MATISSE: Methodologies and Technologies for Industrial Strength Systems Engineering

IST-1999-11435

Event B Reference Manual

MATISSE

June 2001

Project Information

Project Number	IST-1999-11435
Project Title	Methodologies and Technologies for Industrial Strength Systems Engineering (MATISSE)
Website	www.matisse.dera.gov.uk
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 Reference Manual
Workpackage	WP1
Document number	Not referenced
Lead Partner	ClearSy
Editor	T. Lecomte
Contributors	JR Abrial, L. Mussat (DCSSI)
Due date	None

1 Substitutions

1.1 Presentation

Generalized substitutions

A new generalized substitution has been added:

POST	Substitution postcondition

Syntax

The syntax of generalized substitutions is now:

```
Substitution ::=
            Substitution_level1
  |         Substitution_sequencing
  |         Substitution_simultaneous
Substitution_level1 ::=
            Substitution_block
  |         Substitution_identity
  |         Substitution_becomes_equal
  |         Substitution_precondition
  |         Substitution_postcondition
  |         Substitution_assertion
  |         Substitution_limited_choice
  |         Substitution_if
  |         Substitution_select
  |         Substitution_case
  |         Substitution_any
  |         Substitution_let
  |         Substitution_becomes_elt
  |         Substitution_becomes_such_as
  |         Substitution_var
  |         Substitution_call
  |         Substitution_while
```

In the section below, postcondition generalized substitution is described. You are given: its name, the type of components where it may be used, its syntax, its typing and effect rules, its description and an example.

1.2 Postcondition Substitution

Operator

POST	Postcondition

Type

Substitution	of specification	☑
	of refinement	☑
	of implementation	☐

Syntax

Substitution_postcondition ::= "BEGIN" *Substitution* "POST" *Predicate* "END"

Definition

With Q and R as predicates and S a substitution.

$$[\text{BEGIN } S \text{ POST } Q(x,x\$0) \text{ END}]\, R \;\Leftrightarrow\; [S]\, R \wedge [x := x\$0]\,[S]\, R$$

Description

Postcondition substitution enables to assert that substitution S establishes predicate Q and to define more precisely the effects of a substitution. Indeed, with postcondition substitution, it is possible to write properties that substitution S should establishes, in addition to those expressed in the invariant. This substitution is mainly used in event B for Event driven sequential program construction.

As an example, for each new event added in a refinement, we can precise, by using postcondition substitution, that the variant of the system is decreased by the substitution of the added event.

Postcondition substitution allows to precise what the type of the result of an operation is.

Example

```
remove_element =
  BEGIN
          ANY
            element
          WHERE
            element : set &
            enable_remove = TRUE
          THEN
  set := set – {element} ||
            enable_remove := bool(set / {})
          END
POST
            card(set) < card(set$0)
END
```

1.3 Precondition-postcondition Substitution

Syntax

Substitution_precondition_postcondition ::= "PRE" *Predicate* "THEN" *Substitution* "POST" *Predicate* "END"

Definition

With *P*, *Q* and *R* as predicates and *S* a substitution.

PRE *P* THEN *S* POST *Q(x,x$0)* END

can be rewritten as:

BEGIN PRE *P* THEN *S* END POST *Q(x,x$0)* END

Description

Precondition-postcondition substitution is similar to postcondition substitution. This substitution is mainly used with event B and is only a useful abbreviation of postcondition substitution.

1.4 Selection-postcondition Substitution

Syntax

Substitution_selection_postcondition ::= "SELECT" *Predicate* "THEN" *Substitution* "POST" *Predicate* "END"

Definition

With *P*, *Q* and *R* as predicates and *S* a substitution.

SELECT *P* THEN *S* POST *Q(x,x$0)* END

can be rewritten as:

BEGIN SELECT *P* THEN *S* END POST *Q(x,x$0)* END

Description

Selection-postcondition substitution is similar to postcondition. This substitution is mainly used with event B and is only a useful abbreviation of postcondition substitution.

1.5 Unbounded choice-postcondition Substitution

Syntax

Substitution_unbounded_choice_postcondition ::= "ANY" Ident_ren+"," "WHERE" *Predicate* "THEN" *Substitution* "POST" *Predicate* "END"

Definition

With Q, R and T as predicates and S a substitution.

ANY x WHERE T THEN S POST $Q(x,x\$0)$ END

can be rewritten as:

BEGIN ANY x WHERE T THEN S END POST $Q(x,x\$0)$ END

Description

Unbounded choice-postcondition substitution is similar to postcondition. This substitution is mainly used with event B and is only a useful abbreviation of postcondition substitution.

2 Components

An event B development is composed of an abstract component, called abstract system, containing higher level specification, and of refined components, refining the abstract system.

