社団法人 電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
TECHNICAL REPORT OF IEICE.
SS96-5 (1996-07)

視線追跡装置を用いたデバッグプロセスの観察実験

門田 暁人, 高田 義広, 鳥居 宏次

奈良先端科学技術大学院大学 情報科学研究科

〒 630-01 奈良県 生駒市 高山町 8916 番地の 5
akito-m@is.aist-nara.ac.jp

あらまし　ソフトウェアのデバッグのプロセスについて, 効率に影響する要因を明らかにする程度に詳細に分析するためには, デバッグ中に絶えず変化する作業者の意図を追跡する必要があると考えられる. そこで, 我々は, 作業の様子を詳細に観察するために, 視聴覚装置や視線追跡装置などを組入れた観測システムを構築した. そして, バグを発見するまでのプロセスを観測する実験を行なっている. これまでの実験の結果, 注視点の移動や発話の記録などの, 複数の種類のデータを同時に収集し分析することが被験者の意図の追跡に有効であることがわかった. そして, バグの発見に影響すると思われるプロセスの特徴をいくつか確認した.

和文キーワード　視線追跡装置, デバッグプロセス, 観測システム, プロトコル分析, プログラム理解, バグの発見

An Experiment to Observe Debugging Process by Using an Eye Tracking Device

Akito MONDEN, Yoshihiro TAKADA, and Koji TORII

Graduate School of Information Science, Nara Institute of Science and Technology

8916-5 Takayama-Cho, Ikoma-Shi, Nara 630-01
akito-m@is.aist-nara.ac.jp

Abstract　To clarify factors that affect efficiency of debugging, we need to investigate programmers' intentions which continuously vary during debugging. For the purpose, we have developed a system for observing programmers behavior by using an eye tracking device and audio and visual devices. By using the system, we have performed an experiment of observing bug identification processes. This paper presents results of the experiment, including the fact that collecting and analyzing multimodal data, such as motion of gazing point and record of verbal protocol, is effective in investigating programmers' intentions. It also includes several differences among observed processes which are likely to relate to the efficiency of bug identification.

英文 key words　eye tracking device, debugging process, observation system, protocol analysis, program understanding, bug identification

1 はじめに

ソフトウェアを組み込んだシステムが, テストの途中やリリースの後に予期せぬ挙動を示した時には, ソフトウェア中に潜むバグを, 早急に発見し除去することが重要である. ところが, 個々のバグの発見や除去に要する時間には, 極めて大きなばらつきがある [3]. その原因としては, 対象となるソフトウェアの特徴, バグの特徴, 作業者の特徴, 作業のプロセスの特徴, 作業の環境の特徴などの多くの要因が考えられる. ただし, どのような要因がどの程度に影響しているかについては, ほとんど明らかにされていない. デバッグの効率を向上させるためには, デバッグのプロセスを詳細に分析して, 効率に影響する要因を明らかにする必要がある.

ところが, デバッグのプロセスを詳細に分析することは, 容易でない. デバッグ中の作業者の意図を外部から推定することが困難であり, しかも, 作業者の意図がわからなければ, その様子は, 作業者が 2, 3 種類の行動を適当に繰り返しているようにしか見えないからである. 例えば, テキストエディタの操作, プログラムの実行などの繰り返しである. 効率に影響する要因が調べられる程度に詳細にプロセスを分析するためには, 絶えず変化する作業者の意図を追跡する必要があると考えられる.

そこで, 我々は, 視線追跡装置や視聴覚機器などを併用して, バグを発見するプロセスに限定して, 作業者の意図を追跡する実験を行っている. この実験は, プログラムの理解のプロセスを対象とした文献 [1, 5] などの実験と似た方針に基づいているが, バグを発見するプロセスを対象としていることが異なる. 本報告では, 実験のために構築した観測システムと, これまでの実験の結果について報告する.

