ProTest: An Automatic Test Environment for B Specifications

Manoranjan Satpathy[1] Michael Leuschel[2] and Michael Butler[2]

[1] Department of Computer Science

University of Reading, Reading RG6 6AY, UK

[2] School of Electronics and Computer Science

University of Southampton, Highfield, Southampton, So17 1BJ, UK

M.Satpathy@reading.ac.uk, {mal,mjb}@ecs.soton.ac.uk

Abstract

We present ProTest, an automatic test environment for B specifications. B is a model-oriented notation where systems are specified in terms of abstract states and operations on abstract states. ProTest first generates a state coverage graph of a B specification through exhaustive model checking, and the coverage graph is traversed to generate a set of test cases, each being a sequence of B operations. For the model checking to be exhaustive, some transformations are applied to the sets used in the B machine. *The approach also works if it is not exhaustive; one can stop at any point in time during the state space exploration and generate test cases from the coverage graph obtained so far. ProTest then simultaneously performs animation of the B machine and the execution of the corresponding implementation in Java, and assign verdicts on the test results. With some restrictions imposed on the B operations, the whole of the testing process is performed mechanically. We demonstrate the efficacy of our test environment by performing a small case study from industry. Furthermore, we present a solution to the problem of handling non-determinism in B operations.*

keywords: Specification Based Testing; B–Method; Test Environment; Non Determinism.

1 Introduction

Software testing is broadly classified into two categories: *structured testing* and *functional testing* [2, 6]. Structured testing (or white-box testing) derives test cases from the structure of the implementation or part of the implementation. Such test cases are derived from a programmer's perspective with the aim of covering as much as possible the structure of the object under test. This approach is likely to miss out many bugs because it may give all the code coverage that we may need, but it may not give us all of the system coverage that users may expect. The test cases for functional testing (or black-box testing) on the other hand are written from a user's perspective. They

are derived from the external specification of the software behaviour with no consideration given to the internal organisation, logic, control or data flow. Structured tests tell a developer that the code is doing things right while functional tests tell a developer that the code is doing the right things [8].

Functional testing involves executing the implementation under test in relation to a set of test cases and examining the correctness of the generated output. In this context, we have the following issues:

- *Generation of test cases:* How to obtain test cases so that they cover all features of a requirement under all scenarios?
- *Execution of the test cases:* How to execute the test cases which are obtained from requirements or specifications? This may be a difficult task because even if the implementation preserved the intent of the requirement/specification, it may not preserve the structure or the logic of the latter.
- *Validation of test outcomes:* Once we run the test cases, the program would produce some outputs. How to ensure that the results are correct?

If the development process is formal, many of the above issues can be handled in a rigorous manner. Formal specifications precisely define the high level aspects of a software while omitting the detailed structural information; they are more likely to encode all of the required functions and their scenarios. Therefore testers can use the underlying mathematical framework to generate, possibly mechanically, test cases for functional testing.

Even if we obtain test cases from specifications, it may not be easy to use them to execute the implementation. This is because a high level functionality may have been implemented in a variety of ways, and the mapping between the high level functionality and the low level implementation may not be apparent to the tester. Consider an example of a test case being a sequence of high level operations at specification level, but this operation sequence may not map easily to the operations at the implementation level. Some authors have proposed the use of special mappings called *representation mappings* to bridge this semantic gap [17]. In addition, there is the problem of non-determinism. The choice made by a non-deterministic operation may not correspond to the deterministic choice made by the implementation. And then how are we going to use a test case involving non-determinism?

When a system executes a test case, it produces an outcome, and the outcome is often interpreted by the tester to assign a verdict that the system has passed the test. This problem can be tackled by incorporating oracles into the testing process [17, 23]. A test oracle determines if the system behaved correctly in relation to the test case. Test oracles are usually obtained from specifications. The outcome of a test case and the outcome obtained from a test oracle need to be matched to establish the equivalence between abstract outputs and concrete results. There are two issues in this context; first, there must be a mapping between the abstract state of the specification and the concrete state of the implementation, and second, there must be a mechanism to show their equivalence. The first problem can be solved by representation mapping; Antony and Hamlet [4] have discussed how the users could write explicit code for a representation mapping between the concrete data structures of C++ instance variables and the abstraction of the specification. And the second can be addressed though the use of probing or observation operations both at the abstract as well as at the concrete state levels.

In this paper, we discuss *ProTest*, an automatic test environment for B specifications. ProTest is based on ProB, a model checking and animation tools for B [16]. ProTest follows an approach similar to the one by Dick and Faivre [10] (discussed in Section 2) and generates test cases from B specifications by partition analysis of the state invariant and the operation preconditions of a specification. Our method offers some guidelines and if the implementation follows them, then the whole cycle of the testing process can be automated. We also discuss a small industrial case study to illustrate our approach and the test environment. The main results of our paper can be summarised as follows:

- ProTest generates test cases by partitioning and exploring the state space. ProTest then simultaneously animates the specification and runs the implementation with respect to the test cases and assigns verdicts whether the implementation has passed the tests. The whole process is automatic; however, at this stage the test environment imposes some restrictions on operation arguments and results.
- We have presented a solution to handle non-determinism in B operations; however the current implementation of the ProTest does not support this.

The organisation of the paper is as follows. Section 2 discusses the related work. Section 3 presents our approach. Section 4 discusses our implementation and in Section 5 we preset an analysis of our test environment in relation to existing work. Section 6 concludes the paper.

2 Related Work

The concept of specification based testing most probably originated from the work by Hall [13] in which he discussed partitioning the input space by examining predicates in the operations of Z specification [20]. The aim was to induce software correctness based on test results.

The work by Dick and Faivre [10] is a major contribution to the use of formal methods in software testing in which they have discussed a strategy for generating test cases from model oriented formal specifications. A VDM [14] specification has state variables and an invariant (Inv) to restrict the state variables. An operation, say OP, is specified by a pre-condition (OP_{pre}) and a post-condition (OP_{post}). The approach of Dick and Faivre is to partition the input space of OP by converting the expression $OP_{pre} \wedge OP_{post} \wedge Inv$ into its Disjunctive Normal Form (DNF); and each disjunct of it represents an input subdomain of OP. Next, as many operation instances of OP are created as the number of non-contradictory disjuncts in the DNF. An attempt is then made to create a FSA (Finite State Automaton) in which each node represents a possible machine state and an edge represents an application of an operation instance. A set of test cases are then generated by traversing the FSA where each test case is a sequence of operation instances. The work of Dick and Faivre discusses only the mechanisation of the partitioning algorithm.

