



**MATISSE: Methodologies and Technologies for  
Industrial Strength Systems Engineering**

**IST-1999-11435**

**Event B to B Translator User Manual**

MATISSE

June 2001

### **Project Information**

Project Number	IST-1999-11435
Project Title	Methodologies and Technologies for Industrial Strength Systems Engineering (MATISSE)
Website	<a href="http://www.matisse.dera.gov.uk">www.matisse.dera.gov.uk</a>
Partners	QinetiQ Centre National de la Recherche Scientifique -SC Aabo Akademi University Gemplus Siemens Transportation Systems University of Southampton ClearSy

### **Document Information**

Document Title	Event B to B Translator User Manual
Workpackage	WP2
Document number	Not referenced
Lead Partner	ClearSy
Editor	T. Lecomte
Contributors	F. Cave

Due date                      None

The information in this document is provided as is and no guarantee or warranty is given that the information is fit for any particular purpose. The user thereof uses the information at its sole risk and liability.

<b>1</b>	<b>INTRODUCTION.....</b>	<b>4</b>
1.1	Purpose of Document.....	4
1.2	Restrictions.....	4
1.3	To do.....	4
<b>2</b>	<b>STRUCTURE OF THE SOFTWARE.....</b>	<b>5</b>
2.1	Language used.....	5
2.2	Modules.....	5
2.2.1	Launcher.....	5
2.2.2	Parser.....	6
2.2.3	Analyser.....	7
2.2.4	Generator.....	8
<b>3</b>	<b>INTERFACE.....</b>	<b>9</b>
3.1	Input.....	9
3.2	Output.....	9
<b>4</b>	<b>REFERENCES .....</b>	<b>10</b>
<b>5</b>	<b>ANNEX A: DEFINITION OF OPERATORS .....</b>	<b>11</b>
<b>6</b>	<b>ANNEX B: EXAMPLE N°1: DEALING WITH EVENTS .....</b>	<b>16</b>
<b>7</b>	<b>ANNEX C: EXAMPLE N°2: USING MODALITITES .....</b>	<b>22</b>

# 1 Introduction

## 1.1 Purpose of Document

The evt2b tool is intended to implement Event B as described in [1]. By implementing, we mean translating Event B models into "Traditional" B models, as Atelier B toolkit only supports the latter.

This document presents the technological rules that have been adopted for the development of the tool. The structure of the software is described, as well as its interface. For the purpose of the translation, syntactic operators have been defined and are detailed in Annex A.

Two examples are also provided in Annexes B and C. Evt2b tool is demonstrated on two short examples, in order to explain its behaviour and to understand its internals.

## 1.2 Restrictions

Tool is constrained by two restrictions on B models:

- This tool doesn't handle DEFINITIONS. More precisely, DEFINITIONS are not expanded and may lead to erroneous behaviour.
- Module files (Atelier B v3.6 specific feature) are not supported as input.

## 1.3 To do

- Add comments to assertions, to identify them (deadlock-freeness, modalities, event creation).
- Introduce structural refinement while grouping events to ease proof. When  $e_i \sqsubseteq e_2 \dots \sqsubseteq e_n$  is refined by  $e_i \sqsubseteq e_2 \dots \sqsubseteq e_n$  (events grouping),  $O(n^2)$  proof obligations are generated. Structural refinement allows to generate only  $O(n)$  proof obligations.

## 2 Structure of the software

### 2.1 Language used

This software is written in theory language, which has been used to develop all Atelier B proof tools, including TypeChecker, Proof Obligation Generator, Automatic and Interactive Provers, Predicate Prover and Arithmetical Prover.

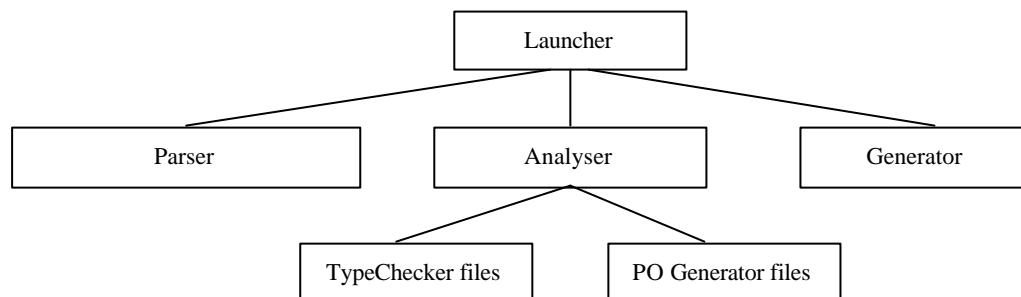
This language, which can be compared to Prolog, implements the B language.

