

Test Criteria Definition for B Models

Salimeh Behnia ¹, Hélène Waeselynck ²

¹ LAAS-CNRS and INRETS

² LAAS-CNRS

7, avenue du Colonel Roche, 31077 Toulouse Cedex 4, France
{behnia, waeselyn}@laas.fr

Abstract. Test criteria are defined in order to guide the selection of subsets of the input domain to be covered during testing. A unification of two categories of test criteria, program based and specification based, is presented. Such a unification is possible for B models because the specification, refinement concepts and implementation are captured in one notation. The notion of control flow graph is extended to handle the abstract constructs of the generalized substitution language, and a link between the coverage of the graph and the coverage of the before-after predicate is established. A set of criteria for the coverage of the control flow graph is proposed. These criteria are partially ordered according to their stringency, so that the coverage strategy may be tuned according to the complexity of the operation under test.

1 Introduction

Testing is a partial verification technique that consists in exercising a target piece of software by supplying it with a sample of input values. Since exhaustive testing is generally not tractable, the tester is faced with the problem of selecting a proper subset of the input domain. The selection is guided by *test criteria* that specify a set of elements to be covered during testing. This paper focuses on the definition of test criteria for B models. It extends previous work establishing a validation framework for the B development process [13]. The aim is to track down specification faults originating from a misunderstanding of the functional requirements, or from the failure to adequately express an understood requirement.

The B formal development process can be seen as a series of stages where more and more concrete models of the application are built, the final code being just a compiled version of the most concrete one. In order to validate these models, we have defined a uniform testing framework, irrespective of the development stage and of whether the test inputs are supplied to the final code or to the formal models [13]. Within the uniform framework, we wish to be equipped with coverage criteria that can be applied not only to the most abstract model (i.e. before refinement), but also to any intermediate model obtained during development. This must be so because, in typical B projects, the smallest meaningful model with respect to the functional requirements is likely to involve a few steps of refinement (see e.g. the modeling approach adopted

by the French railway industry [2]). As a result, the target model to be covered may involve abstract constructs of the B notation, or implementation ones, or a mix of both.

Covering a program and covering an abstract specification correspond to different approaches in the literature on testing. Structural analysis of programs (see e.g. [9]) is usually based on a compact view of their control structure, the control flow graph, that may be supplemented with data flow information. Examples of related criteria are All-Statements, All-Branches or All-Paths, demanding the coverage of the nodes (resp. edges, paths) of the graph: such approaches may be easily transferred to the most concrete B models whose allowed syntactic constructs are similar to program statements. As regards specification based testing, existing approaches vary depending on the used formalism. In case of model-based specifications like Z and VDM, test criteria exploit the structure of the before-after predicates [8, 10]: they demand the coverage of a set of disjunctive cases extracted from the original predicates. As exemplified by [11], such approaches can be transferred to B models, by producing the before-after predicates corresponding to generalized substitutions.

A unification of both categories of criteria, program based and specification based, is presented in this paper. This is done by extending the notion of control flow graph to handle the abstract constructs of the generalized substitution language, and by showing that the coverage of this graph is related to the coverage of the before-after predicate. Such a unification is possible for B models because the specification, refinement concepts and implementation are captured in one notation.

After a very brief overview of the B method (Section 2), we introduce our uniform testing framework for B models, and explain its relation with the problem addressed in this paper (Section 3). In Section 4, we present some specification based and program based structural criteria that have been proposed in the literature. The unification of these structural approaches to cover B models is proposed in Section 5. In Section 6, we define a new set of criteria that are partially ordered according to their stringency. These criteria are illustrated by a simple example.

2 Overview of the B Method

The B method due to J-R. Abrial [1] is a model-based approach for the incremental development of specifications and their refinements down to an implementation. Proof obligations accompany the construction of the software.

The *abstract machine* is the basic element of a B development. It characterizes a machine which has an invisible memory and a number of keys. The values stored in the memory form the *state* of the machine, whereas the various keys are the *operations* that a user is able to activate in order to modify the state. A unique Abstract Machine Notation (AMN) is used for the description of machines at various levels of abstraction, in MACHINE, REFINEMENT and IMPLEMENTATION components.

The *declarative part* of a component describes its encapsulated state according to set-theoretic model and first order logic. The invariant states the static laws that the data must obey, whatever the operation applied.

Various *composition clauses* are defined in the B method, in order to be able to develop large software systems. Refinement is introduced by means of the **REFINES** clause. As soon as the refined version of an abstract machine becomes too complicated, it is decomposed into smaller components, through the **IMPORTS** clause. The **IMPORTS** clause proceeds according to the layered paradigm: the importing machine uses the service offered by the ones imported from lower layers. Those lower layer machines are then independently refined and decomposed into smaller components. Other clauses introduced in [1] are **INCLUDES**, **USES**, **SEES** and **EXTENDS** that enrich the formal text of a component according to specific composition rules.

The *execution part* of a component, which specifies the dynamics, contains the initialization and some operations which are described under the form of a precondition and an action. The corresponding syntactic structures are interpreted in the *generalized substitution language*. The generalized substitutions (see Table 1) are predicate transformers: $[S]R$ denotes the weakest precondition for substitution S to establish postcondition R . For example, for the simple substitution $X := X + 1$ and the postcondition $X = 5$ to be established, we have: $[X := X + 1] X = 5 \Leftrightarrow X + 1 = 5 \Leftrightarrow X = 4$. To facilitate the development of abstract machines, syntactic sugar is introduced. For example, $P \mid S$ is rewritten as **PRE** P **THEN** S **END**.

Table 1. A Subset of Generalized Substitution: x denotes a variable, E is an expression, R and P are predicates, S and T are generalized substitutions

Syntactic Category	Syntax	Semantics
Simple substitution	$x := E$	$[x := E] R \Leftrightarrow$ replacing all free occurrences of x in R by E
no-op	skip	$[\text{skip}] R \Leftrightarrow R$
Preconditioning	$P \mid S$	$[P \mid S] R \Leftrightarrow P \wedge [S] R$
Bounded choice	$S \sqcap T$	$[S \sqcap T] R \Leftrightarrow [S] R \wedge [T] R$
Guarded substitution	$P \Rightarrow S$	$[P \Rightarrow S] R \Leftrightarrow P \Rightarrow [S] R$
Unbounded choice	$@ x . S$	$[@ x . S] R \Leftrightarrow \forall x. [S] R$ where x is not free in R

Preconditioned substitutions are related to the notion of *termination*. Given a substitution S , $\text{trm}(S)$ denotes the predicate that holds if and only if S terminates:

$$\text{trm}(S) \Leftrightarrow [S](x = x)$$

The concept of guarded substitutions is related to the one of feasibility. It would be possible to specify *infeasible* substitutions able to establish any postcondition: this clearly happens when the guard cannot hold. Given a substitution S working with variable x , $\text{mir}(S)$ (miracle) and its negation $\text{fis}(S)$ (feasible) are defined as:

$$\text{mir}(S) \Leftrightarrow [S](x \neq x) \qquad \text{fis}(S) \Leftrightarrow \neg [S](x \neq x)$$

The before-after predicate corresponding to substitution S working with variable x is defined by $\text{prd}_x(S)$, where the after-value is denoted by priming the variable:

$$\text{prd}_x \Leftrightarrow \neg [S](x' \neq x)$$

The following properties are demonstrated in the B-Book:

$$(a) \text{fis}(S) \Leftrightarrow \exists x' . \text{prd}_x(S) \qquad (b) S = \text{trm}(S) \mid @ x' . (\text{prd}_x(S) \Rightarrow x := x')$$

Property (a) indicates that a substitution is feasible (at x) if some after value x' is reachable from the before-value x , and property (b) states that a substitution is completely characterized by `trm` and `prd`.

