Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

CLPS–B - A Constraint Solver to Animate a B Specification

Fabrice Bouquet, Bruno Legeard, Fabien Peureux

Laboratoire d'Informatique (LIFC)
Université de Franche-Comté – CNRS – INRIA
16, route de Gray - 25030 Besançon cedex, France
Tel.: (33) 381 666 664
e-mail: {bouquet,legeard,peureux}@lifc.univ-fcomte.fr

The date of receipt and acceptance will be inserted by the editor

Abstract. This paper proposes an approach to evaluate B formal specifications using Constraint Logic Programming with sets. This approach is used to animate and generate test sequences from B formal specifications. The solver, called CLPS–B, is described in terms of constraint domains, consistency verification and constraint propagation. It is more powerful than most constraint systems, because it allows the domain of variable to contain other variables, which increase the level of abstraction. The constrained state propagates the non-determinism of the B specifications and reduces the number of states in a reachability graph. We illustrate this approach by comparing the constrained state graph exploration with the concrete one in a simple example: Process scheduler. We also describe the automated test generation method that uses the CLPS–B solver to better control of combinational explosion.

Key words: B Notation – Constraint Logic Programming – Set Constraints – Evaluation of specifications – Animation.

1 Introduction

This article presents a constraint solver used to evaluate B formal models. The B method, developed by Jean-Raymond Abrial [Abr96], forms part of a formal specification model based on first order logic extended to set constructors and relations. This method allows refinement of the first abstract level of specification, called a B abstract machine, into an implementation. It has already been successfully used for industrial projects [BDM98]. In our proposal, we only consider B abstract machines (no refinement or implementation level).

1.1 B abstract machine evaluation

A B abstract machine describes the system in terms of an abstract machine defined by a data model (sets, constants and state variables), invariant properties expressed on the variables, and operations. Operations are described in terms of preconditions and substitutions using the language of generalized substitutions (which is an extension of the language of guarded commands). It extends earlier set-based specification notations such as *VDM* [Jon90] and *Z* [Spi92]. More precisely, the data of an abstract machine are specified in the data model by means of a number of mathematical concepts such as sets, relations, functions, sequences and trees. This model, also called the static model, presents the various data of the machine, which were used by the operations, and the invariant properties that the state variables must follow. The dynamics of an abstract machine is expressed through its operations. The role of an operation, which is executed by the computer, is to modify the state of the abstract machine. Each operation must maintain the state invariant.

The objective of the constrained evaluation of B abstract machines, as proposed in this article, is to look into the reachability graph of the system described by the specification: it consists of being able to initialize the machine, evaluate substitutions and check properties of the new computed state. This mechanism is used as a basis to animate B abstract machines [BLP00] and to generate functional test suites from a B abstract machine [LP01, LPU02a].

This approach using constraints manipulates a store of constraints (called constrained states) instead of concrete states, classically handled in the animation of specifications [Dic90, WE92]. This evaluation process makes it possible to maintain the non-determinism of the specifications, and reduces the number of generated states. For example, non-determinism expressed by the B expression:

```
ANY xx WHERE xx ∈ Y THEN  substitution
```

is maintained by the set constraint $xx \in Y$. Substitution is no longer calculated for a particular value, but for a variable xx whose domain is Y. Fig. 1 presents the processing carried out by the CLPS–B solver (**C**onstraint **L**ogic **P**rogamming with **S**et to B) from B specification. The specification is rewritten by a translator into a constraint system. The initial constraint system is rewritten in canonical form (with $\in, \notin, \neq$) by a pre-processor. The concrete structure of the solver is based on reduction and generation procedures. The constrained evaluation of B specifications requires a hypothesis of finite domains: given sets must be replaced by finite enumerated sets. This makes it possible for the CLP-based evaluation to perform much stronger reasoning about the specifications, and this is usually necessary for the animation process to be tractable.

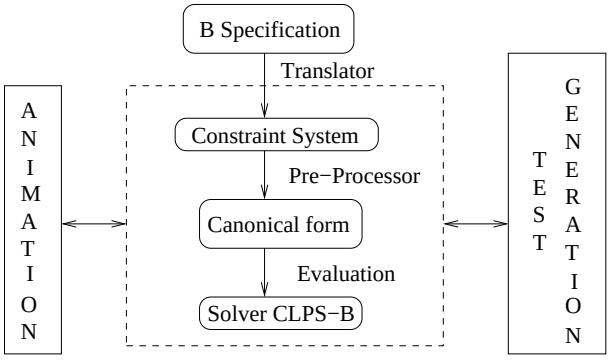


Fig. 1. Evaluation of B specification with CLPS–B

The CLPS–B solver is more general than traditional animation because one evaluation sequence captures the properties of a set of concrete animation sequences. It is less powerful than proof because it requires finiteness assumptions, but it is fully automatic. Thus, each state managed by the evaluator is a store of constraints which represents a set of concrete states of the B abstract machine. The process scheduler example (Fig 2) shows that for n processes, the number of constrained states in the entire reachability graph is at most $(n^2 + 3n + 2)/2$ against more than 3^n concrete states. In general purpose constraint approach, as used in CLPS–B solver, involves a dramatic reduction that makes it possible to animate much larger state spaces than would be possible otherwise.

1.2 Example

As a running example, we use a simple scheduler that manages a set of processes. Fig. 2 gives the B specification with a set *PID* composed of six processes {p1, p2, p3, p4, p5, p6}. The three state variables of the machine are *waiting, ready, active* which respectively represent the waiting, ready to be activated and active processes. In the initial state, the three sets are empty.

Four operations are defined:

- NEW: to create a new process and add it to *waiting*.
- DEL: to kill a process and delete it from *waiting*.
- READY: to activate a process of *waiting* and put it in *active* if this set is empty, add it to *ready* otherwise.
- SWAP: to disable the process of *active* and put it in *waiting*, and activate a process from *ready* if there is one (using a non-deterministic approach).

The evaluation of B expressions and the construction of the reachable states of the system constitute a new problem area for set constraint resolution. The constrained states are built incrementally by substitutions from the initial state. So, if we consider the state of the process scheduler just after the creation of a process xx:

$$waiting = \{xx\} \wedge ready = \{\} \wedge$$
$$active = \{\} \wedge xx \in \{p1, \ldots p6\}$$

```
MACHINE
    SCHEDULER
SETS
    PID = {p1, p2, p3, p4, p5, p6}
VARIABLES
    active, ready, waiting
INVARIANT
    active ⊆ PID ∧    ready ⊆ PID ∧
    waiting ⊆ PID ∧    ready ∩ waiting = {} ∧
    ready ∩ active = {}  ∧  waiting ∩ active = {} ∧
    active ∩ (ready ∪ waiting) = {} ∧
    card(active) ≤ 1∧
    (active = {}) ⇒ (ready = {})
INITIALIZATION
    active := {}||
    ready := {}||
    waiting := {}
OPERATIONS
    NEW(pp)
      PRE
        pp ∈ PID ∧
        pp ∉ (active ∪ ready ∪ waiting)
      THEN
        waiting := (waiting ∪ {pp})
      END;
    it(pp)
      PRE
        pp ∈ waiting
      THEN
        waiting := waiting − {pp}
      END;
    READY(rr)
      PRE
        rr ∈ waiting
      THEN
        waiting := (waiting − {rr})||
        IF (active = {}) THEN
          active := {rr}
        ELSE
          ready := ready ∪ {rr}
        END
      END;
    SWAP
      PRE
        active ≠ {}
      THEN
        waiting := waiting ∪ active||
        IF (ready = {})
        THEN
          active := {}
        ELSE
          ANY pp WHERE pp ∈ ready
          THEN
            active := {pp}||
            ready := ready − {pp}
          END
        END
      END;
```

Fig. 2. B Specification of process scheduler

The evaluation of the operation *new(yy)* is computed using the following rules:

1. the addition of the constraints resulting from the preconditions:

$$yy \in \{p1, ..., p6\} \wedge yy \notin waiting \wedge$$
$$yy \notin active \wedge yy \notin ready$$

2. the evaluation of substitutions:

$$waiting := waiting \cup \{yy\}$$

3. the verification of the invariant properties on the new state.

The sets handled in the computation of substitutions are explicit sets of known cardinality whose elements are either constants or variables. In this context, the approaches of set constraint resolution based on a reduction of set intervals as in CLPS [ALL94] or CONJUNTO [Ger97] do not provide a sufficiently effective propagation of constraints. It is the same for the approaches using set constraints on regular sets [AW93, Koz94] used to analyze programs. This led us to develop a new solver, CLPS-B, based on an explicit representation of variable domains by the intersection of sets of variables and constants.

The remainder of the paper is structured as follows:

- Section 2 characterizes the domain of constraints, then sets out the consistency and reduction rules implemented in the CLPS-B solver, and finally defines the coverage of the operators in the treatment of the B abstract notation,
- Section 3 discusses the use of CLPS–B to animate B abstract machines,
- Section 4 illustrates the application of CLPS–B to animate the process scheduler example,
- Sections 5 and 6 describe the automated test generation approach based on the CLPS–B solver,
- Section 7 presents conclusion and future work.

2 B and Constraint Resolution with CLPS–B

This section introduces the domain of the CLPS–B solver and presents the part of the B notation that is covered. We show the CLPS–B constraint resolution mechanism and the inference rules.

2.1 Restriction on the B notation

As stated earlier, we are only interesting in the more abstract level of the B notation, namely abstract machines. For simplicity, and to improve the efficiency of the process, we impose two main restrictions on these B abstract machines:

- firstly, we only consider single abstract machines, without composition,
- secondly, the given sets must be equal to a finite enumerated set.

Nonetheless, these restrictions are not too strong in practice. We used the CLPS–B solver to animate and generate tests on various industrial applications [BLP00, LP01, LPV01, CGLP01, BJLP02], building each time a specific B formal model with just one machine and enumerated finite sets without any difficulties.