For more information, please refer to [4],[5] and [6].

### 2.2 Modules

The tool is composed of four modules:

- launcher (main module), organising and sequencing actions,
- parser, performing component decomposition into clausal form,
- analyser, transforming Event B clauses into B clauses,
- generator, producing B module file.



This tool also makes use of functions issued from previous software development (TypeChecker, Proof Obligation Generator). These functions are related to substitution calculus (computing  $[S]P$  where  $S$  is a substitution and  $P$  a predicate) and are reused without any modification.

#### 2.2.1 Launcher

This module organises all the steps required to translate Event B to B.

All components need to be in memory prior to any translation, in order to:

- build the complete event list,
- determine which events are created and at which level of refinement, which events are refined,
- store event bodies to be reused for grouping and/or splitting.

So the first step consists in loading the complete refinement column, starting from the most concrete component, up to the more abstract component which should be a SYSTEM component. The REFINES clause is used for each component to determine its abstract counterpart.

Each read component is parsed (using the parser module), rewritten in clausal form and stored in memory.

Then, when all components have been processed that way, translation is performed on all loaded components using a four-step process (using the analyser module)

Once the translation has been performed, the resulting components are saved in a file (using the generator module).

If one of the above steps fails, the main process aborts and an error message is displayed.

### 2.2.2 Parser

The input of the module is a formula like:

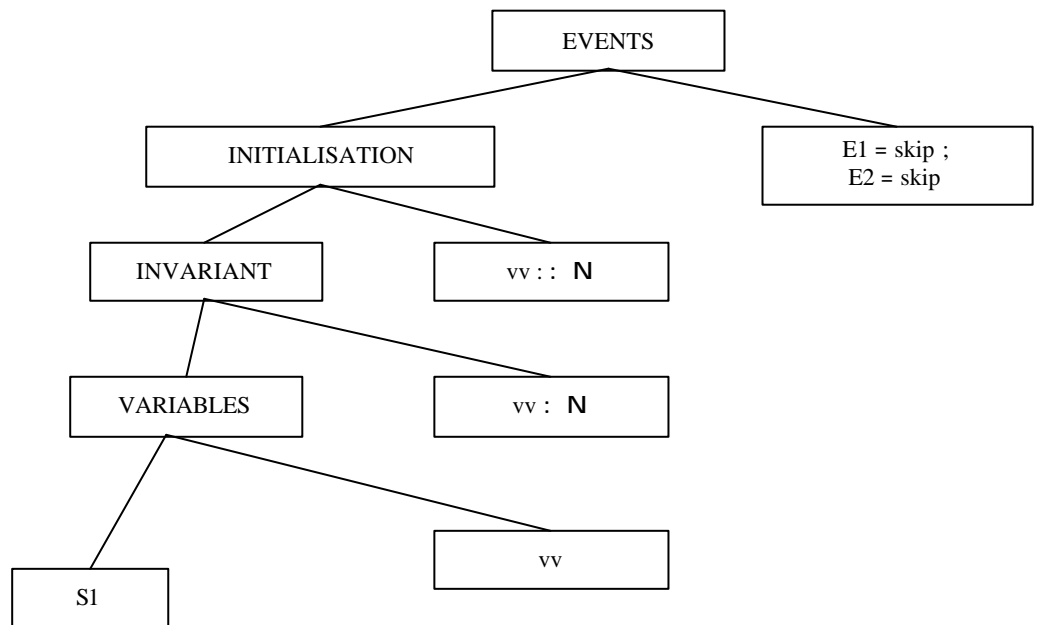
- SYSTEM A END
- REFINEMENT A END

Where  $A$  is put for any formula. Parsing a component means parsing the formula  $A^1$  and generating a semantically equivalent clausal expression.

For example, the component

SYSTEM	S1
VARIABLES	vv
INVARIANT	vv: NAT
INITIALISATION	vv :: NAT
EVENTS	E1 = skip; E2 = skip
END	

is parsed as:



<sup>1</sup> Formulae are parsed using left associativity.

Each clause is associated with its right hand child node. The remaining node is the name of the component.

Its clausal form is

(ClauseSYSTEM | S1) & (ClauseVARIABLES | vv) & (ClauseInvariant | vv:N) & (ClauseInitialisation | vv :: N) & (ClauseEvents | E1 = skip ; e2 = skip)

### 2.2.3 Analyser

Once components are loaded and rewritten in clausal form, transformation of Event B to B is performed in 4 steps:

- Pass 1: list events, determine which events are new, equip new events with POST clause

The complete event list is build while parsing components. The B feature, enabling not to repeat unmodified operation in further refinement, should confuse the tool and should not be used. For each refinement, new events (those which are not explicitly defined in the refined component and which are not defined as refining one or many events) are determined. A post-condition is then added to all these events (see addpost operator in Annex A: definition of operators). Each new event is equipped with skip substitution in abstract components.