Checking the mathematical consistency of a MACHINE component involves proving that its initialization establishes the invariant and that each operation, called within its precondition, terminates and preserves the invariant. Checking the correctness of a REFINEMENT or an IMPLEMENTATION involves checking that the initialization and the operations preserve the semantics of their corresponding more abstract versions. The composition clauses allow the proof to be modular, abstract machines being constructed in an incremental fashion from smaller, already proved components.

3 Purpose of a Uniform Testing Framework

Criteria based on formal models are typically used when testing a program against its specification. However, in our case, the search for such criteria arises in the context of on-going research on the validation of the formal models themselves [4, 12, 13].

The B development process can be seen as a series of stages where more and more concrete models of the application are built. Refinement is typically used both to replace abstract constructs of the notation by programmable ones, *and* to gradually introduce functional requirements into the models. It may be the case that some user requirements are captured only at the end of the process. Then validation may be done by testing the final code, which is a compiled version of the most concrete model. But it is better methodology to capture requirements in earlier stages. This also permits earlier validation, provided that there is a means to test B models. Several B animators already exist or are under development: hence the idea of a uniform framework, irrespective of whether the test cases are supplied to the final code or the models.

Our previous work covers two aspects: (i) identification of the development stages where intermediate testable models are obtained and (ii) formalization of the notions of test sequence and test oracle for the corresponding models, the test oracle being the mechanism for determining acceptance or rejection of the output results.

The first aspect identifies the models that may be the target of validation. The B method calls for the incremental development of abstract machines: the formal text corresponding to an abstract machine is split in smaller components linked by composition clauses. In order to be able to reason about the observable behavior specified in this way, we have defined a flattening algorithm that gives the abstract machine resulting from a set of B components [12]. For example, in Figure 1, *Model 1* is the abstract machine obtained by flattening `Main.mch`, `A.mch` and `B.mch`. The set of B components that can be flattened must satisfy architectural conditions that have been identified. The refinement relation can be applied to flattened abstract machines: in Figure 1, *Model 1* is refined by *Model 2*. Using this partial relation we may construct a hierarchy of flattened machines in decreasing order of abstraction: the top of the hierarchy is the model containing no `REFINES` or `IMPORTS` clauses; the bottom of the hierarchy is the final model to be automatically translated into a programming language; intermediate flattened models form the various development stages.

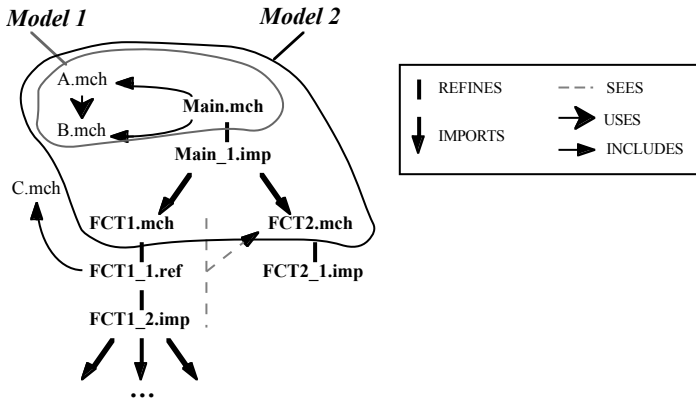


Fig. 1. Identification of development stages

The second aspect specifies what is to be controlled and what is to be observed when testing such models. A test sequence contains a finite sequence of calls to the operations of the machine, beginning with its initialization. It contains no reference to the state variables encapsulated in the machine: due to the hiding principle, the state of an abstract machine must be indirectly controlled and observed through calls to its operations. The oracle checks are defined in relation to the B notion of correctness, namely refinement. This allows us to take advantage of the formality of the B method: given a test sequence, acceptance of the results supplied by a model implies acceptance of the results supplied by proved refinements of this model. We also explored the practical consequences of these definitions in terms of required animation facilities.

The problem of test data selection was not addressed by our previous work. The criteria used for this purpose may refer to elements drawn from the structural analysis of the target B model, or the functional analysis of user requirements, or both. A typical approach could be to select functional test sequences and to supplement them by additional cases to achieve a target coverage of the B model. But the notion of model coverage requires further theoretical investigation: within the uniform framework, we wish to be equipped with criteria that can be applied to abstract models as well as concrete ones. Note that the same problem would arise if our aim was to test the final program against its B specification: the most abstract model is likely to be loose with respect to some required functional features, so that we would also have to consider a refined version of it.

Let us mention that the coverage of models actually involves two problems:

1. identification of test cases for individual operations. Such cases may depend on both the operation's input parameter and the current state of the abstract machine.
2. sequencing of operation calls. As test cases generated in the previous step depend on the internal state, their coverage necessitate to bring the abstract machine to the corresponding before state.

The theoretical contribution of this paper concerns the first problem.

4 Structural Coverage of Specifications and Programs

We briefly discuss the different criteria that have been proposed in the literature for the structural coverage of model-based specifications and procedural programs. Their unification within the framework of the B notation is detailed in Section 5.

4.1 Criteria for the Coverage of Model-based Specifications

In model-based specifications such as Z and VDM, the dynamics of operations are expressed by means of a predicate relating the inputs (input parameters and before state) to the outputs (output parameters and after state). Such a before-after predicate may be used to derive a set of test cases for each operation. Existing approaches first flatten the specification of the operation, recursive definitions being unfolded a bounded number of times. Then they exploit the propositional structure of the resulting predicate to obtain a partition of the input domain: the corresponding test strategy will be to force (at least) one input to come from each subdomain. Since the analyzed predicate is based on the first order logic and set theory, these approaches cannot be fully automated.

The approach proposed by [8] is to reduce the predicate into a Disjunctive Normal Form (DNF) whose disjuncts yield the subdomains of the partition. As the transformation proceeds, the following rules are applied:

- $A \vee B$ is split into three disjoint cases $A \wedge B$, $\neg A \wedge B$ and $A \wedge \neg B$.
- $A \Rightarrow B$ is split into two disjoint cases $\neg A$ and $A \wedge B$.

