Under consideration for publication in Formal Aspects of Computing

Test-Data Generation for Control Coverage by Proof

Ana Cavalcanti[1], Steve King[1], Colin O'Halloran[2], and Jim Woodcock[1]

[1] Department of Computer Science, University of York, UK
[2] Department of Computer Science, University of Oxford, UK

Abstract. Many tools can check if a test set provides control coverage; they are, however, of little or no help when coverage is not achieved and the test set needs to be completed. In this paper, we describe how a formal characterisation of a coverage criterion can be used to generate test data; we present a procedure based on traditional programming techniques like normalisation, and weakest precondition calculation. It is a basis for automation using an algebraic theorem prover. In the worst situation, if automation fails to produce a specific test, we are left with a specification of the compliant test sets. Many approaches to model-based testing rely on formal models of a system under test. Our work, on the other hand, is not concerned with the use of abstract models for testing, but with coverage based on the text of programs. Although our examples present programs using a mathematical notation, they have a direct correspondence to code.

Keywords: control coverage, semantics, UTP, invariants.

1. Introduction

Testing is the most popular software verification technique. For safety-critical systems, certification authorities enforce levels of testing in terms of both requirements and code coverage. In the case of code coverage, widely used white-box testing techniques provide a number of structural criteria based on the conditions and decisions of a program. Conditions are primitive boolean expressions; decisions are conditions, or boolean expressions formed from conditions using operators of the programming language.

Typically, requirements-based testing is conducted first, and then the test set is analysed to establish if the desired structural coverage has been achieved as well. Many tools are available to check whether a test set achieves a required control coverage criterion; some examples are LDRA TestBed [LDR], AdaTest95 [IPL], PureCoverage [Rat], C-cover [Bul], and TCAT [Sof]. If we need to generate further test cases to complete a test set, however, there is not much support at hand. Tools usually indicate which conditions have not been exercised as required, but no test cases or even paths are provided. For criteria that require particular combinations of condition evaluations to be exercised, the information produced is even less useful.

Generation of test sets to guarantee coverage is a computationally complex problem, which is exacerbated

Correspondence and offprint requests to: Ana Cavalcanti (`Ana.Cavalcanti@york.ac.uk`) by the possibility of interactions between decisions and by program preconditions, both of which can make test cases infeasible. A naïve approach, in which all test sets are generated and tried out, is not practical. The large number of test sets that can be considered for a program with a reasonable number of inputs, and the complexity of checking feasibility, make this a very difficult problem. The technique presented in this paper identifies feasible paths to the decisions in the program automatically, and uses that information to generate values, or at least specifications, for the missing test data.

For model-based testing techniques, in which test data is generated using a model of the system (requirements), approaches based on model checking and control coverage criteria based on the model of the system, rather than on the system or its source code, have been very successful [CSE96, HLSU02, RH03]. Here, we are primarily concerned with white-box testing of code, for which models are usually too detailed and large to be tractable by model checking. Our technique does not rely on a model of the program, but only on the semantics of the programming language, and a formal characterisation of the structural coverage criterion.

We rely on traditional techniques of program development: normalisation, loop variants and invariants, and weakest preconditions. For programs that have been subject to rigorous verification, loop variants and invariants are likely to be readily available. If this is not the case, invariant calculators like those described in [EPG$^+$07, BDS06] can be used. They may suggest weak invariants, but our technique is resilient to that: it may fail to provide a test set, but if it does, we can be certain that the required coverage is achieved. For programs that have been developed or verified formally, test data comes as a by-product.

The *de facto* standard "DO-178C" [RCC11] provides development guidelines for avionics systems and has influence in many other domains. For Level A systems, whose failures have catastrophic consequences, DO-178C recommends MC/DC (*Modified Condition/Decision Coverage*) testing [CM94]. It requires that the conditions in all the decisions of a program have an independent effect on its execution. In examples, we consider MC/DC, but our technique applies to other control coverage criteria. Since, as explained above, we can get test data as a by-product of verification, our technique provides further encouragement for the use of formal verification in the context of systems that are subject to stringent testing requirements.

Our language includes the main constructs of imperative programming. It is simple, but sufficient to cover programs written using the restricted subsets of C and Ada usually adopted in the safety-critical systems industry. For programs written in a safe subset of Ada [Bar95], the verification technique in [AC05], for example, is based on a characterisation of programs in a refinement language similar to that we consider here. This verification approach has been successfully applied in industry.

The semantic framework that we use for the formalisation of coverage and of program meaning is the relational model of the unifying theories of programming (UTP) [HH98]. It uses an alphabetised predicate calculus, in the style of Z [WD96] or B [Abr96], for example, to specify relations. In this way, it is very convenient for mechanisation using any theorem prover that supports the first-order predicate calculus [ZC12b]. On the other hand, it is powerful enough to model constructs from a variety of programming paradigms, so that it does not restrict the kind of language to which we can apply our technique.

Advanced and modern programming constructs have already been given a UTP model. Pointers [CHW06], object orientation [ZC12a], concurrency [HH98], and real time [SCHS10] are some examples. In addition, the simple language that we consider here is expressive enough to describe discrete-time control diagrams, in the style described in [ACOS00]. Our technique is described here in the simple setting of a restricted language, but the use of the UTP makes the principle and ideas applicable to a rich set of programs.

In our technique, test-data generation is achieved in four steps. The first step reveals the paths to the decisions in the program. In preparation for the third step, the second step normalises the definition of the paths. In the third step, we calculate the test data used to exercise each decision. In the final step, we determine whether there are decisions not covered, and identify and specify the missing test data, if any.

In the next section, we describe our approach to formalisation of coverage. Section 3 presents our main contribution: a procedure for test-data generation. Section 4 discusses automation and applicability. Section 5 is on related works. Finally, Section 6 summarises our results and suggests future work. An appendix lists the laws of (test) sets that we use in our procedure.

2. Testing semantics

The inputs of our test-data generation procedure are a program and a formalisation of a structural coverage criterion. The formalisation required is a testing semantics, which associates a term (decision or program) to the set of test sets that provide coverage. To illustrate our approach to testing semantics definition, we present a characterisation of a non-trivial criterion: MC/DC (*Modified Condition/Decision Coverage*). We use Z [WD96] as a meta-notation, and explain any use of non-standard mathematical symbols as needed.

MC/DC coverage requires that the conditions in all decisions of a program have an independent effect on its execution. Several varieties of MC/DC differ in the interpretation of what it means for a condition to independently affect the outcome of a decision [KB04]. In Section 2.1, we define unique cause MC/DC for decisions. There, we consider first decisions built up from boolean variables and then general decisions.

Example 2.1. For the decision $a \wedge b$, where a and b are boolean variables, a test set that provides unique cause MC/DC coverage is, for example, $\{\, \{a \mapsto \mathit{true}, b \mapsto \mathit{true}\}, \{a \mapsto \mathit{true}, b \mapsto \mathit{false}\}, \{a \mapsto \mathit{false}, b \mapsto \mathit{true}\}\,\}$. The first two tests can be used to ensure that b has an independent effect on $a \wedge b$, as they give the same value for a, and can establish that, by changing the value of b, the value of $a \wedge b$ changes. For similar reasons, the first and the last tests can establish the independent effect of a on the outcome of $a \wedge b$.

The testing semantics of programs requires that all their decisions are covered, and is independent of a particular criterion for coverage of decisions; it is given in Section 2.3. Section 2.2 gives a brief introduction to the UTP and the (functional) semantics of our programming language. It is needed to define the testing semantics in Section 2.3. For example, to determine if a test set covers a decision in a program, we need to determine, for each test case in the set, the values of the variables at the point at which the decision is evaluated. Existing testing tools usually execute instrumented programs to gather this information; we use a simple relational characterisation of programs in the UTP.

2.1. Decisions

To characterise the test sets that provide coverage for decisions, we define decisions as formulas, and provide an evaluation function that defines the boolean value of a formula for a given test case. This is necessary because some criteria are based on the syntax of the decisions, not their meanings. We first consider decisions built up only from boolean variables: elements of a given set Var; realistic decisions are handled later.

In our approach, we define a test as a function that determines values for the input variables. For decisions, we consider the set T of all tests defined as $T == \mathit{Var} \nrightarrow \mathit{Bool}$. This introduces the set T as the set of all partial functions from variables to boolean values: elements of the set Bool, which can itself be defined as $\mathit{Bool} ::= \mathit{true} \mid \mathit{false}$ to include the elements true and false. A test set is an element of $\mathit{TS} == \mathbb{P}\, T$. Here, we introduce the set TS defined as the power set of T.

The appropriate representation for formulas depends on the coverage criterion of interest. Here, we only specify the existence of a given set $\mathit{Formula}$ of well-formed formulas, and of an evaluation function $\mathit{Ev} : \mathit{Formula} \times T \nrightarrow \mathit{Bool}$. This is a partial function that associates a formula and a test with a boolean value. That is all that we need for unique cause MC/DC.

To formalise unique cause MC/DC coverage, we need to define two concepts: neighbour tests, and input-sensitive tests. The function η defined below gives the set of pairs of tests that are neighbours with respect to a given variable v: unordered pairs of tests that differ only in the value of v. For example, the tests $\{a \mapsto \mathit{true}, b \mapsto \mathit{true}\}$ and $\{a \mapsto \mathit{true}, b \mapsto \mathit{false}\}$ are neighbours with respect to b. The function η is characterised as a total function from Var to sets of UPairT, which is the set of unordered pairs of tests. We represent unordered pairs using sets of two elements, and so we define $\mathit{UPairT} == \{\, ts : \mathit{TS} \mid \# ts = 2 \,\}$.

$$\eta : \mathit{Var} \rightarrow \mathbb{P}\, \mathit{UPairT}$$
$$\forall v : \mathit{Var} \bullet \eta\, v = \{\, t : T \mid v \in \operatorname{dom} t \bullet \{\, t, t \oplus \{\, v \mapsto \mathit{not}(t\, v) \,\}\,\}\,\}$$

The operator $\oplus$ is function overriding, and not is a negation operator for Bool. For a variable v, we have that $\eta\, v$ contains the pairs of tests formed by pairing any test t that gives a value for v with the test obtained by changing (negating) just the value of v in t. The values of all other variables in t are kept.

The function IS gives the input-sensitive tests for a formula f with respect to a variable v. These are the tests that make the evaluation of f change if we vary only the value of v. For example, $\{a \mapsto \mathit{true}, b \mapsto \mathit{true}\}$ is input sensitive for $a \wedge b$ with respect to a and b, but $\{a \mapsto \mathit{false}, b \mapsto \mathit{true}\}$ is input sensitive for $a \wedge b$ with

Table 1. Imperative programming language

Construct	**Syntax**	**Semantics**
assignment	$x :=_{\{x,u\}} e$	$x' = e \land \boldsymbol{u}' = \boldsymbol{u}$
skip	$I\!I_{\boldsymbol{v}}$	$\boldsymbol{v}' = \boldsymbol{v}$
sequence	$p_{\boldsymbol{v},\boldsymbol{v}'} \; ; \; q_{\boldsymbol{v},\boldsymbol{v}'}$	$\exists \boldsymbol{v}_0 \bullet p_{\boldsymbol{v},\boldsymbol{v}_0} \land q_{\boldsymbol{v}_0,\boldsymbol{v}'}$
conditional	$p_{\boldsymbol{v},\boldsymbol{v}'} \lhd d_{\boldsymbol{v}} \rhd q_{\boldsymbol{v},\boldsymbol{v}'}$	$d_{\boldsymbol{v}} \land p_{\boldsymbol{v},\boldsymbol{v}'} \;\lor\; \neg\, d_{\boldsymbol{v}} \land q_{\boldsymbol{v},\boldsymbol{v}'}$
recursion	$\mu\, X \bullet F(X)$	$\bigsqcap\{\, X \mid F(X) \sqsubseteq X \,\}$
variable block	**var** $x : T \bullet p_{x,u,x',u'}$	$\exists\, x, x' \bullet x := \mathcal{DV}(T) \; ; \; p_{x,u,x',u'}$

respect to a, but not b. We define IS as a function from pairs of formulas and variables to a set of tests.

$$IS : Formula \times Var \rightarrow\!\!\!\!\!+\; TS$$

$$\forall f : Formula;\; v : Var \bullet IS(f, v) =$$
$$\{\, t : T \mid (f, t) \in \operatorname{dom} Ev \land (f, t \oplus \{\, v \mapsto not(t\, v) \,\}) \in \operatorname{dom} Ev \land Ev(f, t) \neq Ev(f, t \oplus \{\, v \mapsto not(t\, v) \,\}) \,\}$$

We use $\operatorname{dom} Ev$ to denote the domain of the evaluation function Ev, and consider the tests t for which both (f, t) and $(f, t \oplus \{\, v \mapsto not(t\, v) \,\})$ are in the domain of Ev.

The function ψ gives the set of input-sensitive neighbours of a formula f with respect to a variable v.

$$\psi : Formula \times Var \rightarrow\!\!\!\!\!+\; \mathbb{P}\, UPairT$$

$$\forall f : Formula;\; v : Var \bullet \psi(f, v) = \{\, n : UPairT \mid n \in \eta\, v \land \exists\, t : n \bullet t \in IS(f, v) \,\}$$

In defining the set $\psi(f, v)$, we consider unordered pairs n of tests. We require the tests in n to be neighbours as defined by $\eta\, v$, and that one of these neighbours is input sensitive according to $IS(f, v)$. We observe that, given a pair n of neighbours, either both of them are input sensitive or neither is. So, by requiring that one test in n is input sensitive, we are implicitly requiring that they both are input sensitive.

Finally, the function UC gives the test sets that provide unique cause MC/DC coverage for a formula; we call such tests sets compliant. In the definition of UC, we use a function $FV : Formula \rightarrow \mathbb{P}\, Var$ that gives the set of free variables of a formula. Its definition depends on the particular syntax adopted for decisions.

$$UC : Formula \rightarrow \mathbb{P}\, TS$$

$$\forall f : Formula \bullet UC(f) = \{\, ts : TS \mid \forall\, v : FV(f) \bullet (\exists\, t_1, t_2 : ts \bullet \{\, t_1, t_2 \,\} \in \psi(f, v)) \,\}$$

For every free variable v in the decision represented by the formula f, a compliant test set must include a pair of tests t_1 and t_2 that are neighbours and input sensitive with respect to v as defined by ψ.

A more flexible and more widely used MC/DC variant is masking; its specification requires a more elaborate model of formulas. A formalisation can be found in [CKOW07], where four variants of MC/DC are described using Z. Unique cause MC/DC is, however, enough to illustrate our technique. We describe next our programming language and its (functional) semantics.

2.2. Unifying theories of programming

The UTP is a framework that can be used for a unified characterisation of many programming language paradigms. It is based on an alphabetised relational calculus, and all programming constructs, as well as designs and specifications, are interpreted as relations between two observations of behaviour. Alphabetised predicates in the style of those used in Z or VDM, for example, are used to specify relations.

As already said, our technique requires the definition of loop variants; as a consequence, it applies to terminating programs. For these, the simple general theory of relations of the UTP is enough. The syntax and semantics of our language as a predicate in this theory are presented in the Table 1.

Example 2.2. The semantics of the assignment $x := x + 1$ depends on which programming variables are in scope, since they determine the alphabet of the relation. If there are only two programming variables, x and y, then the semantics is given by the alphabetised predicate: $(\, \{x, y, x', y'\}, x' = x + 1 \land y' = y \,)$. The first element of the pair, which is usually omitted, gives the set of names of the variables in the alphabet;

for a relation, it includes decorated variables like x' and y' to denote the values of the variables x and y in a later observation. The names x and y refer to the values of the variables in the initial observation.

In Table 1, we indicate that the assignment construct determines the set of variables in scope, so that, strictly speaking, we write $x :=_{\{x,y\}} x + 1$, rather than simply $x := x + 1$. In examples, however, we often omit the alphabet when it is clear from the context. We take u and v to be lists of variables, and u' and v' the lists of corresponding dashed variables. The predicate $u = u'$ is a conjunction of equalities that require the value of each value in u to be that of the corresponding dashed variable in u'. Similar comments apply to $v = v'$.

The program $I\!I_{\boldsymbol{v}}$ is skip (with alphabet $\boldsymbol{v}$). It terminates without changing the value of any variables. We write v, v' for the set of variables in these lists, or just v if there are no dashed variables in the set.

A program sequence $p;\ q$ is defined by relational composition, which is expressed using an existential quantification over the intermediate value v_0 of the alphabet variables. In Table 1, we explicitly indicate the alphabets of the programs p and q as subscripts, but leave them implicit in the sequel.

Recursion is defined by least fixed points, but an explicit definition is provided. We use "$\bigsqcap$" to denote the greatest lower-bound in the complete lattice of predicates ordered by the refinement relation "$\sqsubseteq$".

A conditional $p \ \lhd\ d\ \rhd\ q$ behaves like p if d holds, and otherwise behaves like q. We use a similar notation for conditional expressions; in fact, we write $e \ \lhd\ d\ \rhd\ f$ to denote not a single conditional expression, but a list of conditionals: one for each expression in the list e or f. These lists must have the same length.

To avoid nondeterminism, we define that freshly declared variables of a type T take a default value characterised by a function $\mathcal{DV}(T)$. This is different from [HH98], where that initial value is arbitrary. In the UTP, programs are predicates, and can be used interchangeably. For a variable block $\mathbf{var}\ x : T \bullet p_{x,\boldsymbol{u},x',\boldsymbol{u}'}$, the semantics is given by the assignment $x := \mathcal{DV}(T)$ followed in the sequence by the body $p_{x,\boldsymbol{u},x',\boldsymbol{u}'}$ of the block, with the newly introduced variables x and x' existentially quantified.

