An Introduction to using Event-B for Cyber-Physical System Specification and Design

John Colley, Michael Butler
University of Southampton
October 14th 2014
Outline

• ADVANCE Project Overview

• Activities supported by Rodin/Event-B Tools

• The Cyber-Physical Development Process

• The Timing Model
ADVANCE(287563)
Advanced Design and Verification Environment for Cyber-Physical System Engineering

- Cyber-Physical Systems
- Key Innovation
- Technical Approach
- Demonstration and Use
Cyber-Physical Systems

• Integrations of Computing and Physical Mechanisms
  – provide physical services
    • Transportation
    • Energy Distribution
    • Medical Care
    • Manufacturing
  – with increased
    • Adaptability
    • Autonomy
    • Efficiency
    • Safety
Cyber-Physical System Challenges

“.... the lack of temporal semantics and adequate concurrency models in computing, and today’s “best effort” networking technologies make predictable and reliable real-time performance difficult, at best. ”

Cyber-Physical Systems - Are Computing Foundations Adequate?
Edward A. Lee, EECS, UC Berkeley, 2006
Verifying Cyber-Physical Systems

• Most Traditional Embedded Systems are *Closed Boxes*
  – amenable to *Bench Testing*

• Cyber-Physical Systems
  – are typically networked
  – can have complex interactions with their physical environment
  – pose a much greater verification challenge

• How can predictable behaviour and timing be achieved?
Key Innovation of ADVANCE

- Focuses on the key role played by Modelling in Cyber-Physical System Engineering
- Modelling is used at all stages of the Development Process
  - From Requirements Analysis to System Acceptance Testing
- Augments Formal, Refinement-based Modelling and Verification with
  - Simulation
  - Testing
  in a Single Design and Verification Environment
Technical Approach: Overview

- **Formal Modelling** supported by strong Formal Verification Tools to establish deep understanding of Specification and Design
- **Simulation-based Verification** to ensure that the Formal Models exhibit the expected behaviour and timing in the target physical environment
- **Model-based Testing** for the systematic generation of high-coverage test suites
The Multi-Simulation Framework

• Different Simulation tools are better suited to simulating different parts of a Cyber-physical system
  – Environments
  – Controllers
  – Physical Plant

• The Framework manages the co-operation of multiple simulators to enable effective Cyber-physical system verification
Demonstration and Use
### ADVANCE Workpackages

<table>
<thead>
<tr>
<th>WP1</th>
<th>Dynamic Trusted Railway Interlocking Case Study</th>
<th>Alstom</th>
</tr>
</thead>
<tbody>
<tr>
<td>WP2</td>
<td>Smart Energy Grids Case Study</td>
<td>Critical</td>
</tr>
<tr>
<td>WP3</td>
<td>Methods and Tools for Model Construction and Proof</td>
<td>Systerel</td>
</tr>
<tr>
<td>WP4</td>
<td>Methods and Tools for Simulation and Testing</td>
<td>Düsseldorf</td>
</tr>
<tr>
<td>WP5</td>
<td>Process Integration</td>
<td>Southampton</td>
</tr>
<tr>
<td>WP6</td>
<td>External Dissemination and Exploitation</td>
<td>Critical</td>
</tr>
<tr>
<td>WP7</td>
<td>Management</td>
<td>Southampton</td>
</tr>
</tbody>
</table>
Achieving high assurance is not easy

• **Requirements** are poorly understood and analysed

• No software system is self-contained
  – it operates within a potentially **complex environment**
  – complexity of environment means that **hazards / vulnerabilities** in environment are poorly understood

• Designs are verified only **after** implementation
  – **expensive** to fix
  – verification usually **incomplete** – many undiscovered bugs
  – Ensuring coverage of faults/attacks in testing is difficult
Verified Design with Event-B

- **Formal** modelling at **early stages** to prevent errors in understanding requirements and environment

- **Verify conformance** between high-level specifications and designs using incremental approach