An operation op is thus decomposed into disjoint cases op_i characterizing relations between input and output subdomains. Constraints involving only input parameters, or state variables, etc. may be extracted by existentially quantifying the other variables. For example, let us assume that operation op is working with formal parameters v_in , v_out and state variable s . We also adopt the convention that the after value of s is denoted by priming the identifier. Then, $\exists v_out, s' \cdot op_i$ and $\exists v_in, s \cdot op_i$ yield the input and output subdomains for each op_i .

Another approach [10] rewrites the operation predicate as $\bigvee_i (X_i \wedge Y_i)$ where:

- predicates X_i refer only to the inputs;
- predicates Y_i refer to the outputs.

The rewritten operation predicate is not necessarily in DNF, because X_i and Y_i may contain disjunctions. The idea is not to separate constraints that apply to the same category of variables. For example $(v_in = 0 \vee v_in = 1) \wedge (v_out = v_in + 1)$ has the desired form $X_i \wedge Y_i$. As argued by the author, predicate X_i needs not to be further split: it characterizes an input subdomain with uniform behavior. However, at that stage, a partition of the input domain is not yet obtained because the predicates may overlap. Disjoint subdomains are produced by forming their disjunction and applying the same rule as [8] to split it. For example, if the rewritten predicate was:

$$(X_1 \wedge Y_1) \vee (X_2 \wedge Y_2)$$

Then the resulting partition of the input domain would be:

$$X_1 \wedge X_2 \qquad \neg X_1 \wedge X_2 \qquad X_1 \wedge \neg X_2.$$

This partition is coarser than the one of [8] since the individual X_i have not been expanded. However, even [10] faces a scalability problem. As pointed out by the author, a specification in the form $\bigwedge_i (X_i \vee Y_i)$ with $1 \leq i \leq n$ would be rewritten as the disjunction of 2^n predicates.

Both approaches can be further refined by considering boundary cases and expansion of set-theoretic operators. For example, $x \leq a$ may be split into two cases $x = a$, $x < a$; $e \in S_1 \cup S_2$ may be split into three cases $e \in S_1 \cap S_2$, $e \in S_1 - S_2$, $e \in S_2 - S_1$.

The partition analysis of each operation is used by the authors to produce a finite state automaton (FSA) from the model-based specification. The aim is to address the problem of the sequencing of test cases. The corresponding techniques are not presented in this paper.

4.2 Structural Coverage of Programs

Structural criteria have been defined for procedural programs (see e.g. [9]). Structural analysis of procedures is usually based on the analysis of their control flow graph that may be supplemented by data flow information. The source code is divided into basic blocks, i.e. single entry, single exit sequences of statements. Then, the control flow graph is constructed by making each basic block a node and drawing an arc for each possible transfer of control from one basic block to another. The control flow graph may be annotated by data flow information to mark the locations where a variable is assigned a value and the ones where it is used.

The execution of the procedure in response to one input can be seen as the exercising of one path in the graph; hence structural criteria for procedures are path selection criteria. For example, All-Statements (resp. All-Branches, All-Paths) requires the selection of a set of paths such as each node of the graph (resp. edge, path) is traversed at least once. There may be a large – if not infinite – number of paths due to the presence of loops: some specific criteria demand the coverage of a bounded number of iterations. Other data-flow criteria involve the coverage of subpaths between the definitions and uses of the variables. Finally, control and data flow criteria may be refined by considering boundary cases, or by expanding branch Boolean expressions (see e.g. [7]). All these criteria can be partially ordered according to their stringency.

Note that there may be infeasible paths in the graph, i.e. paths for which no test data can be found: such paths are excluded from the above criteria so that All-Paths actually demands the coverage of feasible paths, etc. Since the identification of infeasible paths is undecidable in the general case, none of the above approaches can be fully automated.

Note also that most of the above criteria do not define a partition of the input domain: All-Paths does, because the input subdomains associated to two distinct paths are disjoint; but the input subdomains induced by weaker criteria, like All-Branches, may (and generally will) overlap.

5 Unification of Structural Approaches for B Models

As explained in Section 3, the uniform testing framework has to cope with flattened B models obtained at any development stage. Structural approaches for the coverage of before-after predicates can theoretically be applied whatever the level of abstraction (Section 5.1) but, as already mentioned, these approaches face a scalability problem. Structural approaches defined for procedural programs can be applied to the most concrete models that contain only programmable constructs (Section 5.2). We propose an extension of the notion of control flow graph that can be applied to abstract constructs as well (Section 5.2), so that it can serve as a basis for structural analysis at any development stage. We also show that coverage criteria for this graph can be related to coverage criteria defined from the corresponding before-after predicate.

5.1 Partition Analysis of B Operations from Before-After Predicates

The work introduced in [8] is within the framework of VDM specifications, while [10] is working with Z specifications. However, both approaches may be transferred to other model-based specifications, including B specifications. B operations are specified by generalized substitutions, but it is possible to use the translation into before-after predicate (see Section 2), and perform the partition analysis as described in Section 4.1. Nevertheless, as will be explained below, the partition analysis has to be adapted to the target B notation, in order to account for the notion of well-defined B.

Like [8] in the VDM approach, we assume that the operation is called within its precondition, and that the before state satisfies the invariant. Moreover, for the before-after predicate to have a meaning, we have to take into account contextual information about the sets and constants given for the abstract machine. The predicate corresponding to contextual information is precisely defined in the B-Book (see e.g. the expression of the proof obligation that an operation preserves the invariant).

Now, let us assume that the operation to be analyzed is in the form:

PRE P THEN S END

From Section 2, property (a) of generalized substitutions, we may notice that the existential quantification of the before-after predicate of S , used by [8] to hide the after state and output values, gives us $\text{fis}(S)$. Then, the input domain of the operation to be partitioned into subdomains may be expressed as:

$$\text{context} \wedge \text{invariant} \wedge P \wedge \text{fis}(S) \tag{1}$$

Applying the transformation into disjoint cases, one must be careful not to generate ill-defined cases. To illustrate the problem, let us assume that the partition analysis generates the following intermediate formula, where f is a partial function:

$$\neg(x \in \text{dom}(f) \wedge f(x) = y)$$

Applying the approaches described in Section 4.1, the formula would be further split into three disjoint cases:

$$x \notin \text{dom}(f) \wedge f(x) \neq y \quad x \notin \text{dom}(f) \wedge f(x) = y \quad x \in \text{dom}(f) \wedge f(x) \neq y$$

The first two cases are ill-defined: $f(x)$ is meaningless since x does not belong to the domain of function f .