In [HH98], a suite of laws is presented for each construct studied in the UTP, and normalisation is used to bring order to that collection. For deterministic terminating programs, the normal form is an assignment to all variables. Normalisation is achieved by the exhaustive application of laws to

1. totalise assignments: $(u := e) = (x, u := x, e)$;
2. combine sequences of (normalised) assignments using substitution:
$$(v := e_1;\ v := e_2) = (v := e_2[e_1/v])$$
3. eliminate conditionals (involving normalised assignments) in favour of conditional expressions:
$$((v := e) \ \lhd\ d\ \rhd\ (v := f)) = (v := e \lhd d \rhd f)$$
4. eliminate variable declarations based on the default values
$$(\mathbf{var}\ x : T \bullet v := e) = (v := e[\mathcal{DV}(T)/x])$$

With the definition of variants that guarantee termination, we can also reduce recursions to assignments. In our technique, we use normalisation laws to simplify programs, but the recursions are not eliminated.

2.3. General expressions and programs

In this section, we define a function $[\![\_]\!]^T$ that gives a testing semantics for expressions and programs. It is the definition of this function on programs that is used in our procedure for test-data generation.

First, we consider expressions in general, including decisions involving values of arbitrary types (and not only boolean variables like in Section 2.1). For convenience, we now consider a test case to be an assignment of values to variables, rather than a function, so that a test set is a set of assignments.

Example 2.3. We consider again the decision $a \wedge b$, involving boolean variables a and b. An element of $[\![\, a \wedge b \,]\!]^T$ is the set of tests $\{(a, b := \mathit{true}, \mathit{true}), (a, b := \mathit{true}, \mathit{false}), (a, b := \mathit{false}, \mathit{true})\}$.

The conversion between the different representations of a test is a simple matter, and the view of tests as assignments simplifies the testing semantics of programs. In particular, the definitions of the testing semantics of sequences and conditionals rely on the fact that tests are assignments.

2.3.1. Expressions

An important piece of information used in the definition of the semantics of an expression is its type. We assume that a function $\mathcal{T}$ determines if an expression is a decision involving only boolean variables, is not a decision and involves no decision as a subexpression at all, or involves a mixture of decisions and non-boolean expressions. In this last case, the expression itself may be boolean: a decision like $i < j \land j < 10$ involving non-boolean variables i and j or even more complex non-boolean expressions. If the expression is not boolean, then it has a decision as a subexpression; an example is a conditional expression like $i \lhd i < j \rhd j$, whose integer value is either i or j depending whether $i < j$ or not.

For a given expression, $\mathcal{T}$ gives an element of $Type ::= decision \mid noDecision \mid mixedB \mid mixedNB$. We omit the definition of $\mathcal{T}$, which depends on the expression language, and is trivial, given a specification of the type system. The definition of $[\![e]\!]^T$ depends on $\mathcal{T}(e)$.

Decisions For a decision, the semantics is as in Section 2.1. Below, we use a function $\mathcal{P}$ that produces a formula that represents the expression, and the function UC defined in Section 2.1.

$$[\![e]\!]^T \mathrel{\widehat{=}} \{ ts : UC(\mathcal{P}(e)) \bullet \mathcal{F} (\!| ts |\!) \} \qquad [\textbf{\textit{provided}}\ \mathcal{T}(e) = decision]$$

The definition of $\mathcal{P}$ is a simple parsing problem. The function $\mathcal{F}$ transforms the representation of a test as a function to an assignment. It is applied above to each test of ts; $\_ (\!| \_ |\!)$ is the relational image operator.

No decisions If an expression involves no decisions, then we have the semantics below.

$$[\![e]\!]^T \mathrel{\widehat{=}} TS \qquad [\textbf{\textit{provided}}\ \mathcal{T}(e) = noDecision]$$

Such an expression is not relevant: any test set is acceptable as it can be deemed to cover e.

Non-boolean mixed expressions These are conditionals $e_1 \lhd d \rhd e_2$, for example, which have their semantics defined compositionally as shown below.

We need a new operator; it restricts a test set ts to the tests that satisfy a decision d: those for which $t\ ;\ d$ holds. We observe that $t\ ;\ d$ is a predicate, since programs are predicates in the UTP. In particular, $t\ ;\ d$ states that after the assignment t, the decision d holds.

Example 2.4. The program $(a, b := false, true)\ ;\ (\neg\ a)$ is, according to the definition of sequence in Table 1, the predicate $\exists\ a_0, b_0 \bullet a_0 = false \land b_0 = true \land \neg\ a_0$, which is true.

Accordingly, in filtering a test set ts, we keep the tests t in ts for which $t\ ;\ d$ holds.

Definition 2.1 (Test set filter). $ts \mathbin{\hat{\text{⁏}}} d \mathrel{\widehat{=}} \{t : ts \mid t\ ;\ d\}$

Example 2.5. If ts is the set $\{(a, b := true, false), (a, b := false, true)\}$, then $ts \mathbin{\hat{\text{⁏}}} (\neg\ a)$ includes the test $a, b := false, true$, but not $a, b := true, false$.

For a conditional expression $e_1 \lhd d \rhd e_2$, a compliant test set covers d, and, moreover, covers e_1 if restricted to tests for which d evaluates to true, and covers e_2 when restricted to tests for which d evaluates to false.

$$[\![e_1 \lhd d \rhd e_2]\!]^T \mathrel{\widehat{=}} \{ ts : [\![d]\!]^T \mid (ts \mathbin{\hat{\text{⁏}}} d) \in [\![e_1]\!]^T \land (ts \mathbin{\hat{\text{⁏}}} (\neg\ d)) \in [\![e_2]\!]^T \}$$
$$[\textbf{\textit{provided}}\ \mathcal{T}(e_1 \lhd d \rhd e_2) = mixedNB]$$

Example 2.6. For a simple expression like $i \lhd i < j \rhd j$, since the testing semantics of i and j is TS, we are only required to find a test set ts that covers $i < j$. Even if the filter $ts \mathbin{\hat{\text{⁏}}} (i < j)$ or $ts \mathbin{\hat{\text{⁏}}} (i \geq j)$ gives an empty set of tests, this is not a problem as any element of TS is acceptable to cover i and j.

Mixed decisions If a boolean expression e involves, for example, equalities or inequalities, our definition of its semantics involves a syntactic transformation captured by the function $\mathcal{D}$ defined below. Basically, it defines a decision in which each condition in e is represented by a fresh boolean variable, and records the association; in doing so, it captures the decision structure of e.

Example 2.7. In the case of $i < j \land j < 10$, we need two new variables, say a and b, to represent $i < j$ and $j < 10$. $\mathcal{D}$ associates $i < j \land j < 10$ with $a \land b$ and $(a = (i < j)) \land (b = (j < 10))$ The result of $\mathcal{D}$ is a pair: the first element $\mathcal{D}(e).1$ is the new expression, and the second $\mathcal{D}(e).2$ characterises the new variables. For a pair p, the notation $p.1$ is used to refer to its first element; similarly $p.2$ is its second element.

To define $\mathcal{D}$, we consider a very simple expression language.

$$e \in \mathit{Expression} \quad ::= \quad x \mid e_1 = e_2 \mid e_1 < e_2 \mid e_1 \wedge e_2 \mid e_1 \vee e_2 \mid \neg\, e$$

We give a recursive definition for $\mathcal{D}$.

$$\begin{array}{ll}
\mathcal{D}(x) \;\widehat{=}\; (x, \mathit{true}) & [\textbf{\textit{provided}}\ \mathcal{T}(x) = \mathit{decision}] \\
\mathcal{D}(e_1 = e_2) \;\widehat{=}\; (a, a = (e_1 = e_2)) & [\textbf{\textit{provided}}\ a \text{ is fresh}] \\
\mathcal{D}(e_1 < e_2) \;\widehat{=}\; (a, a = (e_1 < e_2)) & [\textbf{\textit{provided}}\ a \text{ is fresh}] \\
\mathcal{D}(e_1 \wedge e_2) \;\widehat{=}\; (\mathcal{D}(e_1).1 \wedge \mathcal{D}(e_2).1, \mathcal{D}(e_1).2 \wedge \mathcal{D}(e_2).2) & \\
\mathcal{D}(e_1 \vee e_2) \;\widehat{=}\; (\mathcal{D}(e_1).1 \vee \mathcal{D}(e_2).1, \mathcal{D}(e_1).2 \wedge \mathcal{D}(e_2).2) & \\
\mathcal{D}(\neg\, e) \;\widehat{=}\; (\neg\, \mathcal{D}(e).1, \mathcal{D}(e).2) &
\end{array}$$

In the case of a variable x, we have a definition for $\mathcal{D}$ only if x is a boolean variable. If it is not and occurs on its own, then it is not a mixed expression, and $\mathcal{D}$ does not need to be used. If it occurs as part of an equality or inequality, however, then we define $\mathcal{D}$ for the enclosing expression, which does not require a definition of $\mathcal{D}$ for the variable itself. We use a to stand for a boolean variable, which is required to be fresh. In general terms, $\mathcal{D}$ distributes through the structure of expressions, but the second element of the pair that it defines accumulates the conjunction of associations between boolean variables and atomic boolean expressions.

Using $\mathcal{D}$, we define the testing semantics for mixed boolean expressions.

$$[\![\, e \,]\!]^{T} \;\widehat{=}\; \{\, \mathit{tsb} : [\![\, \mathcal{D}(e).1 \,]\!]^{T};\ \mathit{tsp} : \mathit{TS} \mid \mathit{tsb} = \mathcal{C}(\mathcal{D}(e).2)\, (\!|\, \mathit{tsp} \,|\!) \bullet \mathit{tsp} \,\} \qquad [\textbf{\textit{provided}}\ \mathcal{T}(e) = \mathit{mixedB}]$$

The tests in the sets tsb in $[\![\, \mathcal{D}(e).1 \,]\!]^{T}$ assign values to the new boolean variables introduced by $\mathcal{D}$. We are, however, interested in test sets tsp on the original programming variables. The function $\mathcal{C}$ uses the definition of the boolean variables in $\mathcal{D}(e).2$ to characterise the test on the boolean variables that corresponds to a given test on the programming variables. A compliant test set tsp is required to give rise to a test set tsb that is compliant for the decision on the boolean variables, if $\mathcal{C}$ is applied to each of the test cases.

Example 2.8. For the test $i, j := 1, 11$, the corresponding test on the boolean variables is $a, b := \mathit{true}, \mathit{false}$, if the definition of the boolean variables is $(a = (i < j)) \wedge (b = (j < 10))$.

We give the following definition for $\mathcal{C}$.

$$\mathcal{C}(P)(v := e) \;\widehat{=}\; \mathit{bv} := \mathit{lbv} \qquad [\textbf{\textit{provided}}\ P[\mathit{lbv}, e/\mathit{bv}, v]]$$

where v are the programming variables, and bv the boolean variables in P

For a test $v := e$, the result is an assignment $b := \mathit{lbv}$ of a list of literal boolean values lbv (that is, true or false) to a list of boolean variables bv. The values in lbv are those defined for bv in P, given that the variables in v take the values e. We assume that P defines a functional relationship between the programming and the boolean variables, like the second component of the pairs in the range in $\mathcal{D}$.

The definition of the testing semantics of mixed boolean expressions given above is not constructive: the characterisation of tsp using $\mathcal{C}$ is implicit. For our technique, however, what matters is that we have a formal characterisation of the semantics. We do not generate the tests using this definition directly, but a specification that is calculated using this definition, and the others that compose the testing semantics.

2.3.2. Skip and assignment

With the definition of a testing semantics for expressions, the semantics of assignments is quite simple.

$$[\![\, x := e \,]\!]^{T} \;\widehat{=}\; [\![\, e \,]\!]^{T}$$

A test set is compliant for an assignment if it is compliant for the assigned expression.

For $I\!I$, any test set is acceptable.

$$[\![\, I\!I \,]\!]^{T} \;\widehat{=}\; \mathit{TS}$$

This is not necessarily the same testing semantics of an assignment of all variables to themselves, which is a possible definition of the behaviour of II. For the assignment, the presence of boolean variables would impose restrictions on the test sets. This illustrates the fact that the testing semantics depends on the syntax of a program, and not only on its behaviour (functional semantics).

2.3.3. Sequence

In order to define a semantics for sequence, we introduce a new operator.

Definition 2.2 (Lifted sequence). $ts \mathbin{⨾} p \ \widehat{=}\ \{\, t : ts \bullet t \; ; \, p \,\}$

This characterises the test set obtained by executing p for each of the tests t in a test set ts. The assignment t sets up values for the programming variables, which are used and possibly changed by p. The values of the variables after p is executed determine a new test case t; p that is included in $ts \mathbin{⨾} p$.

Well-definedness of this operator requires that $t \; ; p$ is a test: an assignment to all programming variables. Since p is a deterministic terminating program, this follows from the results on normalisation [HH98].

Example 2.9. If ts is $\{(a, b := \mathit{true}, \mathit{false}), (a, b := \mathit{false}, \mathit{true})\}$, then $ts \mathbin{⨾} (a := \mathit{true} \lhd \neg\, a \rhd b := \mathit{true})$ includes only the test $(a, b := \mathit{true}, \mathit{true})$, since this is what we obtain as the result of the normalisation of both $(a, b := \mathit{true}, \mathit{false}) \; ; (a := \mathit{true} \lhd \neg\, a \rhd b := \mathit{true})$ and $(a, b := \mathit{false}, \mathit{false}) \; ; (a := \mathit{true} \lhd \neg\, a \rhd b := \mathit{true})$.

In Definition 2.2, we take advantage of the fact that a test is a program.

A test set compliant for p_1; p_2 is compliant for p_1 and, moreover, the results of each of its tests after p_1 (as defined by the lifted sequence operator) are tests that form a test set compliant for p_2.

$$[\![\, p_1; \; p_2 \,]\!]^{\,T} \ \widehat{=}\ \{\, ts : [\![\, p_1 \,]\!]^{\,T} \mid ts \mathbin{⨾} p_1 \in [\![\, p_2 \,]\!]^{\,T} \,\}$$

This is where the functional and the testing semantics meet. In isolation, p_2 may have a large number of compliant test sets, but we are only interested in those that are a feasible result of the execution of p_1.

2.3.4. Conditional

For a conditional $p_1 \lhd d \rhd p_2$, we have a definition similar to that of conditional expressions.

$$[\![\, p_1 \lhd d \rhd p_2 \,]\!]^{\,T} \ \widehat{=}\ \{\, ts : [\![\, d \,]\!]^{\,T} \mid (ts \mathbin{\hat{⨾}} d) \in [\![\, p_1 \,]\!]^{\,T} \wedge (ts \mathbin{\hat{⨾}} (\neg\, d)) \in [\![\, p_2 \,]\!]^{\,T} \,\}$$

With the use of the filter operator, it takes advantage of the fact that tests are programs, which are predicates.

2.3.5. Variable Blocks

A compliant test set for $\mathbf{var}\ x : T \bullet p$ is a compliant test set for p whose tests all give x the value $\mathcal{DV}(T)$.

$$[\![\, \mathbf{var}\ x : T \bullet p \,]\!]^{\,T} \ \widehat{=}\ \{\, ts : TS \mid ts \mathbin{⨾} x := \mathcal{DV}(T) \in [\![\, p \,]\!]^{\,T} \,\}$$

Since x is local, it is not included in the tests for the block, but only in the tests for its body p.

2.3.6. Recursion

We require the definition of a variant for each recursion. This is an integer expression, whose value is decreased before each recursive call, but never becomes negative; it is the basis of a justification that the recursion terminates. The syntax that we adopt is $\mu X \mid \mathit{vrt} \bullet F(X)$, where vrt is the variant.

To achieve coverage of a recursion, basically, we need to achieve coverage of its body. For that, however, we may take into account any of the possibly several times it is executed due to recursive calls. The variant gives us an upper bound on the number of iterations.

For a recursion body $F(X)$, which is a function on X from programs to programs, we define $F^n(X)$.

$$F^0(X) = X, \qquad \text{and} \qquad F^{(n+1)}(X) = F(F^n(X)), \text{ if } n \geq 0$$

We call this the program iteration operator. With it, we can defined the testing semantics of recursion.

$$[\![\, \mu X \mid \mathit{vrt} \bullet F(X) \,]\!]^{\,T} \ \widehat{=}\ \{\, ts : TS \mid \rho(F, \mathit{vrt}, ts) \in [\![\, F(X) \,]\!]^{\,T} \,\}$$

To determine if a test set ts is compliant for the body $F(X)$ of the recursion, we consider the tests in ts, and all tests carried out as a result of the recursive calls. These are collected in the set $\rho(F, vrt, ts)$ below.

Definition 2.3 (ρ operator). $\rho(F, vrt, ts) \mathrel{\widehat{=}} \{\, t : ts;\ n' : \mathbb{N} \mid t \,;\, (n' \leq vrt) \bullet t \,;\, F^{n'}(\mathit{I\!I}) \,\}$

In this definition, $t \,;\, (n' \leq vrt)$ constrains n' to be a number not exceeding the variant evaluated in the state described by t. When n' is 0, then $F^{n'}(\mathit{I\!I})$ is $\mathit{I\!I}$ and $t;\ \mathit{I\!I}$ is t itself, the original element of ts. This reflects the fact that, when there are no recursive calls, only the test cases in ts are carried out. If there are recursive calls, then extra test cases are tried out: those resulting from the execution of the body of the recursion the appropriate number of times after the variables are initialised with a test case of ts. Examples are given in Section 3. We call applications of the ρ function ρ-expressions.

We also need to define a testing semantics for recursion variables. They do not include decisions, so they do not impose any restriction: $[\![\, X \,]\!]^{T} \mathrel{\widehat{=}} TS$.

The testing semantics for programs does not rely directly on the particular coverage criterion for decisions adopted, but just on the semantics of expressions. To define that, we did refer to the function UC, which formalises unique cause MC/DC coverage. The adoption of a different criterion, however, only requires the use of a different function; we can say that we have a parametrised semantics for structural testing coverage. In the next section, we explain how it can be used to generate compliant test sets.

3. Test generation

In the context of a testing semantics, and given a test set ts and a program p, to establish whether ts provides coverage for p, we need to prove that $\boxed{ts \in [\![\, p \,]\!]^{T}}$. We call this a compliance predicate for p and ts.