- Pass 2 : adding assertions related to MODALITIES and to deadlock-freeness

For each modality and for each event involved in modality, an assertion is computed (see modal operator in Annex A: definition of operators). These assertions are added at the end of current ASSERTIONS clause.

Similarly, an assertion related to deadlock-freeness (see operator df in Annex A: definition of operators) is computed and added at the end of current ASSERTIONS clause.

- Pass 3: transforming POST clauses into LET clauses

Each post-conditioned event body is transformed using LET substitution (see operator post2let in Annex A: definition of operators). Events not matching with following forms are rejected:

-

- Pass 4: events refinement (splitting, grouping)

Splitting is processed first, from abstract to concrete components. Splitting an event means refining a single event with many. An event  $e_i$ , refined in refinement  $R_{i+1}$  with events  $e_{i1}, e_{i2}, \dots, e_{in}$ , is replaced in refinements  $R_0$  to  $R_i$  with events  $e_{i1}, e_{i2}, \dots, e_{in}$  which are specified as  $e_{i1}=e_{i2}=\dots=e_{in}=e_i$ .

Grouping is then processed, from concrete to abstract components. Grouping events means refining more than one event with one. Events  $e_{i1}, e_{i2}, \dots, e_{in}$ , refined in refinement  $R_{i+1}$  with  $e_i$ , are replaced in refinements  $R_0$  to  $R_i$  with events  $e_i$  which is specified as  $e_i = e_{i1}[]e_{i2}[]\dots[]e_{in}$ .

## 2.2.4 Generator

The translated components are saved in a single module file (mod extension), from MACHINE component to most concrete component.

The following CLAUSES are directly saved:

EVENTS, DEFINITIONS, ASSERTIONS, INITIALISATION, INVARIANT, CONCRETE_VARIABLES, ABSTRACT_VARIABLES, VARIABLES, PROPERTIES, CONCRETE_CONSTANTS, ABSTRACT_CONSTANTS, CONSTANTS, , CONSTRAINTS, PROMOTES, EXTENDS, SEES, USES, IMPORTS, INCLUDES, REFINES, IMPLEMENTATION\, REFINEMENT, SYSTEM
--

The following CLAUSES are saved while rewriting clause names as:

SYSTEM → MACHINE EVENTS → OPERATIONS
---

The following CLAUSES are ignored and not saved:

VARIANT, MODALITIES, MACHINE, OPERATIONS
--

## **3 Interface**

### **3.1 Input**

Module files are not supported as input.

An abstract machine (mch extension) or a refinement (ref extension) should be provided. In case of refinement component, all more abstract components in the refinement column are loaded, up to the abstract machine which should be present. The top level component should be a SYSTEM component (not a MACHINE).

Example: if M0.mch is refined by M1.ref, which is refined by M2.ref, which is refined by M3.ref, starting the tool with M2.ref will lead to also load M1.ref and M0.mch.

No implementation should be provided.

Supported component syntax is described in [1]. "Traditionnal B" clauses, like MACHINE or OPERATIONS are not processed, generate error messages and abort the translation process.

### **3.2 Output**

See generator paragraph.

## **4 References**

- [1] Event B Reference Manual (Draft) v1 - ClearSy
- [2] The B-Book: Assigning Programs to Meanings - J.R Abrial
- [3] Guidelines to formal System Studies (Draft version 2 – November 2000) – J.R. Abrial
- [4] Logic Solver: Syntax - v1.2 – ClearSy
- [5] Logic Solver: Semantics - v1.2 – ClearSy
- [6] Logic Solver: Operations and guards - v1.2 – ClearSy

## 5 Annex A: Definition of operators

**grd(S)**: return guards of a substitution as a predicate.

Rules partly come from [2] §6.3.2 Feasability.