Legeard et al. [15] have developed a tool called the BZ Testing Tool (BZ-TT) for deriving test cases from Z or B specifications. Since our approach has many similarities with the BZ-TT, we present it here in some detail. So far

as B specifications are concerned, they assume (i) the specification consists of a single B machine, and (ii) all sets in the B machine are transformed into finite enumerated sets. The test case generation proceeds in the following steps:

- The definition of each B operation Op is transformed into its *normalised form* [1] which looks like:
  $outs \longleftarrow Op(inps) = Pre \mid @s', outs' \cdot Post \implies outs, s := outs', s'$
  where, s is the state variable of the machine, Pre is the precondition (over $inps$ and s) and $Post$ is the postcondition. $inps$ and $outs$ are respectively the operation input and the result. The normal form tells: provided Pre is true, the values s' and $outs'$ are non-deterministically chosen such that $Post$ is satisfied. $Post$ may refer to s and $outs$ as well as s' and $outs'$.

- Pre and $post$ are transformed into their DNF; i.e
  $(\bigvee_i \ Pre_i) \mid @s', outs' \cdot (\bigvee_j Post_j) \implies outs, s := outs', s'$

- The above expression partitions the input space into subdomains of the form:
  $\exists \ inps, s', outs' \cdot (\bigvee_i Pre_i) \bigwedge Post_j$.
  Test cases are generated from the above expression using a CLP (Constraint Logic Programming) solver.

However, in order to generate boundary goals, BZ-TT uses cost functions to partition further the input subdomains. If an input subdomain is represented by the predicate $\#(W \cup R \cup A) < \#\{X_1, X_2, X_3\}$, then some candidates for the maximization and minimization cost functions could be $\#W + \#R + \#A = 2$ and $\#W + \#R + \#A = 0$ respectively. Given a boundary condition, Prolog search techniques are used to generate a test preamble. At a boundary state, all eligible operations are applied to generate test cases as sequences of operation instances. From the test cases, automatic test scripts are generated in the target language, and representation mappings are created manually. Because of problems due to non-determinism and those related to matching between abstract and concrete states, automatic verdict assignment was not implemented. It is to be noted that the BZ-TT does not handle constants, properties and set comprehension, all of which we use in our case study.

The work of Richardson et al. [17] discusses the derivation and use of test oracles for checking test results in the context of multi-lingual and multi-paradigm (formal) specifications. Test oracles are derived from specifications in conjuction with the derivation of test data in relation to some testing criteria. Test execution is monitored and the results are verified against oracles; sometimes the authors considered it useful to compare intermediate results in addition to the end results. To make verification possible, their approach constructs mappings between the name space of the implementation and the name space of the oracle (same as the name space of the specification). There are two kinds of mappings: control and data. Control mappings are between control points in the implementation and locations in the specification where the implementation and the specification should be in same state. Data mappings describe the transformation between the data structures in the implementation and objects in the specification. These mappings are also called representation mappings [15], and usually they are developed manually. The implementation state and the state changes are monitored at the pre-determined control points, and data map-

pings are used to establish the correspondence between the implementation and the specification state as oracle. The authors point out that many of the steps described could be automated.

3 Our Approach

Let us assume that a formal specification has adequately specified all the requirement functions under all possible scenarios. Then our aim is to generate test cases which would test all such functions of the corresponding implementation under the given scenarios. In addition, our test environment would examine the test results for assigning verdicts. ProTest is a test environment for B specifications.

3.1 The B method

The B-method, originally devised by J.-R. Abrial [1] is a theory and methodology for formal development of computer systems. B is model-oriented in the sense of Z and VDM; B is used to cover the whole of the software development cycle; the specification is used to generate code with a sequence of refinement steps in between. At each stage, the current refinement needs to be proved consistent with the previous refinement.

The basic unit of specification in the B method is called a *B machine*. Larger specifications can be obtained by linking B machines in a hierarchical (tree like) manner. This is a design restriction on the B method with view to making proofs compositional. A B machine consists of a set of variables, an invariant to restrict the variables and a set of operations to modify the state variables. A machine has an initial state which initializes the state variables. An operation has a precondition, and an operation invocation is defined only if the precondition holds. The initialization action and an operation body are written as atomic actions coded in a language called the *generalized substitution language* [1]. The language allows specification of deterministic and non-deterministic assignments. An operation transforms the machine state to a new state. The behaviour of a B machine can be described in terms of a sequence of operations; the first operation of the sequence originates from the initial state of the machine.

3.2 Our Example

For our case study, we will consider a component of the teletext system of a commercial television from Philips Electronics [18]. The component description is as follows: a TV screen has a display window consisting of R rows which can display a sequence of N teletext page titles. At any time a subsequence of the transmitted sequence could be displayed and therefore, the display of the N page titles ($N \geq R$) would require scrolling. Page titles could be scrolled by pressing the *up* and the *down* arrow buttons of the TV remote. Every slot of the display window has a default colour and it can display a teletext page title. At any point in time, the cursor resides on exactly one slot which is displayed with a different colour. The component has non-trivial cursor movement operations. A pictorial form of the component has been shown in Figure 1. We have specified this teletext component as a single B machine and a sketch of it has been shown in the Appendix.