In this section, we provide a procedure to prove that a compliance predicate $ts \in [\![\, p \,]\!]^{T}$ is a theorem. If it is, then ts is one of the test sets in $[\![\, p \,]\!]^{T}$, and, therefore, one of those that provide coverage for p. If, on the other hand, the compliance predicate is not a theorem, the procedure generates specifications for the missing tests. A constraint solver can then be used to generate specific additional tests. Using this procedure, we can complete an existing test set, generated, for instance, based on the requirements. In addition, we can start with an empty test set, and generate specifications for tests. Figure 1 describes the procedure. The laws referenced there are listed in Appendix A; we also present and explain these laws later in this section.

To illustrate the approach, we consider the program *Merge* in Figure 2. It takes as input two non-empty sorted arrays a and b of sizes N and M, and produces an ordered array c that contains all the elements in a or b, but without duplicates. The boxes indicate named parts of this program and include the names on their top righthand corners. For clarity, we write an assignment of an expression e to a particular position x of an array v as $v[x] := e$, instead of $v := v \oplus \{x \mapsto e\}$, which is a more precise characterisation of the meaning of the assignment. We also make use of multiple assignments; their functional and testing semantics is a trivial generalisation of that in the previous section.

In *Merge*, the local variables i, j, and k are indices used for the arrays a, b, and c. After initialising the first element of c (C_1), the program loops over a and b (ML) to determine the next smallest element that is not already in c. Elements already stored in c are skipped. When either a or b is exhausted, this main loop terminates; the variant is the sum of the number of remaining elements in a and b. Extra loops copy across the remaining elements of a or b, eliminating duplicates; their variants are the number of remaining elements in the array. As usual, loops are written in terms of recursion: **while** b **do** p **end** $\mathrel{\widehat{=}} \mu\, X \bullet p \,;\, X \lhd b \rhd \mathit{I\!I}$. As previously noted, however, we record the variant vrt of the loop by writing $\mu\ X \mid vrt \bullet p \,;\, X \lhd b \rhd \mathit{I\!I}$.

As already explained, our technique comprises four steps, which we describe in the following sections. In the first step, we rewrite the compliance predicate $ts \in [\![\, p \,]\!]^{T}$ as a conjunction in which each conjunct defines a path to a decision d of p, and requires that after its execution, d is covered if all the tests in ts are executed. Both the tests in ts and the paths involve programs; in the second step, they are all normalised to simplify further reasoning. The third step calculates the result of executing each of the paths for each of the tests in ts. Finally, in the fourth step, we determine if the tests calculated are enough, and if not use weakest preconditions to calculate a specification for the missing tests.

Input: testing semantics $[\![\_]\!]^{T}$, program p, and test set ts

1. Normalise the compliance predicate $\tau \in [\![p]\!]^{T}$, where τ is a fresh meta-variable.
   - (a) Exhaustively expand the applications of $[\![\_]\!]^{T}$ using its definition;
   - (b) Exhaustively apply the lift-semi-assoc law;
   - (c) Eliminate path conjuncts of the form $tsd \in TS$.
2. Normalise the programs (in the normalised compliance predicate and in ts) but keep the recursions.
3. Calculate the test sets.
   - (a) Substitute ts for τ in the normalised compliance predicate;
   - (b) Exhaustively

     For each expression of the form $\sigma \mathbin{⨾} p$, where σ is a set enumeration
     - i Apply Definition 2.2;
     - ii Normalise the programs in the resulting test set, including the recursions.

     For each expression of the form $\rho(F, vrt, \sigma)$, where σ is a set enumeration
     - i Apply Definition 2.3;
     - ii Apply the law set-comp-range-dist;
     - iii Apply the law test-expr-eval to the constraint in each set comprehension;
     - iv Apply the law set-comp-enum to each set comprehension;
     - v Normalise the programs in the resulting test set, including the recursions, and using the definition of program iteration when needed.

     For each expression of the form $\sigma \mathbin{⨾} d$, where σ is a set enumeration
     - i Apply Definition 2.1;
     - ii Apply the law set-comp-test-const using the law test-expr-eval to evaluate the constraint for each test.
4. Check compliance for decisions: for each path conjunct $tsd \in [\![d]\!]^{T}$
   - (a) if it is true, eliminate it;
   - (b) if it is false, for each of its target evaluations $(b_1, \ldots, b_n)$
     - i Calculate $wp.path(tsd).tpred((b_1, \ldots, b_n), d)$ by exhaustively applying the wp definition
       - A To deal with $wp.F^n(I\!I).\phi$, take as input a bound for n;
       - B To deal with $wp.(\mu\ X \bullet F(X) \lhd c \rhd I\!I).\phi$, take the invariant inv of the loop, and calculate $wp.(frame(F(X)) : [inv, inv \wedge \neg\, c]).\phi$.
     - ii Use a constraint solver to determine values for input variables.
   - (c) Revisit the test sets: consider the set $nts = \{t_1, \ldots, t_n\}$ of tests generated in step 4(b)
     - i if nts is empty, finish (the current iteration of step 4);
     - ii otherwise, for each path conjunct $tsd \in [\![d]\!]^{T}$ still to be considered in step 4
       - A consider the definition of that conjunct after step 2, and substitute nts for τ;
       - B apply step 3(b) to it to obtain a conjunct $ntsd \in [\![d]\!]^{T}$;
       - C rewrite the path conjunct to $(ntsd \cup tsd) \in [\![d]\!]^{T}$.

Fig. 1. Test-data generation procedure

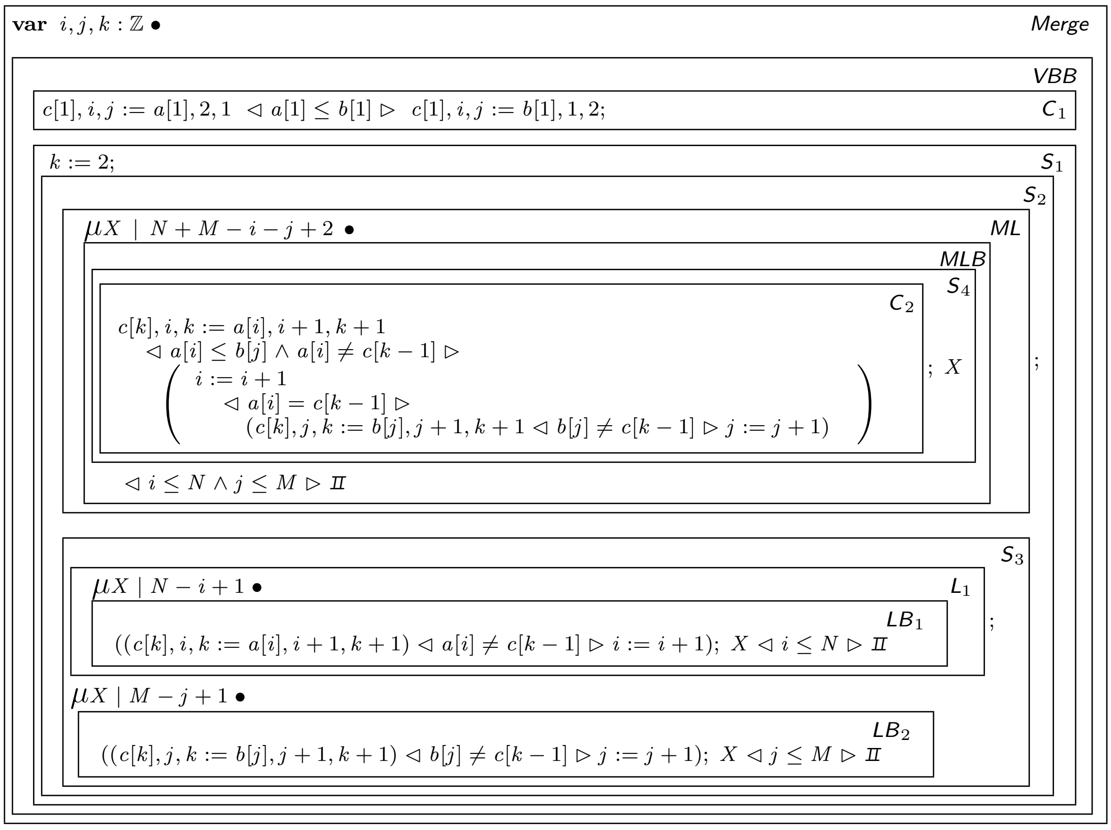
```
$\mathbf{var}\ i, j, k : \mathbb{Z} \bullet$
    $c[1], i, j := a[1], 2, 1 \ \lhd a[1] \leq b[1] \rhd\ c[1], i, j := b[1], 1, 2;$
    $k := 2;$
    $\mu X \mid N + M - i - j + 2 \bullet$
        $c[k], i, k := a[i], i + 1, k + 1$
            $\lhd a[i] \leq b[j] \wedge a[i] \neq c[k-1] \rhd$
            $\Big($ $i := i + 1$
                $\lhd a[i] = c[k-1] \rhd$
                    $(c[k], j, k := b[j], j + 1, k + 1 \lhd b[j] \neq c[k-1] \rhd j := j + 1)$ $\Big)$
        $; X$
        $\lhd i \leq N \wedge j \leq M \rhd \mathit{II}$
    $;$
    $\mu X \mid N - i + 1 \bullet$
        $((c[k], i, k := a[i], i + 1, k + 1) \lhd a[i] \neq c[k-1] \rhd i := i + 1);\ X \lhd i \leq N \rhd \mathit{II}$
    $;$
    $\mu X \mid M - j + 1 \bullet$
        $((c[k], j, k := b[j], j + 1, k + 1) \lhd b[j] \neq c[k-1] \rhd j := j + 1);\ X \lhd j \leq M \rhd \mathit{II}$
```

Fig. 2. Merge sort (input: non-empty sorted arrays a and b; output: array c); N is the size of a and M of b

3.1. Normalising the compliance predicate

In this step, we rewrite the compliance predicate to express it solely in terms of the testing semantics of the decisions in p. More precisely, we write it in a normal form defined as follows.

Definition 3.1 (Normalised compliance predicate). A compliance predicate is normalised if it is a conjunction of path conjuncts. A path conjunct is a predicate of the form $tsd \in [\![\, d \,]\!]^{T}$, where tsd is a test set definition and d is a decision.

For our example, the normalised predicate is shown in Figure 3.

A test set definition specifies how the tests of a test set for a program, which define values for its input variables, are affected by the program behaviour along a particular path; this determines the actual tests that are exercised in the evaluation of a decision at the end of the path. A test set definition tsd is an expression; the BNF below defines the syntax. The set *Name* contains metavariables that represent arbitrary test sets; *Enum* contains all test sets defined in extension, that is, set enumerations. The sets *Dec* and *Prog* are determined by the programming language; the former contains the valid decisions, and the latter, programs.

Definition 3.2 (Test set definition).

$$\begin{array}{lll} \mathit{BTSD} & ::= & \mathit{Name} \mid \mathit{Enum} \mid \rho(F, \mathit{vrt}, \mathit{TSD}) \mid \mathit{TSD} \,\hat{\circ}\, \mathit{Dec} \\ \mathit{TSD} & ::= & \mathit{BTSD} \mid \mathit{BTSD} \,\hat{;}\, \mathit{Prog} \end{array}$$

A test set definition is a basic test set definition (an element of BTSD) either on its own or lifted by a program. A basic test set definition is a metavariable, an explicitly defined set of tests, a ρ-expression applied to a test set definition, or a test set definition filtered by a decision. In simple words, a test set definition cannot include consecutive lifts. This constraint on form simplifies reasoning.

I	$\widehat{=}$	$i, j, k := 0, 0, 0$	vML	$\widehat{=}$	$N + M - i - j + 2$
P_1	$\widehat{=}$	$I;\ C_1;\ k := 2$	vL_1	$\widehat{=}$	$N - i + 1$
P_2	$\widehat{=}$	$P_1;\ ML$	vL_2	$\widehat{=}$	$M - j + 1$
P_3	$\widehat{=}$	$P_2;\ L_1$			

Table 2. Names for programs in path conjuncts and loop variants

$$
\begin{array}{lr}
\tau \fatsemi I \in [\![\, a[1] \le b[1] \,]\!]^T & \mathsf{(A)} \wedge \\
\rho(MLB, vML, \tau \fatsemi P_1) \in [\![\, i \le N \wedge j \le M \,]\!]^T & \mathsf{(B)} \wedge \\
\rho(MLB, vML, \tau \fatsemi P_1) \mathbin{\vdots} (i \le N \wedge j \le M) \in [\![\, a[i] \le b[j] \wedge a[i] \ne c[k-1] \,]\!]^T & \mathsf{(C)} \wedge \\
\left(\begin{array}{l} \rho(MLB, vML, \tau \fatsemi P_1) \\ \quad \mathbin{\vdots} (i \le N \wedge j \le M) \\ \quad \mathbin{\vdots} (a[i] > b[j] \vee a[i] = c[k-1]) \end{array}\right) \in [\![\, a[i] = c[k-1] \,]\!]^T & \mathsf{(D)} \wedge \\
\left(\begin{array}{l} \rho(MLB, vML, \tau \fatsemi P_1) \\ \quad \mathbin{\vdots} (i \le N \wedge j \le M) \\ \quad \mathbin{\vdots} (a[i] > b[j] \vee a[i] = c[k-1]) \\ \quad \mathbin{\vdots} (a[i] \ne c[k-1]) \end{array}\right) \in [\![\, b[j] \ne c[k-1] \,]\!]^T & \mathsf{(E)} \wedge \\
\rho(LB_1, vL_1, \tau \fatsemi P_2) \in [\![\, i \le N \,]\!]^T & \mathsf{(F)} \wedge \\
\rho(LB_1, vL_1, \tau \fatsemi P_2) \mathbin{\vdots} (i \le N) \in [\![\, a[i] \ne c[k-1] \,]\!]^T & \mathsf{(G)} \wedge \\
\rho(LB_2, vL_2, \tau \fatsemi P_3) \in [\![\, j \le M \,]\!]^T & \mathsf{(H)} \wedge \\
\rho(LB_2, vL_2, \tau \fatsemi P_3) \mathbin{\vdots} (j \le M) \in [\![\, b[j] \ne c[k-1] \,]\!]^T & \mathsf{(I)}
\end{array}
$$

Fig. 3. Result of first step

To normalise a compliance predicate, we expand the definition of $[\![\, \_ \,]\!]^T$ to eliminate all its applications to programs, and apply the lift-semi-assoc law, which is defined as $(ts \fatsemi p_1) \fatsemi p_2 = ts \fatsemi (p_1;\ p_2)$ exhaustively. We also eliminate path conjuncts of the form $tsd \in TS$, which are trivially true for any test set definition tsd. The structure of the program and typing information about its expressions guide the rewriting.

For our example, we proceed as shown below. For convenience, we name the programs in the path conjuncts and the variants of the loops as shown in Table 2. Figure 3 gives the normalised predicate in terms of these names. It also names the path conjuncts using letters: (A)-(I).

$$
\begin{array}{ll}
\tau \in [\![\, Merge \,]\!]^T & \\
\equiv \tau \fatsemi I \in [\![\, VBB \,]\!]^T & \text{[semantics of variable blocks]} \\
\equiv \tau \fatsemi I \in [\![\, C_1 \,]\!]^T \wedge (\tau \fatsemi I) \fatsemi C_1 \in [\![\, S_1 \,]\!]^T & \text{[semantics of sequence]} \\
\equiv \quad \tau \fatsemi I \in [\![\, a[1] \le b[1] \,]\!]^T \wedge & \text{[semantics of conditional]} \\
\quad\;\; \tau \fatsemi I \mathbin{\vdots} (a[1] \le b[1]) \in [\![\, c[1], i, j := a[1], 2, 1 \,]\!]^T \wedge & \\
\quad\;\; \tau \fatsemi I \mathbin{\vdots} (a[1] > b[1]) \in [\![\, c[1], i, j := b[1], 1, 2 \,]\!]^T \wedge & \\
\quad\;\; (\tau \fatsemi I) \fatsemi C_1 \in [\![\, S_1 \,]\!]^T & \\
\equiv \quad \tau \fatsemi I \in [\![\, a[1] \le b[1] \,]\!]^T \wedge (\tau \fatsemi I) \fatsemi C_1 \in [\![\, S_1 \,]\!]^T & \text{[semantics of assignment]} \\
\equiv \quad \tau \fatsemi I \in [\![\, a[1] \le b[1] \,]\!]^T \wedge \tau \fatsemi I;\ C_1 \in [\![\, S_1 \,]\!]^T & \text{[}\mathsf{lift\text{-}semi\text{-}assoc}\text{ law]} \\
\equiv \quad \tau \fatsemi I \in [\![\, a[1] \le b[1] \,]\!]^T \wedge \tau \fatsemi I;\ C_1;\ k := 2 \in [\![\, S_2 \,]\!]^T & \\
& \text{[semantics of sequence and assignment, and }\mathsf{lift\text{-}semi\text{-}assoc}\text{ law]}
\end{array}
$$

$$\equiv\ \tau \fatsemi I \in \llbracket\, a[1] \le b[1] \,\rrbracket^{\,T} \wedge \tau \fatsemi I;\ C_1;\ k := 2 \in \llbracket\, ML \rrbracket^{\,T} \wedge \tau \fatsemi I;\ C_1;\ k := 2;\ ML \in \llbracket\, S_3 \rrbracket^{\,T}$$

[semantics of sequence and lift-semi-assoc law]

[semantics of recursion]

$$\begin{aligned}\equiv\ & \tau \fatsemi I \in \llbracket\, a[1] \le b[1] \,\rrbracket^{\,T} \wedge \\ & \rho(MLB, vML, \tau \fatsemi I;\ C_1;\ k := 2) \in \llbracket\, MLB \rrbracket^{\,T} \wedge \tau \fatsemi I;\ C_1;\ k := 2;\ ML \in \llbracket\, S_3 \rrbracket^{\,T}\end{aligned}$$