1. $\text{grd}(A;B)$	$= \text{grd}(A) \circ \text{grd}(B)$
2. $\text{grd}(\text{SELECT } P \text{ THEN } S \text{ POST } Q \text{ END})$	$= \text{grd}(\text{SELECT } P \text{ THEN } S \text{ END})$
3. $\text{grd}(\text{PRE } P \text{ THEN } S \text{ POST } Q \text{ END})$	$= \text{grd}(\text{PRE } P \text{ THEN } S \text{ END})$
4. $\text{grd}(\text{ANY } x \text{ WHERE } P \text{ THEN } S \text{ POST } Q \text{ END})$ $\text{S END})$	$= \text{grd}(\text{ANY } x \text{ WHERE } P \text{ THEN } S \text{ END})$
5. $\text{grd}(\text{BEGIN } S \text{ POST } Q \text{ END})$	$= \text{grd}(\text{BEGIN } S \text{ END})$
6. $\text{grd}(\text{CHOICE } A \text{ OR } B \text{ END})$	$= \text{grd}(A) \circ \text{grd}(B)$
7. $\text{grd}(\text{SELECT } P \text{ THEN } S \text{ END})$	$= P \ \& \ \text{grd}(S)$
8. $\text{grd}(\text{PRE } P \text{ THEN } S \text{ END})$	$= P \ \mathbf{y} \ \text{grd}(S)$
9. $\text{grd}(\text{ANY } x \text{ WHERE } P \text{ THEN } S \text{ END})$	$= \#x.P \ \& \ \text{grd}(S)$
10. $\text{grd}(\text{BEGIN } S \text{ END})$	$= \text{grd}(S)$
11. $\text{grd}(x :: E)$	$= n(E=O)$
12. $\text{grd}(x: P)$	$= \#y.P$ with $y$ free in $P$ and $x$
13. $\text{grd}(S)$	$= \text{btrue}$

## **addpost**

Given that V is the variant of the component:

1.	addpost(ANY x WHERE P THEN S POST X END)	=	ANY x WHERE P THEN S POST (X & exp2post(V)) END
2.	addpost(ANY x WHERE P THEN S END)	=	ANY x WHERE P THEN S POST exp2post(V)) END
3.	addpost(SELECT P THEN S POST X END)	=	SELECT P THEN S POST (X & exp2post(V)) END
4.	addpost(SELECT P THEN S END)	=	SELECT P THEN S POST exp2post(V) END
5.	addpost(BEGIN S POST X END)	=	BEGIN S POST (X & exp2post(V)) END
6.	addpost(BEGIN S END)	=	BEGIN S POST exp2post(V) END
7.	addpost(skip)	=	skip

## modal

Given I is the invariant of the component:

1. modal(BEGIN E MAINTAIN P UNTIL Q VARIANT V END) =  
I & P & nQ y grd(E) &  
I & P & nQ y V : N &  
I & P & nQ y [n:=V][e<sub>i</sub>](nQ y V<n) &  
I & P & nQ y [e<sub>i</sub>](nQ y P)
  
2. modal(ANY x WHERE T THEN E MAINTAIN P UNTIL Q VARIANT V END) =  
I & P & T & nQ y grd(E) &  
I & P & T & nQ y V : N &  
I & P & T & nQ y [n:=V][e<sub>i</sub>](nQ y V<n) &  
I & P & T & nQ y [e<sub>i</sub>](nQ y P)
  
3. modal(BEGIN E ESTABLISH Q END) =  
I y [e<sub>i</sub>]Q
  
4. modal(SELECT P THEN E ESTABLISH Q END) =  
I & P y [e<sub>i</sub>]Q
  
5. modal(ANY x WHERE P THEN E ESTABLISH Q END) =  
I & P y [e<sub>i</sub>]Q

**post2let:** transforming a post-condition into a LET substitution

Z is a list of variables  $v_{i_0}, v_{1_0}, \dots, v_{n_0}$ . Each  $v_{i_n}$  corresponds to variable  $v_i$  contained in Q.

E is a conjunction of equalities  $v_{i_0} = v_i$

$R = [v_1 := v_{1_0}, \dots, v_n := v_{n_0}]Q$

1. `post2let(ANY x WHERE P THEN S POST Q END) =  
ANY x WHERE P THEN LET Z BE E IN BEGIN S;PRE R THEN skip END  
END END END`
2. `post2let(BEGIN S POST Q END) =  
BEGIN LET Z BE E IN BEGIN S;PRE R THEN skip END END END END`
3. `post2let(PRE P THEN S POST Q END) =  
PRE P THEN LET Z BE E IN BEGIN S;PRE R THEN skip END END END  
END`
4. `post2let(SELECT P THEN S POST Q END) =  
SELECT P THEN LET Z BE E IN BEGIN S;PRE R THEN skip END END  
END END`
5. `post2let(ANY x WHERE P THEN S POST Q END) =  
ANY x WHERE P THEN S;PRE Q THEN skip END END).(a = ANY x  
WHERE P THEN S POST Q END`
6. `post2let(BEGIN S POST Q END) =  
BEGIN S;PRE Q THEN skip END END`  
if Q contains no variables.
7. `post2let(PRE P THEN S POST Q END) =  
PRE P THEN S;PRE R THEN skip END END`  
if Q contains no variables.
8. `post2let(SELECT P THEN S POST Q END) =  
SELECT P THEN S;PRE R THEN skip END END`  
if Q contains no variables.