2.2 Computation Domain

The B notation is based on set theory, with four set definitions:

1. Cartesian product: Set $\times$ Set
2. Power-set: $\mathbb{P}$(Set)
3. Set comprehension: { Variable | Predicate}
4. Given set: let $\mathcal{T}$ be the set of all deferred sets.

The next definitions introduce the universe of computation of the CLPS–B variables.

Definition 1 (Set). Let $\mathcal{V}$ be the set of all the variables, $\mathcal{C}$ the set of all the constants, and $\mathcal{O}$ the set of all the pairs over $\mathcal{C} \cup \mathcal{V}$ (including nested pairs). The set $S_{\mathcal{VCO}}$ is defined as follows: $S_{\mathcal{VCO}} = \mathbb{P}(\mathcal{V} \cup \mathcal{C} \cup \mathcal{O})$

Definition 2 (Computation domain). The computation domain of constraints processed in CLPS–B is defined on the set $S_{\mathcal{VCO}} \cup \mathcal{T}$.

The example 1 shows the different kinds of computation domain with B notation. Each expression is explained and we say if it is supported by the CLPS–B solver.

Example 1 (Definition of CLPS–B variables).
List of CLPS–B expressions:

- explicit set: $X \in \{1, 2, 3\}$,
- type: $X \in \mathbb{N}$, because $(\mathbb{N} \in \mathcal{T})$,
- type: $X \subset \mathbb{N}$, because $(\mathbb{N} \in \mathcal{T})$,
- Cartesian product (pairs): $X \in \{1, 2, 3\} \times \{4, 5, 6\}$,
- Set of pairs: $X \in \{(1, 4), (1, 5) \ldots (3, 5), (3, 6)\}$,
- Set defined by comprehension (explicit domain): $\{X \in \mathbb{N} \mid X \leq 3 \wedge X \geq 0\}$.

List of non CLPS–B expressions:

- Set of sets: $X \in \{\{1, 2, 3\}, \{4, 5, 6\}\}$,
- Infinite set: $\{X \in \mathbb{N} \mid X \geq 3\}$.

For each variable of the system, its domains, managed by the CLPS–B solver, is defined by constraints.

2.3 Substitution

The B notation describes actions in the operations by substitution of the state variables. Here, only the definition of a simple substitution is given. The reader can find all other substitution definitions in the B-Book [Abr96].

Definition 3 (Substitution). Let x be a variable, E an expression and F a formula, $[x := E]F$ is the **substitution** of all free occurrences of x in F by E.

Example 2. The result of transformation by substitution of the `swap` operation of the process scheduler is:

```
(active ≠ {}) ∧
   (waiting' := waiting ∪ active) ∧
   (waiting' ⊆ PID) ∧
   (((ready = {}) ∧ (active = {})) ∨
      (¬ (ready = {}) ∧ (@pp.(pp∈ready) ⇒
         (active' := {pp}) ∧ (active' ⊆ PID) ∧
         (ready' := ready - {pp}) ∧
         (ready'⊆ PID)))
```

B Language Operator	Notation	CLPS–B constraint
conjunction	$\wedge$	&
disjunction	$\vee$	or
negation	$\neg$	not
implication	$\Rightarrow$	=>
equivalence	$\Leftrightarrow$	<=>
universal quantification	$\forall$	!
existential quantification	$\exists$	#
comprehension set	$\{\mid\}$	$\{x, y, z\}$ extension set
extension set	$\{x, y, z\}$	$\{x, y, z\}$
empty set	$\{\}$	$\{\}$
set of relation	$\leftrightarrow$	<->
inverse of a relation	$^{-1}$	~
domain of a relation	dom	dom
less than or equal	$\leq$	<=
less than	$<$	<
greater than or equal	$\geq$	>=
greater than	$>$	>
adder	$+$	+
subtracter	$-$	-
multiplier	$*$	*
divisor	$/$	/

B language Operator	Notation	CLPS–B Constraint
range of relation	ran	ran
composition of two relations	; o	; $circ$
identity relation	id	id
domain restriction	$\lhd$	<|
range restriction	$\rhd$	|>
domain subtraction	$\lhd\!\!\!-$	<<|
range subtraction	$\rhd\!\!\!-$	|>>
range of a set under a relation	$[\,]$	[ ]
overriding relation by another	$\leftarrow$	<+
direct product of two relations	$\otimes$	><
parallel product of two relations	$\parallel$	||
first projection	prj_1	prj1
second projection	prj_2	prj2
application of a function	$(\,)$	()
functional abstraction	$\lambda\,.(\mid)$	★
set of partial functions	$\nrightarrow$	+->
set of total functions	$\rightarrow$	-->
set of partial injections	$\rightarrowtail$	>+>
set of total injections	$\rightarrowtail$	>->
set of partial surjections	$\twoheadrightarrow$	+->>
set of total surjections	$\twoheadrightarrow$	-->>
set of partial bijections	$\rightarrowtail\!\!\!\!\twoheadrightarrow$	>+>>
set of total bijections	$\rightarrowtail\!\!\!\!\twoheadrightarrow$	>->>

Table 1. *List of operators in B notation and CLPS–B constraints. The symbol ★ means that the operator is not implemented.*

2.4 Coverage of the B Notation

The coverage of B set operators is high. More than 80% of set operators are supported (Tables 1 and 2). The main integer primitives are implemented using integer finite domain propagation rules [Tsa93] in order to express properties of set cardinality and basic arithmetic operation. There are two reasons of the limitation of the operator coverage: the first concerns the finite tree structures and finite sequences that are not supported. They represent about 15% of the operators. The second is due to the set restriction allowing to use infinite sets $\mathcal{T}$ only to type variables. Int the end of the paper, we only consider the part of B operators covered by the CLPS–B solver.

B language Operator	Notation	CLPS–B Constraint $S_{\mathcal{VCO}}$	CLPS–B Constraint $\mathcal{T}$
membership	$\in$	ins	ins
Cartesian product	$\times$	x	★
set of subsets of a set	$\mathbb{P}$	p_partie	★
set of non-empty subsets of a set	$\mathbb{P}_1$	p1_partie	★
set of finite subsets of a set	$\mathbb{F}$	f_partie	★
set of finite non-empty subsets of a set	$\mathbb{F}_1$	f1_partie	★
inclusion of one set in another	$\subseteq$	sub	sub
union of two sets	$\cup$	union	★
intersection of two sets	$\cap$	inter	★
difference of two sets	$-$	differ	★
non membership	$\notin$	nin	nin
equality	$=$	$=$	$=$
inequality	$\neq$	neq	neq
set equality	$=_S$	eqS	eqS
set inequality	$\neq_S$	neqS	neqS
cardinality of a set	$\#$	card	card

Table 2. *In CLPS–B solver, the set operators are different on explicit $S_{\mathcal{VCO}}$ sets and $\mathcal{T}$ universe sets. Note that only the operators on sets of sets are not implemented(★).*

2.5 Translating B Expressions into Constraint System

In CLPS–B, all the set relations are rewritten into an equivalent system with constraints $\in$, $=$, $\neq$ and $card()$ (cardinality of set) as shown in Table 3 where A, B are sets of $S_{\mathcal{VCO}}$, x and y are elements of $\mathcal{V} \cup \mathcal{C} \cup \mathcal{O}$, and s and r are relations. This translation uses rules or axioms of logic and the semantics of B operators [Abr96].

Example 3. Set constraint transformation: $\{x_1, x_2\} =_S \{y_1, y_2\}$ is rewritten into
$x_1 \in \{y_1, y_2\} \wedge x_2 \in \{y_1, y_2\} \wedge y_1 \in \{x_1, x_2\} \wedge y_2 \in \{x_1, x_2\}$
The domain of each variable is defined with the element of the other set..

The constraints are rewritten into normal disjunctive form and each disjunction is explored by a separate prolog choice point.

Definition 4 (Domain of constraints). We call Ω the **Domain of constraints**. It is the set of all the constraints over $S_{\mathcal{VCO}} \cup \mathcal{T}$. Also, the set constraints over $\mathcal{VCO}$ is called $\Omega_{\mathcal{VCO}}$ and the set of constraints over $\mathcal{T}$ is called $\Omega_{\mathcal{T}}$.

Remark 1. we do not translate all the system at once, but each predicate separately. So, the computation is very fast, because we do not have to calculate the disjunctive normal form of the whole specification.

Theorem 1 (Validity). *The set of constraints obtained after rewriting is semantically equal to the system given by the B specification.*

Proof. All logic identities used (table 3) are the definitions given and proved in the B-Book [Abr96]. The rewriting process always terminates because there is no recursion in the definitions. The consistency of operator definitions ensures the soundness of the method and the termination property ensures its completeness.

