The B Method - Event B
Formal Software Construction

J. Christian Attiogbé

November 2008, maj 11/2013

Outline

Plan
**Introduction**

**Event-B : References**


- *Applying Event and Machine Decomposition to a Flash-Based Filestore in Event-B*. Damchoom, Kriangsak and Butler, Michael; Conference SBMF 2009.


---

**Event B Specification Approach**

**Correct-by-construction**: build correctly the systems. (abstraction, modelling, refinement, composition/decomposition, proof)

Some hints to formal methods:

- Formal methods are rigorous engineering tools.

- Formal methods are means to build executable code from software requirement documents (informal, natural language).

- **Requirement Documents** (provided by clients) should be rewritten after analysis and understanding into Reference Document (where every thing is made clear and properly labelled for traceability).
B Method and Event B

- Event-B is an **extension of the B-method** (J-R. Abrial).
- It is devoted
  - for **system engineering** (both hardware and software)
  - for **specifying and reasoning about complex systems**: concurrent and reactive systems.
- Event-B comes with a new modelling framework called Rodin.
  (like Atelier B tool for the classical B)
- The **Rodin platform** is an eclipse-based open and extensible tool for B model specification and verification.
  It integrates various plug-ins: B Model editors, proof-obligation generator, provers, model-checkers, UML transformers, etc

**Event B Modelling**

Yet used in various case studies and real cases:
- Train signalling system
- Mechanical press system
- Access control system
- Air traffic information system
- Filestore system
- Distributed programs
- Sequential programs
- etc
**Event B Modelling**

**Observe** the behaviour of any system; what matters?
- We see a set of changes of its states.
- But, the **observation distance** does matter!
  (the details may be observed or not: parachutist paradigm)
- The **observation focus** does matter!
  (the observed changes are not the same)
- Different point of views = **abstraction**.

---

**Event B Specification Approach**

Event B Specification ⇒ Abstract systems or Abstract model
An abstract system is a mathematical model of an asynchronous system behaviour

System behaviour : described by **events**

Events are guarded actions/substitutions The events occurrence involve a State-transition model.
- **Abstract System** (or Model) = Specification unit
- **Refinement** (data and events)
  The parachutist paradigm / microscope paradigm (JR Abrial)
- **Decomposition** (of a system into sub-systems)
The behaviour of a discrete system is a sequence of changes (transition system).

- The changes may be internal or enabled by external signals.
- Each event describes the occurrence of a change in the discrete system under modelisation.
  
  event = when Conditions then Effects

- Event B uses Guards and Actions [Dijkstra]
- But, the behaviour of a system may/should be captured gradually.
Events

An event has one of the following general forms (Fig. 1)

\[
\text{name} = /* \text{event name} */ \\
\text{\textbf{WHEN}} /* \text{formely \textbf{SELECT}} */ \\
P(gcv) \\
\text{\textbf{THEN}} \\
GS(gcv) \\
\text{END}
\]

(WHEN/SELECT Form)

\[
\text{name} = /* \text{event name} */ \\
\text{\textbf{ANY}} \ bv \ \text{WHERE} \\
P(bv, gcv) \\
\text{\textbf{THEN}} \\
GS(bv, gcv) \\
\text{END}
\]

(ANY Form)

Figure: General forms of events

\(bv\) denotes the local bound variables of the event;
\(gcv\) denotes the global constants and variables of the abstract;
\(P(bv, gcv)\) a predicate.

An event without guards has the following form:

\[
\text{name} = /* \text{event name} */ \\
\text{BEGIN} \\
GS(gcv) \\
\text{END}
\]
The guard of an event with the WHEN form is: $P(gcv)$.
The guard of an event with the ANY form is: $\exists(bv).P(bv,gcv)$.
The WHEN form is a particular case of the other.
The action associated to an event is modeled with a generalized substitution using the variables accessible to the event: $GS(bv,gcv)$.