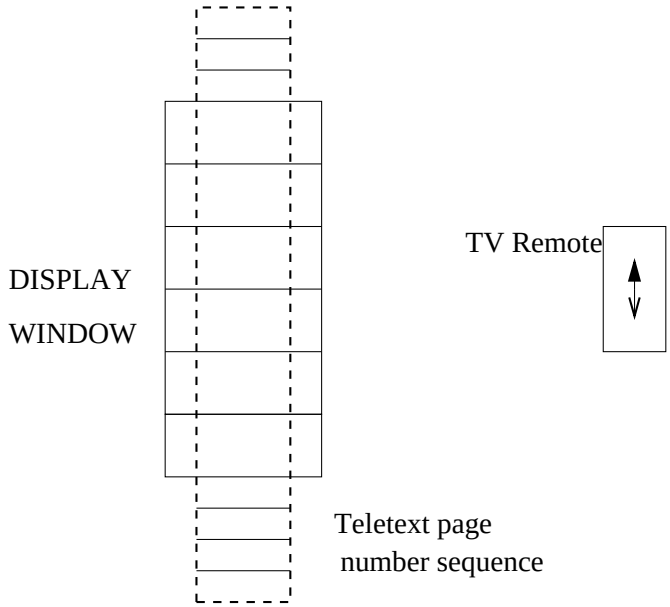

Figure 1. Display of N page titles over the Display window of size R

A B machine has a name, and in our case it is called *Teletext*. The *SETS* clause shows the sets those will be used by the machine. The *CONSTANTS* clause declares the constants used in the machine and the *PROPERTIES* clause tells of their types and values. The *VARIABLES* heading shows the state variables and the *INVARIANT* clause puts restrictions on the state variables in the form of predicates; in the appendix we have shown a fragment of the invariant. The *INITIALISATION* clause initializes the state variables. The *OPERATIONS* clause shows a set of operations which can either probe the machine state or modify it. In our example, the most important operations are $define$, $upCursor$ and $downCursor$. The $upCursor$ tries to move the cursor one position up and it may involve scrolling. The $downCursor$ tries to move the cursor one position down and it may involve scrolling in the opposite direction. All such actions are possible if the $define$ operation has placed the machine in a *defined mode of operation*.

3.3 Partitioning of the input spaces

A B machine has a state which can be modified through the operations of the machine. The *INITIALISATION* clause puts a B machine in its initial state. Thereafter, as and when the precondition of an operation holds, the operation is eligible for application. Application of the eligible operations defines the behaviour of the machine. Following the terminology of Dick and Faivre [10], we will define a test case as a sequence of eligible operations. Every operation has a precondition which defines its input space. Dick and Faivre, partition this input space into subspaces meaning that each subspace defines a possible scenario under which the operation can be applied. We follow the same approach for a B machine. We enumerate our partitioning method in the following steps.

Step 1: Consider a machine operation OP_1. Compute the DNF of the precondition of the operation. As pointed

out by Legeard et al. [15], in practice, the precondition of an operation is sometimes trivial and therefore, a DNF based analysis would not result in interesting partitions. In order to address this problem, we do the following transformation. Consider an operation with an IF construct in its body such as: *IF* $< if\text{-}predicate >$ *THEN* ... *ELSE* ... We then add (through conjunction) the predicate $(< if\text{-}predicate > \vee \neg < if\text{-}predicate >)$ to the precondition. Note that the above is a *tautology*, and therefore, it does not modify the precondition but it results in a better partition of the input space. We do the same for all the if–predicates in the operation. Refer to the operation $upCursor$ in Figure 3.3, and observe how the original precondition has been expanded to create more partitions.

Note that this transformations is not only limited to IF predicates. They are also applied over the predicates of the *CASE* and the *SELECT* statements. The rules for adding predicates to the precondition have been shown in the Appendix. After all these transformations of the precondition, it is subjected to the DNF analysis. Note that, for the moment, situations like an IF statement inside an ANY statement is ignored. The problem is: it might depend on the bound variables which are not part of the input or the initial state.

Step 2: Let the DNF of the precondition be the disjunction of the disjuncts $C_1, C_2, \ldots, C_p$. The way we have lifted tautologies constructed out of the predicates in an operation body, means that some of these disjuncts may be self contradictory, and further some of them might contradict the invariant of the B machine. We then filter out these contradictory disjuncts by subjecting them to a naive theorem prover. Let the disjuncts that remain after filtering are $C_1, C_2, \ldots, C_k$. These disjuncts partition the input space of OP_1 into k subspaces.

Step 3: Create k instances of the operation OP_1; let the instance OP_1^i corresponds to disjunct C_i, $1 \leq i \leq k$. What this means is that the instance OP_1^i is eligible for application when the condition C_i holds. The way we have lifted the predicates to the pre-condition, implies that each operation instance represents a valid control path inside the operation OP_1.

Step 4: Create similar instances for all operations in the machine.

Step 5: The full state space of the B machine is explored to construct a FSM (finite state machine) whose initial node is the initial state of the B machine. Each node in the FSM represents a possible machine state and each edge is labelled by an operation instance. Of course, to explore the full state space, it is assumed that all the sets of the specification are of finite type and they are small in size. The state space search is performed by the ProB tool; more about the implementation will be discussed later. The aim here is that all the operation instances which we have generated in our partition analysis appear at least once in the FSM. It may not be possible since some operation instances may not be reachable.

Step 6: Starting from the initial state, traverse the FSM to generate a set of operation sequences such that each operation instance in the FSM appears in the generated sequences at least once. Each operation sequence should start with the initial state, and an operation instance may appear in more than one sequence. Each such sequence would constitute one test case for the subsequent implementation. And the set of test sequences would be our test suite. The traversal of the FSM to generate an optimal number of test sequences is a NP–complete problem [12]; therefore, we need to follow some heuristic for traversing the graph.

(A)

```
upCursor = PRE   Status = DEFINED  THEN
              IF Selected > 1 THEN
                 Selected := Selected - 1 ||
                 IF (Plist_size >= Display_size) THEN
                     IF Scroll > 0  THEN
                        IF Selected = Scroll + 2  THEN
                            Scroll := Scroll-1 ||  ....
                        ELSE IF   (DisColours(Display_size)=white &
                                   DisPnames(Display_size)=blank)    THEN
                                   DisColours := {nn,cc | ....}
                             ELSE  DisColours := {nn,cc | ......}     END
                        END
                     ELSE  /* Scrolling not necessary */
                             IF Selected >= 2 THEN DisColours := {nn,cc | ...}
                             END
                     END
                 ELSE   DisColours :=  {nn,cc |...}  END
           END END;
```

(B)

```
upCursor =  PRE
     Status = DEFINED & (Selected > 1 or Selected <= 1)          &
     (Plist_size >= Display_size or Plist_size < Display_size)   &
     (Scroll > 0  or  Scroll <= 0)                               &
     (Selected = Scroll + 2   or  Selected /= Scroll +2)         &
     (Selected >= 2  or Selected < 2)                            &
     ((DisColours(Display_size) = white & DisPnames(Display_size) = blank) or
     (DisColours(Display_size) /= white or DisPnames(Display_size) /= blank)) &
     (Selected = 1 or Selected >1) & (Scroll = 0 or Scroll > 0)
THEN  ......  END;
```

Figure 2. Lifting of predicates to a precondition: (A) Definition of operation *upCursor*, (B) New pre-condition of *upCursor*

3.4 Testing strategy

Each test case of the test suite we have generated is nothing but a sequence of operations of the B machine that specified our problem. We will not address the issue of non-determinism here; it will be discussed in a later section. If we could animate the B machine with respect to a test case, at the end of covering the test sequence, we would obtain a state, say State$_{spec}$. Let us assume that we have an implementation of the B-machine and we are able to execute the implementation in relation to the same test case, and let the resulting state be called State$_{impl}$. Now if we are able to match State$_{spec}$ with State$_{impl}$ then we could assign a verdict whether the implementation has passed the test. The whole process has been shown in the figure 3.