以降, 2 章では, 構築した観測システムにより収集するデータの種類と方法について述べる. 3 章では, バグを発見するプロセスを観測する実験について述べる. 4 章では, 実験の結果に基づいて, システムの有効性を論じる. 5 章では, 実験の結果より確認した, バグの発見に影響すると思われるプロセスの特徴を述べる. 6 章は, まとめと今後の課題である.

2 観測システム

構築した観測システムは, 1 名の被験者が計算機を使用して作業を行なう様子を観察して, データを収集するためのシステムである. 構築した観測システムテによって収集しようとしたデータの種類, および, 各種類のデータを収集する仕組みについて述べる.

2.1 収集するデータ

構築した観測システムでは, 被験者の作業についての次の 5 種類のデータを同時に収集する.

(1) 計算機の画面の変化
(2) 画面上の注視点
(3) 発話
(4) 表情や挙動の映像
(5) キーストローク

複数の種類のデータを収集する理由は, 各種類のデータから得られる情報が異なっており, しかも, どの 1 種類のデータからもデバッグ中の被験者の意図を推定する上で十分な情報が得られないと考えたからである. そこで, 意図の推定に少しでも有効であると思われる 5 種類のデータを同時に収集することにした.

ここで推定しようとする被験者の意図とは, 例えば, 次のような詳細さの情報である.

- ある時刻において, ある変数がバグに関係あると断定した.
- ある時刻において, ある手続きにはバグは無いらしいと判断した. そして, 別の手続きを読んで理解することを始めた.
- ある時刻において, ある条件式の中にバグがあると判断した. そして, その条件式に関連する手続きを読み始めた.

このような被験者の意図は, 作業中の被験者自身に絶えず意図を話してもらえば容易にわかると思われるかもしれないが, それは困難である. 文献 [5] でも述べられているように, 作業中に話してもらうことは非常に困難であり訓練を要する. また, 4 章でも詳しく述べるように, 発話から得られる情報には限りがある. [2, 3, 4] では, 被験者のキーストロークから行動を分析する試みも報告されているが, そのようなデータだけでは, 上例のような詳細な意図の推定に不十分である.

2.2 データ収集の機構

構築した観測システムは, 図 1のように 2 室に分割されている. 1 室は, 被験者にデバッグを行ってもらう開発室である. もう 1 室は, 実験者がデータを収集する観測室である. 開発室から実験者が見えないようにしたり, 開発室に防音を施したりすることにより, 被験者に余計なストレスを与えないよう配慮した.

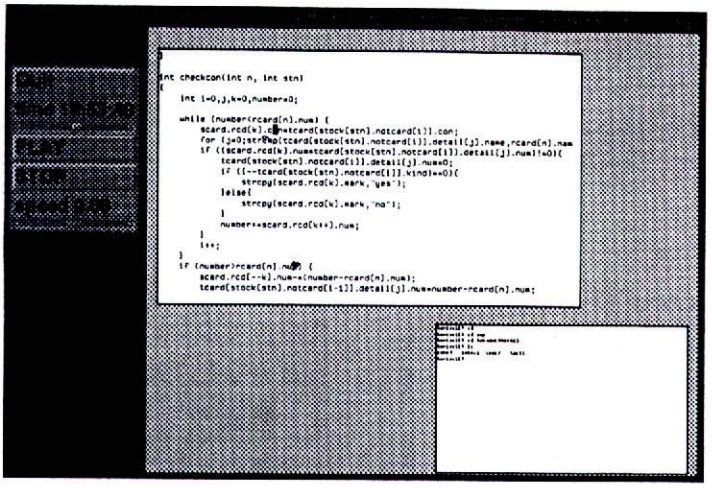

図 2: 観測・分析用ツールの画面

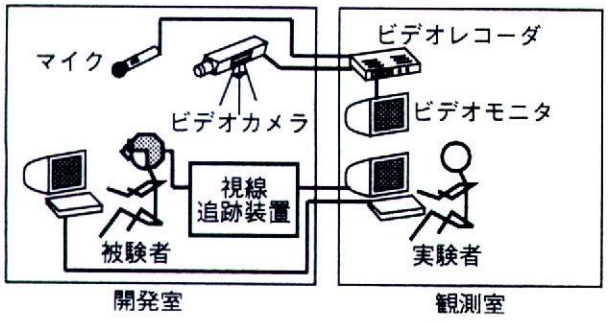

図 1: 構築した観測システム

前節で述べた各データを収集するために組み込んだ装置やソフトウェアツールは, 以下の通りである.