An abstract system describes a mathematical model that simulates the behaviour of a system.
Its semantics arises from the invariant and is enhanced by proof obligations.
The consistency of the model is established by such proof obligations.
Consistency of an event B model:
- PO: the initialisation establishes the invariant
- PO: each event of the abstract system preserves the invariant of the model

$I(gcv)$ the invariant and $GS(bv,gcv)$ the generalized substitution modelling the event action.
Abstract System : Semantics and Consistency

- the initialisation establishes the invariant:

\[ [UI]Inv \]

- each event preserves the invariant:
  In the case of an event with the ANY form, the proof obligation is:

\[ I(gcv) \land P(bv, gcv) \land \text{prd}_v(S) \Rightarrow [GS(bv, gcv)]I(gcv) \]

Moreover the events (e) terminate:

\[ Inv \land eGuard \Rightarrow \text{fis}(eBody) \]

(note that \( Inv \) is \( I(Gcv) \))

The predicate \( \text{fis}(S) \) expresses that \( S \) does not establish \( False \):

\[ \text{fis}(S) \iff \neg [S]False \]

ie

\[ Inv \land eGuard \Rightarrow \neg [S]False \]

The predicate \( \text{prd}_v(S) \) is the \textit{before-after predicate} of the substitution \( S \); it relates the values of state variables just before \((v)\) and just after \((v')\) the substitution \( S \).

The \( \text{prd}_v(\text{ANY } x \text{ WHERE } P(x, v) \text{ THEN } v := S(x, v) \text{ END}) \) is:

\[ \exists x. (P(x, v) \land v' = S(x, v)) \]
Features: Concurrency and synchronization

- Concurrent running of a process consumer which retrieves a data from a buffer filled by another process producer.
- The consumer cannot retrieve an empty buffer and the producer cannot fill in a buffer already full.

An event-driven model of the system is as follows:

```plaintext
system ProdCons /* Model */
sets
  DATA ;
  STATE = {empty, full}
variables
  buffer, bufferstate, bufferc
invariant
  bufferstate ∈ STATE ∧ buffer ∈ DATA ∧ bufferc ∈ DATA
initialization
  bufferstate := empty || buffer := DATA || bufferc := DATA
events
  produce ≡ /* if buffer empty */
    any dd where dd ∈ DATA ∧ bufferstate = empty
    then buffer := dd || bufferstate := full
    end ;
  consume ≡ /* if buffer is full */
    select bufferstate = full
    then bufferc := buffer || bufferstate := empty
    end
end
```
Data refinement
(as usually: new variables + properties; binding invariant)

Event Refinement (extended):
- Strengthening guards (unlike with Classical B)
  More variables are introduced with their properties.
- Each event of the concrete system refines an event of the abstraction.
- Introduction of new events which refine skip, and use new variables.

Let A with Invariant: \( I(\text{av}) \)

\[
\text{evt}_a \equiv */ \text{Abs. ev.}*/ \\
\text{when } P(\text{av}) \\
\text{then } GS(\text{av}) \\
\text{end}
\]

avec \( \text{prd}_v(...) = \text{Ba}(\text{av}, \text{av}') \)

Refined with: Invariant \( J(\text{av}, \text{cv}) \)

\[
\text{evt}_r \equiv */ \text{Conc. ev.}*/ \\
\text{when } Q(\text{cv}) \\
\text{then } GS(\text{cv}) \\
\text{end}
\]

avec \( \text{prd}_v(...) = \text{Bc}(\text{cv}, \text{cv}') \)

Proof obligation:

\[
I(\text{av}) \land J(\text{av}, \text{cv}) \land Q(\text{cv}) \land Bc(\text{cv}, \text{cv}') \Rightarrow \exists \text{cv}'.(\text{Ba}(\text{av}, \text{av}') \land J(\text{av}', \text{cv}'))
\]
Tools

- First generation tools
  - Translation into classical B
  - B4free

- New generation tools: DataBase, Eclipse Plugins, ...
  - Rodin (Deploy Project)

Refinement: structuring models

Refinement= development technique.
Various refinement strategies.

- **Vertical refinement**: From abstract to concrete models.
  Details are gradually introduced in an abstract model in order to make it more concrete.
  The specifier introduces new variables and takes some choices (design), etc.
  Events may be split: event decomposition
  machines may be split too: machine decomposition

- **Horizontal refinement (feature augmentation)**: From a small and abstract to a larger abstract.
  Details are gradually introduced in an abstract model in order to make it more precise
  (wrt to requirements => adding more features, gradually).
Vertical Refinement: event decomposition

A coarse grain event is analysed and described in a more detailed (fine grain) way. Think about the transfer of a file via a network.

- A given change consists of:
  - start by sub-change...;
  - follow by sub-change...;
  - end by sub-change...;

- Hence, at least one sub-change (an event), refines the abstract event.