The problem of well-defined B has been investigated in [3]: the concern was to tackle ill-defined proof obligations. The adopted solution was to 1) generate additional proof obligations of the well-definedness of models; 2) propose a deduction system such that the proof of a well-defined lemma does not introduce ill-defined formulæ. Our work is based on their definition of well-definedness. Assuming that the original predicate is well-defined, the concern is not to introduce ill-definedness during the separation into cases. The well-definedness of predicates is given in [3] by means of Δ_p operator and the well-definedness of substitutions by means of Δ_s operator. In Figure 2, we give a subset of rules defining Δ_p and Δ_s operators to which we will refer afterwards. We also give a theorem defined in [3] which states that predicate transformation of a well-defined substitution is also well-defined.

$\Delta_p \neg P \equiv \Delta_p P$	$\Delta_s (S \parallel T) \equiv \Delta_s S \wedge \Delta_s T$
$\Delta_p (P \wedge Q) \equiv \Delta_p P \wedge (P \Rightarrow \Delta_p Q)$	$\Delta_s (P \Rightarrow S) \equiv \Delta_p P \wedge (P \Rightarrow \Delta_s S)$
$\Delta_p (P \vee Q) \equiv \Delta_p P \wedge (\neg P \Rightarrow \Delta_p Q)$	$\Delta_s (S \sqcap T) \equiv \Delta_s S \wedge \Delta_s T$
$\Delta_p (P \Rightarrow Q) \equiv \Delta_p P \wedge (P \Rightarrow \Delta_p Q)$	$\Delta_s (@x \cdot S) \equiv \forall x \cdot \Delta_s S$
Theorem: $\Delta_p I \wedge \Delta_s S \Rightarrow \Delta_p ([S]I)$	

Fig. 2. Inductive rules from the definition of Δ_p and Δ_s operators in [3]

Note that the rules for Δ_p correspond to an interpretation of the logical *and* (resp. *or*) operator as the *and then* (resp. *or else*) operator provided by some programming languages. Accordingly, a safe separation of the previous formula would give us only two well-defined disjoint cases:



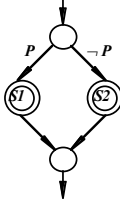
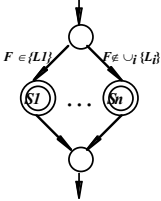

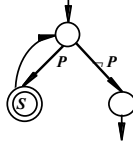
$$x \notin \text{dom}(f) \quad x \in \text{dom}(f) \wedge f(x) \neq y$$

Restricting the formulæ manipulations to safe ones, the partition analysis could be theoretically applied to any intermediate model obtained during a B development: the translation from substitution to before-after predicate can be performed whatever the level of abstraction. Similarly, the application of the approach to refinements was also suggested by [8], in the framework of wide spectrum notations such as VDM-SL. But the problem of scalability is all the more acute as the model is detailed.

5.2 Control Flow Graph of B0 Substitutions

The subset of the B language that can be used in an IMPLEMENTATION component is called B0. In Table 2, we show the substitutions of B0 together with their control flow subgraphs. As these substitutions are similar to the statements of the classical procedural languages, their adopted control flow representations are similar to the ones proposed in the literature for programming language statements.

Table 2. Control flow representation of B0 substitutions: X and W are (lists of) variables; E is an (a list of) expression(s); F is an expression; S , $S1$, $S2$, S_n are substitutions; P , I and V are predicates; $L1$ is a (list of) constant(s); and \odot is the subgraph corresponding to a substitution

Substitution	CF graph	Substitution	CF graph
Assignment substitution: $X := E$		Operation call: $X \leftarrow \text{op}(E)$	
IF substitution: IF P THEN $S1$ ELSE $S2$ END		CASE substitution: CASE F OF EITHER $L1$ THEN $S1$ OR ... ELSE S_n END	
Local variable: VAR W IN S END		WHILE substitution: WHILE P DO S INVARIANT I VARIANT V END	

These substitutions can be sequenced using the “,” operator. This operator is equivalent to the well-known “;” operator in programming languages and means that the related substitutions follow each other. This operator, in terms of construction of a control flow graph, means that the subgraphs of corresponding substitutions must be put one after the other. This is done systematically using the subgraphs of Table 2: the exiting edge of a substitution is merged with the incoming edge of its subsequent substitution. In order to complete the control flow graph, a start node before the first substitution and an end node after the last substitution of an operation are also added. The control flow graph obtained in this way can possibly be simplified into a more compact form. For example, in control flow graphs it is typical to represent by nodes maximum length blocks of uninterrupted sequences of assignments (see e.g. [5] on the control flow graph representation of programs).

The substitutions of B0 are essentially syntactic constructions that are proposed in order to ease the use of the language. They can be rewritten using the basic generalized substitutions. For example, the rewriting of the IF substitution is the following:

$$(P \Rightarrow S1) \sqcap (\neg P \Rightarrow S2)$$

As can be seen in subgraphs of Table 2, the existence of several subpaths leaving a node is because of existence of a decision in the substitution (IF, CASE, WHILE substitutions or other forms of these substitutions). In terms of basic generalized substitutions, this corresponds to the bounded choice between guarded substitutions. If

we rewrite the original substitution S as $\sqcup_i S_i$, each substitution S_i gives us a path on the control flow graph. Note that the control flow graph of the substitution $\sqcup_i S_i$ is not the same as the control flow graph of the original substitution S , but the paths on both graphs are the same. As usual when testing loops, the WHILE substitution will be unfolded to be traversed a bounded number of times.

The rewriting is possible because of the following equivalence rules that have been proved in the B-Book [1]:

$$\begin{array}{ll} (S1 \sqcup S2) ; S3 = (S1 ; S3) \sqcup (S2 ; S3) & S1 ; (S2 \sqcup S3) = (S1 ; S2) \sqcup (S1 ; S3) \\ P \Rightarrow (S1 \sqcup S2) = (P \Rightarrow S1) \sqcup (P \Rightarrow S2) & @x. (S1 \sqcup S2) = (@x. S1) \sqcup (@x. S2) \end{array}$$

According to the Δ_s operator defined in [3] (see Figure 2) for the well-definedness of a substitution, the rewriting does not introduce ill-defined substitutions. We have demonstrated this for all rewriting rules. In Figure 3, we show that the rewriting rule for the guarded substitution is well-defined.

$$\begin{aligned} \Delta_s (P \Rightarrow (S1 \sqcup S2)) &\Leftarrow_{\mathcal{A}_p} P \wedge (P \Rightarrow \Delta_s (S1 \sqcup S2)) \\ &\Leftarrow_{\mathcal{A}_p} P \wedge (P \Rightarrow (\Delta_s S1 \wedge \Delta_s S2)) \\ &\Leftarrow_{\mathcal{A}_p} P \wedge (P \Rightarrow \Delta_s S1 \wedge P \Rightarrow \Delta_s S2) \\ &\Leftarrow_{\mathcal{A}_s} (P \Rightarrow S1) \wedge \Delta_s (P \Rightarrow S2) \\ &\Leftarrow_{\mathcal{A}_s} ((P \Rightarrow S1) \sqcup (P \Rightarrow S2)) \end{aligned}$$

Fig. 3. Well-definedness of the rewriting rule for guarded substitution

Just as in the control flow graph of programs, all of the paths on the graph may not be feasible. The predicate fis of the rewritten substitution gives the feasible paths of the control flow graph: $\text{fis} (\sqcup_i S_i) \Leftarrow \bigvee_i \text{fis} (S_i)$. This shows that path selection techniques can be expressed in terms of predicate coverage.