[semantics of conditional and II]

$$\begin{aligned}\equiv\ & \tau \fatsemi I \in \llbracket\, a[1] \le b[1] \,\rrbracket^{\,T} \wedge \\ & \rho(MLB, vML, \tau \fatsemi I;\ C_1;\ k := 2) \in \llbracket\, i \le N \wedge j \le M \,\rrbracket^{\,T} \wedge \\ & \rho(MLB, vML, \tau \fatsemi I;\ C_1;\ k := 2) \fatsemi (i \le N \wedge j \le M) \in \llbracket\, S_4 \rrbracket^{\,T} \wedge \\ & \tau \fatsemi I;\ C_1;\ k := 2;\ ML \in \llbracket\, S_3 \rrbracket^{\,T}\end{aligned}$$

$$\begin{aligned}\equiv\ & \tau \fatsemi I \in \llbracket\, a[1] \le b[1] \,\rrbracket^{\,T} \wedge \\ & \rho(MLB, vML, \tau \fatsemi I;\ C_1;\ k := 2) \in \llbracket\, i \le N \wedge j \le M \,\rrbracket^{\,T} \wedge \\ & \left(\begin{array}{l} \rho(MLB, vML, \tau \fatsemi I;\ C_1;\ k := 2) \\ \quad \fatsemi (i \le N \wedge j \le M) \end{array}\right) \in \llbracket\, a[i] \le b[j] \wedge a[i] \ne c[k-1] \,\rrbracket^{\,T} \wedge \\ & \left(\begin{array}{l} \rho(MLB, vML, \tau \fatsemi I;\ C_1;\ k := 2) \\ \quad \fatsemi (i \le N \wedge j \le M) \\ \quad \fatsemi (a[i] > b[j] \vee a[i] = c[k-1]) \in \llbracket\, a[i] = c[k-1] \,\rrbracket^{\,T} \end{array}\right) \wedge \\ & \left(\begin{array}{l} \rho(MLB, vML, \tau \fatsemi I;\ C_1;\ k := 2) \\ \quad \fatsemi (i \le N \wedge j \le M) \\ \quad \fatsemi (a[i] > b[j] \vee a[i] = c[k-1]) \\ \quad \fatsemi (a[i] \ne c[k-1]) \end{array}\right) \in \llbracket\, b[j] \ne c[k-1] \,\rrbracket^{\,T} \wedge \\ & \tau \fatsemi I;\ C_1;\ k := 2;\ ML \in \llbracket\, S_3 \rrbracket^{\,T}\end{aligned}$$

[semantics of sequence, conditional, assignment, and recursion variables]

$$\begin{aligned}\equiv\ & \tau \fatsemi I \in \llbracket\, a[1] \le b[1] \,\rrbracket^{\,T} \wedge \\ & \rho(MLB, vML, \tau \fatsemi I;\ C_1;\ k := 2) \in \llbracket\, i \le N \wedge j \le M \,\rrbracket^{\,T} \wedge \\ & \left(\begin{array}{l} \rho(MLB, vML, \tau \fatsemi I;\ C_1;\ k := 2) \\ \quad \fatsemi (i \le N \wedge j \le M) \end{array}\right) \in \llbracket\, a[i] \le b[j] \wedge a[i] \ne c[k-1] \,\rrbracket^{\,T} \wedge \\ & \left(\begin{array}{l} \rho(MLB, vML, \tau \fatsemi I;\ C_1;\ k := 2) \\ \quad \fatsemi (i \le N \wedge j \le M) \\ \quad \fatsemi (a[i] > b[j] \vee a[i] = c[k-1]) \end{array}\right) \in \llbracket\, a[i] = c[k-1] \,\rrbracket^{\,T} \wedge \\ & \left(\begin{array}{l} \rho\,(MLB, vML, \tau \fatsemi I;\ C_1;\ k := 2) \\ \quad \fatsemi (i \le N \wedge j \le M) \\ \quad \fatsemi (a[i] > b[j] \vee a[i] = c[k-1]) \\ \quad \fatsemi (a[i] \ne c[k-1]) \end{array}\right) \in \llbracket\, b[j] \ne c[k-1] \,\rrbracket^{\,T} \wedge \\ & \rho(LB_1, vL_1, \tau \fatsemi I;\ C_1;\ k := 2;\ ML) \in \llbracket\, i \le N \,\rrbracket^{\,T} \wedge \\ & \rho(LB_1, vL_1, \tau \fatsemi I;\ C_1;\ k := 2;\ ML) \fatsemi (i \le N) \in \llbracket\, a[i] \ne c[k-1] \,\rrbracket^{\,T} \wedge \\ & \rho(LB_2, vL_2, \tau \fatsemi I;\ C_1;\ k := 2;\ ML;\ L_1) \in \llbracket\, j \le M \,\rrbracket^{\,T} \wedge \\ & \rho(LB_2, vL_2, \tau \fatsemi I;\ C_1;\ k := 2;\ ML;\ L_1) \fatsemi (j \le M) \in \llbracket\, b[j] \ne c[k-1] \,\rrbracket^{\,T}\end{aligned}$$

[semantics of sequence, recursion, conditional,]
[recursion variables, and assignment, and lift-semi-assoc law]

This example shows that each step of the normalisation is determined by the structure of the program, *Merge*, in this case, and its subprograms, with the law lift-semi-assoc applied exhaustively whenever possible.

3.2. Normalising the programs

In this step, we apply the normalisation procedure in Section 2.2 to the programs in the normalised compliance predicate and to the tests. We, however, leave the recursions (used to write loops) intact.

In our example, I only needs to be extended to an assignment to all variables. For later reference, we call the normalised program $NI \mathrel{\widehat{=}} a, b, c, i, j, k := a, b, c, 0, 0, 0$.

The next program that needs to be normalised is *MLB*; it occurs in the first ρ-expression in the path conjunct (B). Since a recursion variable X is not affected, the normalisation of S_4, and therefore of *MLB*, only changes C_2; we give the steps below. This procedure is standard. Apart from the usual normalisation laws, we use simple conditional laws like $e \lhd d \rhd e = e$, $e_1 \lhd d_1 \rhd (e_1 \lhd d_2 \rhd e_2) = e_1 \lhd d_1 \lor d_2 \rhd e_2$, and $e_1 \lhd d_1 \rhd (e_2 \lhd d_2 \rhd e_1) = e_1 \lhd d_1 \lor \neg\, d_2 \rhd e_2$. The normalised *MLB*, named *NMLB*, is in Figure 4.

$$
\begin{array}{l}
c[k], i, k := a[i], i+1, k+1 \ \lhd a[i] \leq b[j] \land a[i] \neq c[k-1] \rhd \\
\quad \left(\begin{array}{l} i := i+1 \ \lhd a[i] = c[k-1] \rhd \\ \quad (c[k], j, k := b[j], j+1, k+1 \ \lhd b[j] \neq c[k-1] \rhd \ j := j+1) \end{array} \right)
\end{array}
$$

$$
= \begin{array}{l}
a, b, c[k], i, j, k := a, b, a[i], i+1, j, k+1 \ \lhd a[i] \leq b[j] \land a[i] \neq c[k-1] \rhd \\
\left(\begin{array}{l} a, b, c, i, j, k := a, b, c, i+1, j, k \ \lhd a[i] = c[k-1] \rhd \\ \left(\begin{array}{l} a, b, c[k], i, j, k := a, b, b[j], i, j+1, k+1 \ \lhd b[j] \neq c[k-1] \rhd \\ a, b, c, i, j, k := a, b, c, i, j+1, k \end{array} \right) \end{array} \right)
\end{array}
$$

[normalisation of assignments]

$$
= \left(\begin{array}{l} a, \\ b, \\ c[k], \\ i, \\ j, \\ k \end{array} \right) := \left(\begin{array}{l}
a \lhd a[i] \leq b[j] \land a[i] \neq c[k-1] \rhd \\
\quad (a \lhd a[i] = c[k-1] \rhd (a \lhd b[j] \neq c[k-1] \rhd a)), \\
b \lhd a[i] \leq b[j] \land a[i] \neq c[k-1] \rhd \\
\quad (b \lhd a[i] = c[k-1] \rhd (b \lhd b[j] \neq c[k-1] \rhd b)), \\
a[i] \lhd a[i] \leq b[j] \land a[i] \neq c[k-1] \rhd \\
\quad (c[k] \lhd a[i] = c[k-1] \rhd (b[j] \lhd b[j] \neq c[k-1] \rhd c[k])), \\
i+1 \lhd a[i] \leq b[j] \land a[i] \neq c[k-1] \rhd \\
\quad (i+1 \lhd a[i] = c[k-1] \rhd (i \lhd b[j] \neq c[k-1] \rhd i)), \\
j \lhd a[i] \leq b[j] \land a[i] \neq c[k-1] \rhd \\
\quad (j \lhd a[i] = c[k-1] \rhd (j+1 \lhd b[j] \neq c[k-1] \rhd j+1)), \\
k+1 \lhd a[i] \leq b[j] \land a[i] \neq c[k-1] \rhd \\
\quad (k \lhd a[i] = c[k-1] \rhd (k+1 \lhd b[j] \neq c[k-1] \rhd k))
\end{array} \right)
$$

[conditional elimination (three times)]

$$
= \left(\begin{array}{l} a, \\ b, \\ c[k], \\ i, \\ j, \\ k \end{array} \right) := \left(\begin{array}{l}
a, \\
b, \\
a[i] \lhd a[i] \leq b[j] \land a[i] \neq c[k-1] \rhd \\
\quad (c[k] \lhd a[i] = c[k-1] \lor b[j] = c[k-1] \rhd b[j]), \\
i+1 \lhd a[i] \leq b[j] \land a[i] \neq c[k-1] \lor a[i] = c[k-1] \rhd i, \\
j \lhd a[i] \leq b[j] \land a[i] \neq c[k-1] \lor a[i] = c[k-1] \rhd j+1, \\
k+1 \lhd a[i] \leq b[j] \land a[i] \neq c[k-1] \rhd k
\end{array} \right)
$$

[conditional simplification]

The normalisation of P_1, LB_1, and LB_2 is simple; the normalised programs NP_1, NLB_1 and NLB_2 are in Figure 4. In the case of P_2, it involves the loop *ML*, but we do not rewrite loops; so the normalisation of P_2 results in NP_1; $\mu X \mid vML \bullet NMLB$; the situation is similar for P_3, which involves L_1. The final result of this step is a compliance predicate just like that presented in Figure 3 for the previous step, except that the normalised programs presented in Figure 4 are used.

If the particular test set *ts* that is under consideration is not empty, we normalise its tests: since these are assignments, we only need to extend them with trivial assignments to all variables. For our example, we consider the test set containing the five assignments $(a, b := \langle 1 \rangle, \langle 1 \rangle)$, $(a, b := \langle 1 \rangle, \langle 2 \rangle)$, $(a, b := \langle 2, 3 \rangle, \langle 1 \rangle)$, $(a, b := \langle 1, 2 \rangle, \langle 2 \rangle)$, and $(a, b := \langle 1, 3 \rangle, \langle 2, 4 \rangle)$. For example, we obtain the assignment $a, b, c, i, j, k := \langle 1 \rangle, \langle 1 \rangle, c, i, j, k$ when we normalise the first test in our set.

3.3. Calculating the test sets

In the compliance predicate resulting from the second step, there are several occurrences of τ. In this step, we instantiate them to the particular test set *ts* that is under consideration, and eliminate the applications

$$NI \mathrel{\widehat{=}} a, b, c, i, j, k := a, b, c, 0, 0, 0$$

$$NC_2 \mathrel{\widehat{=}} \begin{pmatrix} a, \\ b, \\ c[k], \\ i, \\ j, \\ k \end{pmatrix} := \begin{pmatrix} a, \\ b, \\ a[i] \lhd a[i] \leq b[j] \wedge a[i] \neq c[k-1] \rhd \\ \quad (c[k] \lhd a[i] = c[k-1] \vee b[j] = c[k-1] \rhd b[j]), \\ i+1 \lhd a[i] \leq b[j] \wedge a[i] \neq c[k-1] \vee a[i] = c[k-1] \rhd i, \\ j \lhd a[i] \leq b[j] \wedge a[i] \neq c[k-1] \vee a[i] = c[k-1] \rhd j+1, \\ k+1 \lhd a[i] \leq b[j] \wedge a[i] \neq c[k-1] \rhd k \end{pmatrix}$$

$$NMLB \mathrel{\widehat{=}} NC_2;\ X \ \lhd i \leq N \wedge j \leq M \rhd \ I\!I$$

$$NP_1 \mathrel{\widehat{=}} a, b, c[1], i, j, k := \begin{pmatrix} a, \\ b, \\ a[1] \lhd a[1] \leq b[1] \rhd b[1], \\ 2 \lhd a[1] \leq b[1] \rhd 1, \\ 1 \lhd a[1] \leq b[1] \rhd 2, \\ 2 \end{pmatrix}$$

$$NLB_1 \mathrel{\widehat{=}} \left(\begin{pmatrix} a, \\ b, \\ c[k], \\ i, \\ j, \\ k \end{pmatrix} := \begin{pmatrix} a, \\ b, \\ a[i] \lhd a[i] \neq c[k-1] \rhd c[k], \\ i+1, \\ j, \\ k+1 \lhd a[i] \neq c[k-1] \rhd k \end{pmatrix} \right);\ X \ \lhd i \leq N \rhd \ I\!I$$

$$NP_2 \mathrel{\widehat{=}} NP_1;\ \mu X \mid vML \bullet NMLB$$

$$NLB_2 \mathrel{\widehat{=}} \left(\begin{pmatrix} a, \\ b, \\ c[k], \\ i, \\ j, \\ k \end{pmatrix} := \begin{pmatrix} a, \\ b, \\ b[j] \lhd b[j] \neq c[k-1] \rhd c[k], \\ i, \\ j+1, \\ k+1 \lhd b[j] \neq c[k-1] \rhd k \end{pmatrix} \right);\ X \ \lhd j \leq M \rhd \ I\!I$$

$$NP_3 \mathrel{\widehat{=}} NP_2;\ \mu X \mid vL_1 \bullet NLB_1$$

Fig. 4. Normalised programs in the result of the second step

of $\hat{;}$ and $\hat{:}$, and the ρ-expressions. In the end, we obtain a compliance predicate in which all test set definitions are explicit set enumerations. For that, we exhaustively tackle expressions of the forms $\sigma \mathbin{\hat{;}} p$, $\sigma \mathbin{\hat{:}} d$, and $\rho(F, vrt, \sigma)$, in which σ is already explicitly expressed as a set of tests (like ts). For each of them, we apply the definition of the relevant operator ($\hat{;}$, $\hat{:}$, or ρ), and use normalisation to obtain an explicitly defined set of tests again. In the following sections, we give the details for each operator.

3.3.1. Removing lifted sequence

In this case, we apply the definition, and normalise the programs in the resulting sets. The normalisation, however, now tackles and eliminates the recursion in the usual way.

In our example, the first such term is $ts \mathbin{\hat{;}} NI$ in the conjunct (A). Its evaluation is too simple; a more interesting example is the evaluation of $ts \mathbin{\hat{;}} NP_1$, which occurs in the conjuncts ((B)-(E)). It shows how the definition of particular values for variables in the tests affords significant simplification of the programs in the compliance predicate. In our example, we obtain assignments that characterise tests directly.

$ts \fatsemi NP1$

$$= \left\{ \begin{array}{l} a,b,c,i,j,k := \langle 1\rangle, \langle 1\rangle, c, i, j, k, \\ a,b,c,i,j,k := \langle 1\rangle, \langle 2\rangle, c, i, j, k, \\ a,b,c,i,j,k := \langle 2,3\rangle, \langle 1\rangle, c, i, j, k, \\ a,b,c,i,j,k := \langle 1,2\rangle, \langle 2\rangle, c, i, j, k, \\ a,b,c,i,j,k := \langle 1,3\rangle, \langle 2,4\rangle, c, i, j, k \end{array} \right\} \fatsemi \left(\begin{array}{l} a, \\ b, \\ c[1], \\ i, \\ j, \\ k \end{array} \right) := \left(\begin{array}{l} a, \\ b, \\ a[1] \lhd a[1] \leq b[1] \rhd b[1], \\ 2 \lhd a[1] \leq b[1] \rhd 1, \\ 1 \lhd a[1] \leq b[1] \rhd 2, \\ 2 \end{array} \right) \qquad \text{[instantiation of } ts\text{]}$$

$$= \left\{ \begin{array}{l} (a,b,c,i,j,k := \langle 1\rangle, \langle 1\rangle, c, i, j, k); \left(\begin{array}{l} a, \\ b, \\ c[1], \\ i, \\ j, \\ k \end{array} \right) := \left(\begin{array}{l} a, \\ b, \\ a[1] \lhd a[1] \leq b[1] \rhd b[1], \\ 2 \lhd a[1] \leq b[1] \rhd 1, \\ 1 \lhd a[1] \leq b[1] \rhd 2, \\ 2 \end{array} \right), \\ \ldots \\ (a,b,c,i,j,k := \langle 1,3\rangle, \langle 2,4\rangle, c, i, j, k); \left(\begin{array}{l} a, \\ b, \\ c[1], \\ i, \\ j, \\ k \end{array} \right) := \left(\begin{array}{l} a, \\ b, \\ a[1] \lhd a[1] \leq b[1] \rhd b[1], \\ 2 \lhd a[1] \leq b[1] \rhd 1, \\ 1 \lhd a[1] \leq b[1] \rhd 2, \\ 2 \end{array} \right) \end{array} \right\} \qquad \text{[definition of } \fatsemi\text{]}$$

$$= \left\{ \begin{array}{l} a, b, c[1], i, j, k := \left(\begin{array}{l} \langle 1\rangle, \\ \langle 1\rangle, \\ \langle 1\rangle[1] \lhd \langle 1\rangle[1] \leq \langle 1\rangle[1] \rhd \langle 1\rangle[1], \\ 2 \lhd \langle 1\rangle[1] \leq \langle 1\rangle[1] \rhd 1, \\ 1 \lhd \langle 1\rangle[1] \leq \langle 1\rangle[1] \rhd 2, \\ 2 \end{array} \right), \\ \ldots \\ a, b, c[1], i, j, k := \left(\begin{array}{l} \langle 1,3\rangle, \\ \langle 2,4\rangle, \\ \langle 1,3\rangle[1] \lhd \langle 1,3\rangle[1] \leq \langle 2,4\rangle[1] \rhd \langle 2,4\rangle[1], \\ 2 \lhd \langle 1,3\rangle[1] \leq \langle 2,4\rangle[1] \rhd 1, \\ 1 \lhd \langle 1,3\rangle[1] \leq \langle 2,4\rangle[1] \rhd 2, \\ 2 \end{array} \right) \end{array} \right\} \qquad \text{[normalisation]}$$

$$= \left\{ \begin{array}{l} a, b, c[1], i, j, k := \langle 1\rangle, \langle 1\rangle, 1, 2, 1, 2, \\ a, b, c[1], i, j, k := \langle 1\rangle, \langle 2\rangle, 1, 2, 1, 2, \\ a, b, c[1], i, j, k := \langle 2,3\rangle, \langle 1\rangle, 1, 1, 2, 2, \\ a, b, c[1], i, j, k := \langle 1,2\rangle, \langle 2\rangle, 1, 2, 1, 2, \\ a, b, c[1], i, j, k := \langle 1,3\rangle, \langle 2,4\rangle, 1, 2, 1, 2 \end{array} \right\} \qquad \text{[properties of sequences and propositional calculus]}$$

If a recursion is involved, we need to unfold it using the fixed-point law: $(\mu X \bullet F(X)) = F(\mu X \bullet F(X))$. This is required in the evaluation of $ts \fatsemi NP_2$, since NP_2 includes a loop after NP_1. It occurs in four path conjuncts (B)-(E), and includes all the tests that are carried out for the body of ML given the tests in the original ts. Below, we give the first few steps of the elimination of $ts \fatsemi NP_2$. We use the law lift-semi-assoc (that is, $(ts \fatsemi p_1) \fatsemi p_2 = ts \fatsemi (p_1;\ p_2)$) to allow the reuse of the result for $ts \fatsemi NP_1$. This is a simple optimisation.