- **Rodin**: open source toolset for modelling, verification and simulation
Safety/security properties in Event-B

• Aircraft landing gear:
  
  Gear=retracting ⇒ Door=open

• Railway signalling safety:
  – The signal of a route can only be green when all blocks of that route are unoccupied

  \[ \text{sig}(r) = \text{GREEN} \Rightarrow \text{blocks}[r] \cap \text{occupied} = \emptyset \]

• Access control in secure building:
  – \textbf{if} user \( u \) is in room \( r \), \textbf{then} \( u \) must have sufficient authority to be in \( r \)

  \[ \text{location}(u) = r \Rightarrow \text{takeplace}[r] \subseteq \text{authorised}[u] \]
Refinement in Event-B

- High level models
  - abstract details, allowing focus on system-level properties
- Refined models
  - introduce more requirements or design details
- Conformance:
  - behaviour exhibited by refined model should be allowed by abstract model
- Example, signalling mechanism as a refinement:
  - System level property:
    \[ \text{Gear}=\text{retracting} \Rightarrow \text{Door}=\text{open} \]
  - Design level properties:
    \[ \text{Gear}=\text{retracting} \Rightarrow \text{GearRetractSignal}=\text{TRUE} \]
    \[ \text{GearRetractSignal}=\text{TRUE} \Rightarrow \text{Door}=\text{open} \]
Main features of the Rodin Toolset

• Model Verification
  – Ensure that Event-B models satisfy key properties formulated in a mathematical way

• Model Validation
  – Ensure that Event-B models accurately capture the intended behaviour / requirements of a system

• Model Transformation
  – Transform models from one representation to another, e.g.,
    • graphical to mathematical representation
    • model to code transformation
Simple Verification Example

**Invariant:** \( x \leq y \leq x+C \) (y is bounded by x)

**IncEvent** \( \triangleq \) **when** \( y < x+C \) **then** \( y := y+1 \) **end**

- Assume the the system is initialised to a state that satisfies the invariant.
- Can the system ever get into a state in which the invariant is violated?
- Formulate the question as a **mathematical** problem
  - Is this theorem provable?:
    \[ x \leq y \leq x+C \land y < x+C \Rightarrow x \leq y+1 \leq x+C \]
- NB: theorem and its proof hold for **all values** of x,y,C.
Proof Obligations and Provers

• In Event-B theorems such as these are called **Proof Obligations** (POs)
  – The Rodin tool generates the POs for a model automatically

• The Rodin provers (semi-)automatically construct **mathematical proofs** of the validity of the POs.
Counter examples for invalid theorems

• Suppose our event had a specification error:

\[
\text{Invariant: } x \leq y \leq x+C \quad (y \text{ is bounded by } x)
\]

\[
\text{IncEvent } \triangleq \begin{cases} 
\text{when } y \leq x+C \text{ then } y := y+1 \end{cases}
\]

• A Model Checker can generate counterexamples that demonstrate the consequence of the error in \text{IncEvent}:

  – Before: \(x=0, y=2, C=2\) \quad \text{ok}
  – After: \(x=0, y=3, C=2\) \quad \text{fault}

• Model checker can also generate error traces from initial states:

  – Init: \(x=0, y=0, C=2\) \quad \text{ok}
  – IncEvent: \(x=0, y=1, C=2\) \quad \text{ok}
  – IncEvent: \(x=0, y=2, C=2\) \quad \text{ok}
  – IncEvent: \(x=0, y=3, C=2\) \quad \text{fault}
Model Verification in Rodin

• **Proof Obligation generation**
  – Invariant preservation
  – Refinement checking

• **Automated and interactive proof**
  – Proof manager uses a range of internal and external plug-in theorem proving tools
  – Customisable through proof tactics

• **Model checking with ProB plug-in: automated search for**
  – invariant violations
  – refinement violations
  – deadlocks