Syntax

```
Component ::=
          Abstract_system
  |       Refinement*
```

2.1 Abstract System

Syntax

```
System_abstract ::=
        "SYSTEM" Header
        Clause_system_abstract*
        "END"
Clause_system_abstract ::=
        Clause_constraints
    |   Clause_sees
    |   Clause_sets
    |   Clause_concrete_constants
    |   Clause_abstract_constants
    |   Clause_properties
    |   Clause_concrete_variables
    |   Clause_abstract_variables
    |   Clause_invariant
    |   Clause_assertions
    |   Clause_initialization
    |   Clause_events
    |   Clause_modalities
```

Description

An abstract system is a component that defines in different clauses, data and its properties as well as operations. An abstract system makes up the specification of an event B module. It comprises a header and a certain number of clauses. The order of the clauses in a component is not fixed. The description of clauses is given in the table below.

Clause	Description
CONSTRAINTS	Definition of the type and properties of formal scalar parameters
SEES	List of instances of machines *seen*
SETS	List of abstract sets and definition of listed sets
CONCRETE_CONSTANTS	List of concrete constants
ABSTRACT_CONSTANTS	List of abstract constants
PROPERTIES	Definition of the type and of properties of machine constants
CONCRETE_VARIABLES	List of concrete variables
ABSTRACT_VARIABLES	List of abstract variables
INVARIANT	Declaration of the type and of properties of variables
ASSERTIONS	Definition of properties that are deduced from the invariant
INITIALIZATION	Initialization of variables
EVENTS	List and definition of system events
MODALITIES	List of dynamic properties of the system

Restrictions

1. A clause may only appear at most one time in an abstract machine.

2. If one of the CONCRETE_CONSTANTS or ABSTRACT_CONSTANTS clauses are present, then the PROPERTIES clause must be present.
3. If one of the CONCRETE_VARIABLES or ABSTRACT_VARIABLES clauses is present, then the INVARIANT and INITIALIZATION clauses must be present.

2.2 Refinement

Syntax

```
Refinement ::=
        "Refinement" Header
        Clause_refinement*
        "END"
Clause_refinement ::=
        Clause_constraints
  |     Clause_sees
  |     Clause_variant
  |     Clause_sets
  |     Clause_concrete_constants
  |     Clause_abstract_constants
  |     Clause_properties
  |     Clause_concrete_variables
  |     Clause_abstract_variables
  |     Clause_invariant
  |     Clause_assertions
  |     Clause_initialization
  |     Clause_events
  |     Clause_modalities
```

Description

A refinement is a component that defines in different clauses, data and its properties as well as operations. It comprises a header and a certain number of clauses. The order of the clauses in a component is not fixed. The description of clauses is given in the table below.

Clause	Description
CONSTRAINTS	Definition of the type and properties of formal scalar parameters
SEES	List of instances of machines *seen*
VARIANT	Variant of the system
SETS	List of abstract sets and definition of listed sets
CONCRETE_CONSTANTS	List of concrete constants
ABSTRACT_CONSTANTS	List of abstract constants
PROPERTIES	Definition of the type and of properties of machine constants
CONCRETE_VARIABLES	List of concrete variables
ABSTRACT_VARIABLES	List of abstract variables
INVARIANT	Declaration of the type and of properties of variables
ASSERTIONS	Definition of properties that are deduced from the invariant
INITIALIZATION	Initialization of variables
EVENTS	List and definition of system events
MODALITIES	List of dynamic properties of the system

Restrictions

1. A clause may only appear at most one time in an abstract machine.

2. If one of the CONCRETE_CONSTANTS or ABSTRACT_CONSTANTS clauses are present, then the PROPERTIES clause must be present.
3. If one of the CONCRETE_VARIABLES or ABSTRACT_VARIABLES clauses is present, then the INVARIANT and INITIALIZATION clauses must be present.

2.3 Variant

Syntax

Clause_variant ::= “VARIANT” *Variant*

Variant ::= *Expression_arithmetical*

Description

A VARIANT clause contains a positive integer expression. Every new event introduce in a refinement should decrease the value of this expression. This mechanism allows to guaranty that a new event can’t take the control forever, since variant expression can’t be decreased indefinitely. For each event firing, provided that the invariant *I* of the component and the guard *G* of the event hold, we should demonstrate that the substitution associated to this event decreases the variant.

For example, given an event ev1, V and I respectively the variant and invariant of the system:

ev1 **=** SELECT *P* THEN *S* POST *Q* END

We should demonstrate that

V **:** N **&**

I **&** P **y** [n := V][S](n>V)

Restrictions

1. VARIANT clause may only appear in refinement.

Example

Considering the following system where the only event ev1 non-deterministically chooses a number xx and affects this number to two variables aa and bb:

```
SYSTEM
        toy
VARIABLES
        aa, bb
INVARIANT
        aa : N &
        bb : N &
        aa = bb
INITIALISATION
        aa, bb := 0, 0
EVENTS
        ev1 = ANY xx WHERE xx : N THEN aa, bb := xx, xx END
END
```

We want to refine this system such as:

The concrete event ev1 only modifies the variable aa, by affecting xx value. A new variable bb’, refining bb variable, is given the value 0. Event ev1 can only be fired when aa = bb’.