Terminology	Operator	Definition
membership	$x \in A$	CLPS–B primitive
not member	$x \notin A$	$\{y \mid y \in A \wedge x \neq y\}$
equality	$x = y$	CLPS–B primitive
not equality	$x \neq y$	CLPS–B primitive
subset	$A \subseteq B$	$A \in \mathbb{P}(B)$
set equal	$A =_S B$	$A \subseteq B \wedge B \subseteq A$
set not equal	$A \neq_S B$	$card(A) \neq card(B) \vee \exists x(x \in A \wedge x \notin B) \vee \exists x(x \notin A \wedge x \in B)$
cup	$A \cup B$	$\{x \mid x \in A \vee x \in B\}$
cap	$A \cap B$	$\{x \mid x \in A \wedge x \in B\}$
set minus	$A \setminus B$	$\{x \mid x \in A \wedge x \notin B\}$
cardinality	$card(A)$	CLPS–B primitive
identity	$id(A)$	$\{(x, x) \mid x \in A)\}$
reverse	r^{-1}	$\{(y, x) \mid (x, y) \in r\}$
domain	$dom(r)$	$\{x \mid \exists y((x, y) \in r)\}$
range	$ran(r)$	$\{y \mid \exists x((x, y) \in r)\}$
Terminology	**Expression**	**Definition**
restriction of:		
domain	$A \lhd r$	$\{(x, y) \mid (x, y) \in r \wedge x \in A\}$
range	$s \rhd B$	$\{(x, y) \mid (x, y) \in s \wedge y \in B\}$
subtraction of:		
domain	$A \lhd\!\!\!- r$	$\{(x, y) \mid (x, y) \in r \wedge x \notin A\}$
range	$s \rhd\!\!\!- B$	$\{(x, y) \mid (x, y) \in s \wedge y \notin B\}$
overriding	$s \Leftarrow r$	$\{(x, y) \mid (x, y) \in s \wedge x \notin dom(r) \vee (x, y) \in r\}$
relation	$s \leftrightarrow r$	$\mathbb{P}(s \times r)$
set of partial:		
function	$s \nrightarrow r$	$\{f \mid f \in s \leftrightarrow r \wedge (f^{-1}, f) \subseteq id(r)\}$
injection	$s \rightarrowtail r$	$\{f \mid f \in s \nrightarrow r \wedge f^{-1} \in s \nrightarrow r\}$
subjection	$s \twoheadrightarrow r$	$\{f \mid f \in s \nrightarrow r \wedge ran(f) = r\}$
bijection	$s \rightarrowtail\!\!\!\!\twoheadrightarrow r$	$s \rightarrowtail r \cap s \twoheadrightarrow r$
set of total:		
function	$s \rightarrow r$	$\{f \mid f \in s \nrightarrow r \wedge dom(f) = s\}$
injection	$s \rightarrowtail r$	$s \rightarrowtail r \cap s \rightarrow r$
subjection	$s \twoheadrightarrow r$	$s \twoheadrightarrow r \cap s \rightarrow r$
bijection	$s \rightarrowtail\!\!\!\!\twoheadrightarrow r$	$s \rightarrowtail r \cap s \twoheadrightarrow r$

Table 3. B set operators and their CLPS–B definitions

Example 4. Rewritten predicates of the process scheduler invariant:

B Invariant	CLPS–B Format
$active \subseteq PID \wedge$	$active \in \mathbb{P}(PID) \wedge$
$ready \subseteq PID \wedge$	$ready \in \mathbb{P}(PID) \wedge$
$waiting \subseteq PID \wedge$	$waiting \in \mathbb{P}(PID) \wedge$
$ready \cap waiting = \{\} \wedge$	$\{x \mid x \in ready \wedge x \in waiting\} = \{\} \wedge$
$active \cap (ready \cup$	$\{x \mid x \in active \wedge x \in \{z \mid z \in ready$
$waiting) = \{\} \wedge$	$\vee z \in waiting\}\} = \{\} \wedge$
$card(active) \leq 1 \wedge$	$card(active) \leq 1 \wedge$
$(active = \{\}) \Rightarrow (ready = \{\})$	$card(active) = 0 \Rightarrow card(ready) = 0$

2.6 Constraint Management

The constraint system $\Omega_{\mathcal{VCO}}$ presents some characteristics of the Constraint Satisfaction Problem (CSP). In the CSP, each variable is associated with a domain defined in the set $\mathcal{C}$ of constants. A domain D_x of a variable x is a finite set of possible values which can be assigned to x. Formally, a CSP is denoted by a triplet (V, D, C) where:

- V is a finite set of variables $\{V_1, \ldots, V_n\}$,
- D is a set of domains, D_x for each $x \in V$,
- C is a finite set of constraints on V.

Unlike ordinary CSP, the variables of $\Omega_{\mathcal{VCO}}$ can have several domains D_i^x containing elements of $\mathcal{C} \cup \mathcal{V} \cup \mathcal{O}$ and defined by the constraints $x \in D_i^x$. The resulting domain of x is given by the intersection of the domain $D^x = \bigcap_i D_i^x$. The major difference to CSP is that each D_i^x may contain variables as well as values, whereas in CSP each D_i^x only contains values. Note that $\bigcap_i D_i^x$ cannot always be deterministically computed when the domains D_i^x contain variables. This problem is called V-CSP by analogy with CSP. In the case of all the V-CSP domains contain no variable, it is reduced to a CSP.

Definition 5 (V-domain). A **V-domain** $D_x = \bigcap_i D_i^x$ of a variable x is a finite set of the possible elements (variables or constants) which can be assigned to x. Thus, D_x is included in $\mathcal{C} \cup \mathcal{V} \cup \mathcal{O}$.

Initially, a V-domain D_x is defined by the constraint $x \in D_i^x$. Then, it is modified by the propagation rules. The V-Domain number, n_x, is the number of subdomains D_i^x. It increases when new constraints $x \in D_i^x$ are added, and may decrease when simplification rules are applied.

Definition 6 (V-label). A **V-label** is a pair $< x, v >$ that represents the assignment of the variable x.

The V-label $< x, v >$ is meaningful if v is in a V-domain of x. Note that v can be either a constant or a variable. The concept of V-CSP can also be introduced and used to resolve the constraints on $S_{\mathcal{VCO}}$ by:

Definition 7 (V-CSP). A **V-CSP** is defined by a triplet (V, D, C) where:

- V is a finite set of variables $\{V_1, \ldots, V_n\}$,
- D is a set of V-domains, $\{D_1, \ldots, D_n\}$,
- C is a finite set of constraints of the form $V_i \neq V_j$, where $V_i, V_j \in V$.

Remark 2. In CSP, D can be seen as a function which links a variable of V with a domain. In V-CSP, it is a relation because each variable x can have several domains D_i^x . Moreover, in contrast to CSP, the variables V of a V-CSP can appear in the domains.

2.7 Consistency and Satisfiability

Finally, the definitions of satisfiability and consistency of the constraint system $\Omega_{\mathcal{VCO}}$ have to be extended from CSP to V-CSP by using the V-label procedure instead of the labelling procedure traditionally used with CSP.

Definition 8 (Satisfiability). A V-CSP, noted $(V = \{V_1, \ldots, V_n\}, D, C)$ is **satisfiable** if and only if there is a subset $\mathcal{B} \subseteq D$, called the V-base of V-CSP, and a set of V-label $\mathcal{L} = (< V_1, B_1 >, \ldots, < V_n, B_n >)$ with $B_i \in \mathcal{B} \wedge B_i \subseteq D_i$ such that all the constraints of C can be rewritten with:

1. $B_i \in \{B_1, \ldots, B_i, \ldots, B_k\}$ with $B_i \in \mathcal{B} \wedge B_1 \in \mathcal{B} \wedge \ldots \wedge B_k \in \mathcal{B}$
2. $B_i \neq B_j$ with $i \neq j \wedge B_i \in \mathcal{B} \wedge B_j \in \mathcal{B}$.

Remark 3. constraints like (1) are trivially satisfied. B_i can be a value or a variable with a domain used during the labelling procedure. To simplify in the following, we note the set of sets $\mathcal{B}$ as a set.

Example 5. Given the constraint systems on variables:

- $x_1 \in \{y_1, y_2\} \wedge x_2 \in \{y_1, y_2\} \wedge x_3 \in \{y_1, y_2\} \wedge y_1 \neq y_2 \wedge x_1 \neq x_2 \wedge x_1 \neq x_3 \wedge x_2 \neq x_3$
  It is not satisfiable because there is no V-base or a V-label to make constraints like (1) or (2). If we take $\mathcal{B} = \{y_1, y_2\}$ and $\mathcal{L} = (< x_1, y_1 >, < x_2, y_2 >, < x_3, y_1 >)$, we obtain $y_1 \neq y_1$ (it is due to the constraint $x_1 \neq x_3$).
- $x_1 \in \{y_1, y_2\} \wedge x_2 \in \{y_1, y_2\} \wedge y_1 \neq y_2 \wedge x_1 \neq x_2$
  It is satisfiable. Because all the results lead to $\mathcal{B} = \{y_1, y_2\}$ and $\mathcal{L} = (< x_1, y_1 >, < x_2, y_2 >)$, the resulting system is only defined by constraints like (1) and (2): $y_1 \in \{y_1, y_2\} \wedge y_2 \in \{y_1, y_2\} \wedge y_1 \neq y_2$.

Definition 9 (Consistency). A V-CSP, noted $(V = \{V_1, \ldots, V_n\}, D, C)$ is **consistent** if and only if the two following conditions are verified:

1. $\forall\, i((V_i, D_{V_i}) \in D \Rightarrow \exists\, j(V_j \in D_{V_i} \wedge (V_j \neq V_i) \notin C))$
2. $\forall\, i(V_i \neq V_i) \notin C$

In other words, the domain D_V of a variable V is consistent if and only if there is an element e in this domain and $e \neq V$ is not a constraint of the specification. Arc-consistency is also performed in the constraint graph where the nodes represent variables and the edges represent the constraints $\neq$ (Example 7).