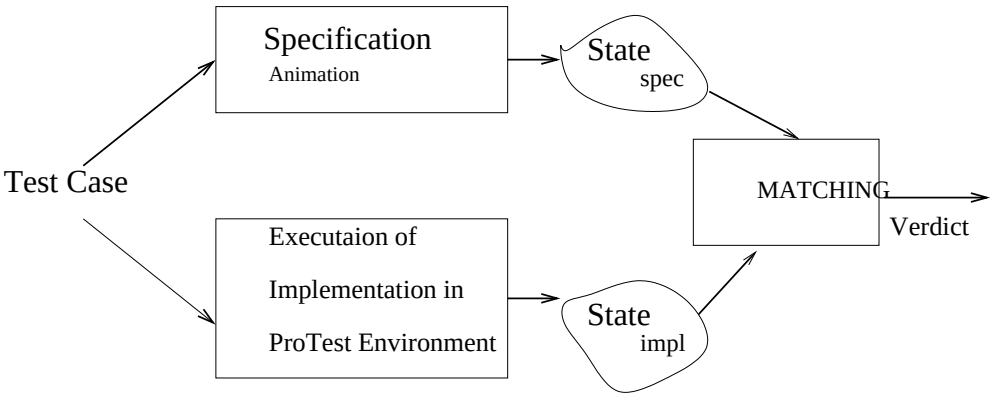

Figure 3. The testing process

3.5 The Matching Problem

The operations of a B machine can be divided into two categories: *update operations* which can modify the machine state, and *probing operations* which only perform queries on the state variables. The probing operations can query the system state to extract out important state aspects. Table 1 shows all the update and the probing operations of our case study. We assume that the implementer implements all the probing as well as the update operations. The probing operations of the specification and those of the implementation can now respectively query important properties of State$_{spec}$ and State$_{impl}$ and match their responses. Of course this would require mapping between the name space of State$_{spec}$ and that of State$_{impl}$, a mapping similar to the work of Richardson et al [17].

4 Implementation: The ProTest Environment

ProTest is a test environment built on top of the ProB tool which has been developed by Leuschel and Butler [16]. In the following, we will present a brief description of it.

4.1 The ProB Tool

The ProB tool is an automated consistency checker of B machines via model checking and constraint-based checking. The activity of consistency checking shows that the operations of a machine preserves the machine invariant. The ProB environment has been developed mainly in SICStus Prolog with a graphical user interface implemented in Tcl/Tk. ProB uses the JBTools [22] package to translate a B machine into XML form, and then the Pillow package [7] is used to to transform the XML files into a Prolog term representation. The ProB front end then uses this Prolog term representation. The ProB animator provides visualization of the state space that has been explored so far by the animator. Further details about the ProB can be found in [16]

The model checker component of ProB tries to explore the state space of a B machine systematically and automatically. It alerts the user as soon as a problem like the invariant violation is found, and then presents the shortest trace within the states already explored that leads from the initial state to the place of error. The model checker also detects when all states for finite state models have been explored, and thus can formally guarantee the absence of errors. For such exhaustive model checking, the sets of the machine are restricted to small finite sets and integer variables are restricted to small numeric ranges. Under these restrictions only, ProB can traverse all the reachable states of the machine. ProB can also animate a B machine. In addition, ProB supports random animation in which eligible operations are applied at random till their number reaches a certain limit given by a user.

In addition to temporal model checking, i.e. model checking of the above type, ProB also supports *constraint based checking*. If there is an invariant violation because of an operation invocation, ProB model checker can find it through systematic exploration. However, constraint based checking finds a state of the machine that satisfies the invariant but where we can apply a single operation to reach a state that violates the invariant. ProB supports this approach through the use of Prolog's co-routining and constraint facility [16]. We will use both these approaches of ProB in our testing environment.

4.2 The ProTest Tool

Our test environment makes the following assumptions:

1. The B specification consists of a single machine. This is because, at this stage, ProB can animate and model check a single B machine.

2. The operation parameters of the machine and those of the implementation are of basic types, and in addition the operations have a single return value of basic type. Our current implementation performs automatic verdict assignment under these restrictions. They keep the representation mapping between the specification and implementation namespaces simple. However, in future, we intend to lift these restrictions.

3. All the machine operations are deterministic. Note that in this paper we will present our solutions to handle non-determinism; however, our current implementation does not support them.

The ProB tool has been augmented with the following enhancements to build the ProTest environment:

- The Partition Analyser: The preconditions and the machine invariants are extracted and both are converted into their DNFs. A naive theorem prover eliminates all the disjuncts from the DNF of the precondition which are either self-contradictory or which contradict the invariant. The remaining disjuncts are used to create partitions of the operation input space, and then the operation instances.
- ProTest has an interface for running Java Programs with respect to test cases, and to explore the execution states through the use of probing operations.
- Coverage Graph Display: ProTest can display the state space coverage in the form of a graph. Nodes in this graph represent the abstract machine states and the edges are labelled with the operation instances. An edge signifies state transformation through the application of the labelled operation instance.

4.3 Mechanical generation of test cases

In a pre-processing phase, the infinite and deferred sets of the B machine are transformed into finite enumerated states; and also the sizes are kept small to facilitate exhaustive model checking. The ProTest partition analyzer partitions the input space of each B operation to generate a set of operation instances. Then the ProB model checker tries to explore the whole state space and generates the state coverage graph. The coverage graph is a directed graph and it has a start state which is also termed the root of the coverage graph. Figure 4 shows the coverage graph for a particular assignment (of the parameters of the $define$ operation) of our case study; each edge has been labelled with an operation instance. Note that a different assignment of the $define$ operation would result in a different coverage graph. Table 1 shows the number of operation instances covered by some of the graphs generated. It can be seen that for operations $upCursor$ and $downCursor$, only a small percentage of instances appear in the coverage graphs. The reasons are: (i) the given initialization and the constant set-up makes many instances unreachable, and (ii) our naive theorem prover at this stage does not remove some partitions which could be inconsistent. It is to be noted that a large majority of the uncovered partitions are contradictions, and at the moment our simple theorem prover does not catch them; we are working on using CLP to catch more.