• **Proof Support for Domain-specific theories**
  – Tables and operators for data manipulation
  – Hierarchical structures (e.g. file system)
  – Train occupancy as chains on a graph
Model Validation

• **Requirements tracing**
  – Validating a formal model against (informal) requirements involves human judgements
  – Strong structuring and traceability helps to ensure that the validation is *comprehensive* and *maintainable*
  – Tracing is supported by ProR plug-in

• **Graphical animation**
  – ProB provides a *simulation engine* for Event-B
  – BMotionStudio allows interactive graphical animations to be constructed, driven by the simulation engine
  – Very valuable for validating model, especially with domain experts
Model Validation (continued)

- **Multi-simulation**
  - Event-B models *discrete* event systems
  - Some environment variables are best represented as *continuous* quantities
    - E.g., voltage, temperature, speed,...
  - **Rodin multi-simulation** framework allows co-simulation of discrete and continuous models
    - links ProB with external simulation tools, e.g., Simulink, Modelica
  - Co-simulation allows us to *validate* a discrete controller model given certain *assumptions* about the (continuous) environment it controls
    - environment variables represented in a continuous model
Continuous / discrete co-simulation

Figure 5. Distribution voltage control system in Modelica

Figure 6. Event-B state machine of the OLTC controller
Co-simulation Results

Figure 7. Co-simulation results of the OLTP voltage control (simulation time = 30s, step size = 0.1s)
Model Transformation

• **UML-B**
  – UML-like graphical notation for Event-B
  – Supports class diagrams and statemachines
  – Graphical representation of refinement

• **Composition and decomposition**
  – Composition: combine models to form larger models
  – Decomposition: split large models into sub-models for further refinement and decomposition
  – Composition and decomposition need to be performed in a disciplined way

• **Code generation**
  – Generate C/Ada/Java from low-level models
  – Customisable
  – Support for generating multi-tasking implementations
UML-B Class diagrams for Bank Accounts

Abstract Class Diagram

Refined Class Diagram
UML-B Statemachines for ATM
Refined model of ATM
Rodin Architecture

• Extension of Eclipse Open Source IDE
• Core Rodin Platform manages:
  – Well-formedness + type checker
  – Consistency/refinement PO generator
  – Proof manager

• Extension points to support plug-ins
  – ProB, Bmotion Studio, ProR
The ADVANCE Process

- Deriving the Safety Constraints from the Functional Requirements using STPA
- Modeling the Safety Constraints in Event-B
  - System-level Safety Constraints
- Determining how Unsafe Control Actions could occur
- Documenting the Requirements and Design Decisions with ProR
- Refining the model and safety constraints to ensure Control Actions are safe in the presence of Hazards
  - Architecture-level Safety Constraints
- Constraint-based test generation and MC/DC coverage
- Shared Event Decomposition
  - Further refinement/ implementation
  - FMI-based Multi-simulation
The Functional Requirements

- System Overview
- Monitored Phenomena
- *Controlled Phenomena*
- Commanded Phenomena
- Mode Phenomena
Controlled Phenomena

Landing Gear Doors

1. The Controller will *open* the Doors when the Pilot moves the Lever to Extend or Retract the Landing Gear
2. The Controller will then *close* the Doors when the Landing Gear is fully Extended or Retracted
3. The Doors will remain *open* while the Landing Gear is Extending or Retracting
Safety Requirements

“Any controller – human or automated – needs a model of the process being controlled to control it effectively”

“Accidents can occur when the controller’s process model does not match the state of the system being controlled and the controller issues unsafe commands.”