**df:** Deadlockfreeness

$$1. \text{df}(E) = \text{I} \text{ grd}(e_i)$$

**expr2post:** transforming a variant expression into a post-condition predicate

$$1. \text{expr2post}(V) = \text{I} (v_i < v_i\$0)$$

where  $v_i$  are all free variables of expression  $V$  and  
 $v_i\$0$  represent variables valuation before modification

## 6 Annex B: Example n°1: Dealing with events

The following example shows how to add/group/split events.

We consider here a simple system (M0). Its state is represented by a variable  $xx$  and one event may occur,  $evol$ , which led variable  $xx$  to evolve according to its definition domain, namely  $\mathbf{N}$ .

```

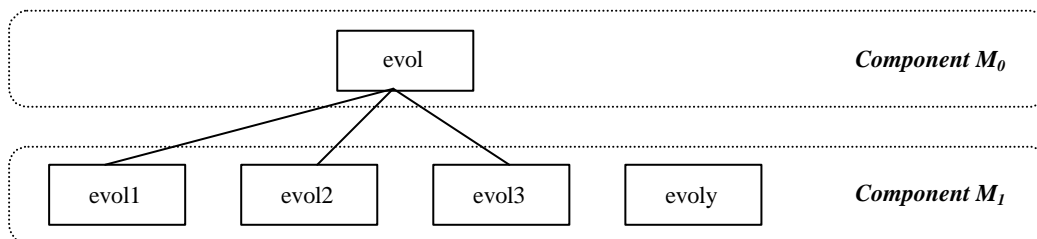
SYSTEM M0
VARIABLES
  xx
INVARIANT
  xx: NAT
INITIALISATION
  xx :: NAT
EVENTS
  evol = BEGIN xx: (xx: NAT) END
END
  
```

In the next refinement (M1), we express that our variable  $xx$  evolves according to the value of an other state variable,  $yy$ , which can be considered as a stimulus.

A new event is created,  $evoly$ , related to the evolution of the variable  $yy$ . In order to ensure that this new event may not take forever the control of the system and prevent event  $evol$  to occur, the number of occurrences of that event should be finite. We introduce in this case a finite set  $zz$ . Values for  $yy$  are chosen in this set and removed from it. When this set is empty, event  $evoly$  can't be fired any more and variable  $yy$  keeps its last value. The VARIANT clause is introduced, containing an expression that should be decreased each time the new event  $evoly$  is fired and should reach zero in a finite number of steps. The VARIANT expression is the cardinal of the set  $zz$ .

Event  $evol$  is split into 3 events, each one describing the evolution of  $xx$  in different cases, depending on the value of  $yy$ . In fact,  $xx$  is modified in order to reach the value of  $yy$ .

The event refinement diagram is given below:



**Figure A-1:** events location in components M0 and M1

In the B model (component M1),  $evol$  is explicitly replaced by  $evol1$ ,  $evol2$  and  $evol3$ . Each one is declared to refine  $evol$ . On the other hand, there is no need to express that  $evoly$  is new event since it can be deduced from previous component. We obtain the event B model below.

```

REFINEMENT M1
REFINES M0
VARIABLES
  xx, yy, zz
INVARIANT
  xx: NAT &
  yy: NAT&
  zz <: NAT &
  zz: FIN(zz)
INITIALISATION
  xx := NAT ||
  yy, zz : (yy: NAT & zz <: NAT & zz /= {} & yy: zz & zz:
FIN(zz))
VARIANT
  card(zz)
EVENTS
  evol1 ref evol = SELECT xx=yy THEN skip END;
  evol2 ref evol = SELECT xx>yy THEN xx:=xx-1 END;
  evol3 ref evol = SELECT xx<yy THEN xx:=xx+1 END;
  evoly =
    SELECT zz /= {} THEN
      ANY val WHERE val: zz THEN
        zz := zz - {val} ||
        yy := val
      END
    END
END
END

```

In the second refinement, *evoly* remains unchanged. *evol1*, *evol2* and *evol3* are grouped into one event to form an IF THEN ELSE substitution. This new event, *evol4*, is explicitly declared to refine *evol1*, *evol2* and *evol3*.