The following heuristic is used to traverse the state coverage graph to generate a set of test cases, each test case being of the form $(preamble(p) :: OP, postulate(N'))$, where OP is the label of an edge joining the node pair (N, N'), $preamble(p)$ is the sequence of labels of a path from the root of the coverage graph to N, $preamble(p) :: OP$ is the sequence obtained by inserting OP at the end of $preamble(p)$, and $postulate(N')$ is the test oracle of the test sequence. $postulate(N)$ is obtained from the node N' which is constituted from the results of the probing operations on the state represented by N'. This heuristic uses Dijkstra's shortest path algorithm [11].

Algorithm: *Generate-Test-Sequences*

input: the coverage graph given by ProB model checker.

output: a set of operation sequences as test cases and postulates for each.

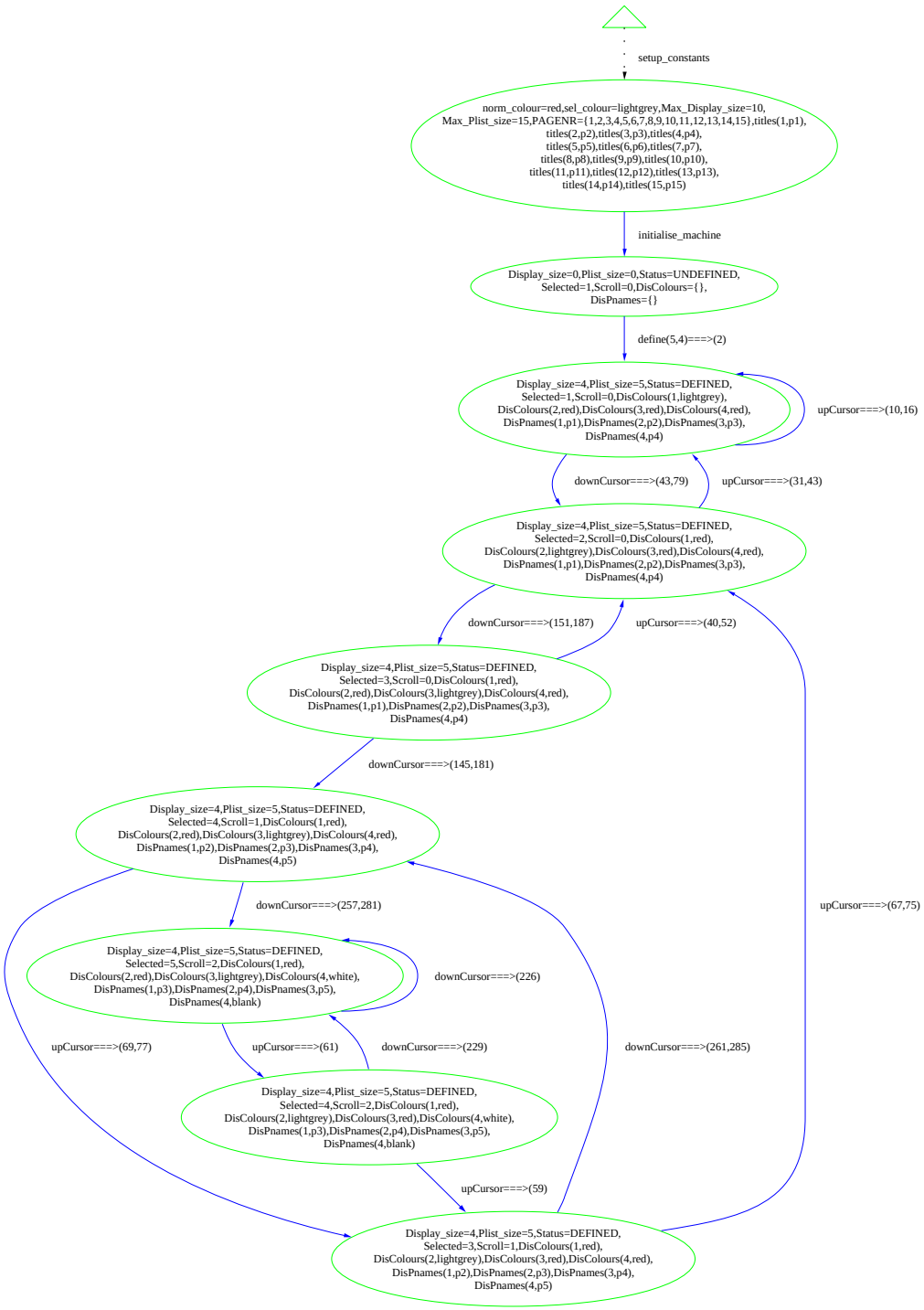

Figure 4. State coverage Graph

Operation name	Type	No. of partitions generated	No. of partitions covered
PageAtColumn1	probing	1	1
ColourAtColumn1	probing	1	1
CurrentCursorPosition	probing	1	1
PageAtCurCursor	probing	1	1
ColourAtCurCursor	probing	1	1
PageAtLastColumn	probing	1	1
ColourAtLastColumn	probing	1	1
define	update	3	3
undefine	update	1	1
upCursor	update	78	15
downCursor	update	288	18

Table 1. partitions for each operation

$< define \rightarrow (2), \mathbf{upCursor} \rightarrow (\mathbf{10}, \mathbf{16}) >$
$< define \rightarrow (2), downCursor \rightarrow (43, 79), \mathbf{upCursor} \rightarrow (\mathbf{31}, \mathbf{43}) >$
$< define \rightarrow (2), downCursor \rightarrow (43, 79), downCursor \rightarrow (151, 187), \mathbf{upCursor} \rightarrow (\mathbf{40}, \mathbf{52}) >$
$< define \rightarrow (2), downCursor \rightarrow (43, 79), downCursor \rightarrow (151, 187), downCursor \rightarrow (145, 181), \mathbf{upCursor} \rightarrow (\mathbf{69}, \mathbf{77}) >$

Table 2. Some test sequences obtained from the graph of Figure 4

{
Mark every edge in the graph as *uncovered*;
$while$ there is an uncovered edge (N, N') with label OP_j {
find the shortest path p from the root of the graph to node N using Dijkstra's algorithm;
Output test case for OP_j: $(preamble(p) :: OP_j, postulate(N'))$;
Mark all edges with label OP_j as covered;
for each uncovered edge (N_a, N_b) with label OP_{ab} occurring in path p {
Output test case for OP_{ab}: $(preamble(p') :: OP_{ab}, postulate(N_b))$, where
p' is the path from $root$ to N_a (p' is a prefix of p);
Mark all edges with label OP_{ab} covered;
}
}

By altering the assignments to the arguments of the $define$ operation to deliberately introduce an error in the implementation, we have generated 43 test cases. Some of these test cases are shown in Table 2; the table does not show the postulates which are generated for each of the test sequences. Furthermore, observe that the test sequences in the table have been generated from the coverage graph of Figure 4.