Engineering a Safer World, Leveson, 2012
System-Theoretic Process Analysis (STPA)

1. Identify Potentially Hazardous Control Actions and derive the Safety Constraints
2. Determine how Unsafe Control Actions could occur
The Door Sub-system Process Models

Controller

**Process Model**

Door Position
- Locked Open
- Locked Closed
- Opening
- Closing
- Unknown

Actuator

OpenDoor
CloseDoor

Sensor

Door is Locked Open
Door is Locked Closed

Human Operator

**Process Model**

Landing Gear
- Extended/ing
- Retracted/ing
- Unknown

Door Sub-system

Controlled Process

Extend Retract
Step I: Identify Potentially Hazardous Control Actions and Derive Safety Constraints

<table>
<thead>
<tr>
<th>Controller Action</th>
<th>Not Providing Causes Hazard</th>
<th>Providing Causes Hazard</th>
<th>Wrong Timing or Order Causes Hazard</th>
<th>Stopped too soon/Applied too long</th>
</tr>
</thead>
<tbody>
<tr>
<td>Open Door</td>
<td>Cannot extend Landing Gear for landing</td>
<td>Not Hazardous</td>
<td>Not Hazardous</td>
<td>Damage to Landing Gear/Not Hazardous</td>
</tr>
<tr>
<td>Close Door</td>
<td>Not Hazardous</td>
<td>Damage to Landing Gear</td>
<td>Damage to Landing Gear</td>
<td>Not Hazardous/Not Hazardous</td>
</tr>
</tbody>
</table>

**Safety Constraints**

1. If the Landing Gear is Extending, the Door must be Locked Open
2. If the Landing Gear is Retracting, the Door must be Locked Open
3. A “Close Door” command must only be issued if the Landing Gear is Locked Up or Locked Down
4. An “Open Door” command must only be issued if the Landing Gear is Locked Up or Locked Down
Deriving the Formal Safety Constraints

• Natural Language Constraints developed systematically by the *Domain Experts*

1. If the Landing Gear is Extending, the Door must be Locked Open
2. If the Landing Gear is Retracting, the Door must be Locked Open
3. A “Close Door” command must only be issued if the Landing Gear is Locked Up or Locked Down
4. An “Open Door” command must only be issued if the Landing Gear is Locked Up or Locked Down

• Formal, Event-B Safety Constraints
  – Derived systematically from the Natural Language Descriptions
  – Linked to Requirements
    • ProR
Deriving the Formal Safety Constraints

• Natural Language Constraints developed systematically by the *Domain Experts*

1. If the Landing Gear is Extending, the Door must be Locked Open
2. If the Landing Gear is Retracting, the Door must be Locked Open
3. A “Close Door” command must only be issued if the Landing Gear is Locked Up or Locked Down
4. An “Open Door” command must only be issued if the Landing Gear is Locked Up or Locked Down

```plaintext
gearstate ∈ {locked_down, locked_up} ∨ doorstate = locked_open
```

```plaintext
event Close
  where
    @grd1 gearstate ∈ {locked_down, locked_up}
    @grd2 doorstate ∈ {opening, locked_open}
  then
    @act1 doorstate := closing
end
```
The Model Extended FSM

- **G locked_up**
  - **D locked_closed**
  - **CompleteClose**
  - **Open**
  - **Close**

- **G locked_up**
  - **D opening**
  - **CompleteOpen**
  - **Open**

- **G locked_up**
  - **D closing**
  - **Close**

- **G extending**
  - **D locked_open**
  - **CompleteExtend**
  - **Extend**

- **G retracting**
  - **D locked_open**
  - **Retract**

- **G locked_down**
  - **D locked_open**
  - **CompleteOpen**
  - **Close**

- **G locked_down**
  - **D closing**
  - **CompleteClose**

- **G locked_down**
  - **D locked_closed**

**Actions:**
- **Open**
- **Close**
- **Extend**
- **Retract**
Refinement: Introducing the Handle and Timing
Refinement: Introducing the Handle and Timing

ProB

- Model Check
- Generate Tests (Constraint-based)
- Measure Coverage (MC/DC)
  - all the guards of all the events can be set independently to FALSE for all states
Refinement: The Component View Architecture-Level

Controller

Landing Gear Sub-system

gear_extended/
gear_retracted

door_open/
door_closed

open/close
Refinement: The Component View Architecture-Level

Formal Shared Event Decomposition

- Further refinement/implementation of the Controller
- FMI-based Multi-simulation

gear_extended/
gear_retracted

door_open/
door_closed

open/close

© Accellera Systems Initiative 43
Why Model Timing?