5.3 Structural Coverage of Generalized Substitutions

In the previous section, only B0 substitutions have been discussed. We will now propose an extension to the notion of control flow graph in order to be able to also consider abstract substitutions of the B language.

The bounded choice substitution can be used at different levels of development in MACHINE, REFINEMENT and IMPLEMENTATION components. In an IMPLEMENTATION component the specification must be deterministic. Therefore, the bounded choice substitution is used only in the context of a decision, i.e. the component substitutions are guarded with mutually exclusive predicates whose disjunction is true (e.g. P and $\neg P$). In a MACHINE or REFINEMENT component the bounded choice substitution may be used in a non-deterministic manner: the substitution $S = S1 \sqcup S2$ means that the substitution S can be implemented by further implementing either $S1$ or $S2$. In order to represent this substitution in a control flow graph, we propose the subgraph of Table 3. As can be noticed, this subgraph is similar to the subgraph of B0's IF substitution but there are no predicates associated to the incoming arcs of subgraphs of

$S1$ and $S2$; $S1$ and $S2$ may or may not be guarded substitutions, or the guards may not be exclusive. Therefore, the coverage of this subgraph must be given an operational interpretation. We adopt the operational interpretation that is based on the notion of correct executable interpretation of model-based specifications investigated for Z notation in [6], also discussed in the context of testing B models via animation [13]. It is argued that a correct interpretation of a non-deterministic specification should offer many alternative outputs. In the same manner, if for some input data both component substitutions of the bounded choice are feasible, both branches of the subgraph will be covered.

Table 3. Control flow representation of bounded choice, parallel and guarded substitutions: S , $S1$, $S2$ are substitutions; P is a predicate; \odot is the subgraph corresponding to a substitution

Substitution	CF graph	Substitution	CF graph
Bounded Choice Substitution: CHOICE $S1$ OR $S2 \equiv S1 \sqcup S2$ END		Parallel Substitution: $S1 \parallel S2$	
Guarded Substitution: SELECT P THEN $S \equiv P \Rightarrow S$ END			

As the substitutions of an IMPLEMENTATION component are deterministic, an execution of these substitutions will correspond to only one complete path on the corresponding control flow graph. This will no more be the case for more abstract substitutions of MACHINE or REFINEMENT components that may contain non-deterministic substitutions. An “execution” (by animation) of such substitutions may not correspond to only one path on the control flow graph but to a set of paths. We call a *path-combination* the set of paths that are covered at the same time by test data.

Besides non-determinism, another issue that must be considered is the notion of infeasible specifications. There is no proof of feasibility in the B method. While it would be possible for a MACHINE or REFINEMENT component to contain infeasible substitutions, if the development ends up with a set of proved IMPLEMENTATIONS, then it is sure that the specification is feasible. If in a machine or refinement component a miracle has been specified, there may be no feasible paths for some input data. This will be the case when all guards evaluate to false. Because of infeasible substitutions, a path-combination may also be the empty set.

In Table 3, we also propose a control flow subgraph for guarded and parallel substitutions. The guarded substitution $P \Rightarrow S$ means that the substitution S is only

feasible when the guard is true. In the same manner, the subgraph of the substitution S is covered only if the guard is true. The parallel substitution $S = S1 \parallel S2$ indicates the simultaneous execution of substitutions $S1$ and $S2$. The parallelism in this case means that both substitutions, working with distinct variables, can be performed independently. A parallel substitution is feasible if and only if both component substitutions are feasible. So if for some input cases one of the substitutions is not feasible, the subgraph of the parallel substitution is not covered.

Like the other substitutions, an original substitution S containing parallel substitutions can be rewritten in the form $\square_i S_i$ so that each S_i will be a path on the control flow graph. The following equivalence rules have been proved in the B-Book:

$$(S1 \square S2) \parallel S3 = (S1 \parallel S3) \square (S2 \parallel S3) \quad S1 \parallel (S2 \square S3) = (S1 \parallel S2) \square (S1 \parallel S3)$$

Note that for parallel substitutions a subpath on a control flow graph is not the same as the classical subpaths on a program control flow graph. For example considering the control flow graph of Figure 4, we only have two paths and not three. These are the paths that we get after rewriting the original substitution.

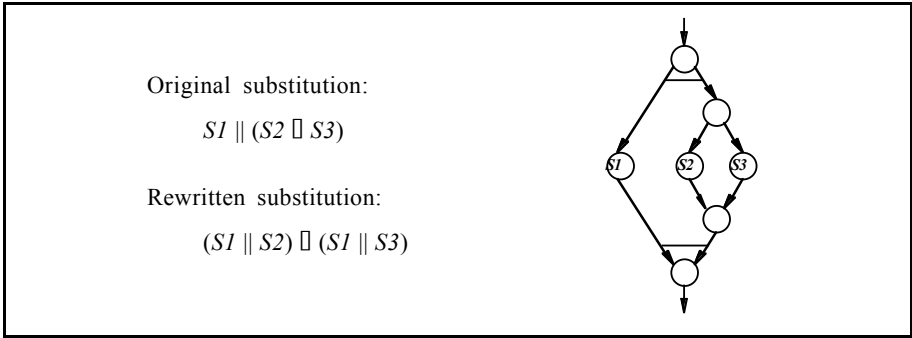


Fig. 4. An example of a control flow graph for a parallel substitution. The paths on the graph are $S1 \parallel S2$ and $S1 \parallel S3$

The last basic generalized substitution that must be considered is preconditioned substitution. Actually, the construction of control flow graph assumes that the precondition, the invariant and the context holds. This assumption, which is the same as Equation 1, in Section 5.1, when analyzing before-after predicates, is necessary in order to be able to reason about the feasibility of substitutions. For example, consider the feasibility of preconditioned substitution $P \mid S$: $\text{fis}(P \mid S) \Leftrightarrow P \Rightarrow \text{fis}(S)$. According to [3], S and $\text{fis}(S)$ may be ill-defined if the precondition does not hold.