(1) 計算機の画面の変化を収集するツール

観測室の計算機では, 開発したソフトウェアツールにより, 被験者が使用している計算機の画面の変化のデータ, および, キーストロークを収集し, 観測室の計算機へ実時間で送信する. 収集する画面の変化のデータは, マルチウインドウ環境に対応しており, 各ウインドウの位置や大きさに関する情報, 各ウインドウ内の文字の変化やテキストカーソルの移動, マウスカーソルの移動などである.

(2) 視線追跡装置

米国 Applied Science Laboratories 社製の 4000 シリーズの視線追跡装置を使用している. この装置は, 被験者の頭に被ってもらうヘルメット型の部分と, 据え置き型の部分とからなる. 開発したツールにより, 機器の出力から画面上の注視点の座標を計算して, 座標データを 0.1 秒ごとに観測室の計算機へ実時間で送信する. 注視点の座標の計算にはパラメータの推定が必要であるため, キャリブレーション用のツールも作成した. 現時点の注視点の測定精度は, 平均誤差が 15～20mm になる程度である.

(3) マイク

被験者に考えを絶えず話してもらい, それを録音する.

(4) ビデオカメラ

被験者の表情や挙動を撮影する.

(5) キーストロークを収集するツール

開発したツールにより, 画面の変化のデータと同様に, 観測室の計算機からキーストロークを収集し, 観測室の計算機へ実時間で送信する. ただし, キーストロークは, 今回の実験では分析に用いなかった.

(6) ビデオレコーダとモニタ

被験者の映像や発話をモニタしたり, 記録したり, 再生する.

(7) 観測・分析用ツール

実験室の計算機で開発室からのデータを受信し, 表示したり, 記録したり, 再生するために開発したソフトウェアツールである. 画面の変化と注視点を同時に再生している様子を図 2に示す. 図 2では, ある時刻において被験者が開いていた 2 個のウィンドウが再現されており, 瞳型の小アイコンによって注視点が表されている. つまり, このツールにより, 任意の時間に被験者がソースリスト中のどこを見ていたかを調べることができる. コマ送り, 早送り, 巻戻しの機能も備えている.

3 実験

構築した観測システムを使用して, バグを発見するプロセスを観測する実験を行なっている. これまでに, 3 名の被験者に対して, のべ 5 回の試行を実施した.

3.1 方法

各試行においては, 1 個のバグの入った 1 個の短いプログラムを用意し, 1 名の被験者にそのバグを発見し除去することを試みてもらった. 具体的な手順は, 次の通りである.

Step 1. 被験者にプログラムの仕様書を与え, また, 口頭でも仕様を説明する.

Step 2. 試行の度に視線追跡装置の調整が必要であるので, そのための 5〜15 分の作業を被験者に行ってもらう.

Step 3. プログラムを実行して, バグの症状を 1 回だけ被験者に見せて, 仕様と異なることを説明する.

Step 4. ソースプログラムを与えて, バグを探し始めてもらう. そして, 考えを絶えず独り言のように話すよう要求する.

Step 5. バグを除去したと被験者が宣言するまで, 作業を続けてもらい, その間に 2.1 で述べたデータを収集する.

Step 6. 記録したデータを一通り再生することにより, 被験者と一緒に作業を追体験する. そして, 被験者の意図があいまいな点があれば, 被験者に質問する.

Step 7. バグを発見するまでのプロセスを非形式的に詳細に記述する. 必要に応じて, 記録の再生を繰り返す.

以上の手順で得たデータやプロセスの記述を, 比較したり被験者に再確認することにより, 観測システムの有効性の評価とプロセスの特徴の調査を行なった.