A major challenge that CPS present to systems modelling is that a well-developed notion of time needs to be introduced..

[Lee and Seshia, Introduction to Embedded Systems A Cyber-Physical Approach 2011]

.. and often this is necessary quite early in the model refinement process.

*We want to reason formally about the temporal properties of a Cyber-Physical System.*
Why Synchronous?

• Critical timing paths can be identified through formal static timing analysis
  – Prove that the clock period is sufficiently long
• Proven synthesis route to a hardware implementation
• Enables interrupt-free software implementations
  – Easier to verify for safety-critical implementation
Milner's Synchronous Calculus of Communicating Systems (SCCS)

- \( P \xrightarrow{a} P' \)
  - An agent \( P \) may perform an atomic action \( a \) and become \( P' \) in doing so.

- Assuming time is discrete
  - \( P \) at time \( t \) becomes \( P' \) at time \( t + 1 \)
  - Actions are atomic in the sense that they are indivisible *in time* (but not indivisible in every sense)

- A system of 3 agents \( P, Q \) and \( R \) where
  - \( P \xrightarrow{a} P' \), \( Q \xrightarrow{b} Q' \), \( R \xrightarrow{c} R' \)
  - can perform the product (\( X \)) of \( a, b \) and \( c \) *simultaneously* and \( X \) is associative and commutative

- An agent that cannot perform an action must at least be able to perform an *idle* action \( i \)
  - Otherwise “disaster”

*Calculi for Synchrony and Asynchrony, 1982*
Abstract Untimed Specification

event EstablishCommsLink
  where
  @grd1 ControllerActive = FALSE
  then
  @act1 ControllerActive := TRUE
end
Not an Atomic process, so

\[
\text{event } \text{InitiateCommsLink} \\
\text{where} \\
\quad @\text{grd1} \text{ControllerActive} = \text{FALSE} \\
\quad @\text{grd2} \text{TCInit} = \text{FALSE} \\
\text{then} \\
\quad @\text{act1} \text{TCInit} := \text{TRUE} \\
\text{end}
\]

\[
\text{event } \text{CompleteCommsLink} \text{ refines EstablishCommsLink} \\
\text{any } \text{pack} \\
\text{where} \\
\quad @\text{grd1} \text{pack} = \text{TRUE} \\
\quad @\text{grd2} \text{TCInit} = \text{TRUE} \\
\quad @\text{grd3} \text{ControllerActive} = \text{FALSE} \\
\text{then} \\
\quad @\text{act1} \text{ControllerActive} := \text{TRUE} \\
\text{end}
\]

AND THEN …
Handle Soft and Hard Errors

**event** InitiateCommsLink

**where**
- @grd1 ControllerActive = FALSE
- @grd2 TCInit = FALSE

**then**
- @act1 TCInit := TRUE

**end**

**event** CompleteCommsLink **refines** EstablishCommsLink

**any pack**

**where**
- @grd1 ControllerActive = FALSE
- @grd2 TCInit = FALSE

**then**
- @act1 TCInit := TRUE

**end**

**event** RetryCommsLink

**any** pack

**where**
- @grd1 pack = FALSE
- @grd2 TCInit = TRUE
- @grd3 ControllerActive = FALSE
- @grd4 RetryCount > 0

**then**
- @act1 RetryCount := RetryCount - 1

**end**

**event** CompleteCommsLinkFail

**any pack**

**where**
- @grd1 pack = TRUE
- @grd2 TCInit = TRUE
- @grd3 ControllerActive = FALSE

**then**
- @act1 ControllerActive := TRUE

**end**
Are we Done?

- \( p \xrightarrow{a} p' \)
  - We have a set of four atomic actions \( a \)
  - At least one event in the system is always enabled
  - Under the interpretation “the evaluation of an event advances discrete time” we have implemented SCCS