To conclude, consider an operation $\text{PRE } P \text{ THEN } S \text{ END}$, in which S may contain abstract and concrete substitutions. We have seen in Section 5.1 that in order to have a partition of the input domain, predicate transformations, like the ones proposed by [8], may be performed on $\text{fis}(S)$. We have also seen that a control flow graph can be constructed from S , and that after rewriting the substitution S so that the choices are pushed out, the feasible paths on the graph are $\text{fis}(\square_i S_i) \Leftrightarrow \bigvee_i \text{fis}(S_i)$. This means that path coverage criteria defined for this control flow graph are equivalent to partitioning $\text{fis}(S)$ into (possibly overlapping) cases.

6 Coverage Criteria for Generalized Substitutions

The extended control flow graph permits us to define a hierarchy of criteria that are applicable to models in different levels of abstraction. The hierarchy of criteria allows to tune the test stringency according to the abstraction level of the analyzed B model.

6.1 Definition of Criteria

The first three criteria that we will define in this section, concern the coverage of the control flow graph. Each criterion can be further refined by considering the coverage of predicates of guards.

All-paths criterion. Like the classical all-paths criterion, this criterion demands that all feasible paths of the control flow graph be covered. We have seen that in a substitution that is rewritten in the form $\Box_i S_i$, each S_i is a path on the control flow graph. So this criterion requires that test data be generated that satisfy each predicate $\text{fis}(S_i)$, which is not equivalent to false.

All-path-combinations criterion. This criterion demands that all combinations of feasible paths on the control flow graph be covered by test data. The combinations of the feasible paths are obtained by the separation of the predicate $\bigvee_i \text{fis}(S_i)$ into disjoint cases. Consider the example of a substitution S which is rewritten to $S1 \Box S2$. The predicate fis of this substitution is $\text{fis}(S1) \vee \text{fis}(S2)$. Assuming that $S1$ and $S2$ are well-defined, the predicates $\text{fis}(S1)$ and $\text{fis}(S2)$ are also well-defined (see theorem in Figure 2, defined in [3]). So the predicate $\text{fis}(S1) \vee \text{fis}(S2)$ can be separated into the following three disjoint cases, without generating ill-defined terms:

$$\text{fis}(S1) \wedge \text{fis}(S2) \quad \text{or} \quad \neg \text{fis}(S1) \wedge \text{fis}(S2) \quad \text{or} \quad \text{fis}(S1) \wedge \neg \text{fis}(S2)$$

The separation of the predicate $\bigvee_i \text{fis}(S_i)$ will generate at most $2^i - 1$ cases. This is an upper bound because some of the cases may be simplified to false. For example, there may be some paths on the control flow graph that are exclusive, that is if one is feasible the other one will always be infeasible. For the substitutions of IMPLEMENTATION components, i.e. deterministic and feasible operations, this criterion is equivalent to all-paths criterion.

All-expanded-path-combinations criterion. In the previous criterion, it has been required to consider combinations in which some of paths on the control flow graph are infeasible. There may also be several cases in which a path is infeasible: the substitution $P \Rightarrow S$ may be infeasible either if the guard P is false or if the substitution S is infeasible; the substitution $S \parallel T$ may be infeasible if either substitutions S or T , or both of them are infeasible. The all-expanded-path-combinations demands that each of these cases be considered separately. In order to do

this, we will separate the negation of the predicate *fis* of guarded and parallel substitutions into disjoint cases.

According to the definition of Δ_s operator for guarded substitutions defined in [3] (see Figure 2), the component substitution of a guarded substitution may be ill-defined if the predicate of guard is false. To avoid generating ill-defined terms, we will use a safe separation into disjoint cases for the negation of the predicate *fis* of guarded substitutions:

$$\neg \text{fis} (P \Rightarrow S) \Leftrightarrow \neg (P \wedge \text{fis} (S)) \quad \text{is separated into} \quad \neg P \quad \text{or} \quad P \wedge \neg \text{fis} (S)$$

For the parallel substitutions, the negation of the predicate *fis* can be separated into three disjoint cases. This is because the Δ_s of the parallel substitution (see Figure 2) states that both component substitutions are well-defined if the parallel substitution is well-defined.

$$\neg \text{fis} (S \parallel T) \Leftrightarrow \neg (\text{fis} (S) \wedge \text{fis} (T)) \quad \text{is separated into} \quad \begin{array}{ll} \neg \text{fis} (S) \wedge \neg \text{fis} (T) & \text{or} \\ \neg \text{fis} (S) \wedge \text{fis} (T) & \text{or} \\ \text{fis} (S) \wedge \neg \text{fis} (T) & \end{array}$$

Coverage of predicates of guards. In the previous criteria, we have considered the structure of the operators of the Generalized Substitution Language and we have proposed different ways in which they can be covered. We may also consider the structure of predicates of guards. If a predicate of guard can be broken into simpler predicates, we may define different ways in covering these predicates. In order to cover a predicate of guard, we propose the following separation into disjoint cases for disjunction and implication operators and for negation of a conjunction operator:

$$\begin{array}{lll} A \vee B & \text{is separated into} & A \quad \text{or} \quad \neg A \wedge B \\ A \Rightarrow B & \text{is separated into} & \neg A \quad \text{or} \quad A \wedge B \\ \neg(A \wedge B) & \text{is separated into} & \neg A \quad \text{or} \quad A \wedge \neg B \end{array}$$

All of the above criteria can be further refined by requiring that each predicate of guard be separated at least once during testing. For example, all-paths plus predicate coverage criterion requires that all feasible paths on the control flow graph be covered and that every predicate of guard be separated into disjoint cases at least once. The above separations are safe separations that will not generate ill-defined terms. They are based on the definition of the well-definedness operator Δ_p for predicates [3] (see Figure 2).

There are other predicate coverage criteria that have been proposed in the literature for the structural coverage of decisions in a program (e.g. [7] in avionics domain). Such criteria could be used as well.

6.2 Hierarchy of the Criteria

The hierarchy of the criteria that we have defined in the previous section is shown in Figure 5. The relation between criteria is the *inclusion* relation [9]. Criterion A includes criterion B, if and only if each test set that covers A also covers B. The

inclusion relation between our criteria is easily found according to their definition. The all-path-combination criterion demands that all combinations of feasible paths be covered and in this way includes all-paths criterion which demands only that all feasible paths be covered. The all-expanded-path-combination criterion demands that all combinations of feasible paths be covered, distinguishing between the different cases in which a path may not be feasible, and so it includes all-path-combination criterion. Each of these criteria is included by the same criteria plus the coverage of predicates of guards, in the way that it demands that each predicate be separated at least once during testing.

These criteria are defined in order to cover specifications in different levels of abstraction, which may contain non-deterministic and infeasible constructs. After the all-paths criterion we may retrieve the classical structural coverage criteria that have been defined for programs.