$ts \fatsemi NP_2$

$= ts \fatsemi NP_1;\ \mu X \bullet NMLB$ [definition of NP_2]

$= (ts \fatsemi NP_1) \fatsemi \mu X \bullet NMLB$ [lift-semi-assoc law]

$$= \left\{\begin{array}{l} a, b, c[1], i, j, k := \langle 1\rangle, \langle 1\rangle, 1, 2, 1, 2, \\ a, b, c[1], i, j, k := \langle 1\rangle, \langle 2\rangle, 1, 2, 1, 2, \\ a, b, c[1], i, j, k := \langle 2, 3\rangle, \langle 1\rangle, 1, 1, 2, 2, \\ a, b, c[1], i, j, k := \langle 1, 2\rangle, \langle 2\rangle, 1, 2, 1, 2, \\ a, b, c[1], i, j, k := \langle 1, 3\rangle, \langle 2, 4\rangle, 1, 2, 1, 2 \end{array}\right\} \hat{;}\ \mu X \bullet \mathit{NMLB} \qquad \text{[previous result]}$$

$$= \left\{\begin{array}{l} (a, b, c[1], i, j, k := \langle 1\rangle, \langle 1\rangle, 1, 2, 1, 2);\ \mu X \bullet \mathit{NMLB}, \\ (a, b, c[1], i, j, k := \langle 1\rangle, \langle 2\rangle, 1, 2, 1, 2);\ \mu X \bullet \mathit{NMLB}, \\ (a, b, c[1], i, j, k := \langle 2, 3\rangle, \langle 1\rangle, 1, 1, 2, 2);\ \mu X \bullet \mathit{NMLB}, \\ (a, b, c[1], i, j, k := \langle 1, 2\rangle, \langle 2\rangle, 1, 2, 1, 2);\ \mu X \bullet \mathit{NMLB}, \\ (a, b, c[1], i, j, k := \langle 1, 3\rangle, \langle 2, 4\rangle, 1, 2, 1, 2);\ \mu X \bullet \mathit{NMLB} \end{array}\right\} \qquad \text{[definition of } \hat{;}\text{]}$$

We now present the evaluation of the loop $\mu X \bullet \mathit{NMLB}$ for the fourth test listed above, which is obtained after the execution of P_1 (or NP_1); we omit the variant, since it is not used. We use a simple law called assumption-intr: $(x := e;\ p) = (x := e;\ (x = e \wedge p))$, provided x is not free in e. It relies on a program being a predicate: it uses a conjunction to combine an equality with p. Using this law, it is possible to take advantage of information from the test in the evaluation of the body of the recursion. For recursions that encode a loop, this strategy is always applicable.

$$(a, b, c[1], i, j, k := \langle 1, 2\rangle, \langle 2\rangle, 1, 2, 1, 2);\ \mu X \bullet \mathit{NMLB}$$

$$= (a, b, c[1], i, j, k := \langle 1, 2\rangle, \langle 2\rangle, 1, 2, 1, 2);\ \mathit{NMLB}(\mu X \bullet \mathit{NMLB}) \qquad \text{[fixed-point law]}$$

$$\begin{array}{l} = (a, b, c[1], i, j, k := \langle 1, 2\rangle, \langle 2\rangle, 1, 2, 1, 2); \\ \quad (NC_2;\ (\mu X \bullet \mathit{NMLB}) \lhd i \leq N \wedge j \leq M \rhd I\!I) \end{array} \qquad \text{[definition of } \mathit{NMLB}\text{]}$$

$$\begin{array}{l} = (a, b, c[1], i, j, k := \langle 1, 2\rangle, \langle 2\rangle, 1, 2, 1, 2); \\ \quad i = 2 \wedge j = 1 \wedge (NC_2;\ (\mu X \bullet \mathit{NMLB}) \lhd i \leq N \wedge j \leq M \rhd I\!I) \end{array} \qquad \text{[assumption-intr law]}$$

$$\begin{array}{l} = (a, b, c[1], i, j, k := \langle 1, 2\rangle, \langle 2\rangle, 1, 2, 1, 2); \\ \quad i = 2 \wedge j = 1 \wedge (NC_2;\ (\mu X \bullet \mathit{NMLB}) \lhd \mathit{true} \rhd I\!I) \end{array} \qquad \text{[propositional calculus, } N = 2\text{, and } M = 1\text{]}$$

$$= (a, b, c[1], i, j, k := \langle 1, 2\rangle, \langle 2\rangle, 1, 2, 1, 2);\ NC_2;\ (\mu X \bullet \mathit{NMLB})$$

$$\text{[}p_1 \lhd \mathit{true} \rhd p_2 = p_1\text{, and assumption-intr law]}$$

$$= (a, b, c[1, 2], i, j, k := \langle 1, 2\rangle, \langle 2\rangle, (1, 2), 3, 1, 3);\ (\mu X \bullet \mathit{NMLB}) \qquad \text{[normalisation: sequence elimination]}$$

$$\begin{array}{l} = (a, b, c[1, 2], i, j, k := \langle 1, 2\rangle, \langle 2\rangle, (1, 2), 3, 1, 3); \\ \quad i = 3 \wedge j = 1 \wedge (NC_2;\ (\mu X \bullet \mathit{NMLB}) \lhd \mathit{false} \rhd I\!I) \end{array} \qquad \text{[fixed-point law, } \mathit{NMLB}\text{, and assumption-intr law]}$$

$$= \boxed{a, b, c[1, 2], i, j, k := \langle 1, 2\rangle, \langle 2\rangle, (1, 2), 3, 1, 3} \qquad \text{[}p_1 \lhd \mathit{false} \rhd p_2 = p_2\text{, assumption-intr law, and } p;\ I\!I = p\text{]}$$

As shown above, we introduce an assumption that gives the values of the variables in the loop decision, and evaluate the conditional. If the body of the loop is unfolded, we normalise the result, and evaluate the recursion again. If the result of the evaluation of the conditional is $I\!I$, we use the law $p;\ I\!I = p$ to conclude the evaluation. The result above gives the values of the variables after the execution of P_2 for the input given by the fourth test case in the original ts: a has been copied across to c; and b is yet to be considered.

For our example, we can eliminate all occurrences of $\hat{;}$ straightaway; in this way, the last arguments of all ρ-expressions in the compliance predicate are defined explicitly.

3.3.2. Removing ρ-expressions

We use the definition of ρ and normalisation to calculate the tests. This corresponds to calculating the tests that are carried out for loop bodies. As explained in Section 2.3, a single test may lead to the execution of a loop body several times, and so to several tests of its decisions.

If the last argument ts of a ρ-expression is explicitly given as a set of tests, we can write the ρ-expression as a union of sets of tests; there is a set for each test t in ts. This follows from the simple property of set comprehensions, which is captured by the law set-comp-range-dist below.

$$\{tc : \{t_1, \ldots, t_n\};\ n' : T \mid \phi(tc) \bullet e(tc)\} = \{n' : T \mid \phi(t_1) \bullet e(t_1)\} \cup \ldots \cup \{n' : T \mid \phi(t_n) \bullet e(t_n)\}$$

In this law, we consider a set comprehension over a set tc and some other variable (or variables) n', where the declaration of tc enumerates its possible values $t_1, \ldots, t_n$, with a constraint $\phi(tc)$ and a term $e(tc)$ where tc occurs free. The law establishes that the set comprehension can be unfolded to a union where each value of tc is considered individually. As an example, we present below the first few steps of the elimination of $\rho(NMLB, vML, ts \mathbin{⨟} NP_1)$, which appears in the path conjuncts (B)-(E).

$$\rho(NMLB, vML, ts \mathbin{⨟} NP_1)$$

$$= \{\, tc : ts \mathbin{⨟} NP_1;\ n' : \mathbb{N} \mid (tc;\ n' \leq vML) \bullet tc;\ NMLB^{n'}(\mathit{I\!I}) \,\} \qquad [\text{definition of } \rho]$$

$$= \left\{ \begin{array}{l} n' : \mathbb{N} \mid ((a, b, c[1], i, j, k := \langle 1\rangle, \langle 1\rangle, 1, 2, 1, 2);\ n' \leq vML) \bullet \\ \quad (a, b, c[1], i, j, k := \langle 1\rangle, \langle 1\rangle, 1, 2, 1, 2);\ NMLB^{n'}(\mathit{I\!I}) \end{array} \right\} \cup \qquad [ts \mathbin{⨟} NP_1 \text{ and law set-comp-range-dist}]$$
$$\left\{ \begin{array}{l} n' : \mathbb{N} \mid ((a, b, c[1], i, j, k := \langle 1\rangle, \langle 2\rangle, 1, 2, 1, 2);\ n' \leq vML) \bullet \\ \quad (a, b, c[1], i, j, k := \langle 1\rangle, \langle 2\rangle, 1, 2, 1, 2);\ NMLB^{n'}(\mathit{I\!I}) \end{array} \right\} \cup$$
$$\left\{ \begin{array}{l} n' : \mathbb{N} \mid ((a, b, c[1], i, j, k := \langle 2, 3\rangle, \langle 1\rangle, 1, 1, 2, 2);\ n' \leq vML) \bullet \\ \quad (a, b, c[1], i, j, k := \langle 2, 3\rangle, \langle 1\rangle, 1, 1, 2, 2);\ NMLB^{n'}(\mathit{I\!I}) \end{array} \right\} \cup$$
$$\left\{ \begin{array}{l} n' : \mathbb{N} \mid ((a, b, c[1], i, j, k := \langle 1, 2\rangle, \langle 2\rangle, 1, 2, 1, 2);\ n' \leq vML) \bullet \\ \quad (a, b, c[1], i, j, k := \langle 1, 2\rangle, \langle 2\rangle, 1, 2, 1, 2);\ NMLB^{n'}(\mathit{I\!I}) \end{array} \right\} \cup$$
$$\left\{ \begin{array}{l} n' : \mathbb{N} \mid ((a, b, c[1], i, j, k := \langle 1, 3\rangle, \langle 2, 4\rangle, 1, 2, 1, 2);\ n' \leq vML) \bullet \\ \quad (a, b, c[1], i, j, k := \langle 1, 3\rangle, \langle 2, 4\rangle, 1, 2, 1, 2);\ NMLB^{n'}(\mathit{I\!I}) \end{array} \right\}$$

To proceed, in each of the set comprehensions, we evaluate the value of n' for the relevant test. This uses the law test-expr-eval defined as $(v := e_1);\ e_2(n', v) = e_2(n', e_1)$, where v is the list of undashed variables in the alphabet, and provided v are not free in e_1 and n' is not in the alphabet.

$$\begin{array}{ll} = & \{\, n' : \mathbb{N} \mid n' \leq 1 \bullet (a, b, c[1], i, j, k := \langle 1\rangle, \langle 1\rangle, 1, 2, 1, 2);\ NMLB^{n'}(\mathit{I\!I}) \,\} \cup \qquad [\text{law test-expr-eval}] \\ & \{\, n' : \mathbb{N} \mid n' \leq 1 \bullet (a, b, c[1], i, j, k := \langle 1\rangle, \langle 2\rangle, 1, 2, 1, 2);\ NMLB^{n'}(\mathit{I\!I}) \,\} \cup \\ & \{\, n' : \mathbb{N} \mid n' \leq 2 \bullet (a, b, c[1], i, j, k := \langle 2, 3\rangle, \langle 1\rangle, 1, 1, 2, 2);\ NMLB^{n'}(\mathit{I\!I}) \,\} \cup \\ & \{\, n' : \mathbb{N} \mid n' \leq 2 \bullet (a, b, c[1], i, j, k := \langle 1, 2\rangle, \langle 2\rangle, 1, 2, 1, 2);\ NMLB^{n'}(\mathit{I\!I}) \,\} \cup \\ & \{\, n' : \mathbb{N} \mid n' \leq 3 \bullet (a, b, c[1], i, j, k := \langle 1, 3\rangle, \langle 2, 4\rangle, 1, 2, 1, 2);\ NMLB^{n'}(\mathit{I\!I}) \,\} \end{array}$$

Each set in the union includes the test t itself and all tests carried out as a consequence of the recursive calls generated by executing the program when the variables have the values defined by t. By calculating the value n' of the variant for t, we determine the maximum number of recursive calls. To proceed, we evaluate the set comprehensions to obtain set enumerations using a specialisation of the law set-comp-range-dist presented above. We call the new law set-comp-enum; it is defined as $\{n' : \mathbb{N} \mid n' \leq l \bullet e(n')\} = \{e(0), \ldots, e(l)\}$.

To finalise the calculation of the tests, we normalise each of the programs $tc;\ F^{n'}(\mathit{I\!I})$. We proceed as we did for recursive programs in the previous section, but instead of the fixed-point law, we use the definition of $F^n(X)$. Below, we give the calculation of the set that corresponds to the fourth test in $ts \mathbin{⨟} NP_1$.

$$\{\, n' : \mathbb{N} \mid n' \leq 2 \bullet (a, b, c[1], i, j, k := \langle 1, 2\rangle, \langle 2\rangle, 1, 2, 1, 2);\ NMLB^{n'}(\mathit{I\!I}) \,\}$$

$$\begin{array}{ll} = \{ & (a, b, c[1], i, j, k := \langle 1, 2\rangle, \langle 2\rangle, 1, 2, 1, 2);\ NMLB^{0}(\mathit{I\!I}), \qquad [\text{law set-comp-enum}] \\ & (a, b, c[1], i, j, k := \langle 1, 2\rangle, \langle 2\rangle, 1, 2, 1, 2);\ NMLB^{1}(\mathit{I\!I}), \\ & (a, b, c[1], i, j, k := \langle 1, 2\rangle, \langle 2\rangle, 1, 2, 1, 2);\ NMLB^{2}(\mathit{I\!I}) \,\} \end{array}$$

$$\begin{array}{ll} = \{ & (a, b, c[1], i, j, k := \langle 1, 2\rangle, \langle 2\rangle, 1, 2, 1, 2), \qquad [F^0(\mathit{I\!I}) = \mathit{I\!I} \text{ and } p;\ \mathit{I\!I} = p] \\ & (a, b, c[1], i, j, k := \langle 1, 2\rangle, \langle 2\rangle, 1, 2, 1, 2);\ NMLB^{1}(\mathit{I\!I}), \\ & (a, b, c[1], i, j, k := \langle 1, 2\rangle, \langle 2\rangle, 1, 2, 1, 2);\ NMLB^{2}(\mathit{I\!I}) \,\} \end{array}$$

$$\left\{\begin{array}{l} a, b, c[1], i, j, k := \langle 1\rangle, \langle 1\rangle, 1, 2, 1, 2, \\ a, b, c[1], i, j, k := \langle 1\rangle, \langle 2\rangle, 1, 2, 1, 2, \\ a, b, c[1], i, j, k := \langle 2, 3\rangle, \langle 1\rangle, 1, 1, 2, 2, \\ a, b, c[1], i, j, k := \langle 1, 2\rangle, \langle 2\rangle, 1, 2, 1, 2, \\ a, b, c[1, 2], i, j, k := \langle 1, 2\rangle, \langle 2\rangle, (1, 2), 3, 1, 3, \\ a, b, c[1], i, j, k := \langle 1, 3\rangle, \langle 2, 4\rangle, 1, 2, 1, 2, \\ a, b, c[1, 2], i, j, k := \langle 1, 3\rangle, \langle 2, 4\rangle, (1, 2), 2, 2, 3, \\ a, b, c[1, 2, 3], i, j, k := \langle 1, 3\rangle, \langle 2, 4\rangle, (1, 2, 3), 3, 2, 4 \end{array}\right\}$$