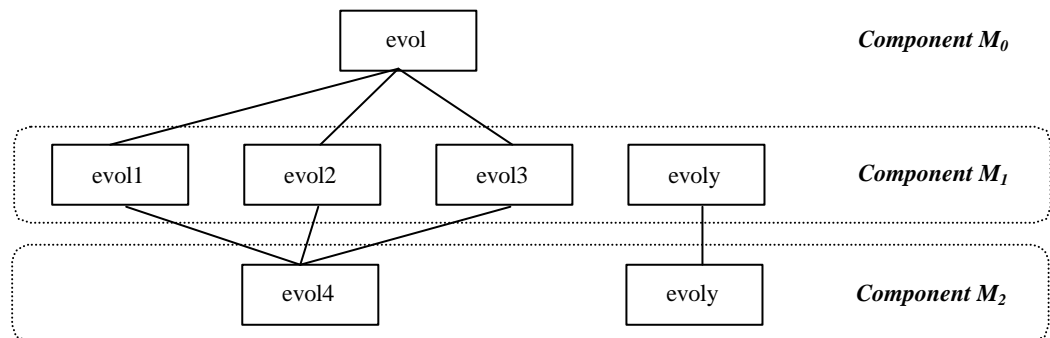


Figure A-2: events location in components M0, M1 and M2

We obtain the event B model below.

```

REFINEMENT M2
REFINES M1
VARIABLES
    xx, yy, zz
INVARIANT
    xx: NAT &
    yy: NAT&
    zz <: NAT &
    zz: FIN(zz)
INITIALISATION
    xx := NAT ||
    yy, zz : (yy: NAT & zz <: NAT & zz /= {} & yy: zz & zz:
FIN(zz))
EVENTS
    evol4 ref evol1, evol2, evol3 =
        BEGIN
            IF xx>yy THEN xx:=xx-1
            ELSIF xx<yy THEN xx:=xx+1
            ELSIF xx=yy THEN skip
            END
        END
;
    evoly =
        SELECT zz /= {} THEN
            ANY val WHERE val: zz THEN
                zz := zz - {val} ||
                yy := val
            END
        END
END
END

```

Now let us translate these event b components, using evt2b tool.

We decide to translate M0, M1 and M2. We should start translation from the most concrete component of the refinement column. So we type in:

```
evt2b M2.ref
```

and we obtain the following messages

```

REFINEMENT Component M2 loaded
REFINEMENT Component M1 loaded
SYSTEM Component M0 loaded
Starting Pass 1
Events found:
    evol
    evol1
    evol2
    evol3
    evoly
    evol4
Suppressing Refs:...
Adding new events declaration with skip substitution:.

Starting Pass 2
Transforming POST clauses: .OK

Starting Pass 3
Pass3a (DeadlockFreeness): OK
Pass3b (Modalities): OK
Pass3c (Post-processing assertions): OK

Starting Pass 4

```

Splitting events: OK  
 Aggregating events: OK

Saving system M0 as M0.mod file

Event structure in figure A-2 can't be kept in the final B models, since event names and signatures should be constant within a refinement column. So structure should be adapted.

Concerning *evoly*, we just need to add an event *evoly* in component M0 with skip substitution.

Things are more complicated for *evol*. Since *evol* is split into *evol1*, *evol2* and *evol3*, *evol* disappears in M0 and is replaced by these 3 events. The body of *evol1*, *evol2* and *evol3* in M0 is made equal to the body of *evol*. But, since *evol1*, *evol2* and *evol3* are also refined by *evol4*, those events should be replaced by *evol4*. In M1, *evol4* is now equal to the non-deterministic choice of the substitutions of those events, namely

CHOICE *body<sub>evol1</sub>* OR *body<sub>evol2</sub>* OR *body<sub>evol3</sub>* END

The same procedure is repeated in M0 to obtain *evol4* body. We obtain the final event structure described in figure A-3.

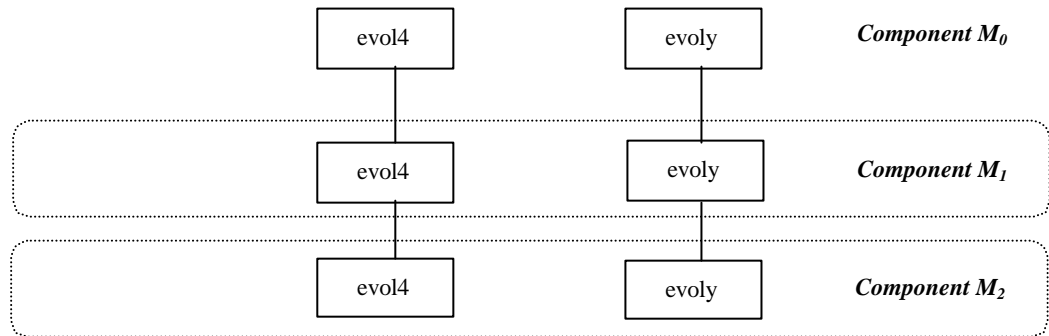


Figure A-3: events location in components M0, M1 and M2, after translation

The translated components are given below.

```

MACHINE
  M0
VARIABLES
  xx
INVARIANT
  xx: NAT
INITIALISATION
  xx:: NAT
OPERATIONS
  evoly = skip
;   evol4 = CHOICE
      BEGIN
        xx: (xx: NAT)
      END
    OR BEGIN
        xx: (xx: NAT)
      END
    OR BEGIN
        xx: (xx: NAT)
      END
  END
END
END
  