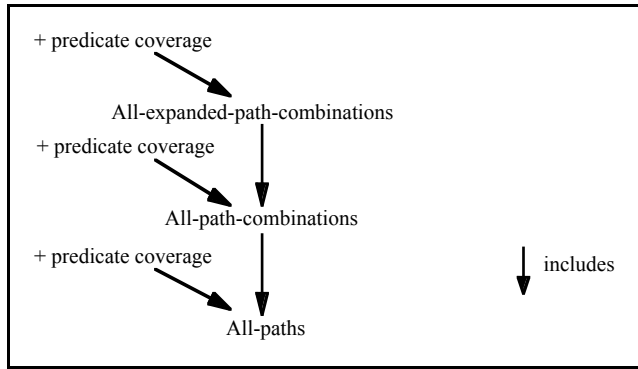


Fig. 5. The hierarchy of the coverage criteria.

It is also interesting to compare these criteria with the approach proposed in [8] for the generation of test cases from VDM specifications. In this approach, test cases are generated for each operation by partition analysis of the corresponding before-after predicate: logical disjunction and implication operators are decomposed into disjoint cases (see Section 4.1). This approach is more stringent than our criteria. First, let us mention that the cases that we have identified from the structure of the substitution operators correspond to disjunctions in the before-after predicate. For example, $\text{prd}_x(S1 \sqcap S2) \Leftrightarrow \text{prd}_x(S1) \vee \text{prd}_x(S2)$. The same applies to the cases identified from the guard predicates. Then, the criterion of [8] is at least as stringent as all-expanded-path-combinations plus the coverage of all of the predicates of guards. But in addition, in [8], test cases may be generated that partition the output space. Consider the following simple example:

IF $x > 0$ THEN $y := x$ ELSE $y := 1$ END

The approach of [8] will generate three test cases:

- (1) $x = 1$ (2) $x > 1$ (3) $x \leq 0$

For the first branch of IF, i.e. when $x > 0$, two cases are generated: in case (1) the output result is the same as the one that would be obtained through the other branch of IF, i.e. $y = 1$; in case (2) the output result is different. The distinction of these cases arises from a partition of output space. It is also worth noting that if the approach of [8] is applied to B models, generated test cases may contain ill-defined terms.

Let us recall that the approach proposed in [8] causes serious problems of scalability. Even for a small and simple specification a large number of test cases may be generated. As we work directly with the substitutions and not with the before-after predicates, we are able to distinguish between cases that are generated from the structure of substitution operators and cases that correspond to the structure of guard predicates. This has permitted us to define criteria with different degrees of stringency, so that the coverage strategy may be tuned according to the complexity of the operation.

6.3 Example

In this section, we will illustrate by means of an example, the criteria that we have defined in the previous section. The application of these criteria will permit to identify (possibly overlapping) subsets of the input domain in order to generate test data.

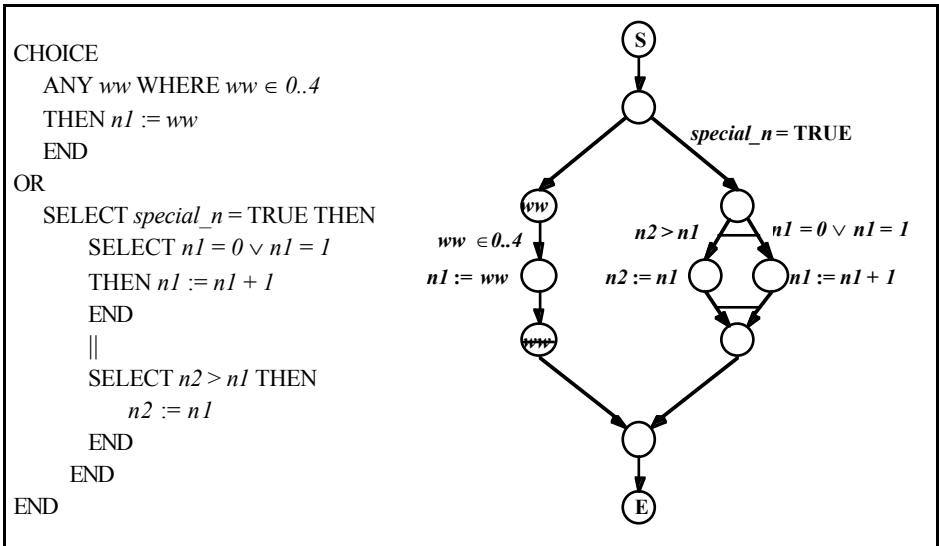


Fig. 6. The control flow graph of a substitution. The substitution is considered in the following context : $n1 \in \text{NAT} \wedge n2 \in \text{NAT} \wedge \text{special_n} \in \text{BOOL}$

Consider the example of Figure 6. The specification can be written using the basic substitutions:

$$\begin{aligned}
 S = & (@ww \cdot ww \in 0..4 \Rightarrow n1 := ww) \sqcap \\
 & (special_n = \text{TRUE} \Rightarrow (((n1 = 0 \vee n1 = 1) \Rightarrow n1 := n1 + 1) \parallel \\
 & (n2 > n1 \Rightarrow n2 := n1)))
 \end{aligned}$$

This substitution needs not be rewritten. Let $S1$ and $S2$ designate each component of the bounded choice substitution. The predicate $\text{fis}(S)$ is:

$$\begin{aligned} \text{fis}(S) &\Leftrightarrow \text{fis} (@ww \cdot ww \in 0..4 \Rightarrow n1 := ww) \vee \\ &\quad \text{fis} (special_n = \text{TRUE} \Rightarrow (((n1 = 0 \vee n1 = 1) \Rightarrow n1 := n1 + 1) \parallel \\ &\quad \quad (n2 > n1 \Rightarrow n2 := n1))) \\ &\Leftrightarrow \text{fis}(S1) \vee \text{fis}(S2) \end{aligned}$$

The all-paths criterion requires the coverage of each feasible path on the control flow graph, so we have to produce test data that satisfy the following two predicates:

$$\begin{aligned} \text{fis}(S1) &\Leftrightarrow \exists ww \cdot ww \in 0..4 \\ \text{fis}(S2) &\Leftrightarrow special_n = \text{TRUE} \wedge (n1 = 0 \vee n1 = 1) \wedge n2 > n1 \end{aligned}$$

The first path is always feasible ($\text{fis}(S1)$ always holds). In order to cover the second path a single test data, for example $special_n = \text{TRUE} \wedge n1 = 0 \wedge n2 = 1$, is sufficient. If we want also to cover the predicates of guards, the predicate $(n1 = 0 \vee n1 = 1)$ will be separated. In order to cover this predicate, the previous test can be completed by test data $special_n = \text{TRUE} \wedge n1 = 1 \wedge n2 = 2$.