- **BUT**
  - This is a very simple system
  - For complex CPS it is not usually feasible to represent the action as a single event
  - Recall “Actions are atomic in the sense that they are indivisible in time (but not indivisible in every sense)”
  - and we have only considered a single process ....
Implementing SCCS with Actions comprising multiple events

- After the system initiates the comms link it WAITs for a response
- If there is no response, it retries and WAITs again
- So, we implement an action as a sequence of events:
  - Evaluate, E1, E2, ... Wait
Multi-event action: $C \rightarrow C'$

**event** CEvaluate

where
- $\text{@grd1 CEvaluated} = \text{FALSE}$
- $\text{@grd2 Cstep} = 0$

then
- $\text{@act1 Cstep} := 1$

end

**event** CWait

where
- $\text{@grd1 Cstep} = 2$

then
- $\text{@act1 Cstep} := 0$
- $\text{@act2 CEvaluated} := \text{TRUE}$

end

**event** InitiateCommsLink

where
- $\text{@grd1 ControllerActive} = \text{FALSE}$
- $\text{@grd2 TCInit} = \text{FALSE}$
- $\text{@grd3 Cstep} = 1$

then
- $\text{@act1 TCInit} := \text{TRUE}$
- $\text{@act2 Cstep} := 2$

end

..

**event** CompleteCommsLink reﬁnes EstablishCommsLink

any *pack*

where
- $\text{@grd1 pack} = \text{TRUE}$
- $\text{@grd2 TCInit} = \text{TRUE}$
- $\text{@grd3 ControllerActive} = \text{FALSE}$
- $\text{@grd4 Cstep} = 1$

then
- $\text{@act1 ControllerActive} := \text{TRUE}$
- $\text{@act2 Cstep} := 2$

end
**Update Event advances Time**

---

**event** CEvaluate  
where  
@grd1 CEvaluated = FALSE  
@grd2 Cstep = 0  
then  
@act1 Cstep := 1  
end

**event** CWait  
where  
@grd1 Cstep = 2  
then  
@act1 Cstep := 0  
@act2 CEvaluated := TRUE  
end

**event** Update  
where  
@grd1 CEvaluated = TRUE  
then  
@act1 CEvaluated := FALSE  
end

---

**event** InitiateCommsLink  
where  
@grd1 ControllerActive = FALSE  
@grd2 TCInit = FALSE  
@grd3 Cstep = 1  
then  
@act1 TCInit := TRUE  
@act2 Cstep := 2  
end

..  

**event** CompleteCommsLink refines  
EstablishCommsLink

**any** pack  
where  
@grd1 pack = TRUE  
@grd2 TCInit = TRUE  
@grd3 ControllerActive = FALSE  
@grd4 Cstep = 1  
then  
@act1 ControllerActive := TRUE  
@act2 Cstep := 2  
end
2 processes: $C \rightarrow C'$, $T \rightarrow T'$

**event** CEvaluate

where

@grd1 CEvaluated = FALSE
@grd2 Cstep = 0
then
@act1 Cstep := 1
end

...  

**event** CWait

where

@grd1 Cstep = 2
then
@act1 Cstep := 0
@act2 CEvaluated := TRUE
end

**Event** TEvaluate

where

@grd1 TEvaluated = FALSE
@grd2 Tstep = 0
then
@act1 Tstep := 1
end

...  