4.4 Automating the test execution and verdict assignment

ProTest environment at this stage can deal with Java implementations. Let us assume, we have a Java implementation which has encoded all the update and the probing operations of a B machine. In this sense, the

```
test_case_generate: Test case successful: test(downCursor,177)
test_case_generate: Test case successful: test(downCursor,132)
test_case_generate: Test case successful: test(upCursor,21)
test_case_generate: Test case successful: test(upCursor,31)
test_case_generate: Test case successful: test(downCursor,151)

bjasper: Java and B Return Value Mismatch: (blank\==p7)
bjasper: Java and B Return Value Mismatch: (white\==red)
test_case_generate: Test case has failed: test(downCursor,253)
```

Table 3. Some test outputs of passed and failed test cases

implementation has been directed by the specification. Richardson et al. [17] point out that while running a test case, examination of the test result is not the only interesting observation; sometimes intermediate results can be examined at strategic points which they call control points. In our case, control points could be the positions before and after operation (or operation instance) invocations. ProTest has the capability to perform simultaneous specification animation and code execution, and at selected control points, both the specification and the execution states are examined by invoking their respective probing operations. The results of the probing operations are matched by using the representation mappings, and if there is a mismatch, it is reported to the user.

As mentioned earlier, by altering the assignments to the parameters of operation $define$, we have generated 43 test cases. In our first attempt, we obtained 22 test cases in relation to $define(5, 4)$ out of which 21 failed. The reason was that the argument ordering for the $define$ operation was swapped in the implementation. After it was corrected, 4 test cases still did not pass. This time the reason was an error in the specification; one slot in the display column was getting assigned to a wrong value. Interestingly, this error was not discovered during the model checking since it was not violating the invariant. The specification was corrected and after that all the test cases passed the tests. In the end, all the 43 test cases passed their tests. Table 3 shows test outputs generated by ProTest for some of the above test cases.

4.5 Handling non-determinism

B supports two types of non-determinism: bounded choice through the syntactic construct *SELECT* and unbounded choice through the construct *ANY*. In a SELECT construct, there are a finite number of guarded substitutions and a branch whose guard evaluates to ture is non-deterministically chosen. In an ANY construct, an element of a set is non-deterministically chosen.

Let us consider a B operation *OP* having non-deterministic constructs. In order to handle the non-deterministic choices made by OP, we require that OP makes its choices visible by delivering them through result parameters (in addition to other result parameters of the operation). If OP has made k non-deterministic choices, then the operation looks like:

$r_1, r_2, \ldots, r_k, result \longleftarrow$ OP = PRE P

THEN . . . END

Here $r_1, \ldots, r_k$ are the k non-deterministic choices made in the course of operation *OP*. Let us consider the case when ProTest is doing simultaneous animation and execution during testing with respect to a test case, and we have reached the operation OP. At this point ProTest observes what choices the implementation has made. Thereafter, ProTest will follow the choices made by the implementation. We term this approach *testing on the fly*. Note that the current version of the ProTest does not support this aspect.

5 Discussion

The following are the highlights of the ProTest environment.

1. Partition Analysis: Many other works like [10, 15] partition the input space by considering both the pre- and the postconditions of an operation. The reason they cite is that usually the operation preconditions are trivial in nature, and a DNF analysis over them would hardly result in worthwhile partitions. In our case, we strengthen an operation precondition by lifting predicates used within the operation bodies. We have observed that our approach generate the same number of partitions as the one by Legeard et al [15]. However, partition analysis in presence of non-determinism may need some more analysis with a view to creating further partition of the input space.

2. ProTest tool performs simultaneous specification animation and code execution to demonstrate that both exhibit equivalent behaviour with respect to test cases. Further ProTest makes it easier to check and validate intermediate results. ProTest performs automatic verdict assignment through the use of representation mappings. However at present this task is easier because we only consider simple types for operation arguments and results. Automatic verdict assignment in presence of complicated data structures would be a challenging task. One solution could be to choose probing operations judiciously which can extract relevant and important information out of complicated data structures; this will keep the matching of specification and implementation states within reasonable complexity. However, this aspect need to be further explored though larger case studies.

3. Non determinism: Handling non-determinism is a novelty of our approach. The requirement of making the non-deterministic choices visible does not pose any additional burden on the specifier; however, the implementer needs to be instructed to make the corresponding deterministic choice visible by some mechanism such as the use of output operation parameters. In other words, an implementer need to be faithful to many such recommendations from the specifier(s). Our approach of making testing on-the-fly may bring out interesting test cases, which the static analysis may not reveal.

4. Once input subdomains are derived after a DNF based analysis, the BZ-TT approach uses some (minimization/maximization) cost functions to further partition the input space and then test cases are generated. At

this stage though the ProTest approach does not use cost functions, it can use them in future to create further partitions. This is just an enhancement which can be easily integrated into the ProTest environment.

5. Reachability Analysis: ProTest performs exhaustive state space search to generate a state coverage graph. If it finds an invariant violation in the process, it not only reports it to the user, but it also informs the shortest sequence of operations that led to the invariant violation. This information can be used by the tester to perform intelligent debugging of the code.

   There may be operation instances generated by partition analysis which are not reachable in the course of exhaustive model checking. One possible reason may be the initialization condition of the machine which does not make it possible to reach the operation instance; however, there may be a different initialization which can make this operation instance reachable. This can be found out by the constraint based checking facility of ProB. ProB can even suggest an initialization condition which can make this operation instance reachable. In addition, it can be checked if application of this operation instance can lead to invariant violation.

   ProTest uses this facility of ProB to generate a set of robust test cases. Given an operation instance, not reachable through exhaustive model checking, it can say whether from a different initialization of the machine the operation instance is reachable. If so, the same initialization condition can be passed to the tester/implementer so that the implementation can be re-initialized. ProB can also give the set of operation sequences which can make the original operation sequence reachable.

6. The approach of BZ-TT is the closest to that of ProTest; however, there are important differences. First, the approach to partition analysis is different though both result in similar partition sets. Second, ProTest is different in the sense that it performs simultaneous specification animation and code execution to establish the correspondence. Third, ProTest performs automatic verdict assignment. In addition, We address the issue of non-determinism.

