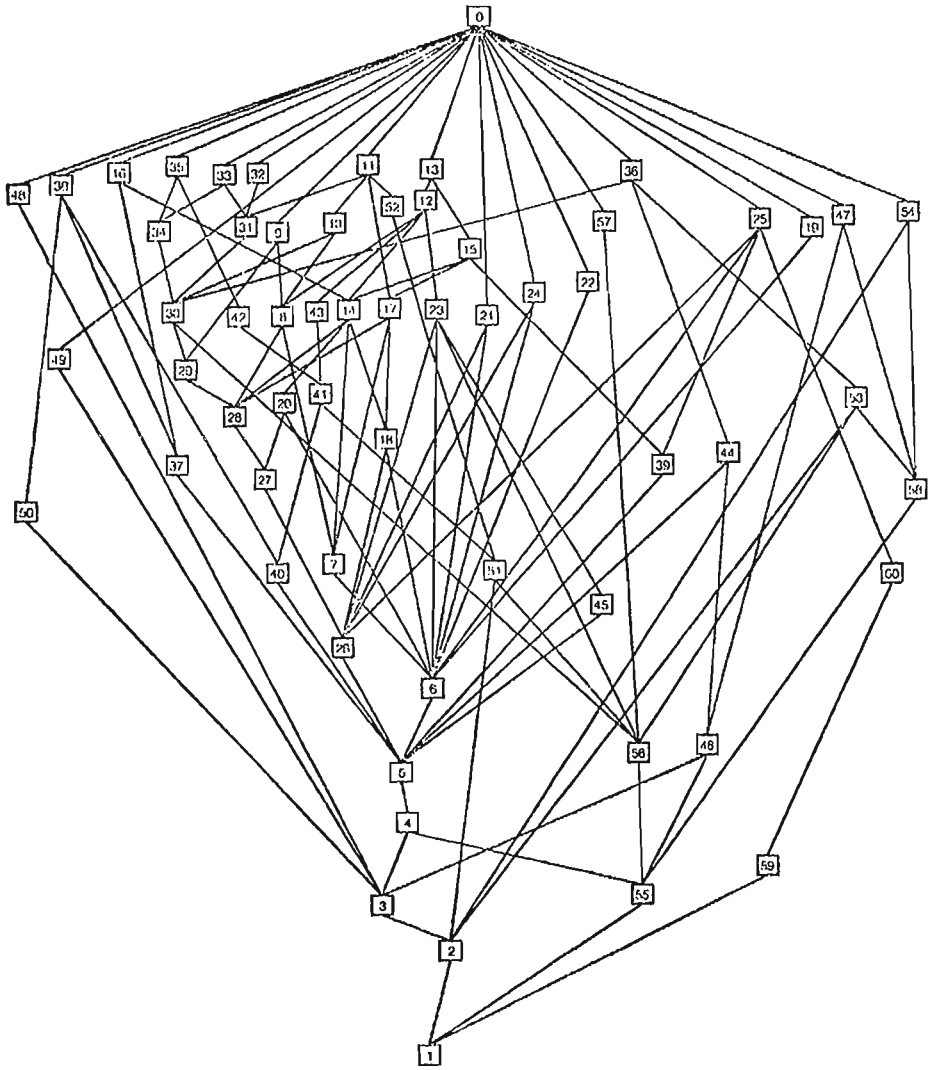
distinctions that are inappropriate for this skill.) Which things constitute subskills when left out of the correct skill can be determined empirically from the database of bugs.

A subskill is defined to be any isolatable part of the skill that it is possible to mislearn. This is determined from the database of primitive bugs by considering how they partition the set of all possible problems. Every bug separates the set of all possible problems into two partitions: those for which it generates the correct answer and those for which it generates an incorrect answer. Different bugs will often generate different partitions, though it is possible for two bugs to generate the same ones. For example, the bugs **0-n=0** and **0-n=n** generate equal partitions because any problem that causes **0-n=0** to produce an incorrect answer also causes **0-n=n** to produce an incorrect answer. The bugs are grouped into equivalence classes by the equality of their partitionings of the set of all possible problems. Each of the equivalence classes that arises from this analysis determines a *subskill.* This gives rise to subskills that are much finer grain than normally recognized. While in theory this leads to the consideration of an infinite number of problems, in practice one need only consider enough problems to completely distinguish between bugs.

The subskills can be placed naturally into a lattice by the relationship **gets correct answers on all the same problems and more.** The maximal element in the lattice is the skill of knowing how to subtract all possible problems (having all problems in the "correct" partition) and the minimal element is the lack of any subtraction skill at all (having no problems in the "correct" partition). Figure 4 presents the skill lattice for subtraction based on bugs we have observed. We obtained supporting evidence for this notion of subskill when, after determining the lattice, it was possible to identify and name all of the subskills produced. It is also worth noting that the subskills are independent of the representation chosen for the bugs; it stems only from their behavior.