Example 6. An inconsistent constraint system:
$x_1 \in \{y_1, y_2\} \wedge y_1 \neq y_2 \wedge x_1 \neq y_1 \wedge x_1 \neq y_2$

Theorem 2. *A satisfiable constraint system on $S_{\mathcal{VCO}}$ is consistent.*

Proof. by negation,
Let S be an inconsistent V-CSP $(\{V_1, \ldots, V_n\}, D, C)$, $\mathcal{B} = \{B_1, \ldots, B_m\}$ a V-Base and $\{< V_1, B_{j_1} >, \ldots, < V_n, B_{j_n} >\}$ the V-label with $j_1, \ldots, j_n \in \{1, \ldots, m\}$. Two cases are possible according to definition 9:

1. S inconsistent with the point 1 of definition 9, we have $\exists\, i((V_i, D_{V_i}) \in D \wedge \forall\, j(V_j \notin D_{V_i} \vee (V_j \neq V_i) \in C))$. We replace the V variables by their range (value or variable) in the V-label, we obtain $\exists\, i((B_{j_i}, D_{V_i}) \in D \wedge \forall\, k(B_{j_k} \notin D_{V_i} \vee (B_{j_k} \neq B_{j_i}) \in C))$. If it is true for all k, it is true for $k = i$ and we have $(B_{j_i} \notin D_{V_i} \vee (B_{j_i} \neq B_{j_i}) \in C)$. The first disjunctive is not satisfiable by definition 8: B_{j_i} is an element of domain D_{V_i}. It is the same for the second because an element can not be different to itself. So S is non satisfiable.
2. S inconsistent with the point 2 of definition 9, we have $\exists\, i(V_i \neq V_i) \in C \Rightarrow \exists\, i(B_{j_i} \neq B_{j_i}) \in C$. It is the negation of the point 2 of definition 8. So S is non satisfiable.

Thus, in both cases, S inconsistent $\Rightarrow$ S non satisfiable. □

Remark 4. The reciprocal is not true: for example, the following constraint system is consistent but not satisfiable:

$$x_1 \in \{y_1, y_2\} \wedge x_2 \in \{y_1, y_2\} \wedge x_3 \in \{y_1, y_2\} \wedge y_1 \neq y_2 \wedge x_1 \neq x_2 \wedge x_1 \neq x_3 \wedge x_2 \neq x_3$$

The inconsistency of a system can also be detected by the constraints $x \in \{\}$ and $x \neq x$. These concepts define the formal framework to resolve the $\Omega_{\mathcal{VCO}}$ system.

The correctness of the reduction procedure is ensured by two points: deleted values in the domains are inconsistent values (see rule P_1 below), and deleted constraints are trivially satisfied (see rules P_2 and P_3 below).

The reduction procedure does not ensure the assignment of the variables to an element of the domain, and is thus not complete. The completion can also be performed by a generation procedure, which is a variation of the *forward − checking* algorithm [Nad89].

2.8 Inference Rules

The notion of consistency establishes the conditions which the elements of a domain must satisfy. If the consistency is not verified, the domain is reduced, i.e. elements are deleted in order to make it consistent. In the following, the element e_i belongs to $\mathcal{V} \cup \mathcal{C} \cup \mathcal{O}$ and τ belongs to $\mathcal{T}$. The notation $\Omega \cup \{C_1, C_2, \ldots, C_n\}$ describes the conjunction of the current constraint system Ω and the constraints $C_1, C_2, \ldots, C_n$. Ω is divided into two subsets $\Omega_{\mathcal{VCO}}$ and $\Omega_{\mathcal{T}}$ which respectively correspond to the constraints on the elements of $\mathcal{V} \cup \mathcal{C} \cup \mathcal{O}$ and $\mathcal{T}$.
Rule P_1 ensures the consistency on $\Omega_{\mathcal{VCO}}$:

$$P_1 : \frac{\Omega \cup \{e \in \{e_1, \ldots, e_{i-1}, e_i, e_{i+1}, \ldots, e_n\}, e \neq e_i\}}{\Omega \cup \{e \in \{e_1, \ldots, e_{i-1}, e_{i+1}, \ldots, e_n\}, e \neq e_i\}}$$

The following two rules are simplification rules:

$$P_2 : \frac{\Omega \cup \{e_i \in \{e_1, \ldots, e_i, \ldots, e_n\}\}}{\Omega}$$

$$P_3 : \frac{\Omega \cup \{e \in \{e_1, \ldots, e_n\}, e \in \{e_1, \ldots, e_n, \ldots, e_{n+m}\}\}}{\Omega \cup \{e \in \{e_1, \ldots, e_n\}\}}$$

When a domain is reduced to one single variable, unification is carried out:

$$P_4 : \quad \frac{\Omega \cup \{e_i \in \{e_j\}\}}{\Omega \cup \{e_i = e_j\}}$$

Two additional inference rules describe cases where the constraint system $\Omega_{\mathcal{VCO}}$ is inconsistent:

$$P_5 : \frac{\Omega \cup \{e \in \{\}\}}{\mathit{fail}} \quad P_6 : \frac{\Omega \cup \{e \neq e\}}{\mathit{fail}}$$

The following inference rule describes the case where the constraint system $\Omega_{\mathcal{T}}$ is inconsistent:

$$T_1 : \frac{\Omega \cup \{e \in \tau, e \notin \tau\}}{fail}$$

Rule T_2 infers a new constraint on $\Omega_{\mathcal{VCO}}$ from the system $\Omega_{\mathcal{T}}$:

$$T_2 : \quad \frac{\Omega \cup \{e_i \in \tau, e_j \notin \tau\}}{\Omega \cup \{e_i \in \tau, e_j \notin \tau, e_i \neq e_j\}}$$

These inference rules are used until a fixed point is obtained, i.e. until no rules can be applied.

One of the problems of the B expression evaluation lies in the verification of the invariant properties. States computed by operations are represented by a set of constraints. The verification of the invariant is computed on this constraint system by a subsumtion test. Let E be the constraint system representing the B machine state and φ_I the invariant. The invariant properties are verified if and only if:

$$E \Rightarrow \varphi_I \text{ THEN } E \supseteq \varphi_I \ (1)$$

In order to ensure the efficiency of the inclusion test(1), four additional inference rules, which generate new constraints, are now defined:

– on $\mathcal{T}$ (where τ is a set of $\mathcal{T}$)

$$T_3 : \frac{\Omega \cup \{e \in \{e_1, \ldots, e_n\}, e_1 \in \tau, \ldots, e_n \in \tau\}}{\Omega \cup \{e \in \{e_1, \ldots, e_n\}, e_1 \in \tau, \ldots, e_n \in \tau, e \in \tau\}}$$

$$T_4 : \frac{\Omega \cup \{e \in \{e_1, \ldots, e_n\}, e_1 \notin \tau, \ldots, e_n \notin \tau\}}{\Omega \cup \{e \in \{e_1, \ldots, e_n\}, e_1 \notin \tau, \ldots, e_n \notin \tau, e \notin \tau\}}$$

– on $S_{\mathcal{VCO}}$ (where *set* is a $S_{\mathcal{VCO}}$ set)

$$P_7 : \frac{\Omega \cup \{e \in \{e_1, \ldots, e_n\}, e_1 \in set, \ldots, e_n \in set\}}{\Omega \cup \{e \in \{e_1, \ldots, e_n\}, e_1 \in set, \ldots, e_n \in set, e \in set\}}$$

$$P_8 : \frac{\Omega \cup \{e \in \{e_1, \ldots, e_n\}, e_1 \neq k, \ldots, e_n \neq k\}}{\Omega \cup \{e \in \{e_1, \ldots, e_n\}, e_1 \neq k, \ldots, e_n \neq k, e \neq k\}}$$

Justification. These rules are used to transmit the common properties of an explicit set *ens* to an element x with $x \in ens$. For example, if the type of all the elements of *ens* is T, then a variable which belongs to *ens* is also of type T.

Example 7. Given the following system:

$$x_0 \in \{x_1, x_2, x_3\} \wedge x_0 \in \{x_1, x_2, x_4\} \wedge$$
$$x_5 \in \{x_3, x_4\} \wedge x_0 \neq x_5$$

When the constraint $x_3 \neq x_5$ is added to the system, the rules infer the following reductions (the modified constraints have an underline):

– $x_0 \in \{x_1, x_2, x_3\} \wedge x_0 \in \{x_1, x_2, x_4\} \wedge \underline{x_5 \in \{x_3, x_4\}} \wedge x_0 \neq x_5 \wedge \underline{x_3 \neq x_5}$

$\xRightarrow{P_1}$ $x_0 \in \{x_1, x_2, x_3\} \wedge x_0 \in \{x_1, x_2, x_4\} \wedge \underline{x_5 \in \{x_4\}} \wedge x_0 \neq x_5 \wedge x_3 \neq x_5$, domain of x_5 is reduced.

$\xRightarrow{P_4}$ $x_0 \in \{x_1, x_2, x_3\} \wedge \underline{x_0 \in \{x_1, x_2, x_4\}} \wedge \underline{x_0 \neq x_4} \wedge x_3 \neq x_4$, x_5 is unified to the unique variable x_4 of its domains. x_5 disappear of the constraint system because all x_5 occurrences are replaced by x_4.

$\xRightarrow{P_1}$ $\underline{x_0 \in \{x_1, x_2, x_3\}} \wedge x_0 \in \{x_1, x_2\} \wedge x_0 \neq x_4 \wedge x_3 \neq x_4$, domain of x_0 is reduced.

$\xRightarrow{P_3}$ $x_0 \in \{x_1, x_2\} \wedge x_0 \neq x_4 \wedge x_3 \neq x_4$, remove the largest domain of x_0

The reduced system is consistent and satisfiable, and offers two solutions:

1. x_0 is unified to the variable x_1:
   $\mathcal{L} = (< x_0, x_1 >, < x_5, x_4 >)$
   $\mathcal{B} = \{x_1, x_2, x_3, x_4\}$
   $C = (x_1 \neq x_4 \wedge x_3 \neq x_4)$
2. x_0 is unified to the variable x_2:
   $\mathcal{L} = (< x_0, x_2 >, < x_5, x_4 >)$
   $\mathcal{B} = \{x_1, x_2, x_3, x_4\}$
   $C = (x_2 \neq x_4 \wedge x_3 \neq x_4)$

3 Animation of B machines with CLPS–B

This part describes the constrained evaluation process of specification called animation. It consists in resolving set logical B formulas with CLPS–B solver. This process manages the evolution of the constrained state when operations are executed (the first is called from the initial state of the B machine).