Machine Decomposition: structuring models

A coarse grain model is analysed and described in a more detailed (fine grain) way. Think about a system involving software and physical devices.

- A given model is made of variables that model purely physical devices, and events are associated only to these variables.
- The splitting is based on variables splitting (but not always straightforward).
- Divide and conquer: a small model is more tractable than a huge one. Decomposition enables one to break complexity, to structure and develop more easily.
Machine Decomposition: structuring models

- **Decomposition with Abrial's style (shared variables):** Machine variables and events are partitioned into sub-machines. The sub-machines may interact with each other via shared variables. Shared variables are duplicated, new external-events are introduced in each machine that has a shared variable in order to ensure consistency of changes.

- **Decomposition with Butler's style:** Machine variables and events are partitioned into sub-machines. The variables are not shared; an event which uses variables in separate machines, is shared (then separated-duplicated). The sub-machines may interact with each other via synchronisation over shared parameterised events.

*Event-B Model Decomposition, Carine Pascal(Systerel) and Renato Silva(University of Southampton)*

---

Structuring Event-B Models

An event-B model is structured with

- **Contexts** that contain carrier sets, axioms and theorems (seen by various machine)

- **Machines** which sees the contexts and defines a state space (static part: variables + labelled invariants) and a dynamic part made of some events.

- A context may be extended; A machine may be refined.
Event-B Model - Example: File transfer protocol

Specification of a file transfer between two sites: a sender and a receiver.

A file is made of a set of data records. From a very abstract level, the transfer is done instantaneously.
But, a file is made of a set of data records which are to be transferred through a channel.

From a more concrete level, the transfer is achieved step by step, one record after the other.
There are some intermediary operations, to send data on the channel from the sender side, to receive data from the channel from the receiver side. In the same way acknowledgements are sent/received.

Only after all the intermediary operations, the transfer will be completed.
**Event-B Model - Example: File transfer protocol**

- Sender file = some data records = 1..nr → DATA
  
  \{1 \map data1, 2 \map data2, \ldots\}

- A channel is a set such data records.

- At each time, the channel contains a part (set inclusion) of the sender’s file

- The receiver acknowledges the received records numbers.

- The file transfer is completed when all the records are acknowledged.

- Failure: loss of data/ack in the channels.

We have the model!
MACHINE Transfer
SETS DATA
CONSTANTS nr /* file size : number of records */
PROPERTIES nr : NAT & nr > 1
VARIABLES
sf /* sender file */
, rf /* receiver file */

INVARIANT
& sf : 1..nr -> DATA /* all records of sf */
& rf : 1..nr +-> DATA /* probably part of records of sf */

INITIALISATION
sf := {} || rf := {}

EVENTS
transf = /* instantaneous transfer, from far way */
BEGIN
rf := sf
END

/* but, technically, we will need to anticipate the intermediary events */

sendta = skip
recdta = skip
sendac = skip
recvac = skip

/* the followings are events that simulate the non-reliability of channels */
rmvData = skip
rmvAck = skip

END
**Introduction**

**Event-B Model Example: File transfer protocol**

**REFINEMENT**

Transfer_R1

**REFINES** Transfer

**VARIABLES**

- cs /* current record to be sent */
- cr /* current record received */
- rf /* sender file */
- erf /* effectively received file */
- dataChan /* data channel */
- ackChan /* ack channel */

**INVARIANT**

- cs : 1..nr+1 /* current to be sent */
- & cr : 0..nr /* current received */
- & cr <= cs /* current received is <= current sent */
- & cs <= cr+1 /* cs <= cr+1 */
- & erf = (1..cr) <| sf
- & dataChan <: (1..cs) <| sf
- & ackChan <: 1..cr

**INITIALISATION**

- cs := 1
- || cr := 0
- || rf := {}
- || sf := {}
- || erf := {}
- || dataChan := {}
- || ackChan := {}

**EVENTS**

- transf = WHEN cs = (nr + 1) /* that is all cs are received (last ack received) */ THEN rf := erf /* not necessary, effective copy of the received file in the receiver */ END

... (continued)

/** new events introduced (ie. we "forget" the anticipation in the abstract model) */

; sendta = WHEN cs <= nr THEN dataChan(cs) := sf(cs) /* now wait for the ack, before updating cs */ END