The most surprising thing about the skill lattice for subtraction is its complexity. There are 58 subskills necessary for even this simple skill (approximately one for every two bugs). Another surprising thing is that many of the subskills are taken for granted; seemingly because the situations in which they occur are *not* recognized as decision points in the final skill. For example, students must learn to ignore the bottom number in a column they are borrowing from. This captures the intuition that part of expertise is knowing when to ignore what.

Figure 4
Skill lattice for subtraction



0 Correct skills
1 No subtraction skills
2 Subtract 0 from a number whose right digit is 0
3 Subtract 0 from a number
4 Subtract columns which are n-n or n-0
5 Subtraction without borrowing
6 Borrow from somecolumns with a digit on the bottom
7 Borrow from columns whose answer is the same as their bottom digit
8 Borrow from a column with a 9 or a larger number on top

9 Borrow from a column with a larger top digit
10 Borrow from a column with zero on top when blank on bottom
11 Borrow from a column with zero on both top and bottom
12 Borrow from a column unless one is on top and a non-zero digit is on the bottom
13 Borrow from a column unless a one on top and bottom
14 Borrow in two consecutive columns
15 Borrow from columns with the same digits on top and bottom
16 Borrow from columns that have nine as the bottom digit
17 Borrow from columns with the same non-zero digits on top and bottom
18 Borrow from columns which have an answer of zero
19 Borrow from the leftmost column when it has a non-blank in the bottom
20 Borrow more than once per problem
21 Can borrow then not borrow
22 Borrow from columns with two on the bottom
23 Borrow from leftmost columns or columns that have a non-one on top
24 Borrow from columns with top digits smaller than the bottom digits
25 Borrow from columns that have a top digit one less than the bottom digit
26 Borrow from the leftmost column
27 Borrow once in a problem
28 Borrow from columns that have the top digit larger than the bottom
29 Borrow from a column with the top digit greater than or equal to the bottom
30 Borrow from a column with a zero on top
31 Borrow from a middle (not leftmost) column with a zero to the left
32 Borrow from leftmost column of the form one followed by one or more zeroes
33 Borrow from a column with a zero on top and a zero to the left
34 Borrow from or into a column with a zero on top and a zero to the left
35 Borrow into a column with a zero on top and a zero to the left
36 Borrow from a column with a zero on top and a blank on bottom
37 Borrow into a column with a nonzero digit on top
38 Borrow into a column with a one on top
39 Borrow when difference is 5
40 Subtract columns with a one or a zero in top that require borrowing
41 Subtract columns with a zero in the top number that require borrowing
42 Subtract a column with a zero on top that was not the result of decrementing a one
43 Borrow into a column with a zero on top when next top digit is zero
44 Borrow from a column with a blank on the bottom
45 Borrow from a column with a one on top
46 Subtract numbers of the same lengths
47 Subtract a single digit from a large number
48 Subtract columns unless the same digit is on top and bottom
49 Borrow all the time
50 Subtract when one is in top
51 Subtract when neither number has a zero unless the 0 is over a 0
52 Subtract columns when the bottom is a zero and the top is not zero
53 Subtract when a column has a zero over a blank
54 Subtract numbers which have a one over a blank that is not borrowed from
55 Can subtract a number from itself
56 Subtract numbers with zeros in them
57 Subtract problems that do not have a zero in the answer
58 Subtract numbers when the answer is no longer than the bottom number
59 Subtract leftmost columns that have top and bottom digits the same
60 Subtract columns that have top and bottom digits the same

*4.1 Uses of skill lattice in test generation:*

A subskill lattice can be created from any test by considering the way the bugs partition the problems on the test. The resulting lattice will be a homomorphic image of the maximal lattice resulting from considering all possible problems. Any nodes in the maximal lattice that are mapped onto the same node in the test lattice are not distinguishable by that problem set (assuming one is considering an answer as either right or wrong). This provides a measure of the extent to which one subskill may be hidden by another in the test. When designing a test for a diagnostic system that works by considering only the correctness of the answers, this mapping information is very useful. Testing two subskills together prevents students from demonstrating that they do know one of them if they don't know the other. For example it may be that every problem that involves subtracting a zero from a digit also involves borrowing. If a student cannot borrow, we will not know whether he can subtract a zero or not. Diagnostic systems such as DEBUGGY which try to predict the exact wrong anwers given have a better opportunity to separate the student's subskills. However, even these systems can benefit from subskill independent tests when producing partial diagnoses for students who are not consistent.