A new event ev2 progressively increases the value of bb’ by one, if bb’ < aa.

The gluing invariant

(bb’ = aa **y** bb = aa)

indicates when synchronisation between abstract variable bb and concrete variable bb’ occurs (ie they have the same value when bb’ reaches aa).

```
REFINEMENT
        toy_1
REFINES
        toy
VARIABLES
        aa, bb’
INVARIANT
        bb’ : 0..aa & (bb’=aa y bb’=bb)
VARIANT
        aa-bb’
INITIALISATION
        aa, bb’ := 0, 0
EVENTS
        ev1 = ANY xx WHERE xx : N & bb’ = aa THEN aa, bb’ := xx, 0 END
        ;
        ev2 = SELECT bb’<aa THEN bb’ := bb’ +1 END
END
```

We can clearly see that concrete ev1 event refines its abstract counterpart and ev2 refines skip.

The proof obligations related to the VARIANT clause are:

aa **:** **N** **&** bb’ **:** 0..aa **y** aa-bb **:** **N**

aa **:** **N** **&** bb’ **:** 0..aa **&** bb’ < aa **y** aa-(bb’+1) < aa-bb’

2.4 Modalities

Syntax

```
Clause_modalities ::= "MODALITIES" Modality+";"
Modality ::=
                    Modality_maintain
          |         Modality_establish
Modality_maintain ::=
                    "ANY" Ident+"," "WHERE" Predicate "THEN" Event_list "MAINTAIN"
                    Predicate "UNTIL" Predicate "VARIANT" Variant "END"
          |         "BEGIN"  Event_list "MAINTAIN" Predicate "UNTIL" Predicate
                    "VARIANT" Variant "END"
Modality_establish ::=
                    "ANY" Ident+"," "WHERE" Predicate "THEN" Event_list
"ESTABLISH"
                    Predicate "END"
          |         "SELECT" Predicate "THEN" Event_list "ESTABLISH" Predicate
                    "END"
          |         "BEGIN" Event_list "ESTABLISH" Predicate "END"
  Event_list ::=
                    "ALL"
          |         Ident+","
```

Description

MODALITIES clause allows to express dynamic properties of a system. Two modalities can be used : MAINTAIN et ESTABLISH.

General form of MAINTAIN modality is:

ANY *x* WHERE *T* THEN *E* MAINTAIN *P* UNTIL *Q* VARIANT *V* END

where:

- *x* are local variables,
- *T* is a typing predicate for local variables *x*,
- *E* is an event list (E can be replaced by the keyword *ALL* in case *E* contains all the events of the system),
- *P* and *Q* are predicates,
- V is positive integer arithmetical expression.

Note that "ANY *x* WHERE *T* THEN" can be replaced by "BEGIN" if there is no need to introduce local variables.

This modality means that the events contained in *E* should lead to establish the property *Q* (while looping), while maintaining property *P*. For each event *Ei* of *E*, we should demonstrate that (*I* is the invariant of the system):

I **&** T **&** P **& n**Q **y O**$_{\text{i=1..n}}$ grd(Ei)

and

I **&** T **&** P **& n**Q **y** V **: N &**[Ei](**n**Q **y** P) **&** [n := V][Ei](**n**Q **y** V<n)

<u>ESTABLISH modality is simpler. Its general form is</u>:

ANY *x* WHERE *P* THEN *E* ESTABLISH *Q* END

Note that "ANY *x* WHERE *P* THEN" can be replaced by "SELECT *P* THEN" or by "BEGIN".

This modality means that, if *P* holds, events contained in *E* should establish property *Q* in one shot.

Associated proof obligation is (with I invariant of the system):

I **&** P **y** [Ei]Q

For each event *Ei* contained in *E*.

Example

BEGIN *init_set* ESTABLISH *set* **d** *{}* END;

BEGIN *suppress_event*, *modify_set* MAINTAIN *element* **:** *set*
UNTIL *set* = *{}* VARIANT card(*set*) END

2.5 Events

Syntax

```
Clause_events ::= "EVENTS" Event+";"
Event ::=          Ident_ren [ref Ident_ren+","] "="
                   Substitution_event_body
Substitution_event_body ::=
                   Substitution_bloc
              |    Substitution_postcondition
              |    Substitution_selection
              |    Substitution_selection_postcondition
              |    Substitution_unbounded_choice
              |    Substitution_unbounded_choice_postcondition
```

Restrictions

1. An event can’t be named *ALL*.

Description

A system contains a description of its state as well as a number of events. These events are defined in the clause EVENTS. Each event is composed of a guard and an action. The guard is the necessary condition to enable the firing of the event. Once the guard holds, an event may be fired at any time (but it may also never be fired). Once its guards doesn’t hold, an event can’t be fired. Events are atomic. If guards of many events hold simultaneously, only one event may be fired at a time, non-deterministically (it is called external non-determinism) .