The all-path-combination criteria is more demanding than all-paths criterion. It requires that all combinations of feasible paths be covered by test data. The predicate $\text{fis}(S)$ will be separated into disjoint cases:

$$\begin{aligned} \text{fis}(S1) \wedge \text{fis}(S2) &\Leftrightarrow special_n = \text{TRUE} \wedge (n1 = 0 \vee n1 = 1) \wedge n2 > n1 \\ \neg \text{fis}(S1) \wedge \text{fis}(S2) &\Leftrightarrow \text{FALSE} \\ \text{fis}(S1) \wedge \neg \text{fis}(S2) &\Leftrightarrow \neg (special_n = \text{TRUE} \wedge (n1 = 0 \vee n1 = 1) \wedge n2 > n1) \end{aligned}$$

Two test data will be sufficient in order to satisfy the above predicates. For example, we may complete test data $special_n = \text{TRUE} \wedge n1 = 0 \wedge n2 = 1$ by test data $special_n = \text{FALSE}$.

Finally, the all-expanded-path-combination criterion requires to distinguish between different cases in which a path may be infeasible. Consider the last predicate generated in the all-path-combinations criterion, in which the substitution $S2$ is not feasible. Each component of $S2$, that can make the substitution infeasible, will be considered separately. So we have to complete the previous test data in order to satisfy the following predicates:

$$\begin{aligned} special_n &= \text{FALSE} \\ special_n = \text{TRUE} &\wedge \neg(n1 = 0 \vee n1 = 1) \wedge \neg(n2 > n1) \\ special_n = \text{TRUE} &\wedge (n1 = 0 \vee n1 = 1) \wedge \neg(n2 > n1) \\ special_n = \text{TRUE} &\wedge \neg(n1 = 0 \vee n1 = 1) \wedge (n2 > n1) \end{aligned}$$

These cases are the result of the safe separation of the guarded substitution and the separation of the parallel substitution into disjoint cases.

7 Conclusion

A unified approach to the structural coverage of B models has been proposed. This approach was initially considered in a testing framework in which intermediate B models obtained in different stages of development are object to validation. As the smallest meaningful model with respect to user requirements may involve several steps of refinement and decomposition into logical layers, the B models that will be validated may contain abstract and concrete constructs at the same time. But the proposed approach reaches a more global aim: it can also be used wherever a structural analysis of B models is needed.

We have seen how a control flow graph can be constructed from the substitutions of a B operation, and how the coverage of (sets of) paths in this graph can be linked with the coverage of before-after predicates proposed for model-based specifications in the literature. The fact that we work directly with substitutions, and not with their translation into before-after predicate, has allowed us to distinguish between cases that may be extracted from the structure of substitution operators (like the bounded choice) and cases that may be extracted from the structure of guard predicates. Moreover, the approach takes into account the notion of well-definedness for both predicates and substitutions, so that ill-defined cases are never generated. We end up with a set of criteria whose stringency can be partially ordered according to the inclusion relation. The aim is to be able to tune the test strategy to the complexity of the model to be analyzed. While the fine partition analysis of [8] may be applicable to simple abstract models, it is expected to become untractable as complexity is gradually entered into the models; for the most concrete models obtained at late development stages, only weak path selection criteria (e.g. statement coverage) may be adopted. Our criteria fill the gap between these specification based and program based strategies.

The presented criteria apply to the coverage of individual B operation. As mentioned earlier, separate analysis of each operation is not sufficient: their joint dependency on the encapsulated state must also be considered to design test sequences. The approaches proposed in [8, 10] yield a finite state automaton representing all valid sequences of operation calls: both the states and transitions of the automaton are derived from the partition analysis of the operations. We will investigate whether such approaches are applicable to realistic B models, at least at the earliest stages of the development process.

As regards tool support, it would be worth studying whether our criteria may be implemented using existing environment like B-CASTING [11]. CASTING is a generic approach meant for the generation of test data from formal specification: a first prototype instantiating this approach in the case of the B notation has been developed. The extraction of test cases works with an attributed grammar of the notation, while the generation of data is performed using a constraint solver. We are currently investigating the possibility of expressing the parsing of B substitutions induced by our criteria in terms of rules similar to the ones given in [11].

However, it is not expected that the generation of test data from the models can be fully automated. An alternative approach could be to generate test sequences using some other means, and to obtain an a posteriori assessment of the supplied coverage.

The assessment could be performed by instrumenting the target B models, so that we obtain a trace of the parts of the operation substitutions that are traversed during animation. We have started preliminary work to address this problem. Our first conclusion is that it should be possible to mark the nodes that are traversed by insertion of simple substitutions, but the coverage analysis from the resulting trace has still to be formalized.

Acknowledgments. We sincerely thank our colleague, Pascale Thévenod-Fosse, for her detailed reading and helpful suggestions in the preparation of this paper.

References

1. Abrial, J.R.: The B-Book – Assigning Programs to Meanings, Cambridge University Press (1996)
2. Behm, P., Desforges, P., Mejia, F.: Application de la méthode B dans l'industrie ferroviaire, In: Application des techniques formelles au logiciel, ARAGO 20, OFTA, ISBN 2-906028-06-1 (1997) 59-87
3. Behm, P., Burdy, L., Meynadier, J.M.: Well Defined B, In: Proceedings of 2nd International B Conference, LNCS No. 1393, April (1998) 29-45
4. Behnia, S., Waeselynck, H.: External Verification of a B Development Process, In: Proceedings of 9th European Workshop on Dependable Computing, May (1998) 93-96
5. Bieman, J.M., Baker, A.L., Clites, P.N., Gustafson, D.A., Melton, A.C.: A Standard Representation of Imperative Language Programs for Data Collection and Software Measures Specification, The Journal of Systems and Software 8 (1998) 13-37
6. Breuer, P.T., Bowen, J.P.: Towards Correct Executable Semantics for Z, In: Proceedings of the Z User Workshop, (1994) 185-209
7. Chilenski, J.J., Miller, S.P.: Applicability of Modified Condition/Decision Coverage to Software Testing, Software Engineering Journal, September (1994) 193-200
8. Dick, J., Faivre, A.: Automating the generation and sequencing of test cases from model-based specifications, In: Proceedings of the International Formal Methods Europe Symposium, (1993) 268-284
9. Frankl, P.G., Weyuker, E.J.: An Applicable Family of Data Flow Testing Criteria, IEEE Transaction on Software Engineering, Vol. 4, No. 10, October (1988) 1483-1498
10. Hierons, R.M.: Testing from a Z Specification, Software Testing, Verification and Reliability, Vol. 7 (1997) 19-33
11. Van Aertryck, L., Benveniste, M., Le Métayer, D.: CASTING: a Formally Based Software Testing Generation Method, In: Proceedings of 1st International Conference on Formal Engineering Methods, IEEE Computer Society, November (1997)
12. Waeselynck, H., Behnia, S.: Towards a Framework for Testing B Models, LAAS Report No. 97225, June (1997)
13. Waeselynck, H., Behnia, S.: B Model Animation for External Verification, In: Proceedings of 2nd International Conference on Formal Engineering Methods, IEEE Computer Society, December (1998) 36-45