Definition 10 (Constrained state). A **constrained state** is a pair (V, C_V) where V is a set of state variables of the specification, and C_V is a set of constraints based on the state variables of the specification.

The constrained evaluation models the evolution of the B machine state. It changes one constrained state to another by executing operations.

Definition 11 (Constrained evaluation). Given a constrained state (V, C_V) and φ constraints of the specification. The **constrained evaluation** is a relation called $\mathcal{EVAL}$, which associates a constrained state to the next constrained state:

$$\mathcal{EVAL} : (V, C_V) \mapsto (V', C_V \wedge \varphi)$$

where V' represents state variables V after substitution calculation φ.

More accurately, three procedures, based on the calculus of logical set B formula, have been defined to perform this evaluation. These procedures can establish preconditions, compute substitutions and verify the invariant properties. The CLPS–B solver ensures the reduction and propagation of constraints given by the B specifications.

3.1 Activating an Operation

From the initial state, any operation can be activated. An activation consists in verifying the preconditions of the operation, computing substitutions and verifying the invariant properties for the different computed states. CLPS–B evaluates each substitution, with eventual choice points, which give one or more new generated states. If the preconditions are not verified then the operation is not activated and no state is generated. This process is shown in Section 4.1.

3.1.1 Precondition Processing

The operation preconditions are defined by a constraint set based on specification variables and local operation variables. Given the constrained state of the specification $\theta = (V, C_V)$ and φ_{pre} precondition constraints, the processing of preconditions adds the constraints φ_{pre} to C_V. The result is a system of constraints reduced by the CLPS–B solver, where Red^i represents the i^{th} rewritten constraints:

$$\bigcup_i Red^i(C_V \cup \varphi_{pre})$$

Finally, the processing of preconditions changes the constrained state θ to the constrained states $\theta_i^{pre} = (V, Red^i(C_V \cup \varphi_{pre}))$.

The operation is activated from θ_i^{pre} if the constraint system $Red^i(C_V \cup \varphi_{pre})$ is satisfiable. To ensure satisfiability, a solution can be generated. Only the satisfiable states θ_i^{pre} are retained to activate operations. These are called activation states.

3.1.2 Substitution Computation

φ_{sub} are the constraints induced by the substitutions. φ_{sub} incrementally builds a constraint model over the state variables. Thus, each state variable is always represented by a CLPS–B variable introduced by the last constrained substitution.

Substitution is computed by reduction of the constraint system $C_V \cup \varphi_{sub}$. As in the precondition processing of φ_{sub}, the reduction can introduce choice points. Thus, substitution computation φ_{sub} can change the constrained state $\theta = (V, C_V)$ to the constrained states $\theta_i^{sub} = (V', Red^i(C_V \cup \varphi_{sub}))$ such as $\forall X' \in V', (X' \in V \vee (X' = exp \wedge X \in V))$. A solution can be generated to verify satisfiability of each resulting state. Only the satisfiable states θ_i^{sub} are retained.

3.1.3 Invariant Verification

The goal of this stage is to validate the constrained state $\theta = (V, C_V)$ given by the invariant $\varphi_I(V)$. The verification is performed by an inclusion test in the constraint graph.

The procedure $\mathcal{P}_{NP}$ gives a disjunction of the system of constraints given by φ_I. It is called $\bigvee_i \varphi_I^i$. The invariant is verified if:

$$\exists\, i.(C_V \Rightarrow \varphi_I^i(V))) \Leftrightarrow \exists\, i.(\varphi_I^i(V) \subseteq C_V))$$

Assuming the inclusion test $\varphi_I^i(V) \subseteq C_V$ (that means all the constraints of C_V subsume the constraints of the invariant) does not allow isomorphism of the sub-graph, the variables of the constrained state V verify the invariant. In this case, there is no advantage in using the constrained state instead of the concrete state. Efficiency of constrained evaluation is based on the minimization of the number of enumerated constrained states.

3.1.4 Synthesis

An operation can produce a model from a pair $(\varphi_{pre}^i, \varphi_{sub}^i)$ where φ_{pre}^i are the precondition constraints and φ_{sub}^i are the substitution constraints. Evaluation of the operation $(\varphi_{pre}, \varphi_{sub})$ changes the system from a constrained state $\theta_i = (V, C_V)$ to a constrained state $\theta_{i+1} = (V', C_V \cup \varphi_{pre} \cup \varphi_{sub})$. Initially, state $\theta_i^{pre} = (V, C_V \cup \varphi_{pre})$ is computed with the preconditions. In the second stage, the state $\theta_{i+1} = (V', C_V \cup \varphi_{pre} \cup \varphi_{sub})$ is computed by adding the substitution constraints. In the last stage, the verification of the invariant $\varphi_I(V')$ is performed with the state θ_{i+1}.

3.2 Complexity

In CLPS–B, the constraint satisfaction is based on the following facts:

- an element can possess several domains,
- a domain can possess variables,
- adding a constraint can generate new constraints $\in$, $\notin$, $\neq$.

3.2.1 The $S_{\mathcal{VCO}}$ Constraints:

Adding a new constraint ($\in$ or $\neq$) implies, by propagation, the creation of other constraints ($\in$ and $\neq$). In the worst case, propagation generates **n.(n-1)/2** new constraints of difference, if all the variables are different from each other. This complexity is theoretical and, in practice, the number of the system variables are linear.

Property 1 (Number of membership constraints) *Given a V-CSP composed of n variables, d is the size of the largest domain and n_d is the highest number of variable domains. The maximum number of membership constraints inferred is* $\mathbf{n^2 \times n_d \times d}$.

Proof (Outline of proof). The membership constraints are inferred by propagation given by the P_1 and P_7 rules.

P_7 adds an element e to the common domain of the other elements. This rule does not create a new domain in the system.

P_1 substitutes a new domain exp' to exp with the relation: $exp' \subset exp \wedge (\#exp' = \#exp - 1)$. Thus, for each domain, d new membership constraints can be generated.

For a variable in the worst case, the number of inferred membership constraints is equal to the $number\_of\_maximum\_domain * size\_of\_domain$, i.e. $(n * n_d) * d$. Finally, the number of inferred membership constraints for all variables is limited by $n^2 \times n_d \times d$. □

3.2.2 The $\mathcal{T}$ constraints:

Given a number of set variables n_v and a number of elements n_e, the worst case is $\mathbf{n_e} \times \mathbf{n_v}$ membership constraints and no membership constraint is generated by propagation.

4 Application to the Process Scheduler

The B abstract machine example is the process scheduler introduced in Section 1.2.

The first part presents the constrained evaluation of the sequence of operations: NEW(PP1), NEW(PP2), READY(RR1), where PP1, PP2 and RR1 are the input variables of the operations.

The second part deals with comparison between concrete and constrained reachability graphs of the process scheduler example.

4.1 Constrained Evaluation

This part shows the evolution of the constrained state in a CLPS–B operation evaluation process. This evolution is described in Table 4, whereas Fig. 3 presents the interface of animation during this short example. Only CLPS–B reduced constraints are added to the store.

In order to evaluate the first and the second operations, the computation is deterministic and does not use any new variables. The last operation, defined by the formula $waiting^{(3)} = \{PP1, PP2\} \setminus \{RR1\}$, is deterministic because $RR1 \in \{PP1, PP2\}$. The result $waiting^{(3)}$ is a set of a single element: $waiting^{(3)} = \{PP3\}$ with the constraints $PP3 \in \{PP1, PP2\} \wedge PP3 \neq RR1$.

Finally, the variables of the specification (*waiting*, *ready* and *active*) are represented by their latter valuations, respectively $waiting^{(3)}$, *ready* and $active'$. The constraints describe a set of properties linked to variables.

The verification by entailment of the invariant properties is detailed for the constrained state obtained after the $READY(RR1)$ operation:

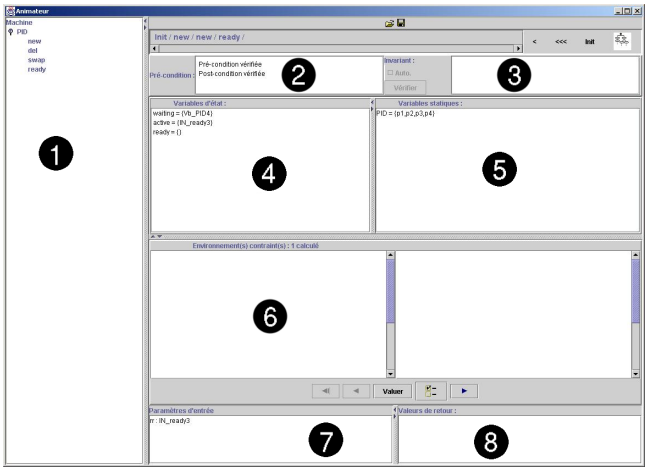


Fig. 3. IHM of the Animation tool for the PID example and the sequence operations INIT,NEW,NEW,READY. ① list of the operations, ② message about validation of precondition and postcondition of the activated operation,③ message about verification of the invariant, ④ state variables, ⑤ sets of abstract machine, ⑥ constraints store, ⑦ and ⑧ input and output parameters of the activated operation

1. $active \subseteq PID$ becomes $\{RR1\} \subseteq PID$. This constraint is written as $RR1 \in PID$ which entails $RR1 \in \{PP1, PP2\}$, $PP1 \in PID$ and $PP2 \in PID$ (rule T_3).
2. $ready \subseteq PID$ becomes $\{\} \subseteq PID$. This constraint is always satisfied.
3. $waiting \subseteq PID$ becomes $\{PP3\} \subseteq PID$. This constraint is written as $PP3 \in PID$ which entails $PP3 \in \{PP1, PP2\}$, $PP1 \in PID$ and $PP2 \in PID$ (rule T_3).
4. $ready \cap waiting = \{\}$ becomes $\{\} \cap \{PP3\} = \{\}$. This constraint is always satisfied.
5. $active \cap (ready \cup waiting) = \{\}$ becomes $\{RR1\} \cap \{PP3\} = \{\}$. $\{RR1\} \cap \{PP3\}$ computes to $\{\}$ and is satisfied.
6. $card(active) \leq 1$ becomes $card(\{RR1\}) \leq 1$ and $1 \leq 1$. This constraint is always satisfied.
7. $(active = \{\}) \Rightarrow (ready = \{\})$ becomes $(\{RR1\} = \{\}) \Rightarrow (\{\} = \{\})$. The constraint $\{RR1\} \neq \{\}$ is satisfied because the two sets have different sizes. So $(\{RR1\} \neq \{\}) \vee (\{\} = \{\})$ is satisfied.