```

In M1, we notice that:

- an assertion (clause ASSERTIONS), related to deadlockfreeness, has been added. It consists in the disjunction of the guards of all the events.
- As *evoly* is a new event, this event should decrease the VARIANT expression. This is accomplished first by adding a post condition ( $\text{card}(zz) < \text{card}(zz\$0)$ ) to *evoly*, then by transforming this post condition in a LET substitution.

```

REFINEMENT
  M1
REFINES
  M0
VARIABLES
  xx,yy,zz
INVARIANT
  xx: NAT
&   yy: NAT
&   zz <: NAT
&   zz: FIN(zz)
INITIALISATION
  xx:: NAT || yy,zz: (yy: NAT & zz <: NAT & zz/={}) & yy: zz & zz:
FIN(zz))
ASSERTIONS
  xx = yy or xx>yy or xx<yy or zz/={ }
OPERATIONS
  evoly = SELECT zz/={ } THEN
    LET zzx_0 BE
      zzx_0 = zz
    IN
      BEGIN
        ANY val WHERE
          val: zz
        THEN
          zz:=zz-{val} ||
          yy:=val
        END;
        PRE
          card(zz)<card(zzx_0)
        THEN
          skip
        END
      END
    END
  END
  evol4 = CHOICE
    SELECT xx = yy THEN
      skip
    END
    OR SELECT xx>yy THEN
      xx:=xx-1
    END
    OR SELECT xx<yy THEN
      xx:=xx+1
    END
  END
END

```

M2 is similar to its event B counterpart, except event B features that have been removed or replaced (EVENTS by OPERATIONS, *ref* suppressed).

```

REFINEMENT
  M2
REFINES
  M1
VARIABLES
  xx,yy,zz
INVARIANT
  xx: NAT
&   yy: NAT
&   zz <: NAT
&   zz: FIN(zz)
INITIALISATION
  xx:: NAT || yy,zz: (yy: NAT & zz <: NAT & zz/={}) & yy: zz & zz:
  FIN(zz))
OPERATIONS
  evol4 = BEGIN
    IF xx>yy THEN
      xx:=xx-1
    ELSIF xx<yy THEN
      xx:=xx+1
    ELSIF xx = yy THEN
      skip
    END
  END
;   evol5 = SELECT zz/={}) THEN
    ANY val WHERE
      val: zz
    THEN
      zz:=zz-{val} ||
      yy:=val
    END
  END
END

```

The components are then proved. Status project is given below.

Project status

COMPONENT	TC	POG	nPO	nUn	%Pr	BOC	C	Ada	C++	HIA
M0	OK	OK	0	0	100	-				
M1	OK	OK	13	0	100	-				
M2	OK	OK	15	0	100	-				
TOTAL	OK	OK	28	0	100	-	OK	OK	OK	OK

## 7 Annex C: Example n°2: Using modalities

The following example shows how to use modalities.

We consider here a simple system (S0). Its state is represented by two variables *xx* and *input*. *xx* is linked with *input* such as, when *input* evolves (event *ev1*), then *xx* is likely to evolve too (events *ev2* to *ev7*). We can consider that *ev1* represents a kind a stimulus (from the environment) and *ev2* to *ev7* represent the response to this stimulus. Modalities can be used to write invariant properties restricted to some events (i.e. *ev2*, *ev3*, ... *ev7*). In this case, we would like to verify that the response complies with some general law, exhibited in the ESTABLISH clause.

```
SYSTEM
  S0
SETS
  EE = {e0, e1, e2, e3, e4};
  INPUTS = {v1, v2, v3}
VARIABLES
  xx,
  input
INVARIANT
  xx: EE &
  input : INPUTS
INITIALISATION
  xx := e0 ||
  input :: INPUTS
EVENTS
  ev1 =
  ANY in WHERE in : INPUTS
  THEN
    input := in
  END
  ;
  ev2 = SELECT input=v1 & xx = e0 THEN xx := e2 END ;
  ev3 = SELECT input=v1 & xx /= e0 THEN xx := e0 END ;
  ev4 = SELECT input=v2 & xx = e0 THEN xx := e2 END ;
  ev5 = SELECT input=v2 & xx /= e0 THEN xx := e3 END ;
  ev6 = SELECT input=v3 & xx = e1 THEN xx := e4 END ;
  ev7 = SELECT input=v3 & xx /= e1 THEN xx := e1 END
MODALITIES
  BEGIN
    ev2, ev3, ev4, ev5, ev6, ev7
  ESTABLISH
    input|->xx : {
      (v1|->e0), (v1|->e2), (v1|->e3),
      (v2|->e2), (v2|->e3),
      (v3|->e1), (v3|->e3), (v3|->e4)
    }
  END
END
```