3.2 被験者

被験者は, Y, A, S の 3 名である. Y には 3 回, A と S にはそれぞれ 1 回の試行を行なってもらった. Y は 10 年以上のプログラミングの経験があった. A と S とは, プログラミングの経験が浅く, 他人の作成したプログラムを読んでデバッグを行なったことがなかった.

3.3 プログラム

用意したプログラムは, 100〜402 行の短さであり, C 言語で書かれていた. 3 目並べゲームや酒屋の在庫管理のプログラムであった. このように短いプログラムを選んだ理由は, Step 6, 7 の分析を現実的な時間と労力で終らせるためである. 実際に分析に要した時間については, 4 章で述べる.

3.4 バグ

各プログラムのバグは, そのプログラムの作成者が実際に混入させた誤りであり, 作為的に作り出した誤りでない. 配列の添字の誤りや if 文の条件式の誤りなどであった.

4 観測システムの有効性

前述の実験の結果, 被験者の意図を追跡する上では, 各種類のデータが以下のような特性を持つことがわかった.

まず, 画面の表示の記録については, それだけからでも, 被験者の注目しているプログラムの箇所や被験者の意図の推定が, 時として可能である. ウィンドウ内に表示されている内容から, 注目しているログラム中の箇所が限定でき, しかも, 被験者は, 注目している箇所にテキストカーソルを移動することが多いからである. また, 注目している箇所にマウスカーソルを移動することもある. ただし, それだけからは, ほとんどの時刻について意図を推定できない.

発話の記録については, 被験者の意図の中の特定の部分についての詳細な情報を与えるが, 全く情報を与えない部分も多い. 話される内容が断片的であるためである. その主な理由としては, 一定の時間に話せる量に限りがあること, そして, 被験者が作業に集中して話すことを忘れることが多いことが挙げられる. 発話されていても, 文章が不完全であったり丁寧でなかったり聞きとれなかったりするために理解できないことも多い. 図 3は, 記録の例である. 図中で "???" は, 聞き取れなかった部分を表している. 作業しながら話すには, 訓練が必要であると思われる. また, 被験者の思考と発話とが時間的にずれることも多い.

注視点の移動の記録については, それだけからは意図を推定できないことも多いが, 詳細な情報が絶えず得られる. 注視点は, 1 分間に数十回の頻度で瞬間的に移動するが, 停留している間は, 注目しているプログラム中の文や変数なども識別できる. 例えば, プログラムをスクロールしている最中に, 一瞬, ある変数に目が止まったこともわかる. タイプの時に, 一瞬, キーボードを見たことまでわかる. ただし, それだけからは, 意図を推定できないことも

.....	.....
13:47:35	出力を計算するところ出庫処理のリストを作ってると思うんやけど, そこがおかしいんやな. だから, そこを探しに行きます.
13:47:44	出庫処理のリストを作ってるところ. えーと, 出庫の頭文字は大文字のSやから, 大文字のSでキーワードサーチをかけます.
13:48:04	来た. いきなり.
13:48:09	データイン. データインで. えー, 引数はcで, そのcがSならsscanf. 違う. ただのscanfの???で, えー, 引数に???.
13:48:31	で, 読んだら, もういきなりリターンしよると. データインというのは, 読んだらいきなりリターンするのか.
13:48:44	他にSっていうのないし, キーワードないからほぼ間違いないやろ.
13:48:49	て言うことは, 今度はデータインを呼んでるところやけど.
13:48:55	メイン. メイン. 頭一文字を読んでデータインに渡す.
.....	.....