7. Snook and Butler [19] have developed a tool called *U2B* which mechanically translates UML specifications to B. Of course there are some restrictions on the UML classes so that when translated they do not violate the hierarchical structure of the B machines. Our ProTest environment could be integrated with the U2B tool which would facilitate mechanical generation of test cases from UML specifications. Further by testing the implementation against the generated B specification, this approach would indirectly establish the correspondence between the UML specification and the implementation.

6 Conclusion and Future Work

In this paper, we have presented ProTest, a testing environment for B specifications. The highlights of this tool are that it performs in parallel the animation of the specification and code execution with respect to test cases, and it assigns verdicts on the test results. We have also offered a solution to handle non-determinism in the B

operations. We have discussed the efficacy of the ProTest tool by performing a small case study from industry. We have also demonstrated how through the use of temporal model checking and constraint based checking, we can obtain a set of robust test cases.

The ProTest environment can be extended in many dimensions; in particular, we plan to do the following in future:

- Enhancing the ProTest environment to handle non-deterministic operations and to support on-the-fly testing.
- Integrating the U2B tool with the ProTest for generating test cases for UML specifications.

References

[1] J.-R. Abrial, *The B Book: Assigning Programs to Meanings*, Cambridge University Press, 1996.

[2] W. R. Adrion, M.A. Branstad and J.C. Cherniavsky, Validation, Verification and Testing of Computer Software, ACM Computing Surveys, Vol. 14(2), June 1982.

[3] F. Ambert, F. Bouquet, S. Chemin, S. Guenaud, B. Legeard, F. Peureux, N. Vacelet and M. Utting, BZ-TT: A Tool-Set for Test Generation from Z and B using Constraint Logic Programming, Formal Approaches to Testing of Software, Satellite Workshop of CONCUR02, August 24th, Brno, Czech Republic, 2002.

[4] S. Antoy and D. Hamlet, Automatically Checking an Implementation against its Formal Specification, IEEE Transactions on Software Engineering, Vol. 26(1), January 2000, pp.55–69.

[5] E. Bernard, B. Legeard, X. Luck and F. Peureux, Generation of Test Sequences from Formal Specifications: GSM 11–11 Standard Case Study, Unpublished Draft.

[6] B. Beizer, *Black-Box Testing: Techniques for Functional Testing of Software and Systems*, John Wiley, 1995.

[7] D. Cabeza and M. Hermenegildo, *The PiLLoW Web Programming Library*. The CLIP Group, School of Computer Science, Technical University of Madrid, 2001. Available at http://www.clip.dia.fi.upm.es/

[8] J. Canna, Testing, fun? Really? (Using unit and functional tests in the development process), website: http://www.106-ibm.com/developerworks/library/j-test.html.

[9] D. Carrington and P. Stocks, A Tale of Two Paradigms: Formal Methods and Software Testing, Proc. of the Eighth Z User Meeting (Eds. J.P. Bowen and J.A. Hall), Cambridge, Springer Verlag, 1994.

[10] J. Dick and A. Faivre, Automating the generation and sequencing of test cases from model-based specifications, Proc. of the FME'93: Industrial Strength Formal Methods Europe, LNCS 670, 1993, pp. 268–284.

[11] E.W. Dijkstra, A note on two problems in connection with graphs, *Numerische Mathematik*, Vol. 1, 1959.

[12] J. Gross, J. Yellen, *Graph Theory and its Applications*, CRC Press, 1999.

[13] P.A.V. Hall, Relationship between Specifications and Testing, Information and Software Technology, Jan/Feb 1991.

[14] C.B. Jones, *Systematic Software Development using VDM*, 2^{nd} Edition, Prentice Hall, 1990.

[15] B. Legeard, F. Peureux and M. Utting, Automated Boundary Testing from Z and B, Proc. of the FME'02 (Formal Methods Europe) Conference, LNCS No. 2391, 2002, pp. 21–40.

[16] M. Leuschel and M. Butler, ProB: A model-Checker for B, FM 2003: 12th International FME Symposium, Pisa, September 2003.

[17] D.J. Richardson, S.L. Aha, T.O. O'Malley, Specification-based Test oracles for Reactive Systems, Proc. of the 14th ICSE, ACM Press, 1992, 105–118.

[18] M. Satpathy, R. Harrison, C. Snook, M. Butler, A Comparative Study of Formal and Informal Specifications through an Industrial Case Study, IEEE/IFIP Joint Workshop on Formal Specifications of Computer Based Systems, Washington DC, April 2001.

[19] C. Snook and M. Butler, Verifying Dynamic Properties of UML Models by Translation to the B Language and Toolkit. In UML 2000 Workshop on Dynamic Behaviour in UML Models: Semantic Questions, October 2000.

[20] J.M. Spivey, *Understanding Z*, Cambridge University Press, 1988.

[21] P. Stocks and D. Carrington, A Framework for Specification-Based Testing, IEEE Transactions on Software Engineering, Vol. 22(11), 1996, pp. 777–793.

[22] B. Tatibouet, *The JBTools Package*, 2001, available at http://lifc.univ-fcomte.fr/PEOPLE/tatibouet/JBTOOLS/BParser_en.html.

[23] E.J. Weyuker, On Testing Nontestable Programs, The computer Journal, Vol 25(4), 1982, pp. 465–470.