; recdta = WHEN cr+1 : dom(dataChan) THEN erf(cr+1) := dataChan(cr+1) || cr := cr + 1 /* the next data to be received */ END

; sendac = WHEN cr /= 0 /* send ack for the received cr data */ /* may be observed repeatedly until the next data */ THEN ackChan := ackChan {cr} END

**recvac** = WHEN cs : ackChan /* ack for the already sent cs */ THEN cs := cs + 1 /* now the next to be sent */ END /* Simulating non-reliability of channels, data/ack may be lost */

; rmvData = ANY ii, dd WHERE ii |->dd : dataChan THEN dataChan := dataChan - { ii |->dd } END

; rmvAck = ANY ii WHERE ii : ackChan THEN ackChan := ackChan - {ii} END
Case Study: Multiprocess specification
(Readers/writers)

- **Description**
  - Multiple processes: readers, writers
  - Shared resources between the processes
  - Several readers may read the resource
  - Only one writer at a time

- **Property:**
  Mutual exclusion between readers and writers

- **Improvement:**
  no starvation → as a new property
  (using refinements)

```plaintext
MACHINE readWrite2
SETS WRITER /* set of writer processes */
; READER /* set of reader processes */

VARIABLES writers /* current writers */
, activeWriter
, waitingWriters
, readers /* current readers */
, waitingReaders
, activeReaders /* we may have svrl readers simultan. */
```

J. Christian Attiogbé (November 2008, maj 11)
Multiprocess specification

**INVARIANT**

writers <: WRITER & activeWriter <: WRITER & card(activeWriter) <= 1 & writers \ waitingWriters = {} & activeWriter \ waitingWriters = {} & activeWriter \ writers = {} /* merge */ & readers <: READER & waitingReaders <: READER & activeReaders <: READER & card(activeReaders) >= 0 & readers \ waitingReaders = {} & activeReaders \ waitingReaders = {} & activeReaders \ readers = {} /*-----safety properties -----*/ & not((card(activeWriter) = 1) & (card(activeReaders) >= 1))

**INITIALISATION**

activeWriter := {}
|| waitingWriters := {}
|| activeReaders := {}
|| readers :: POW(READER)
|| writers :: POW(WRITER)
|| waitingReaders := {}/
**Multiprocess specification**

```plaintext
want2write = /* observed when a process wants to write */
  ANY ww WHERE
  ww : writers
  & ww /: waitingWriters
  & ww /: activeWriter
  THEN
  waitingWriters := waitingWriters \ {ww}
  || writers := writers - {ww}
  END

writing =
  ANY ww WHERE
  ww : waitingWriters
  & activeReaders = {} & activeWriter = {} 
  THEN
  activeWriter := {ww}
  || waitingWriters := waitingWriters - {ww}
  END

endWriting =
  ANY ww WHERE
  ww : activeWriter
  THEN
  writers := writers \ {ww}
  || activeWriter := {}
  END

want2read =
  ANY rr WHERE
  rr : readers
  & rr /: waitingReaders
  & rr /: activeReaders
  THEN
  waitingReaders := waitingReaders \ {rr}
  || readers := readers - {rr}
  END
```
Multiprocess specification

```
reading =
  ANY rr WHERE
  rr : waitingReaders
  & activeWriter = {}
  THEN
  activeReaders := activeReaders \ {rr}
  || waitingReaders := waitingReaders - {rr}
  END
;
endReading =
  /* one of the active readers finishes and leaves
  the competition to the shared resources */
  ANY rr WHERE
  rr : activeReaders
  THEN
  activeReaders := activeReaders - {rr}
  || readers := readers \ {rr}
  END

newWriter = /* a new Writer */
  ANY ww
  WHERE ww : WRITER
  & ww /: (writers \ waitingWriters \ activeWriter)
  THEN
  writers := writers \ {ww}
  END
;
leaveWriters = /* a writer leaves the group */
  ANY ww
  WHERE
  ww : writers
  THEN
  writers := writers - {ww}
  END
```
Multiprocess specification

newReader = /* a new reader joins the readers */
ANY rr WHERE
rr : READER
& rr : (readers \ waitingReaders \ activeReaders)
THEN
readers := readers \ {rr}
END

; leaveReader =
ANY rr WHERE
rr : readers & card(readers) > 1
THEN
readers := readers - {rr}
END

Example: Flash-based Filestore in Event-B

To be studied, and summarized.