図 3: 発話の記録の例

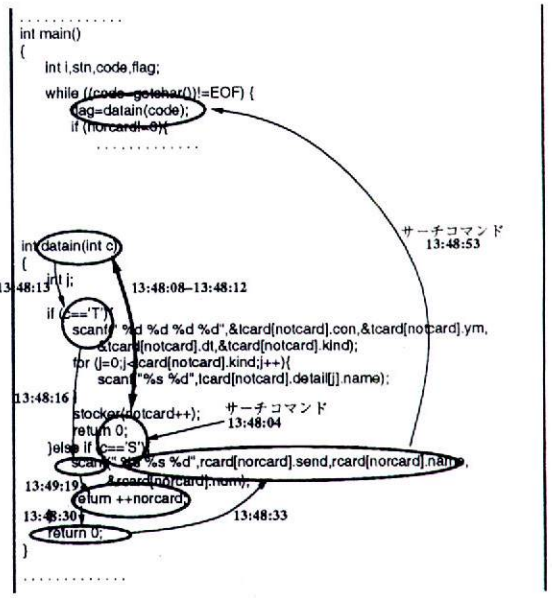

図 4: 画面上の注視点の軌跡の例

多い. 図 4は, 記録の例である.

被験者の挙動の映像については, 意図の推定にほとんどの場合に役立たないが, 時として被験者が意図を思い出すことに役立つことがある. ほとんどの時刻において, 被験者は淡々とキーボードを叩いているだけであり, そのような映像からは何の情報も得られない. 稀に, 首を捻ったり, 両手で頭を抱えたりすることがあり, そのような特徴的な挙動の映像は, 役立つことがある.

以上のことから, 2.1 で述べたデータのどの 1 種類だけを使用しても, 全ての時刻の被験者の意図を正確に推定することは困難であると言える. 被験者の意図を追跡するためには, 今回の実験でのように, 2.1 で述べたデータを同時に収集すること, そして, デバッグの終了の直後にそれらのデータを併用して被験者に質問することが重要であると言える.

Step 7 によって得られたプロセスの記述の一部を, 図 5に例示する. この例は, 被験者 Y が酒屋の在庫管理プログラムのバグを探しているプロセスの記述の一部であり, 図 3 や図 5 などのデータを元に作成した. 記述の妥当性を被験者へ再確認した結果, どの試行についても, 記述の誤りは認められなかった. 構築した観測システムは, 被験者の意図の追跡に有効であったと言える.

また, 各試行において被験者がデバッグに要した時間は 15 分～38 分であった. それに対して, その後の分析に要した時間は, 2～5 時間であった. プロセスの約 8 倍もの時間を要したものの, 現実的な時間で分析することができた.

5 バグ発見のプロセスの特徴

実験において得たデータやプロセスの記述を比較することにより, バグの発見の効率に影響すると思われる要因の抽出を試みた. 具体的には, まず, 要因が明らかになると思われる程度に詳細に, 各時刻の被験者の状態を分類し, バグを発見するプロセスの状態遷移モデルを定義した. 次に, そのモデルに基づいて, 各試行における状態遷移を分析した. そして, 分析の結果を比較することにより, 効率に影響すると思われる相違点を調査した.

5.1 プロセスのモデル

5.1.1 状態の分類

各時刻における被験者の状態は, まず, 3 種類に基本的に分類できた. プログラムを読んで理解している状態と, プログラムを実行している状態と, その他の考え込んでいる状態とである.

考え込んでいる状態においては, 主に, バグの位置や種類についての仮説を立てたり, その仮説の妥当性について考えたりしていた. 例えば, 図 5の先頭では, “出庫処理のリストを作っている関数があ

.....	.....
13:47:46	(酒屋の在庫管理プログラムにおける) 出庫処理のリストを作っている関数があり, その中に誤りがあると推測する.
13:48:04	その場所を探すために, 出庫依頼書の識別子である'S' でキーワードサーチして, 唯一ヒットした場所を読み始める.
13:48:08	その関数の名前が datain であることを確認する.
13:48:17	datain の引数に, 依頼書の識別子'T' または'S' が入っていることを知る.
13:48:44	datain という関数は, 入庫依頼書, および, 出庫依頼書を入力から読み込んで構造体に保存する関数であることを理解する.
13:48:49	main では, 入力の頭一文字だけを読んで datain に渡していることを確認する.
.....	.....