All the invariant constraints are satisfied or are entailed by the constrained state. Thus, no generation stage is needed to ensure the satisfiability of the constrained state.

Constrained evaluation makes it possible to animate B abstract machine by preserving non-determinism and without using concrete arguments. For the creation of a new process, a variable defined on set *PID* is used instead of a specific one of the set *PID*: the new process does not need to be valued.

In the example, we can apply activation to any of the processes without specifying it. Moreover, for each computed state, the invariant properties of the system can be verified by entailment (without enumeration).

	Operation		CLPS–B constraint store	
			B Variables	**Other constraints**
INIT	SUB	$waiting = \{\}$ $ready = \{\}$ $active = \{\}$	$waiting = \{\}$ $ready = \{\}$ $active = \{\}$	
NEW(PP1)	PRE	$PP1 \in PID$ $PP1 \notin \{\}$ $PP1 \notin \{\}$	$waiting' = \{PP1\}$ $ready = \{\}$ $active = \{\}$	$PP1 \in PID$
	SUB	$waiting' = \{\} \cup \{PP1\}$		
NEW(PP2)	PRE	$PP2 \in PID$ $PP2 \notin \{\}$ $PP2 \notin \{PP1\}$	$waiting'' = \{PP1, PP2\}$ $ready = \{\}$ $active = \{\}$	$PP1 \in PID$ $PP2 \in PID$ $PP1 \neq PP2$
	SUB	$waiting'' = \{PP1\} \cup \{PP2\}$		
READY(RR1)	PRE	$RR1 \in \{PP1, PP2\}$	$waiting^{(3)} = \{PP3\}$ $ready = \{\}$ $active' = \{RR1\}$	$PP1 \in PID$ $PP2 \in PID$ $PP1 \neq PP2$ $RR1 \in \{PP1, PP2\}$ $PP3 \in \{PP1, PP2\}$ $RR1 \neq PP3$
	SUB	$waiting^{(3)} = \{PP1, PP2\} \setminus \{RR1\}$ $active' = \{RR1\}$		

Table 4. Constrained evaluation

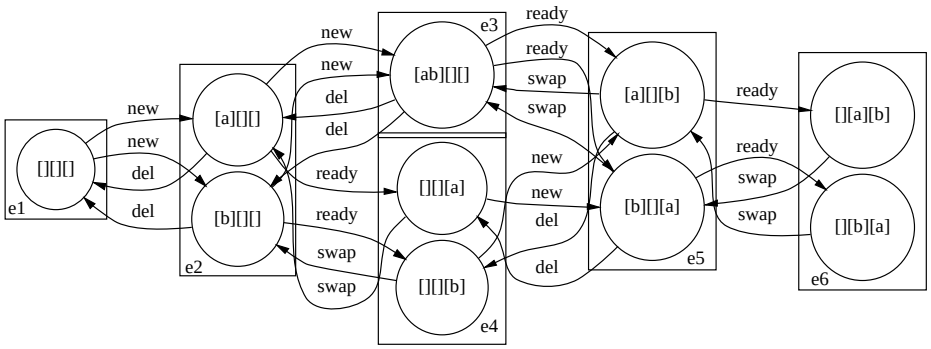


Fig. 4. Reachability concrete graph with $max = 2$

4.2 Experimental Results

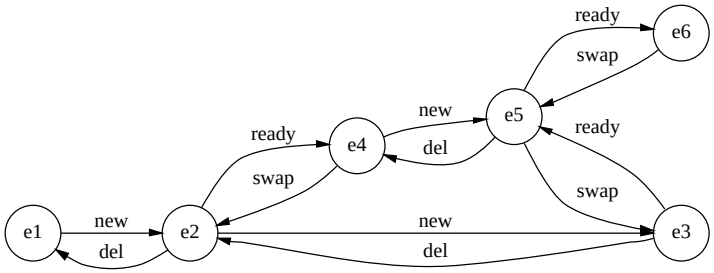
The whole reachability graph of the process scheduler B machine was built. The number of processes was bound to *max* by adding the following precondition in the *NEW* operation: $card(waiting \cup ready \cup active) < max$. Fig. 4 and Fig. 5 respectively present the constrained reachability graph and the concrete one for $max = 2$ and $PID = \{a, b\}$.

Fig. 5 shows the advantage of the constrained evaluation to build the reachable graph of a system. The number of states is dramatically reduced because one constrained state represents several concrete states. The table of Fig. 5 summarizes, according to the *max* number of parameters, the evolution of the state number in the reachability graph of the process scheduler example.

5 Test generation

The CLPS–B constraint solver is used for test generation [LPU02a, LPU02b]. In [LPU02b], we presented a new approach for formal-specification-based test generation called B-Testing-Tools (B-TT). This method is based on the CLPS–B solver, and makes intensive use of a set-oriented constraint technology. Its goal is to test every operation of the system at every *boundary state* using all *input boundary values* of that operation. The unique features of the B-TT method are the following:

- takes a B abstract machine as input;
- avoids the construction of a complete finite state automaton (FSA) for the system;
- produces boundary-value test cases (both boundary states and boundary input values);



	number of states	
max	constrained	concrete
1	3	3
2	6	10
3	10	35
4	15	124
5	21	437
6	28	1522
7	36	5231
n	$\frac{n^2+3n+2}{2}$	$> 3^n$

Fig. 5. Constrained reachability graph with $max = 2$

– produces both negative and positive test cases;
– is fully supported by tools.

5.1 Overview of the B-TT test generation method

Our goal is to test some implementation, which is not derived via refinement from the formal model. The implementation is usually a state machine with hidden state. We specify this state machine by a B formal specification, which has a state space (consisting of several state variables) and a number of operations that modify this state.

A behavior of such a system can be described in terms of a sequence of operations (a trace) where the first is activated from the initial state of the machine. However, if the precondition of an operation is false, the effect of the operation is unknown, and any subsequent operations are of no interest, since it is impossible to determine the state of the machine. Thus, we define a positive test case to be any legal trace, i.e. any trace where all preconditions are true. A positive test case corresponds to a sequence of system states representing the value of each state variable after each operation invocation. The submission of a legal trace is a success if all the output values returned by the concrete implementation during the trace are equivalent (through a function of abstraction) to the output values returned by its specification during the simulation of the same trace (or included in the set of possible values if the specification is non-deterministic). A negative test case is defined as a legal trace plus a final operation whose precondition is false. The generation of negative test cases is useful for robustness testing.

The B-TT method consists of testing the system when it is in a *boundary state*, which is a state where at least one state variable has a value at an extremum – minimum or maximum – of its sub-domains. At this boundary state, we want to test all the possible behaviors of the specification. That is, the goal is to invoke each *update operation* with extremum values of the sub-domains of the input parameters. The test engineer partitions the operations into *update operations*, which may modify the system state, and *observation operations*, which may not.

We divide the trace constituting the test case into four subsequences:[1]

Preamble: this takes the system from its initial state to a boundary state.
Body: this invokes one update operation with input boundary values.
Identification: this is a sequence of observation operations to enable a pass/fail verdict to be assigned.
Postamble: this takes the system back to the boundary state, or to an initial state. This enables test cases to be concatenated.

The body part is the critical test invocation of the test case. Update operations are used in the preamble, body and postamble, and observation operations in the identification part.

The B-TT generation method is defined by the following algorithm, where $\{bound_1, bound_2, ..., bound_n\}$ and $\{op_1, op_2, ..., op_m\}$ respectively define the set of all boundary states and the set of all the update operations of the specification:

```
for i←1 to n          % for each boundary state
   preamble(bound_i);  % reach the boundary state
   for j←1 to m        % for each update operation
      body(op_j);      % test op_j
      identification;  % observe the state
      postamble(bound_i); % return to the
   endfor                 % boundary state
   postamble(init);    % return to the initial state
endfor
```

This algorithm computes positive test cases with valid boundary input values at body invocations. A set of one or more test cases, concatenated together, defines a test sequence. For negative test cases, the body part is generated with invalid input boundary values, and no identification or postamble parts are generated, because the system arrives at an indeterminate state from the formal model point of view. Instead, the test engineer must manually define an oracle for negative test cases (typically something like *the system terminates without crashing*).

[1] The vocabulary follows the ISO9646 standard [ISO].

After positive and negative test cases are generated by this procedure, they are automatically translated into executable test scripts, using a test script pattern and a reification relation between the abstract and concrete operation names, inputs and outputs.

6 The Test Generation Modules

The test generation method is performed in two main stages. Firstly, we compute a set of *boundary goals* from the Disjunctive Normal Form (DNF) of the operations of the B abstract machine. Secondly, the preamble is computed to reach, from the initial state, a state verifying the *boundary goal* that is called *boundary state*. Finally, the *boundary states* of the specification are used as a basis to generate test cases.