Fig. 5. Tests in $\rho(NMLB, vML, ts \,⨟\, NP_1)$

$$\begin{array}{ll} = \{ & (a, b, c[1], i, j, k := \langle 1, 2\rangle, \langle 2\rangle, 1, 2, 1, 2), \qquad [NMLB,\ F^0(\mathit{II}) = \mathit{II}, \text{ and } p;\ \mathit{II} = p] \\ & (a, b, c[1], i, j, k := \langle 1, 2\rangle, \langle 2\rangle, 1, 2, 1, 2);\ NMLB^2(\mathit{II}), \\ & (a, b, c[1], i, j, k := \langle 1, 2\rangle, \langle 2\rangle, 1, 2, 1, 2);\ (NC_2 \ \lhd i \leq N \wedge j \leq M \rhd\ \mathit{II}) \} \end{array}$$

$$\begin{array}{ll} = \{ & (a, b, c[1], i, j, k := \langle 1, 2\rangle, \langle 2\rangle, 1, 2, 1, 2), \\ & (a, b, c[1], i, j, k := \langle 1, 2\rangle, \langle 2\rangle, 1, 2, 1, 2);\ NMLB^2(\mathit{II}), \\ & (a, b, c[1], i, j, k := \langle 1, 2\rangle, \langle 2\rangle, 1, 2, 1, 2);\ NC_2 \} \end{array}$$

[assumption-intr law, propositional calculus, $p_1 \lhd true \rhd p_2 = p_1$]

$$\begin{array}{ll} = \{ & (a, b, c[1], i, j, k := \langle 1, 2\rangle, \langle 2\rangle, 1, 2, 1, 2), \qquad [\text{normalisation}] \\ & (a, b, c[1], i, j, k := \langle 1, 2\rangle, \langle 2\rangle, 1, 2, 1, 2);\ NMLB^2(\mathit{II}), \\ & (a, b, c[1, 2], i, j, k := \langle 1, 2\rangle, \langle 2\rangle, (1, 2), 3, 1, 3) \} \end{array}$$

$$\begin{array}{ll} = \{ & (a, b, c[1], i, j, k := \langle 1, 2\rangle, \langle 2\rangle, 1, 2, 1, 2), \qquad [F^{n+1}(X) = F(F^n(X)) \text{ and } NMLB] \\ & (a, b, c[1, 2], i, j, k := \langle 1, 2\rangle, \langle 2\rangle, (1, 2), 3, 1, 3), \\ & ((a, b, c[1], i, j, k := \langle 1, 2\rangle, \langle 2\rangle, 1, 2, 1, 2);\ (NC_2;\ NMLB(\mathit{II})\ \lhd i \leq N \wedge j \leq M \rhd\ \mathit{II})) \} \end{array}$$

$$\begin{array}{ll} = \{ & (a, b, c[1], i, j, k := \langle 1, 2\rangle, \langle 2\rangle, 1, 2, 1, 2), \\ & (a, b, c[1, 2], i, j, k := \langle 1, 2\rangle, \langle 2\rangle, (1, 2), 3, 1, 3), \\ & (a, b, c[1], i, j, k := \langle 1, 2\rangle, \langle 2\rangle, 1, 2, 1, 2);\ NC_2;\ NMLB(\mathit{II}) \} \end{array}$$

[assumption-intr law, propositional calculus, $p_1 \lhd true \rhd p_2 = p_1$]

$$\begin{array}{ll} = \{ & (a, b, c[1], i, j, k := \langle 1, 2\rangle, \langle 2\rangle, 1, 2, 1, 2), \qquad [\text{normalisation}] \\ & (a, b, c[1, 2], i, j, k := \langle 1, 2\rangle, \langle 2\rangle, (1, 2), 3, 1, 3), \\ & (a, b, c[1, 2], i, j, k := \langle 1, 2\rangle, \langle 2\rangle, (1, 2), 3, 1, 3);\ NMLB(\mathit{II}) \} \end{array}$$

$$\begin{array}{ll} = \{ & (a, b, c[1], i, j, k := \langle 1, 2\rangle, \langle 2\rangle, 1, 2, 1, 2), \qquad [NMLB \text{ and normalisation}] \\ & (a, b, c[1, 2], i, j, k := \langle 1, 2\rangle, \langle 2\rangle, (1, 2), 3, 1, 3) \} \end{array}$$

It can be observed that the test generated by the last execution of the loop body is the same calculated in the evaluation of $ts \,⨟\, NP_2$ for the same input test. The reason for this is that the ρ-expression includes all tests used for the loop body MLB, and the test that leads to termination is used for the program S_3 that follows the loop. The set $ts \,⨟\, NP_2$, on the other hand, includes exactly all the tests used for S_3. Figure 5 gives the complete set of tests generated by the evaluation of $\rho(NMLB, vML, ts \,⨟\, NP_1)$.

3.3.3. Removing the filters

Returning to Figure 3, we observe that in the path conjuncts several filters are (consecutively) applied to $\rho(NMLB, vML, ts \,⨟\, NP_1)$, whose value is shown in Figure 5. To remove them, first we apply the definition of the filter operator; this results in a set comprehension ranging over tests t in set given by an explicit enumeration, and whose constraint is of the form $t;\ d$, for a decision d. We next apply another specialisation of law seq-comp-range-dist to this set comprehension: law seq-comp-test-const defined as $\{t : \{t_1, \ldots, t_n\} \mid \phi(t)\} = \{i : I \bullet t_i\}$ where for every i such that $\phi(t_i)$ holds, $i \in I$. To determine whether $t;\ d$ holds, we use law test-expr-eval. Since d is a decision of the program, this evaluation is decidable. In accordance with law seq-comp-test-const, only those tests that satisfy this constraint are kept.

In our example, the application of the filter with decision $i \leq N \wedge j \leq M$ in the conjunct (C), for instance, removes all but three of the tests: the fourth, the sixth, and the seventh in Figure 5; they lead to the execution of the loop body, rather than to straight termination. These are the tests that may exercise the decisions in the body of the loop. An extra filter is applied to this set in the path conjunct (D); the decision $a[i] > b[j] \lor a[i] = c[k-1]$ is the negation of the outer decision in C_2. This filter characterises the set of tests that exercise the else part of C_2. Its only test is $a, b, c[1], i, j, k := \langle 1, 3\rangle, \langle 2, 4\rangle, 1, 2, 1, 2$, so we can conclude that its decisions are not covered, but this is the concern of the next step. The final filter $a[i] \neq c[k-1]$, in the conjunct (E), does not remove this test.

Due to the structure of *Merge*, and the consequent form of the compliance predicate, there is no need to iterate to consider again any of the operators $\hat{;}$, ρ, $\hat{:}$. If we consider them in this order, we can eliminate them all straightaway. In general, however, as emphasised in Figure 1, this third step of our procedure is an iterative step that tackles these operators exhaustively, and in any order.

3.4. Checking compliance for decisions

Since all test set definitions in the path conjuncts are now expressed as set enumerations, it is feasible to check them using tools like [IPL, LDR] that check for the appropriate coverage criterion. In this step, we iteratively tackle one path conjunct at a time. If they are all true, the original test set is compliant.

If, on the other hand, we find a path conjunct $tsd \in [\![d]\!]^T$ that does not hold, then we have a decision d that is not covered. We need to calculate extra tests to cover d. We proceed as follows.

The decision d is reached along the path characterised by tsd. We can describe this path as a program $path(tsd)$ defined as shown below; mv and en stand for arbitrary metavariables and set enumerations.

Definition 3.3 (Path programs).

$path(mv) = I\!I$
$path(en) = I\!I$
$path(btsd \mathbin{\hat{;}} p) = path(btsd);\ p$
$path(\rho(F, vrt, tsd)) = path(tsd);\ \bigsqcup n \leq vrt \bullet F^n(I\!I)$
$path(tsd \mathbin{\hat{:}} d) = path(tsd);\ d_\perp$

The path for a lifted sequence $btsd \mathbin{\hat{;}} p$ is that of $btsd$ followed by the program p. The path for a ρ-expression $\rho(F, vrt, tsd)$ is that of tsd, which leads to the recursion, followed by a choice ($\bigsqcup$) over a number n of iterations of the recursion. In the UTP, choice is defined by disjunction. The number of iterations is limited by the variant vrt. For filter expressions $tsd \mathbin{\hat{:}} d$, the decision is turned into an assumption. For a decision d, the assumption $d_\perp$ is the program that skips if d holds, but aborts otherwise. In the UTP, we have $d_\perp \mathrel{\widehat=} I\!I \lhd d \rhd true$. Abort is characterised just by $true$; it is the program that can exhibit any behaviour.

The problem posed by a conjunct $tsd \in [\![d]\!]^T$ that does not hold can be characterised by a truth table that gives the missing assignments of truth values to the conditions of d that are required for coverage, and $path(tsd)$. For each row in the table, we need inputs that, when modified by the program $path(tsd)$, result in the conditions being evaluated as required in the table row.

For our example, we first tackle the conjunct (A); its decision is covered by the tests in $ts \mathbin{\hat{;}} I$ that originate from the tests $(a, b := \langle 1\rangle, \langle 1\rangle)$ and $(a, b := \langle 2, 3\rangle, \langle 1\rangle)$ in ts. Indeed, the calculations in the previous steps reveal that, along the path defined by $ts \mathbin{\hat{;}} I$, these tests become $(a, b, c, i, j, k := \langle 1\rangle, \langle 1\rangle, c, 0, 0, 0)$ and $(a, b, c, i, j, k := \langle 2, 3\rangle, \langle 1\rangle, c, 0, 0, 0)$, which do provide the required coverage. Afterwards, we consider (B); the decision is covered by the first, the second, and the fourth tests in Figure 5.

The decision in the conjunct (C), that is, $a[i] \leq b[j] \land a[i] \neq c[k-1]$ is considered next. It is not covered by the three tests under consideration: the fourth, the sixth, and the seventh in Figure 5, which remained after the application of the filter with $i \leq N \land j \leq M$. What we are missing is a test that drives the program so that the condition $a[i] \leq b[j]$ evaluates to *true*, but $a[i] \neq c[k-1]$ evaluates to *false*. This kind of information can be provided by existing testing tools [IPL, LDR]. What current tools cannot do, however, is to support the definition of a test that solves the problem. In our technique, we report the problem with the one-line truth table in Figure 6, and with the path below, which we name PP_1.

$NP_1;\ \bigsqcup n \leq vML \bullet NMLB^n(I\!I);\ (i \leq N \land j \leq M)_\perp$

For a given conjunct $tsd \in [\![d]\!]^T$ that does not hold, we call the corresponding truth table a target evaluation

$a[i] \leq b[j]$	$a[i] \neq c[k-1]$
true	*false*

Fig. 6. Partial specification of a coverage problem

for d. They are assignments of truth values $b_1, \ldots, b_n$ to the conditions $c_1, \ldots, c_n$ of d. With these target evaluations, we proceed to calculate the extra tests required.

3.4.1. Enriching the original test set

We consider in turn each target evaluation $(b_1, \ldots, b_n)$ for d, and the associated program p that characterises the path to the decision, that is $\mathit{path}(\mathit{tsd})$. There may be many or even an infinite number of satisfactory tests: input values that give rise to the required valuation of the conditions.

These input values can be precisely characterised using weakest preconditions [Dij76]. They are values that satisfy the weakest precondition for p to establish the required valuation of conditions. We call this the target predicate of $(b_1, \ldots, b_n)$ for d defined as $\mathit{tpred}((b_1, \ldots, b_n), d) \mathrel{\widehat{=}} \bigwedge i \bullet \mathit{def}(c_i, b_i)$, where the c_i are the conditions of d, and $\mathit{def}(c, \mathit{true}) = c$ and $\mathit{def}(c, \mathit{false}) = \neg\, c$.

We write the weakest precondition for a program p to establish a postcondition ϕ as $\mathit{wp}.p.\phi$; the function wp is a predicate transformer, and its definition for our language is standard [Dij76]. Reduction of applications of wp can be achieved with direct use of its definition, except only for the case of iterated recursion bodies $F^n(X)$ and recursions $(\mu X \bullet F(X))$; we provide a procedure to tackle these in the sequel.

With this result, a constraint solver can be used to generate concrete test cases. Even if a constraint solver cannot provide a result, we are still left with a formal characterisation of the required test, and theorem proving can be used to establish coverage of a test generated in some other way.

Iterations For the missing test in our example, we present the weakest precondition calculation below. As in the UTP, we use dashes to distinguish references to the final values of the variables from references to their initial values. A definition for wp in terms of this convention is found in [CW99]. For brevity, we name the condition $a'[i'] \leq b'[j'] \wedge a'[i'] = c'[k'-1]$ as ϕ.

$$\mathit{wp}.\mathit{PP}_1.\phi$$

$$= \left(\begin{array}{l} a[1] \leq b[1] \Rightarrow \\ \quad (\mathit{wp}.(\bigsqcup n \leq \mathit{vML} \bullet \mathit{NMLB}^n(\mathit{II});\ (i \leq N \wedge j \leq M)_\perp).\phi)'[a, b, a[1], 2, 1, 2/a', b', c'[1], i', j', k'] \end{array}\right) \wedge$$
$$\left(\begin{array}{l} a[1] > b[1] \Rightarrow \\ \quad (\mathit{wp}.(\bigsqcup n \leq \mathit{vML} \bullet \mathit{NMLB}^n(\mathit{II});\ (i \leq N \wedge j \leq M)_\perp).\phi)'[a, b, b[1], 1, 2, 2/a', b', c'[1], i', j', k'] \end{array}\right)$$

[wp for sequences and conditionals]

$$= \left(\begin{array}{l} a[1] \leq b[1] \Rightarrow \\ \quad (\exists\, n \leq \mathit{vML} \bullet \mathit{wp}.\mathit{NMLB}^n(\mathit{II}).(i' \leq N \wedge j' \leq M \wedge \phi))'[a, b, a[1], 2, 1, 2/a', b', c'[1], i', j', k'] \end{array}\right) \wedge$$
$$\left(\begin{array}{l} a[1] > b[1] \Rightarrow \\ \quad (\exists\, n \leq \mathit{vML} \bullet \mathit{wp}.\mathit{NMLB}^n(\mathit{II}).(i' \leq N \wedge j' \leq M \wedge \phi))'[a, b, b[1], 1, 2, 2/a', b', c'[1], i', j', k'] \end{array}\right)$$

[wp for choice, sequence, and assumptions]

$$\begin{array}{ll} = & (\exists\, n \leq N + M - 1 \bullet a[1] \leq b[1] \Rightarrow (\mathit{wp}.\mathit{NMLB}^n(\mathit{II}).(i' \leq N \wedge j' \leq M \wedge \phi))[a[1], 2, 1, 2/c[1], i, j, k]) \wedge \\ & (\exists\, n \leq N + M - 1 \bullet a[1] > b[1] \Rightarrow (\mathit{wp}.\mathit{NMLB}^n(\mathit{II}).(i' \leq N \wedge j' \leq M \wedge \phi))[b[1], 1, 2, 2/c[1], i, j, k]) \end{array}$$

[predicate calculus and substitution]

The existential quantification reflects the possibility of achieving coverage with any number of iterations of the loop. The calculations capture the fact that we are required to find inputs such that, after some number n of iterations, if $a[1] \leq b[1]$, then $i \leq N \wedge j \leq M$ holds, so that we reach the body of the loop, and then both $a[i] \leq b[j]$ and $a[i] = c[k-1]$ hold, if the initial values of $c[1]$, i, j, and k are $a[1]$, 2, 1, and 2. And similarly for when $a[1] > b[1]$, but with different initial values for $c[1]$, i, j, and k.

To handle the existential quantifications, or more generally, the programs $F^n(\mathit{I\!I})$, we look for an n for which we can find a solution for the weakest precondition predicate. With a value for the variant to bound n, calculating $wp.F^n(\mathit{I\!I})$ is again a simple matter. The difficulty is that the variant may depend on the inputs. This is the case in our example: it depends on the sizes N and M of the input arrays. In this situation, user assistance is required to define a value for the constants. If, for a reasonable value of the constants, no test can be constructed, then there is an indication that coverage is not possible. In this case, theorem proving may be used to establish that the weakest precondition is *false*, so that coverage cannot be achieved.

For our example, a value for N or M of at least 2 is needed to allow a test in which the arrays are long enough to lead to the execution of the loop body *NMLB*. For $n = 0$, we can already find the test $a, b := \langle 1, 1\rangle, \langle 2\rangle$. In this case, $a[1] > b[1]$ is false, so that the second existential quantification above is true. For the first quantification, 0 is a witness for n: $NMLB^0(\mathit{I\!I})$ is $\mathit{I\!I}$, and since $wp.\mathit{I\!I}.\phi = \phi$, the required precondition follows by propositional calculus and substitution.

For each target evaluation, we need to calculate an extra test. With them added to the original test set, we can be certain that the decision d of interest is now covered. Afterwards, however, we need to proceed with the examination of the remaining path conjuncts as required in this step. We do not need to consider again any of those that have already been analysed, but the new tests may be relevant for coverage when analysing the remaining conjuncts. In Section 3.4.2 we discuss how the test sets in the compliance predicate can be enriched to take the extra tests into account. In simple terms, we need to revisit the third step.

Recursions If the path p of the problematic conjunct involves recursions, to avoid fixed point calculations in the rewriting of $wp.p$, we use the specifications of the loops, based on their invariants. In our example, such an issue arises from the analysis of the conjunct (G).

As explained above, to cater for the problem with (C), we include $a, b := \langle 1, 1\rangle, \langle 2\rangle$ in the test set. After that, we find that (D) holds, (E) requires the addition of, for example, $a, b := \langle 1, 2\rangle, \langle 1\rangle$, and (F) holds. For (G), we have to deal with the recursive program *ML* (with normalised body). The tests in the test set in (G) are $a, b, c[1], i, j, k := \langle 2, 3\rangle, \langle 1\rangle, 1, 1, 2, 2$ and $a, b, c[1, 2], i, j, k := \langle 2, 3\rangle, \langle 1\rangle, (1, 2), 2, 2, 3$. For both of them, $a[i] \neq c[k-1]$ holds, so we need a test to achieve $a[i] = c[k-1]$. The weakest precondition calculation is shown below, where we refer to $a'[i] = c'[k'-1]$ as ϕ.