Now let us translate this event b component, using evt2b tool. So we type in:

```
evt2b S0.mch
```

and we obtain the following messages

```
SYSTEM Component S0 loaded
Starting Pass 1
```

```

Events found:
  ev1
  ev2
  ev3
  ev4
  ev5
  ev6
  ev7
Suppressing Refs:.
Adding new events declaration with skip substitution:

Starting Pass 2
  Transforming POST clauses: OK

Starting Pass 3
  Pass3a (DeadlockFreeness): OK
  Pass3b (Modalities): .OK
  Pass3c (Post-processing assertions): OK

Starting Pass 4
  Nothing to do

Saving system S0 as S0.mod file

```

Now let us examine generated assertions.

The first one is related to deadlockfreeness. It is constituted of the disjunction of the guards of all the events (*ev1* to *ev7*).

The six other assertions are related to ESTABLISH modality, ie provided that the invariant holds, we should demonstrate that each event establishes the modality. For each event *ev<sub>i</sub>*, and for the modality  $P_{modality}$ , we have:

$$\text{Inv } \mathbf{y} [S_{ev_i}]P_{modality}$$

The resulting B model is given below.

```

MACHINE
  S0
SETS
  EE = {e0,e1,e2,e3,e4}; INPUTS = {v1,v2,v3}
VARIABLES
  xx,input
INVARIANT
  xx: EE
&   input: INPUTS
INITIALISATION
  xx:=e0 || input:: INPUTS
ASSERTIONS
  #in.(in: INPUTS) or (input = v1 & xx = e0) or (input = v1 &
xx/=e0) or (input = v2 & xx = e0) or (input = v2 & xx/=e0) or (input
= v3 & xx = e1) or (input = v3 & xx/=e1)
;   xx: EE & input: INPUTS => (input = v1 & xx = e0 => input|->e2:
{v1|->e0,v1|->e2,v1|->e3,v2|->e2,v2|->e3,v3|->e1,v3|->e3,v3|->e4})
;   xx: EE & input: INPUTS => (input = v1 & xx/=e0 => input|->e0:
{v1|->e0,v1|->e2,v1|->e3,v2|->e2,v2|->e3,v3|->e1,v3|->e3,v3|->e4})
;   xx: EE & input: INPUTS => (input = v2 & xx = e0 => input|->e2:
{v1|->e0,v1|->e2,v1|->e3,v2|->e2,v2|->e3,v3|->e1,v3|->e3,v3|->e4})
;   xx: EE & input: INPUTS => (input = v2 & xx/=e0 => input|->e3:
{v1|->e0,v1|->e2,v1|->e3,v2|->e2,v2|->e3,v3|->e1,v3|->e3,v3|->e4})
;   xx: EE & input: INPUTS => (input = v3 & xx = e1 => input|->e4:
{v1|->e0,v1|->e2,v1|->e3,v2|->e2,v2|->e3,v3|->e1,v3|->e3,v3|->e4})

```

```

;      xx: EE & input: INPUTS => (input = v3 & xx/=e1 => input|->e1:
{v1|->e0,v1|->e2,v1|->e3,v2|->e2,v2|->e3,v3|->e1,v3|->e3,v3|->e4})
OPERATIONS
  ev1 = ANY in WHERE
    in: INPUTS
    THEN
      input:=in
    END
;      ev2 = SELECT input = v1 & xx = e0 THEN
      xx:=e2
    END
;      ev3 = SELECT input = v1 & xx/=e0 THEN
      xx:=e0
    END
;      ev4 = SELECT input = v2 & xx = e0 THEN
      xx:=e2
    END
;      ev5 = SELECT input = v2 & xx/=e0 THEN
      xx:=e3
    END
;      ev6 = SELECT input = v3 & xx = e1 THEN
      xx:=e4
    END
;      ev7 = SELECT input = v3 & xx/=e1 THEN
      xx:=e1
    END
END

```

The component is then proved. Status project is given below.

Project status

COMPONENT	TC	POG	nPO	nUn	%Pr	BOC	C	Ada	C++	HIA
S0	OK	OK	34	0	100	-				
TOTAL	OK	OK	34	0	100	-	OK	OK	OK	OK