図 5: プロセスの記述の例

り, その中に誤りがある" という仮説を立てていたことが記述されている.

3 種類の状態の中で, プログラムを読んで理解している状態と実行している状態とについては, 更に, それぞれ 2 種類に分類できた. 考え込んでいる状態において立てた何らかの仮説に基づいて行動している状態と, 何の仮説も意識せずに行動している状態とである. この 2 種類の状態の間では, 5.2 に詳しく述べるように, バグの発見の効率に関して明らかに違いがあった.

また, 上述の状態のどれにも分類できない状態として, 次の状態があった. それは, バグの位置についての不確かな仮説を持ち, それが不確かであることを自覚していながらプログラムを修正しようとしている状態である. その後のプログラムの実行によりバグの症状が無くなるかどうかを確かめることにより, 変更した箇所がバグと関連しているかどうかを知ろうとする意図がある. 被験者 Y の試行において, 2 回, 観察された.

5.1.2 モデルの定義

前述の分類に基づいて, 図 6のように, 6 個の状態と 1 個の終了状態とからなる遷移モデルを定義した.

状態 H. バグの位置や種類についての仮説を立てている, あるいは, その妥当性を考えている, あるいは, 検証する方法を考えている. バグの位置とは, プログラム中に実在する手続きや文とは限らない. 被験者がその存在を仮定する手続きや条件分岐の位置も含む.

状態 R1. バグについて何の仮説も意識しないでプログラムを読んで理解を進めている.

状態 R0. バグについての特定の仮説に基づいてプログラムを読んで理解を進めている. 後述するように, 状態 R1 と比べると理解の戦略に違いがある.

状態 E1. バグについて何の仮説も意識しないでプログラムを実行している. プログラムに対する入力が, ほとんどランダムに選ばれる.

状態 E0. バグについての特定の仮説に基づいてプログラムを実行している. プログラムに対する入力が, 仮説に基づいて選ばれる.

状態 M. バグの位置についての仮説が不確かであると自覚しているにもかかわらず, プログラムを修正しようとしている. プログラムの編集の後にコンパイルや実行を行なっている状態も, この状態の一部であると考える.

5.1.3 状態遷移の例

被験者 Y の 1 回目の試行と, 被験者 S の試行における状態遷移を, 図 7に例示する. 横軸は, 時間を表しており, それぞれの図の左上端がデバッグを開始した時刻を表しており, 折り返している点がその 10 分後を表している.

なお, 被験者 S については, 6 個の状態のどれにも含まれない "その他の状態" が記されている. これは, 例えば, ウインドウの大きさや位置を変えたり, 文字のフォントを変えたりしている状態である. どの被験者にも観察されるが, バグの発見と無関係な行動であり通常は短い行動であるので, 図 6のモデルには含めていない.

5.2 観察された相違点

各試行における状態遷移は, 図 6のモデルに従っていたが, 各状態へ遷移する頻度や留まる時間は, 図 7のように, 試行の間で著しく異なっていた. また, 各状態における行動にも相違が観察された. 以下では, バグの発見の効率に影響すると思われる, 7 個の主な相違点を列挙する.

5.2.1 状態 R0 と R1 とへの遷移の頻度

全ての被験者について, 状態 R0 と R1 へのそれぞれの遷移が観察されたが, その頻度に明らかな相違があった. また, 5 回の試行におけるバグの発見は, 全て, 仮説に基づいてプログラムを読んでいる