6.1 Boundary Goals Generation

For test generation purposes, the postcondition of each operation is transformed into DNF, $\bigvee_j Post_j(op)$. The DNF transformation is not making in all specification, but only computes each condition described in operation one by one. Then we project each of these disjuncts into the input state, using the formula [LPU02b]:

$$(\exists\, inputs, state', outputs \bullet Pre \wedge Post_j)$$

We call these state subsets *precondition sub-domains*. The aim of boundary goal generation is to find boundaries within each of these precondition sub-domains.

In practice, the CLPS–B solver reduces each subdomain to a set of constraints. We compute boundary goals on the basis of the partition analysis by minimization and maximization using a suitable metric function chosen by the test engineer. (e.g., minimize or maximize the sum of the cardinalities of the sets). According to the optimization function, this results in one or several minimal and maximal boundary goals for each predicate.

Given the invariant properties Inv, a precondition subdomain predicate PS_i, a vector of variables V_i which comprises all the free state variables within PS_i, and f an optimization function, the boundary goals are computed as follows:

- $BG_i^{min} = minimize(f(V_i), Inv \wedge PS_i)$
- $BG_i^{max} = maximize(f(V_i), Inv \wedge PS_i)$

The optimization function $f(V_i)$, where V_i is a vector of variables $v_1 \ldots v_m$, is defined as $g_1(v_1) + g_2(v_2) + \ldots + g_m(v_m)$, where each function g_i is chosen according to the type of the variable v_i.

For example, from the predicate PS of the process scheduler example, boundary goals BG^{min} and BG^{max} are computed with the optimization function $f(V_i) = \sum_{v \in V_i} \#v^2$. The result of constraint solving is a set of constraints on the cardinalities of the set variables *waiting*, *ready* and *active* such that: $BG^{min} = \{waiting = \{\} \wedge ready = \{\} \wedge active = \{\}\}$, and $BG^{max} = \{waiting = \{X_1\} \wedge ready = \{X_2\} \wedge active = \{X_3\}\}$ where $(\forall\, i \cdot X_i \in \{p_1, p_2, p_3, p_4\})$ and $(\forall\, i \cdot i \neq j \Rightarrow X_i \neq X_j)$. It should be noted that other optimization functions could be used ($\sum_{v \in V_i} \#v$, $\sum_{v \in V_i} \sqrt{v}$,...).

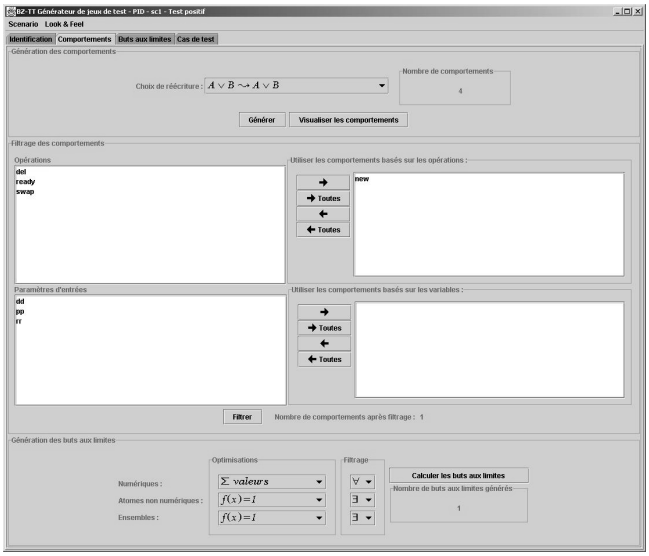


Fig. 6. IHM of the test generation for the PID.

6.2 Test Case Generation

This section describes the generation process of each test case, which is comprised of a preamble, a body, an identification and a postamble part [LP01]. Fig. 6 shows a window of the interface used to drive test generation.

6.2.1 Preamble Computation

Each boundary goal is instantiated to one or more reachable boundary states by exploring the reachable states of the system, starting from the initial state. The CLPS–B solver is used to simulate the execution of the system, recording the set of possible solutions after each operation. A best-first search [Pre01] is used to try to reach a boundary state that satisfies a given boundary goal. Preamble computation can thus be viewed as a traversal of the reachability graph, whose nodes represent the constrained states built during the simulation, and whose transitions represent an operation invocation. A consequence of this path computation is that state variables, which are not already assigned a value by the boundary goal, are assigned a reachable value of their domain.

Some boundary goals may not be reachable via the available operations (this happens when the invariant is weaker than it could be). By construction, every boundary goal satisfies the invariant, which is a partial reachability check. In addition to this, we bind the search for the boundary state during the preamble computation,

so that unreachable boundary goals (and perhaps some reachable goals) are reported to the test engineer as being unreachable. If all boundary goals in a precondition sub-domain PS are unreachable, we relax our boundary testing criterion and search for any preamble that reaches a state satisfying PS.

6.2.2 Input Variable Boundary and Body Computation

The purpose of the body computation is to test, for a given boundary state, all the update operations, with all boundary values of their input variables. For the boundary values which satisfy the precondition, we get a positive test case, otherwise we get a negative test case. Note that, from the same preamble and boundary state, several bodies are usually obtained for each operation, with differing input values.

The process of boundary analysis for input variables is similar to that for state variables, except that invalid input values are kept, which is not the case for unreachable boundary states. Given an operation Op with a set of input variables I_i and a precondition Pre, let BG_i be a boundary goal. Note that BG_i is a set of constraints over the state variables, typically giving a value to each state variable. Then, given f an optimization function chosen by the test engineer, the input variable boundaries are computed as follows:

- for positive test cases:
  $minimize(f(I_i), Pre \wedge BG_i)$
  $maximize(f(I_i), Pre \wedge BG_i)$

- for negative test cases:
  $minimize(f(I_i), \neg Pre \wedge BG_i)$
  $maximize(f(I_i), \neg Pre \wedge BG_i)$

6.2.3 Identification and Postamble

The identification part of a test case is simply a sequence of all observation operations whose preconditions are true after the body. The postamble part is computed similarly to the preamble, using a best-first search.

6.3 Industrial Case-study results

The B-TT method has been validated in several industry case studies. Each study contributes to validate a part of test generation process. To evaluate the case-study specification complexity, some elements are now introduced:

- GSM 11-11 smart card software [LPV01]: 12 pages of B specifications with 11 operations and 10 state variables with an average domain size of 5 elements. 38 boundary goals and about 1000 test cases were automatically generated from the B model [LP01]. This study shown that the B-TT generated test cases covered 85% of high quality manually-designed test cases. 15% of manually-designed test cases were not generated with the automatic procedure: it is mainly because only one preamble is computed to reach the boundary state from the initial state. Using several preambles with specific operations would make it possible to automatically cover all the manually-designed test cases.
- Metro/RER ticket validation algorithm [CGLP01]: 11 pages of B specifications with only 1 operation and 46 states variables with an average domain size of 6 elements. About 100 boundary goals and then 100 test cases were generated.
- Java Card Virtual Machine (JCVM) Transaction mechanism [BJLP02]: 14 pages of B specifications with 21 operations and 10 variables with an average domain size of 12 elements. 45 boundary goals and about 6000 test cases were generated. An automatic process to translate abstract test cases into executable test scripts were used.
- Automobile windscreen wiper controller: 12 pages of B specifications with 25 operations and 15 variables with an average domain size of 7 elements. About 500 boundary goals and 10 000 test cases were generated.

The method is currently being used in two other industrial projects: a process to validate payment transaction for card terminal in an urban parking system and a process to manage the key security in novel 3G smart card application.

7 Related work

This study is part of the research field of using CLP techniques for software verification. Over the last few years, constraint technology has been used for various purposes, such as model checking [DP99], formal model animation [Gri00] and automated test generation that is either code-based [GBR00] or specification-based [MA00, ABlM97]. A common interest of all these approaches of using CLP techniques for software verification is to have available a resolution engine to symbolically execute the formal model.Others works carry on with the same goal, but with different techniques. For example, [Jac00] use relational logic and a SAT solver to symbolically execute formal specification such as Z, OCL or Alloy.

Mostly, all these techniques use existing constraint solvers (Boolean and finite domains in general). Due to the specificity of set-oriented B notation, we developed the original CLPS–B solver able to treat constraints over sets, relations and mappings (CLPS–B co-operates with the integer finite domain solver).

On the other hand, the B-Testing-Tools approach is related to the emerging model-based testing techniques like [FHNS02, EFW01]. The research on model-based automated test generation tools is currently very active. These tools use as input a formal model of the system under test and allow the validation engineer to drive

the test generation automated process. More particularly, set-oriented formal specification notations, such as Z [Spi92], VDM [Jon90] and B [Abr96] have been recognized since the beginning of the 1990s as a suitable basis for model-based testing [DF93, HP95, Hie97, ABlM97, Meu99].

In this framework, test cases are generated on the basis of a partition analysis of the formal model: the formal model is used as an input in order to validate an implementation under test.

8 Conclusion and Future Works

This article introduced a constraint resolution system adapted to the evaluation of B formal specifications. The objective was to enable the construction of a reachability graph or finite state automaton of the specifications, in particular to animate and check the model. The constrained states, rather than concrete ones, propagate the non-determinism of the specifications, and dramatically reduce the number of states of the reachability graph.

The key point of this approach is the expression of domains of constraints by explicit sets where the elements of the domains can be variables (constrained) as well as constants. Rules of propagation and consistency reduce the need of enumeration (using entailment) and consistency tests during the computation of substitutions, the treatment of the preconditions and the checking of the invariant properties.

Classic Logic Programming is often used for animation of formal specifications but it is mostly a valued animation [SCT96, NS99]. The CLPS–B constraint solver makes it possible to evaluate B abstract machine for constrained animation purpose as ZETA used a concurrent constraint resolution [Gri00] to animate Z schema.

Test generation on the ground of CLP is known to be a flexible and efficient framework [OJP99], in structural testing [GBR00, Meu01] and in specification-based test generation [LP00, MA00]. Our proposal consists of using the CLPS–B solver to generate functional test cases from B abstract machines.