Appendix

```
MACHINE Teletext
SETS  COLOURS =   {red,white,lightgrey};
      STATUS  =   {DEFINED,UNDEFINED};
      PAGENAMES = {blank,p1,p2,p3,p4,p5,p6,p7...p14,p15}
CONSTANTS     PAGENR, Max_Plist_size, Max_Display_size,
              sel_colour, norm_colour, titles
PROPERTIES
        PAGENR <: NAT & PAGENR =  1..15 &
        Max_Plist_size : NAT & Max_Plist_size = 15 &
        Max_Display_size : NAT & Max_Display_size = 10 &

        sel_colour : COLOURS & sel_colour = lightgrey &  /*Colour of slot at cursor*/
        norm_colour: COLOURS & norm_colour = red &       /*Usual colour of  slots  */
        titles     : PAGENR --> PAGENAMES &              /* a total function       */
        titles = { 1 |-> p1, 2 |-> p2, 3 |-> p3, ...  15 |-> p15}
VARIABLES
    Plist_size,  /* Actual number of pagess to be displayed */
    DisColours,  /* Colour of slots from 1st position onwards */
    DisPnames,   /* Page names from first position onwards */
    Scroll,      /* No. of pages crolled above the display column */
    Selected,    /* Serial number of selected page in transmission*/
    Status,
    Display_size  /* Size of the display column */
DEFINITIONS
              PageListRange0 == 0..15; PageListRange1 == 1..15;
              DisplaySizeRange0 == 0..10; DisplaySizeRange1 == 1..10
INVARIANT   .
```

```
            .
        ((Status = UNDEFINED)  =>
         (Plist_size = 0 & DisPnames = {} & DisColours = {}))  &
        ((Status = DEFINED)    =>
          (Plist_size > 0 & DisPnames /= {} & DisColours /= {} &
           DisColours(Selected - Scroll) = sel_colour &
          (Scroll > 0  => Selected > 1)              &
          (Scroll = 0 =>  Selected < Display_size)   &
          Selected <= Plist_size             &
          ((Plist_size < Display_size)  => Scroll = 0)&
          (Plist_size  = Display_size => (Scroll = 0 or Scroll =1)) &
          ((Plist_size > Display_size) => Scroll <= Plist_size-Display_size+1)))
INITIALISATION    Display_size := 0 || Plist_size   := 0 ||
                  Status   := UNDEFINED || Selected := 1 ||
                  Scroll := 0 || DisColours,DisPnames := {},{}
OPERATIONS    /* 7  probing operations */
     pp <-- PageAtColumn1 = ...;  /* returns page title at 1st slot */
     cc <-- ColourAtColumn1 = ...; /* returns colour of 1st slot    */
     pos <-- CurrentCursorPosition = ...; /* returns slot position at cursor */
     pp <-- PageAtCurCursor = ....;       /* returns page title at cursor   */
     cc <-- ColourAtCurCursor = ... ;     /* returns slot colour at cursor  */
     pp <-- PageAtLastColumn =  ... ;    /* returns page title at last slot */
     cc <-- ColourAtLastColumn = ...;    /* returns colour of last slot    */
    /* 4 update operations **/
    define(trans_size,disp_size) =
          /*defines display window size and  number of pages in transmission*/
        PRE
             Status = UNDEFINED & trans_size : PageListRange1 &
             trans_size >0 & trans_size <6
             & disp_size > 1 & disp_size <5 &
             disp_size  : DisplaySizeRange1 &
             ((trans_size < disp_size) or (trans_size >= disp_size))
        THEN
            Status := DEFINED ||
            Plist_size := trans_size ||
            Display_size := disp_size ||
            Selected := 1 || Scroll := 0 ||
              IF trans_size >=  disp_size THEN
                 DisPnames :=  {nn,tt | ..} ||
                 DisColours := {nn,cc | ..}
              ELSE         /* trans_size < disp_size */
                 DisPnames :=  {nn,tt | ... } ||
                 DisColours := {nn,cc | ... }
             END
          END;

    undefine = /* puts teletext in an undefined mode of operation */
               PRE Status = DEFINED THEN
               Status   := UNDEFINED || Selected := 1 ||
               Scroll := 0   || Plist_size := 0 ||
               Display_size := 0 || DisColours,DisPnames:= {},{}
               END;
```

Syntax	Predicates to be lifted
IF P THEN S END	$P \vee \ \neg P$
IF P THEN S ELSE T END	$P \vee \ \neg P$
IF P_1 THEN S_1 ELSEIF P_2 THEN S_2 ... ELSEIF P_k THEN S_k ELSE T END	$P_1 \vee$ $(\neg P_1 \ \wedge P_2) \vee$ ... $(\neg P_1 \ \wedge \ldots \neg P_{k-1} \ \wedge P_k) \ \vee$ $(\neg P_1 \ \wedge \ldots \neg P_{k-1} \ \wedge \neg P_k)$
IF P_1 THEN S_1 ELSEIF P_2 THEN S_2 ... ELSEIF P_k THEN S_k END	$P_1 \vee$ $(\neg P_1 \ \wedge P_2) \ \vee$ ... $(\neg P_1 \ \wedge \ldots \neg P_{k-1} \ \wedge P_k) \ \vee$ $(\neg P_1 \ \wedge \ldots \neg P_{k-1} \ \wedge \neg P_k)$
SELECT P THEN S END	$P \vee \ \neg P$
SELECT P_1 THEN S_1 WHEN P_2 THEN S_2 ... WHEN P_k THEN S_k END	$(P_1 \ \wedge \ \neg P_2 \ldots \wedge \neg P_k) \ \vee$ $(\neg P_1 \ \wedge \ P_2 \ldots \wedge \neg P_k) \ \vee$ ... $(\neg P_1 \ \wedge \ \ldots \neg P_{k-1} \wedge P_k) \ \vee$ $(\neg P_1 \ \wedge \ldots \ \neg P_{k-1} \ \wedge \neg P_k)$
SELECT P_1 THEN S_1 WHEN P_2 THEN S_2 ... WHEN P_k THEN S_k ELSE T END	$(P_1 \ \wedge \neg P_2 \ \ldots \wedge \neg P_k) \ \vee$ $(\neg P_1 \ \wedge P_2 \ldots \wedge \neg P_k) \ \vee$ ... $(\neg P_1 \ \wedge \ \ldots \neg P_{k-1} \wedge P_k) \ \vee$ $(\neg P_1 \ \wedge \ \ldots \neg P_{k-1} \wedge \neg P_k)$
CASE E OF EITHER l_1 THEN S_1 ... OR l_k THEN S_k END END	 $(E \in \{l_1\} \wedge E \notin \{l_2, \ldots, l_k\}) \vee$ ... $(E \in \{l_k\} \wedge E \notin \{l_1, \ldots, l_{k-1}\}) \vee$ $(E \notin \{l, \ldots, l_k\})$
CASE E OF EITHER l_1 THEN S_1 ... OR l_k THEN S_k ELSE U END END	 $(E \in \{l_1\} \wedge E \notin \{l_2, \ldots, l_k\}) \vee$ ... $(E \in \{l_k\} \wedge E \notin \{l_1, \ldots, l_{k-1}\}) \vee$ $(E \notin \{l, \ldots, l_k\})$

Table 4. Rules for extracting clauses

```
    upCursor =  /* defines action when up arrow button is pressed */
         PRE  ...  THEN ... END;

    downCursor =  /* defines action when down arrow button is pressed */
       PRE
            Status = DEFINED   & ...
       THEN
           ....
       END
END  /* End of Machine declaration */
```