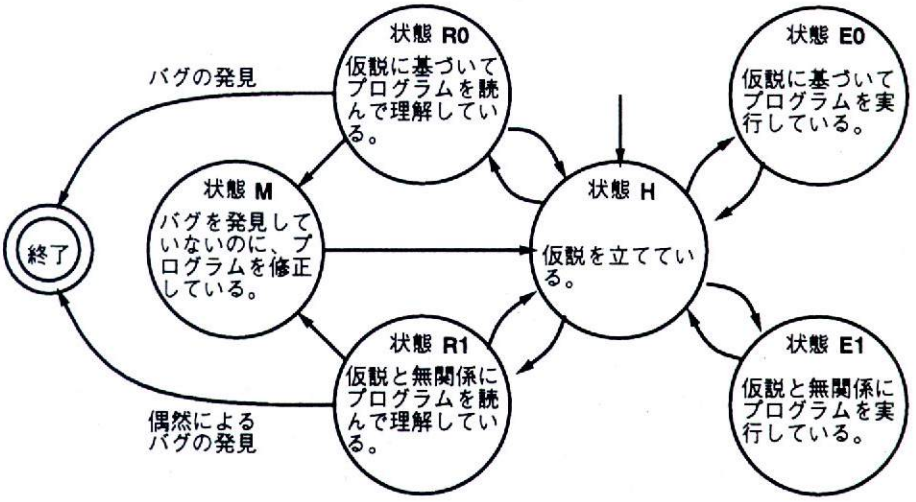

図 6: バグ発見のプロセスのモデル

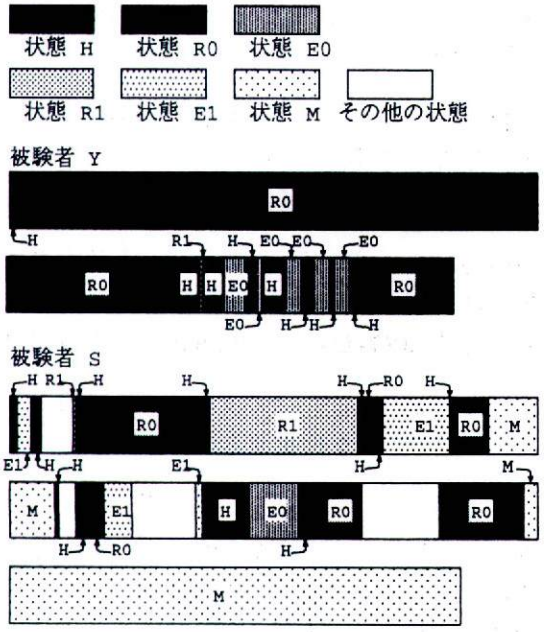

図 7: 被験者 Y と S の状態遷移の例

状態 R0 において起こり, 仮説を意識せずに読んでいる状態 R1 では起こらなかった.

5.2.2 プログラムを読む前の推定の量

プログラムを読む前に特定の手続きや条件分岐を仮定することについては, 試行によって, 全く行なわれない場合と, よく行なわれる場合とが観察された. 特に, 被験者 Y は, そのような仮定を行なっていた. 仮定が正しかった場合には, 短時間でバグが発見できていた. このような仮定が行なえるかどうかは, 被験者の経験に依存すると思われる.

5.2.3 理解の困難な部分に遭遇した後の行動

状態 R0 や R1 において理解の困難なプログラムの部分に遭遇した時には, 次のような何通りかの行動が観察された. 1 つは, その部分を理解することを諦めて, 別の部分を理解しようとすることである. もう 1 つは, その部分を長い間眺めて, 何とか理解しようと勤めることである. また, その部分にバグがあると考えた場合に, 状態 M への遷移が見られる場合があった.

5.2.4 理解するための戦略

状態 R0 や R1 のプログラムの理解においては, 次のような多様な戦略が採られていた. しかも, 1 回の試行の中でも採られる戦略が頻繁に変わっていた.

制御フローを辿る. C 言語の main 関数の先頭を起点として, 実行される順にプログラムを辿る. 状態 R0 と R1 の両方で主に採られていた戦略である.

リストを上から下に読み進む. プログラムの制御フローやデータフローには関係なく, リストの順に読んで理解する. 状態 R0 よりも R1 において多く観察された.

出力からデータフローを遡る. バグを出現させた出力文を起点として, データフローを逆に辿る. 2 回の試行において観察されたが, 状態 R1 では観察されずに, R0 においてだけ観察された.