The path for (G) (as defined by *path*) is $NP_1;\ (\mu X \bullet NMLB);\ (\bigsqcup n \leq vL_1 \bullet NLB_1^n(\mathit{I\!I}));\ (i \leq N)_\perp$. In the weakest precondition calculation, we use the following invariant for *ML*, which we name *invML*.

$$
\begin{array}{l}
(1 \leq i \leq N+1) \wedge (1 \leq j \leq M+1) \wedge (2 \leq k \leq i+j-1) \wedge \\
(i \neq N+1 \vee j \neq M+1) \wedge \\
(1\,..\,k-1) \upharpoonright c = merge(rd((1\,..\,i-1) \upharpoonright a), rd((1\,..\,j-1) \upharpoonright b)) \wedge \\
((1\,..\,k-1) \upharpoonright c \leq (i\,..\,N) \upharpoonright a) \wedge ((1\,..\,k-1) \upharpoonright c \leq (j\,..\,M) \upharpoonright b)
\end{array}
$$

We have the usual restrictions on the values over which i, j, and k range, and the extra constraint that only one of i and j reaches the end of the array. For c, we record that up to the element in the position $k-1$, what we have is the result of the merge of a and b, up to the positions $i-1$ and $j-1$, after duplicates are removed. The array (sequence) $(x\,..\,y) \upharpoonright s$ is the subsequence of s that contains the elements in the positions x up to and including y. The function *rd* removes duplicates and the function *merge* defines merge of arrays; both functions have simple definitions. The last conjuncts of *invML* state that the elements already in c, that is, those up to the position $k-1$ are smaller than or equal to the remaining elements of a and b. We use $\leq$ to compare sequences: $s_1 \leq s_2$ means that all elements of s_1 are smaller than all the elements of s_2.

A loop can be characterised by a specification statement [Mor94] whose precondition is the invariant, and whose postcondition is the invariant conjoined with the negation of its termination condition. For *ML*, this is $i \leq N \vee j \leq M$. The frame of the specification, that is, the variables that can be changed, is the list of variables that are assigned to in the body of the loop; in our example, c, i, j, and k. The definition of the syntactic function *frame*(p), which applies to any program p and defines the list of variables assigned in p, and is used in the construction of the specification statement (see Figure 1) is very simple. A weakest precondition semantics for specification statements is also provided in [Mor94, CW99].

In handling the weakest precondition calculation for the path conjunct (G), we proceed as follows.

$$wp.(NP_1;\ (\mu X \bullet NMLB);\ (\bigsqcup n \leq vL_1 \bullet NLB_1^n(\mathit{I\!I}));\ (i \leq N)_\perp).\phi$$

$$= wp.NP_1.(wp.(\mu X \bullet NMLB).(wp.(\bigsqcup n \leq vL_1 \bullet NLB_1^n(\mathit{I\!I})).i' \leq N \wedge \phi)')'$$

[*wp* for sequences and assumptions]

$$= wp.NP_1.\left(\begin{array}{c} wp.(c,i,j,k : [invML, invML' \wedge (i' > N \vee j' > M)]). \\ (wp.(\bigsqcup n \leq vL_1 \bullet NLB_1^n(\mathit{II})).i' \leq N \wedge \phi)' \end{array}\right)' \qquad [\text{definition of } (\mu X \bullet NMLB)]$$

$$= wp.NP_1.\left(invML \wedge \left(\begin{array}{c} \forall c', i', j', k' \bullet invML' \wedge (i' > N \vee j' > M) \Rightarrow \\ (wp.(\bigsqcup n \leq vL_1 \bullet NLB_1^n(\mathit{II})).i' \leq N \wedge \phi)' \end{array}\right)[\_/']\right)'$$

[wp for specification statements]

$$= wp.NP_1.\left(invML \wedge \left(\begin{array}{c} \forall c', i', j', k' \bullet invML' \wedge (i' > N \vee j' > M) \Rightarrow \\ (\exists n \leq vL_1 \bullet wp.NLB_1^n(\mathit{II}).i' \leq N \wedge \phi)' \end{array}\right)[\_/']\right)' \qquad [wp \text{ for choice}]$$

$$= \left(\begin{array}{l} a[1] \leq b[1] \Rightarrow \\ \quad \left(invML \wedge \left(\begin{array}{c} \forall c', i', j', k' \bullet invML' \wedge (i' > N \vee j' > M) \Rightarrow \\ (\exists n \leq vL_1 \bullet wp.NLB_1^n(\mathit{II}).i' \leq N \wedge \phi)' \end{array}\right)[\_/']\right)[a[1], 2, 1, 2/c[1], i, j, k] \end{array}\right) \wedge$$
$$\left(\begin{array}{l} a[1] > b[1] \Rightarrow \\ \quad \left(invML \wedge \left(\begin{array}{c} \forall c', i', j', k' \bullet invML' \wedge (i' > N \vee j' > M) \Rightarrow \\ (\exists n \leq vL_1 \bullet wp.NLB_1^n(\mathit{II}).i' \leq N \wedge \phi)' \end{array}\right)[\_/']\right)[a[1], 1, 2, 2/c[1], i, j, k] \end{array}\right)$$

[wp for sequences and conditionals]

$$= \left(\begin{array}{l} a[1] \leq b[1] \Rightarrow \\ \quad \left(invML[a[1], 2, 1, 2/c[1], i, j, k] \wedge \left(\begin{array}{c} \forall c', i', j', k' \bullet invML' \wedge (i' > N \vee j' > M) \Rightarrow \\ (\exists n \leq vL_1 \bullet wp.NLB_1^n(\mathit{II}).i' \leq N \wedge \phi)' \end{array}\right)[\_/']\right) \end{array}\right) \wedge$$
$$\left(\begin{array}{l} a[1] > b[1] \Rightarrow \\ \quad \left(invML[b[1], 1, 2, 2/c[1], i, j, k] \wedge \left(\begin{array}{c} \forall c', i', j', k' \bullet invML' \wedge (i' > N \vee j' > M) \Rightarrow \\ (\exists n \leq vL_1 \bullet wp.NLB_1^n(\mathit{II}).i' \leq N \wedge \phi)' \end{array}\right)[\_/']\right) \end{array}\right)$$

[property of substitution]

$$= \left(a[1] \leq b[1] \Rightarrow \left(\begin{array}{c} \forall c', i', j', k' \bullet invML' \wedge (i' > N \vee j' > M) \Rightarrow \\ (\exists n \leq vL_1 \bullet wp.NLB_1^n(\mathit{II}).i' \leq N \wedge \phi)' \end{array}\right)[\_/']\right) \wedge$$
$$\left(a[1] > b[1] \Rightarrow \left(\begin{array}{c} \forall c', i', j', k' \bullet invML' \wedge (i' > N \vee j' > M) \Rightarrow \\ (\exists n \leq vL_1 \bullet wp.NLB_1^n(\mathit{II}).i' \leq N \wedge \phi)' \end{array}\right)[\_/']\right)$$

[substitution, properties of sequences (arrays), and definitions of rd and $merge$]

The result of the weakest precondition calculation again involves a program iteration: $NLB^n(\mathit{II})$. If we instantiate N and M to 2 and 1, that is, if we search for tests with a of size 2 and b of size 1, we can proceed with the calculation, and discover that $a, b := \langle 2, 2 \rangle, \langle 1 \rangle$, for example, is satisfactory to provide the coverage required by (G). The procedure is that already explained and exemplified above for $NMLB^n(\mathit{II})$.

3.4.2. Revisiting the test sets

With the inclusion of new tests $t_1, \ldots, t_n$ in the original test set, we need to recalculate some of the test sets generated in the third step: those in the path conjuncts that have not yet been visited in this step. For that, we normalise the new test cases, and revisit the third step. Every test set definition is originally defined in terms of the original test set ts. If a definition tsd is in a conjunct yet to be visited, it needs to be reevaluated because we have a new larger ts; we need to insert the new normalised tests in ts, and reevaluate tsd.

Fortunately, the reevaluation can be incremental: a new test generates one or more tests in tsd, but all the tests already there are kept. We can apply the procedure of the third step to the instantiation of tsd for a ts defined as $\{t_1, \ldots, t_n\}$, and add the tests in the resulting set to the existing set. This is justified by the distributivity of $⨾$, $\hat{⨾}$, and ρ over set union. More precisely, $((ts_1 \cup ts_2) ⨾ p) = ((ts_1 ⨾ p) \cup (ts_2 ⨾ p))$, $((ts_1 \cup ts_2) \hat{⨾}\, d) = ((ts_1 \hat{⨾}\, d) \cup (ts_2 \hat{⨾}\, d))$, and finally $\rho(F, vrt, ts_1 \cup ts_2) = \rho(F, vrt, ts_1) \cup \rho(F, vrt, ts_2)$.

In our example, the first three path conjuncts (A), (B), and (C) are already covered: the first two were satisfied straightaway, and an extra test $a, b := \langle 1, 1 \rangle, \langle 2 \rangle$ was added to sort out (C). If we evaluate the test set definition in (D) for the singleton set ts including only the normalised additional test $a, b, c, i, j, k := \langle 1, 1 \rangle, \langle 2 \rangle, c, i, j, k$, then we get the singleton containing $a, b, c[1], i, j, k := \langle 1, 1 \rangle, \langle 2 \rangle, 1, 2, 1, 2$. (The application of the third step to $\rho(NMLB, vML, ts ⨾ NP_1)$ for the new ts shows that the recursive call introduces another new test, $a, b, c[1], i, j, k := \langle 1, 1\rangle, \langle 2\rangle, 1, 3, 1, 2$, but it is filtered out by the condition $i \leq N \wedge j \leq M$.) We include the unique extra test in the coverage analysis of the decision in (D).

As we explained in Section 3.3.3, the test set calculated for (D) based on the original test set included only $a, b, c[1], i, j, k := \langle 1, 3\rangle, \langle 2, 4\rangle, 1, 2, 1, 2$. It could not possibly achieve coverage. With the extra test included to cater for (C), however, the coverage required in (D) for $a[i] = c[k - 1]$ is achieved as well; since this is just a condition, we need only two tests that lead to different evaluations: *true* and *false*. We still need, however, to proceed with the analysis of the remaining path conjuncts. It is only when all of them have been analysed that we finish. In the previous section, we mentioned that (E) required a new test, that (F) holds, and we explained how to proceed in relation to (G). Finally, (H) and (I) hold.

4. Discussion

Due to the (mathematical) nature of the concepts and steps used in our procedure for test-data generation, it can be automated with the use of tactics of proof and constraint solvers. In the worst case, where proof or constraint solving fails, we are left with a specification of the missing test data based on the decisions of the program. Such a specification can be used, for instance, to check the suitability of test data generated by other means. Moreover, if the specification is not satisfiable, then we have proof that it is not possible to cover the program. This is a very useful result that avoids waste of effort to achieve a coverage that is actually not possible. Usually, the required course of action, in this case, is refactoring of the program.

In the UTP assignments (or programs in general) are predicates. This is an apparently trivial issue, but this flexible framework guarantees that we can reason about programs and their tests using first-order predicate logic. This greatly facilitates automation using powerful theorem provers [Jon92, ORS92, NWP02].

All steps of our procedure can be automated; the structure of the program provides the needed guidance for the application of laws. For the first step, what we have is basically a simple rewriting procedure that involves the exhaustive application of a definition (that of the testing semantics) and a law. The structure of the program guides the applications of the definition of the testing semantics.

The second step is again a normalisation procedure, applied exhaustively to all programs in the normalised compliance theorem and in the test set. Its automation is a standard exercise; an example of such mechanisation (for a much more complex programming language) can be found in [LCS02]. For optimisation, normalisations of programs in a path conjunct can be reused in the normalisation of programs in conjuncts that correspond to extensions of that path; for example, in the normalisation of P_1, we use NI, in the normalisation of P_2 we use NP_1, and so on. We also observe that in this case the standard procedure to normalise programs is simplified, since recursions are not removed.

In the third step, we again have just the application of definitions (of test set operators $\hat{;}$, $\hat{:}$, and ρ) and of a normalisation procedure. We now, however, also normalise (and eliminate) the recursions and program iterations. Since the decisions are written in the notation of a programming language, their evaluation is a simple matter. As demonstrated in our example, optimisation can be achieved by applying the law lift-semi-assoc to expressions $ts \,\hat{;}\, (p_1;\ p2)$ to obtain $(ts \,\hat{;}\, p_1);\ p2$, and use the (existing) evaluation and normalisation of $(ts \,\hat{;}\, p_1)$. In addition, it is also possible to reuse results of the evaluation of filter expressions $ts \,\hat{:}\, d$. Figure 3 shows how they occur repeatedly in different path conjuncts. These optimisations are not essential, but are likely to be of much relevance in the application of our technique to large examples.

We observe, however, that in areas like avionics and the automotive industry, where techniques like MC/DC testing are relevant, applications are typically written in safe subsets of imperative languages and individual procedures in these programs are typically tens of lines long, not hundreds. In these (restricted) domains, as already explained, there are successful examples of translation of programs to a formal language like that considered here. For unit testing of individual procedures in this area of application, experience with verification based on theorem proving indicates that industrial application is feasible.

For the fourth step of our procedure, current tools [IPL, LDR] that have been extensively used in industrial settings are useful. As we explained previously, at the end of the third step, the compliance theorem is expressed as a conjunction of predicates of the form $tsd \in [\![\, d \,]\!]^{T}$, where tsd is an explicit enumeration of tests. So, what we have are simple propositions stating that a specific set of tests covers a specific decision. The calculations in the previous steps have tackled the need to consider paths in the program and their feasibility. To tackle each of these propositions, tools like [IPL, LDR] are very effective. For MC/DC, in particular, techniques based on boolean derivatives [Kuh99, OBY04], for example, are also effective.

Implementation of our procedure can, therefore, be best achieved in one of two ways. We can either integrate a theorem prover with a testing tool or extend a testing tool to provide program transformation facilities. Soundness is better achieved and more easily justified if the first approach is followed, and theorem provers like Isabelle [NWP02], for instance, provide mechanisms for integration with external tools.

Weakest precondition calculations required in the fourth step can be achieved using a theorem prover as well. An example of such an implementation is provided by the DAZ tool [CO06] associated with the ProofPower theorem prover. For optimisation, the weakest precondition calculations for a program p can be reused in the weakest precondition calculation for any sequence $p\ ;\ q$, since $wp.(p\ ;\ q).\phi = wp.p.(wp.q.\phi)$. Such sequences are likely to arise from path conjuncts whose path is an extension of p.

In our procedure, as already said, we may need to take two kinds of extra inputs: bounds for variants and loop invariants. If a variant bound is too low, it may be the case that no tests are found, when some actually exist. We, however, do not get unsound results: tests that do not provide the required coverage. Similarly, if an invariant is not strong enough to specify the loop precisely, we do not get unsound results. For example, if we leave the last conjunct out of the invariant *ML*, our calculations lead to the weakest precondition *false*. A weak invariant means that the specification of the loop is nondeterministic, and it may not be possible to drive a nondeterministic program to achieve the required tests. If a test is obtained, however, we can be certain that it provides the required coverage. This makes it possible to use invariant calculators [EPG$^+$07, BDS06] to further automate this step if an invariant for the loops is not available.

For validation, we have mechanised our specifications of coverage criterion, including that for unique cause MC/DC presented here. We have used the Jaza and PiZA animators for Z, and the Z/Eves theorem prover [Mei00]. To check that the sets of tests identified by our semantics indeed provide the required coverage, we have used the LDRA TestBed [LDR]. This tool can check coverage of a given test set.

5. Related works

Model-based testing is a well established approach, where models of the system under testing are used to guide the generation and evaluation of requirements-based black-box test. A large corpus of work is available in this area and formal models are considered widely [DF93, LY96, TS91, SU93, GJ98, HSS01]. Here, we use a very concrete model of the program to generate white-box tests.

Path-oriented test-case generation techniques [Kor90] reduce the problem to that of finding inputs that result in the execution of a path to the element of the program that needs to be covered. Paths are selected, possibly automatically, and either symbolic execution or instrumentation and actual program execution is used to suggest appropriate inputs. The use of symbolic execution to support testing is not a novel idea [Kin75]; in this line of work, it is complemented with facilities to choose paths that are of interest, perhaps due to the need for achieving coverage according to a specific criterion.

In goal-oriented techniques [Kor92, MM98] there is no path selection; they define functions (on values of variables at specific points of the program) whose values are minimal for test sets that provide the required coverage, and use results on function minimization to produce the tests. In the chaining approach [FK96], the search for minimal values takes data dependency as well as control flow information into account. When a path-oriented, goal-oriented, or chaining technique is used, if the test data generation fails due, for instance, to a time out, or lack of resources, no further support is provided.

In [XVM05], the use of model checking to generate test cases for (Modified Condition/Decision Coverage [CM94]) coverage of C programs is explored in a realistic case study. To cope with the size of the model, abstraction techniques based on weakest preconditions are used. Besides the extra modelling effort that is required, again, when tests fail to be generated, no extra support is provided.

Theorem proving has been used to generate tests based on partitions of the input domain; data in the same partition are assumed to cause the same error, or no error at all. Work has been carried out for Z [HNS97, SCS97, BCM00] and Prolog [BGM91] models; they are concerned with requirements-based testing. Tactics are used to manipulate specifications to generate predicates that suggest tests. The tactics embed partitioning heuristics based, for example, on the disjunctive normal form or on case analysis.

Another approach for exploiting proof in testing is presented in [Mah99]; it extracts information from case analysis in proofs of correctness of functional programs to generate specifications of tests. The concern is again requirements-based testing, and automation is indicated as future work.

Like ours, the work in [Hor02] is motivated by lack of tool support. It proposes metrics to help in the choice of decisions to be targeted. They assess the difficulty to find tests to cover particular statements, based on the number of conditions that determine whether the path to those statements is followed. They also assess the benefit of executing a statement, in terms of the number of other statements that are executed as a result. To help with the definition of tests, a tool presents a reduced control-flow graph of the relevant path, annotated with the values that the decisions are required to have. We can benefit from these results by using the metrics in [Hor02] to define the order in which the path conjuncts that we generate in the first step are tackled in the fourth step. Our procedure envisages 100% coverage, but it can be adapted for partial coverage. Once a decision is chosen, however, we present not a suggestion, but a specification of the required tests; it is suitable for calculation using a theorem prover.