*4.2 Use of skill lattice in diagnosis:*

The skill lattice provides a useful heuristic for attacking the problem of having one bug hide the symptoms of another. In some situations, the diagnostic program is faced with the problem that its best current hypothesis {A B} explains some of the students' answers but not all of them. One explanation for this situation is that there is some other bug C which is also present, that is, the student actually has bug {A B C}. If {A B} is not covering C (i.e., there exists some problem for which C generates a wrong answer but both A and B generate the right answer), there will be some independent evidence that C is present (the problem that C predicts to be wrong) and the combination heuristic will try compounding {A B} and C. If, however, {A B} does cover C (because every problem for which C generates a wrong answer either A or B generates a wrong answer), there will never be any evidence for C separately, and {A B} would (potentially) have to be compounded with all other primitive bugs to discover {A B C}.

The skill lattice contains exactly the information to identify candidates for C. We first calculate the skill lattice using the problems the student has answered and find the union of A's and B's "incorrect partitions". This union defines a cut through the lattice such that all of the skills below this cut have "incorrect partitions" that are subsets of the union. The skills below the cut are covered by {A B} and any bugs of these skills are candidates for the hidden bug C. If this set is large, the problems predicted wrongly by the bug {A B} can be used to produce the candidates for C ordered by how well their "incorrect partition" matches the answers not explained by {A B}.

*4.3 Diagnosis to the subskill level:*

In the light of our characterization of subskills, we reconsider the possibility of diagnosing only on the basis of whether a student's answers are right or wrong. The method we used to define subskill depends only on which problems were missed, so it should be possible to determine which subskills are errant by only considering right versus wrong. Scoring test items as right or wrong is fast and there is the possibility that a very fast, simple diagnosis system could be built based on this technique by locating the student's template of right and wrong answers in the lattice. It is our belief that a fast system could be developed around this paradigm.

There is a limiting problem with this approach. From a practical standpoint, bugs of the same subskill may require different remediation. Consider two bugs of the borrow subskill: **leaving blanks in the answer in any column which requires a borrow**, and **decrementing the minuend digit in the first column that requires a borrow before adding ten to it.** These two bugs will be grouped together by any classification scheme that looks only at right vs wrong because they cause wrong answers on exactly the same set of problems (any problem that requires borrowing). However, it is clear that a student with the second of these bugs understands quite a lot about borrowing while the student with the first probably understands very little.

The use of skill lattice also presents interesting possibilites in combination with methods used in DEBUGGY. In particular it provides a clue to the problem of "noise" by characterizing the problem-partitionings that are possible within the boundaries of the theory. If the student has a partitioning that is not in the lattice or constructable by combination of partitions that are in the lattice, the student's test must have "noise." Of course knowing that a student's solution contains noise is quite different from knowing where the noise is or what is producing it, but it is an interesting case of the program recognizing that something is beyond the limits of its theory.

## Conclusions

We have described a system which has been used to diagnose consistent procedural bugs in more than a thousand students and which is still being used in classrooms. From this experience we have learned that diagnosis of real students using only the student's answers is difficult but possible if one has a strong model of the correct skill and its bugs. We have described techniques for implicitly reducing the size of the search considered during diagnosis and searching the reduced space in an efficient order. An important lesson from this discussion is that the diagnostic heuristics go hand in hand with test generation; one can be designed to overcome (to some extent) the limitations of the other. We also presented an empirically based definintion of subskill and explored some of its applications.

## References

Brown, J.S. and Burton, R.R. Diagnostic models for procedural bugs in basic mathematical skills, *Cognitive Science*, 2, (1978) 155-192.

Brown, J.S. and VanLehn, K. Repair theory: A generative theory of bugs in procedural skills, *Cognitive Science*, 2 (1980).

Durnin, J.H. and Scandura, J.M. An algorithmic approach to assessing behavioral potential: Comparison with item forms and hierarchical technologies, *Journal of Educational Psychology*, 65, 2 (1973), 262-272.

Friend, J. Domain Referenced Adaptive Testing, Palo Alto, CA: Xerox Cognitive and Instructional Sciences Report 10, 1981.

Friend, J. and Burton, R. R. Teacher's Manual of Subtraction Bugs. Palo Alto, CA: CIS Working Paper, Xerox Palo Alto Science Center, August 1980.

Sacerdoti, E. A structure for plans and behavior, *The Artificial Intelligence Series*, New York: Elsevier North-Holland, 1977.

VanLehn, K. Bugs are not enough: Empirical studies of bugs, impasses and repairs in procedural skills. Palo Alto, CA: Xerox Cognitive and Instructional Sciences Report 11. 1981.

**Acknowledgements:**

I would like to thank John Seely Brown and Kurt VanLehn for their support during the design and documentation of DEBUGGY, Jamesine Friend for pouring over diagnoses and ferreting out troubles with the system and Bruce Buchanan for his many helpful comments on an early draft of this paper.