Action associated to an event indicates how state variables would evolve when the event is fired. An event can possibly contain a post-condition. A post-condition is a condition that should hold just after the event is fired.

<u>System consistency</u>

Once a system is built, consistency should be proved. A system is consistent if each event preserves the invariant of the system. More precisely, it should be proved that each event modifies state variables in such way that new invariant can be proved, provided that invariant holds with older variable values and event guard holds.

For example, for an event ev **=** SELECT *P* THEN *S* END

We should demonstrate that

$$I \,\&\, P \;\; \mathbf{y} \;\; [S]I$$

If a post-condition Q is associated to the event, we should then demonstrate that

$$I \,\&\, P \;\; \mathbf{y} \;\; [S]I \,\&\, [S]Q$$

Refining a system

Refining a system consists in refining its states and its events. Each event of the abstract system is refined in a more concrete component. An event y (more concrete level) refines an event x (abstract level) when the guard of event y is stronger than the guard of x (guard strengthening). Joint action of both events should also preserve gluing invariant of the system.

Considering an abstract system with a state *v* and an invariant *I(v)*, and a refinement of that system with a state *w* and a gluing invariant *J(v, w)*, if an abstract event and its refinement are of the form:

```
ANY x WHERE                   ANY y WHERE
      P(x, v)                       Q(y, w)
THEN                          THEN
      v := E(x, v)                  w := F(y, w)
END                           END
```

Then we should demonstrate that:

```
I(v) & J(v, w) & Q(y, w)
 y
#x . (P(x, v) & J(E(x, v), f(y, w)))
```

Adding new events

With event B, new events can be introduced during a development. Each new event is supposed to refine an event doing nothing (skip). In this case, proof obligations of the refinement are trivial. Each new event should decrease the variant of the system (see VARIANT clause).

Splitting events

When refining, one event can be refined into many, by using ref keyword. For example, abstract event *evt_x* is refined by *evt_y*, *evt_z* and *evt_t*:

$$\begin{tikzcd}
 & evt\_y \text{ ref } evt\_x = S1 \\
evt\_x = S \arrow[ur, no head] \arrow[r, no head] \arrow[dr, no head] & evt\_z \text{ ref } evt\_x = S2 \\
 & evt\_t \text{ ref } evt\_x = S3
\end{tikzcd}$$

Notice that *evt_x* disappear in the concrete model. In this case, we should demonstrate that *evt_y* refines *evt_x*, *evt_z* refines *evt_x* and *evt_t* refines *evt_x*.

Grouping events

A contrario, many events can be grouped into one concrete event when refining the system.

For example, abstract events evt_x, evt_y and evt_z are grouped into one concrete event : evt_t.

$$\begin{tikzcd}
evt\_x = S1 \arrow[dr, dash] & \\
evt\_y = S2 \arrow[r, dash] & evt\_t \text{ ref } evt\_x, evt\_y, evt\_z = S \\
evt\_z = S3 \arrow[ur, dash] &
\end{tikzcd}$$

Notice that concrete events evt_x, evt_y and evt_z disppear in the concerte model. We should demonstrate that evt_t refines evt_x, evt_y and evt_z.

Example

```
SYSTEM
      maxi_1
CONSTANTS
      nn, tt
PROPERTIES
      nn : NATURAL1 &
      tt : 1..nn 3 NATURAL
VARIABLES
      mm
INVARIANT
      mm : ran(tt)
INITIALISATION
      mm :: ran(tt)
EVENTS

aprog =
      BEGIN
      mm : (mm : ran(tt) & !ii.(ii : 1..nn y tt(ii) <= mm))
      END
END

REFINEMENT
      maxi_2
REFINES
      maxi_1
VARIABLES
      mm, kk
INVARIANT
      kk : 1..nn &
      !ii.(ii : 1..kk y tt(ii) <= mm)
INITIALISATION
      kk := 1 || mm := tt(1)
```

```
EVENTS

aprog =
      SELECT
            kk = nn
      THEN
            skip
      END;

test_1 =
      SELECT
            kk /= nn &
            tt(kk+1) <= mm
      THEN
            kk := kk+1
      POST
            nn - kk >= 0 & nn - kk < nn - kk$0
      END;

test_2 =
      SELECT
            kk /= nn &
            tt(kk+1) > mm
      THEN
            kk := kk+1 ||
            mm := tt(kk+1)
      POST
            nn - kk >= 0 & nn - kk < nn - kk$0
      END
END
```