Several non-standard coverage criteria have been studied in [WGS94]; they are all based on the idea of establishing the impact of individual conditions on the outcome of a decision, and MC/DC corresponds to one of the variants. A tool for test generation is described in [WGS94], but the coverage criteria are applied to boolean specifications, rather than to programs. The decisions are, first of all, written in disjunctive normal form, and this is not appropriate for coverage of programs, since any connection to object code is lost. As we remarked previously, some coverage criteria are dependent on the syntax of the decisions, and certainly on the syntax of the programs. Programs with the same functionality, or logically equivalent decisions, are not necessarily covered by the same test sets. Therefore, techniques that rewrite decisions are appropriate for requirements-based testing, but not for white-box testing.

Concerns with cost have motivated work on optimal test sets, which achieve coverage with the minimal number of tests. This is an NP-hard problem [HCL$^+$03], and several heuristics have been studied [Agr94, BM96, Chu87, GS93, GN04]. They aim at finding minimal sets of entities (decisions, conditions, statements, and so on) whose coverage is enough to achieve total coverage. Such results can be used in conjunction with our procedure, for instance, to establish an order in which decisions should be tackled. Using our procedure we can check, after considering only the decisions indicated, whether total coverage is indeed achieved.

An algorithm for generating test data to provide coverage according to a specific variety of MC/DC has been recently presented in [CHL12], where good results in terms of fault detection and execution time are reported. The focus of the algorithm is decisions based on boolean conditions, and like other commercial tools already mentioned, could be used in conjunction with our technique. This approach can both check for coverage of decisions and identify the missing tests that define the target evaluation table that we use here.

The tool in [PXTdH10] uses the modern technology based on dynamic symbolic execution to generate tests to provide MC/DC coverage. The approach favours reuse of tools for structural coverage. It rewrites programs so that coverage, according to a weaker requirement, of the new version amounts to coverage, considering stringent requirements like MC/DC or boundary value analysis, of the original program is achieved. Experiments show that high levels of coverage can be achieved, but no support is offered for completion.

6. Conclusions

We have presented a strategy for generation of tests to achieve control coverage of programs, whenever possible. It is based on a precise definition of the coverage criterion; we showed how to provide such a specification using the complex requirements of MC/DC as an example. Basically, we need to define the sets of all test sets that provide coverage for an arbitrary program.

While there are several formalisations of coverage for decisions in the literature [AOH03], we are not aware of any such results for programs. A formal analysis of MC/DC is presented in [KB04]. We took a step further and used Z to formalise all the concepts; this enabled the mechanisation we discussed.

Our test-generation strategy uses standard techniques of programming and verification: normalisation, variants, invariants, and weakest preconditions. It can be automated using tactics of proof, and moreover, if the strategy fails to generate a test, we are still left with an explicit and formal characterisation of the adequate missing tests. This goes well beyond the support provided by existing testing tools. The specification of the missing tests can be used to prove that a test generated by other means is adequate, or that the program is not coverable. Proof of adequacy of a new test can be achieved applying our technique again, but tactics of proof to establish uncoverability are left as future work.

The problem of infeasible paths is a hindrance to many techniques [MM98]. In our work, feasibility is considered in the testing semantics; the definition for sequences $p;\ q$ plays a central role, since it requires that q is covered by tests obtained after execution of p. In the procedure, normalisation of the programs in the second step eliminates any infeasible paths, so that the test sets calculated only include feasible tests.

Verification of the Typhoon's flight and engine control software [AC05] directly uses the weakest precondition approach employed here to generate verification conditions. The tools deployed for Typhoon are the same required to generate our specifications of missing tests. We plan to employ that verification technology as part of a new software engineering approach for avionics [TACO04]. It would be relatively straightforward to support the approach described in this paper using the same technology, thus achieving the cost reductions anticipated in [TACO04] while still satisfying a coverage criterion like MC/DC, for instance.

Our technique is entirely based on algebraic techniques and proof tactics, and so it is amenable to automation. We also expect that it is possible to develop effective proof tactics to handle the result of a weakest precondition calculation when there is the indication that it is not possible to achieve coverage. The form of these results follows a pattern determined by the definition of *wp* and by the shape of path programs. We will also explore this uniformity in our future work.

Proof is the basis of the tool HOL-TestGen described in [BW12]. It provides facilities for test specification and generation based on Isabelle/HOL, and can support interactive theorem proving as well as test generation. HOL-TestGen is suitable for white-box (as well as black-box) testing, and so exploring the implementation of our procedure based on its theoretical framework is a promising avenue for future work.

Acknowledgements

Ana Cavalcanti is partially funded by the Royal Society of London. Our case study was suggested by Marie-Claude Gaudel. We discussed this work with Emine Aydal. We are also grateful to anonymous referees for very helpful suggestions.

References

[Abr96] J.-R. Abrial. *The B-Book: Assigning Progams to Meanings*. Cambridge University Press, 1996.

[AC05] M. M. Adams and P. B. Clayton. Cost-Effective Formal Verification for Control Systems. In K. Lau and R. Banach, editors, *ICFEM 2005: Formal Methods and Software Engineering*, volume 3785 of *Lecture Notes in Computer Science*, pages 465 – 479. Springer-Verlag, 2005.

[ACOS00] R. Arthan, P. Caseley, C. M. O'Halloran, and A. Smith. ClawZ: Control laws in Z. In *3rd International Conference on Formal Engineering Methods*, pages 169 – 176. IEEE Press, 2000.

[Agr94] H. Agrawal. Dominators, Super Blocks, and Program Coverage. In *21st ACM Symposium on Principles of Programming Languages*, pages 25 – 34, 1994.

[AOH03] P. Ammann, J. Offutt, and H. Huang. Coverage Criteria for Logical Expressions. *14th International Symposium on Software Reliability Engineering*, 00:99 – 107, 2003.

[Bar95] G. Barrett. Model checking in practice: The T9000 Virtual Channel Processor. *IEEE Transactions on Software Engineering*, 21(2):69 – 78, 1995.

[BCM00] S. Burton, J. Clark, and J. A. McDermid. Testing, Proof and Automation: An Integrated Approach. In *1st International Workshop on Automated Program Analysis, Testing and Verification (WAP-ATV 2000)*, pages 57 – 63, 2000.

[BDS06] M. Boshernitsan, R. Doong, and A. Savoia. From Daikon to Agitator: lessons and challenges in building a commercial tool for developer testing. In *International Symposium on Software Testing and Analysis*, pages 169 – 180. ACM Press, 2006.

[BGM91] G. Bernot, M.-C. Gaudel, and B. Marre. Software testing based on formal specifications: a theory and a tool. *Software Engineering Journal*, 6(6):387 – 405, 1991.

[BM96] A. Bertolino and M. Marré. How Many Paths are Needed for Branch Testing? *The Journal of Systems and Software*, 35(2):95 – 106, 1996.

[Bul] Bullseye Testing Technology. *C-Cover*. `www.bullseye.com`.

[BW12] Achim D. Brucker and Burkhart Wolff. On Theorem Prover-based Testing. *Formal Aspects of Computing*, 2012.

[CHL12] J.-R. Chang, C.-Y. Huang, and P.-H. Li. An investigation of classification-based algorithms for modified condition/decision coverage criteria. In *6th International Conference on Software Security and Reliability*, pages 127 – 136. IEEE, 2012.

[Chu87] T. Chusho. Test Data Selection and Quality Estimation Based on the Concept of Essential Branches for Path Testing. *IEEE Transactions on Software Engineering*, 13(5):509 – 517, 1987.

[CHW06] A. L. C. Cavalcanti, W. Harwood, and J. C. P. Woodcock. Pointers and Records in the Unifying Theories of Programming. In S. Dunne and B. Stoddart, editors, *Unifying Theories of Programming*, volume 4010 of *Lecture Notes in Computer Science*, pages 200 – 216. Springer-Verlag, 2006.

[CKOW07] A. L. C. Cavalcanti, S. King, C. M. O'Halloran, and J. C. P. Woodcock. A scientific investigation of MC/DC testing. Technical Report YCS-2007-411, University of York, Department of Computer Science, 2007.

[CM94] J. J. Chilenski and S. P. Miller. Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal*, pages 193 – 200, 1994.

[CO06] P. Clayton and C. O'Halloran. Using the compliance notation in industry. In A. L. C. Cavalcanti, A. C. A. Sampaio, and J. C. P. Woodcock, editors, *Refinement Techniques in Software Engineering*, volume 3167 of *Lecture Notes in Computer Science*, pages 269 – 314. Springer-Verlag, 2006.

[CSE96] J. Callahan, F. Schneider, and S. Easterbrook. Specification-based testing using model checking. In *SPIN Workshop*, 1996.

[CW99] A. L. C. Cavalcanti and J. C. P. Woodcock. ZRC—A Refinement Calculus for Z. *Formal Aspects of Computing*, 10(3):267—289, 1999.

[DF93] J. Dick and A. Faivre. Automating the generation and sequencing of test cases from model-based specifications. In *Formal Methods Europe*, volume 670 of *Lecture Notes in Computer Science*, pages 268 – 284. Springer-Verlag, 1993.

[Dij76] E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[EPG+07] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 2007.

[FK96] R. Ferguson and B. Korel. The Chaining Approach for Software Test Data Generation. *ACM Transactions on Software Engineering and Methodology*, 5(1):63 – 86, 1996.

[GJ98] M.-C. Gaudel and P. J. James. Testing algebraic data types and processes : a unifying theory. *Formal Aspects of Computing*, 10(5-6):436 – 451, 1998.

[GN04] J. Grabowski and B. Nielsen. Using Model Checking for Reducing the Cost of Test Generation. In J. Grabowsk and B. Nielsen, editors, *Formal Approaches to Software Testing*, volume 3395 of *Lecture Notes in Computer Science*, pages 110 – 124. Springer-Verlag, 2004.

[GS93] R. Gupta and M. L. Soffa. Employing Static Information in the Generation of Test Cases. *Software Testing, Verification and Reliability*, 3(1):29 – 48, 1993.

[HCL+03] H. S. Hong, S. D. Cha, I. Lee, O. Sokolsky, and H. Ural. Data Flow Testing as Model Checking. In *25th International Conference on Software Engineering*, pages 232 – 242, 2003.

[HH98] C. A. R. Hoare and J. He. *Unifying Theories of Programming.* Prentice-Hall, 1998.

[HLSU02] H. S. Hong, I. Lee, O. Sokolsky, and H. Ural. A Temporal Logic Based Theory of Test Coverage and Generation. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems*, 2002.

[HNS97] S. Helke, T. Neustupny, and T. Santen. Automating Test Case Generation from Z Specifications with Isabelle. In J. P. Bowen, M. G. Hinchey, and D. Till, editors, *International Conference of Z Users*, volume 1212 of *Lecture Notes in Computer Science*, pages 52 – 71. Springer-Verlag, 1997.

[Hor02] S. Horwitz. Tool Support for Improving Test Coverage. In D. Le Métayer, editor, *European Symposium on Programming*, volume 2305 of *Lecture Notes in Computer Science*, pages 162 – 177. Springer-Verlag, 2002.

[HSS01] R. M. Hierons, S. Sadeghipour, and H. Singh. Testing a system specified using statecharts and Z. *Information and Software Technology*, 43(2):137 – 149, 2001.

[IPL] IPL. *AdaTEST 95.* `www.ipl.com/products/tools/pt600.php`.

[Jon92] R. B. Jones. ICL ProofPower. *BCS FACS FACTS*, Series III, 1(1):10 – 13, 1992.

[KB04] K. Kapoor and J. P. Bowen. A formal analysis of MCDC and RCDC test criteria. *Software Testing, Verification and Reliability*, 15:21 – 40, 2004.

[Kin75] J. C. King. A new approach to program testing. In *International Conference on Reliable software*, pages 228–233. ACM, 1975.

[Kor90] B. Korel. Automated Software Test Data Generation. *IEEE Transactions on Software Engineering*, 16(8):870 – 879, 1990.

[Kor92] B. Korel. Dynamic Method for Software Test Data Generation. *Journal of Software Testing, Verification and Reliability*, 4:203 – 213, 1992.

[Kuh99] D. R. Kuhn. Fault Classes and Error Detection Capability of Specification-Based Testing. *ACM Transactions on Software Engineering and Methodology*, 8(4):411 – 424, 1999.

[LCS02] B. O. Lira, A. L. C. Cavalcanti, and A C. A. Sampaio. Automation of a Normal Form Reduction Strategy for Object-oriented Programming. In *Proceedings of the 5th Brazilian Workshop on Formal Methods*, pages 193 – 208, 2002.

[LDR] LDRA. *LDRA TestBed.* `www.ldra.co.uk/testbed.asp`.

[LY96] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines - A survey. In *Proceedings of the IEEE*, volume 84, pages 1090 – 1126, 1996.

[Mah99] S. Maharaj. Towards a Method of Test Case Extraction from Correctness Proofs. In *14th International Workshop on Algebraic Development Techniques*, pages 45 – 46, 1999.

[Mei00] I. Meisels. *Software Manual for Windows Z/EVES Version 2.1.* ORA Canada, 2000. TR-97-5505-04g.

[MM98] C. C. Michael and G. McGraw. Automated software test data generation for complex programs. In *Automated Software Engineering*, pages 136 – 146, 1998.

[Mor94] C. C. Morgan. *Programming from Specifications.* Prentice-Hall, 2nd edition, 1994.

[NWP02] T. Nipkow, M. Wenzel, and L. C. Paulson. *Isabelle/HOL: a proof assistant for higher-order logic.* Springer, 2002.

[OBY04] V. Okun, P. E. Black, and Y. Yesha. Comparison of Fault Classes in specification-Based Testing. *Information and Software Technology*, 46(8):525 – 533, 2004.

[ORS92] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748 – 752. Springer, 1992.

[PXTdH10] R. Pandita, T. Xie, N. Tillmann, and J. de Halleux. Guided test generation for coverage criteria. In *26th IEEE International Conference on Software Maintenance*, pages 1 – 10. IEEE Computer Society, 2010.

[Rat] Rational Software. *PureCoverage.* `www.rational.com/products/pqc/index.jsp`.

[RCC11] RTCA/DO-178C/ED-12C. Software Considerations in Airborne Systems and Equipment Certification, 2011.

[RH03] S. Rayadurgam and M. P. E. Heimdahl. Generating MC/DC Adequate Test Sequences Through Model Checking. In *Software Engineering Workshop, 28th Annual NASA Goddard*, pages 91 – 96, 2003.

[SCHS10] A. Sherif, A. L. C. Cavalcanti, J. He, and A. C. A. Sampaio. A process algebraic framework for specification and validation of real-time systems. *Formal Aspects of Computing*, 22(2):153 – 191, 2010.

[SCS97] H. Singh, M. Conrad, and S. Sadeghipour. Test Case Design based on Z and the Classification-tree Method. In M. G. Hinchey and S. Liu, editors, *1st International Conference on Formal Engineering Methods (ICFEM 1997)*, pages 81 – 90. IEEE Computer Society Press, 1997.

[Sof] Software Research, Inc. *TCAT*. `www.soft.com/products/web/tcat.java.html`.

[SU93] H. V. D. Schoot and H. Ural. Data flow oriented test selection for LOTOS. *Computer Networks and ISDN Systems*, 27(7):1111 – 1136, 1993.

[TACO04] N. Tudor, M. Adams, P. Clayton, and C. M. O'Halloran. Auto-coding/Auto-proving flight control software. In *IEEE Digital Avionics Systems Conference*, 2004.

[TS91] P. Tripathy and B. Sarikaya. Test Generation from LOTOS Specifications. *IEEE Transactions on Computers*, 40(4):543–552, 1991.

[WD96] J. C. P. Woodcock and J. Davies. *Using Z—Specification, Refinement, and Proof.* Prentice-Hall, 1996.

[WGS94] E. Weyuker, T. Goradia, and A. Singh. Automatically Generating Test Data from a Boolean Specification. *IEEE Transactions on Software Engineering*, 20(5):353 – 363, 1994.

[XVM05] S. Xia, B. Di Vito, and C. Munoz. Automated Test Generation for Engineering Applications. In *Automated Software Engineering*, pages 283 –286, 2005.

[ZC12a] F. Zeyda and A. L. C. Cavalcanti. Higher-Order UTP in Theories of Object-Orientation. In *4th International Symposium on Unifying Theories of Programming*, Lecture Notes in Computer Science, 2012.

[ZC12b] F. Zeyda and A. L. C. Cavalcanti. Mechanical Reasoning about Families of UTP Theories. *Science of Computer Programming*, 77(4):444 – 479, 2012.

A. Laws

Law [lift-semi-assoc] $(ts \mathbin{\hat{;}} p_1) \mathbin{\hat{;}} p_2 = ts \mathbin{\hat{;}} (p_1;\ p_2)$

Law [set-comp-range-dist]

$$\{tc : \{t_1, \ldots, t_n\};\ n' : T \mid \phi(tc) \bullet e(tc)\} = \{n' : T \mid \phi(t_1) \bullet e(t_1)\} \cup \ldots \cup \{n' : T \mid \phi(t_n) \bullet e(t_n)\}$$

Law [test-expr-eval] $(v := e_1);\ e_2(n', v) = e_2(n', e_1)$
where *v is the list of all undashed variables in the alphabet.*
provided *the variables of v are not free in e_1 and n' is not in the alphabet.*

Law [set-comp-enum] $\{n' : \mathbb{N} \mid n' \leq l \bullet e(n')\} = \{e(0), \ldots, e(l)\}$

Law [seq-comp-test-const] $\{t : \{t_1, \ldots, t_n\} \mid \phi(t)\} = \{i : I \bullet t_i\}$
provided *I identifies the tests t_i in $\{t_1, \ldots, t_n\}$ for which $\phi(t_i)$ holds.*