The constrained evaluator, based on the CLPS–B solver, is also embedded in the B-Testing-Tools tool-set for two main applications:

- animation of B abstract machines for model validation purpose,
- functional black-box test generation on the basis of the B abstract machine.

This technology was consolidated using real-life size industrial applications, and is currently being improving and consolidated in the following ways for delivery to the scientific community:

- generalizing the input notations like Z, UML, StateChart with a common format based on preconditioned before-after predicates. Fig. 7 presents the general architecture, where all tools use an extended constraint solver based on the CLPS–B concepts ,
- taking into account numerical constraints on continuous domains,
- consolidating the inference rules to improve propagation.

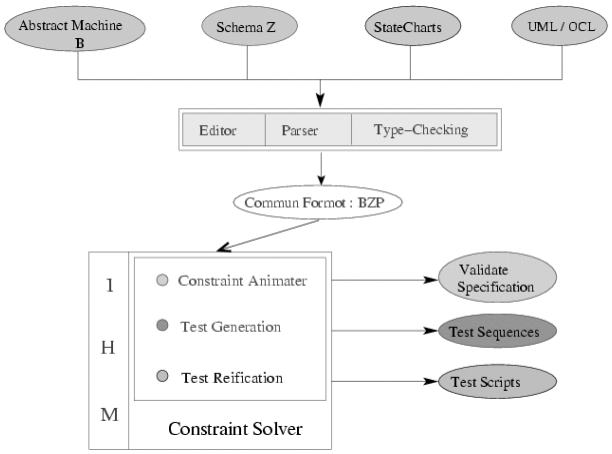


Fig. 7. Architecture of System tools for Animation and Tests Generation

References

[ABlM97] L. Van Aertryck, M. Benveniste, and D. le Metayer. CASTING: a formally based software test generation method. *In 1st IEEE International Conference on Formal Engineering Methods (ICFEM'97)*, pages 99–112, 1997.

[Abr96] J-R. Abrial. *The B-BOOK: Assigning Programs to Meanings*. Cambridge University Press, 1996. ISBN 0 521 49619 5.

[ALL94] F. Ambert, B. Legeard, and E. Legros. Constraint Logic Programming on Sets and Multisets. In *Proceedings of ICLP'94 - Workshop on Constraint Languages and their use in Problem Modeling*, pages 151–165, New York, November 1994.

[AW93] A. Aiken and E.L. Wilmmers. Type Inclusion Constraints and Type Inference. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, pages 31–41, Copenhagen, Denmark, 1993.

[BDM98] P. Behm, P. Desforges, and J.M. Meynadier. METEOR : An Industrial Success in Formal Development. In *Proceedings of the 2nd International Conference on the B method*, Montpellier, 1998.

[BJLP02] F. Bouquet, J. Julliand, B. Legeard, and F. Peureux. Automatic reconstruction and generation of JCVM functional tests patterns (confidential). Technical Report TR-01/02, LIFC - University of Franche-Comté and Schlumberger Montrouge Product Center, 2002.

[BLP00] F. Bouquet, B. Legeard, and F. Peureux. Constraint Logic Programming with sets for animation of B formal specifications. In *CL'00 Workshop*

on (Constraint) Logic Programming and Software Engineering (LPSE'00), London, UK, July 2000.

[CGLP01] N. Caritey, L. Gaspari, B. Legeard, and F. Peureux. Specification-based testing – Application on algorithms of Metro and RER tickets (confidential). Technical Report TR-03/01, LIFC - University of Franche-Comté and Schlumberger Besançon, 2001.

[DF93] J. Dick and A. Faivre. Automating the generation and sequencing of test cases from model-based specifications. In *FME'93*, volume 670 of *LNCS*, pages 268–284. Springer Verlag, April 1993.

[Dic90] J. Dick. Using Prolog to animate Z specifications. In *Z User Meeting 1989. Workshops in Computing*. Springer-Verlag, 1990.

[DP99] G. Delzanno and A. Podelski. Model Checking in CLP. In *Proc. Tools and Algorithms for Construction and Analysis of Systems (TACAS'99)*, pages 223–239, 1999.

[EFW01] I. El-Far and J.A. Whittaker. Model-based software testing. *Encyclopedia of Software Engineering*, 1:825–837, 2001.

[FHNS02] G. Friedman, A. Hartman, K. Nagin, and T. Shiran. Projected State machine Coverage for Software Testing. In *ISSTA'02 Symposium*, volume 27, pages 134–143, Rome, Italy, July 2002. ACM SIGSOFT.

[GBR00] A. Gotlieb, B. Botella, and M. Rueher. A CLP framework for computing structural test data. In *Proceedings of the First International Conference on Computational Logic (CL'00)*, pages 399–413, London, UK, July 2000. Springer-Verlag.

[Ger97] C. Gervet. Interval Propagation to reason about Sets: Definition and Implementation of a Practical Language. *Constraints*, 1(2):191–246, 1997.

[Gri00] W. Grieskamp. A computation model for Z based on concurrent constraint resolution. In *Proceedings of the International Conference of Z and B Users (ZB'00)*, volume 1878 of *LNCS*, pages 414–432, September 2000.

[Hie97] R. Hierons. Testing from a Z specification. *The Journal of Software Testing, Verification and Reliability*, 7:19–33, 1997.

[HP95] H.M. Hörcher and J. Peleska. Using formal specifications to support software testing. *Software Quality Journal*, 4(4):309–327, 1995.

[ISO] ISO. Information Processing Systems, Open Systems Interconnection. *OSI Conformance Testing Methodology and Framework – ISO 9646*.

[Jac00] D. Jackson. Automating First-Order Relational Logic. In *Proceedings of the International Conference on Foundations of Software Engineering*, pages 130–138. ACM SIGSOFT, November 2000.

[Jon90] C.B. Jones. *Systematic Software Development Using VDM*. Prentice-Hall, 2^{nd} edition, 1990.

[Koz94] D. Kozen. Set Constraints and logic Programming (abstract). In *Proceedings of the 1st International Conference on Constraints in Computational Logics (CCL'94)*, volume 845 of *LNCS*, pages 302–303, Germany, September 1994. Springer-Verlag.

[LP00] H. Lötzbeyer and A. Pretschner. AutoFocus on constraint logic programming. In *CL'00 Workshop on (Constraint) Logic Programming and Software Engineering LPSE'00*, London, UK, July 2000.

[LP01] B. Legeard and F. Peureux. Generation of functional test sequences from B formal specifications – Presentation and industrial case-study. In *Proceedings of the 16^{th} International Conference on Automated Software Engineering (ASE'01)*, pages 377–381, San Diego, USA, November 2001. IEEE Computer Society Press.

[LPU02a] B. Legeard, F. Peureux, and M. Utting. A comparison of the BTT and TTF test-generation methods. In *Proceedings of the International Conference on Formal Specification and Development in Z and B (ZB'02)*, volume 2272 of *LNCS*, pages 309–329, Grenoble, France, January 2002. Springer-Verlag.

[LPU02b] Bruno Legeard, Fabien Peureux, and Mark Utting. Automated boundary testing from Z and B. In Lars-Henrik Eriksson and Peter Lindsay, editors, *Formal Methods Europe, 2002*, volume 2391 of *LNCS*, pages 21–40. Springer-Verlag, July 2002.

[LPV01] B. Legeard, F. Peureux, and J. Vincent. Automatic generation of functional tests from the GSM 11-11 specification (confidential). Technical Report TR-01/01, LIFC - University of Franche-Comté and Schlumberger Montrouge Product Center, 2001.

[MA00] B. Marre and A. Arnould. Test Sequence generation from Lustre descriptions: GATEL. In *Proceedings of the 15^{th} International Conference on Automated Software Engineering (ASE'00)*, pages 229–237, Grenoble, France, 2000. IEEE Computer Society Press.

[Meu99] C. Meudec. Tests Derivation from Model Based Formal Specifications. In *Proceedings of the 3^{rd} Irish Workshop on Formal Methods (IWFM'99)*, Galway, Eire, July 1999. BCS, electronic Workshops in Computing. ISBN 1 902505 23 9.

[Meu01] C. Meudec. ATGEN: Automatic test data generation using constraint logic programming and symbolic execution. *The Journal of Software Testing, Verification and Reliability*, 11(2):81–96, 2001.

[Nad89] B.A. Nadel. Constraint Satisfaction Algorithms. *Computer Intelligence*, 5:188–224, 1989.

[NS99] D. Neilson and I.H. Sorensen. *The B-Technologies: a system for computer aided programming*. B-Core (UK) Limited, Kings Piece, Harwell, Oxon, OX11 0PA, 1999. http://www.b-core.com/btoolkit.html.

[OJP99] A.J. Offutt, Z. Jin, and J. Pan. The dynamic domain reduction procedure for test data generation. *The Journal of Software Practice and Experience*, 29(2):167–193, 1999.

[Pre01] A. Pretschner. Classical search strategies for test case generation with Constraint Logic Programming. In *Proceedings of the CONCUR'01 Workshop on Formal Approaches to Testing of Software (FATES'01)*, pages 47–60, Aalborg, Denmark, August 2001. BRICS.

[SCT96] L. Sterling, P. Ciancarini, and T. Turnidge. On the animation of "Not executable" specification by prolog. *Journal of Software Engineering and Knowledge Engineering*, 6(1):63–87, 1996.

[Spi92] J.M. Spivey. *The Z notation: A Reference Manual.* Prentice-Hall, 2^{nd} edition, 1992. ISBN 0 13 978529 9.

[Tsa93] E. Tsang. *Foundations of Constraint Satisfaction.* Academic Press, 1993.

[WE92] M.M. West and B.M. Eaglestone. Software Development: Two approaches to animation of Z specifications using Prolog. *Software Engineering Journal*, 7(4):264–276, 1992.