**Event** TWait

where

@grd1 Tstep = 3
then
@act1 Tstep := 0
@act2 TEvaluated := TRUE
end

**event** Update

where

@grd1 CEvaluated = TRUE
@grd2 TEvaluated = TRUE
then
@act1 CEvaluated := FALSE
@act2 TEvaluated := FALSE
end
Inter-process Communication

**event** InitiateCommsLink
   
   **where**
   
   @grd1 ControllerActive = FALSE
   @grd2 TCInit = FALSE
   @grd3 Cstep = 1
   
   **then**
   
   @act1 TCInit := TRUE
   @act2 Cstep := 2

**end**

**..**

**event** CompleteCommsLink  **refines** EstablishCommsLink

**any** pack
   
   **where**
   
   @grd1 pack ∈ BOOL
   @grd2 TCInit = TRUE
   @grd3 TCAcknowledgeInit = TRUE
   @grd4 ControllerActive = FALSE
   @grd5 Cstep = 1
   
   **then**
   
   @act1 ControllerActive := TRUE
   @act2 Cstep := 2

**end**

**event** TAcknowledgeInit

**any** pack
   
   **where**
   
   @grd1 pack ∈ BOOL
   @grd2 TCInit = TRUE
   @grd3 TStep = 1
   
   **then**
   
   @act1 TCAcknowledgeInit := pack
   @act2 TStep := 2

**end**

**..**

**event** Update

   **where**
   
   @grd1 CEvaluated = TRUE
   @grd2 TEvaluated = TRUE
   
   **then**
   
   @act1 CEvaluated := FALSE
   @act2 TEvaluated := FALSE

**end**

© Accellera Systems Initiative 55
Inter-process Communication

**event InitiateCommsLink**

```plaintext
where
    @grd1 ControllerActive = FALSE
    @grd2 TCInit = FALSE
    @grd3 Cstep = 1
then
    @act1 TCInit := TRUE
    @act2 Cstep := 2
```

**event TAcknowledgeInit**

```plaintext
any pack
where
    @grd1 pack ∈ BOOL
    @grd2 TCInit = TRUE
    @grd3 Tstep = 1
then
    @act1 TCAcknowledgeInit := pack
    @act2 Tstep := 2
```

**Evaluation of C and T is order dependent!**

Does not preserve *commutativity* required by SCCS

**Race Condition**
Preserving evaluation order independence

**event** InitiateCommsLink
where
@grd1 ControllerActive = FALSE
@grd2 TCInit = FALSE
@grd3 Cstep = 1
then
@act1 TCInitprime := TRUE
@act2 Cstep := 2
end

..

**event** CompleteCommsLink refines EstablishCommsLink
any pack
where
@grd1 pack ∈ BOOL
@grd2 TCInit = TRUE
@grd3 TCAcknowledgeInit = TRUE
@grd4 ControllerActive = FALSE
@grd5 Cstep = 1
then
@act1 ControllerActive := TRUE
@act2 Cstep := 2
end

..

**event** TAcknowledgeInit
any pack
where
@grd1 pack ∈ BOOL
@grd2 TCInit = TRUE
@grd3 Tstep = 1
then
@act1 TCAcknowledgeInitprime := pack
@act2 Tstep := 2
end

..

**event** Update
where
@grd1 CEvaluated = TRUE
@grd2 TEvaluated = TRUE
then
@act1 CEvaluated := FALSE
@act2 TEvaluated := FALSE
@act3 TCInit := TCInittprime
@act4 TCAcknowledgeInit := TCAcknowledgeInitprime
end
Preserving evaluation order independence

**event InitiateCommsLink**

where
- @grd1 ControllerActive = FALSE
- @grd2 TCInit = FALSE
- @grd3 Cstep = 1

then
- @act1 TCInit := TRUE
- @act2 Cstep := 2

end

**event CompleteCommsLink**

refines

**any pack**

where
- @grd1 pack ∈ BOOL
- @grd2 TCInit = TRUE
- @grd3 TCAcknowledgedInit = TRUE
- @grd4 ControllerActive = FALSE
- @grd5 Cstep = 1

then
- @act1 ControllerActive := TRUE
- @act2 Cstep := 2

end

**event T AcknowledgeInit**

any pack

where
- @grd1 pack ∈ BOOL

then
- @act1 TCAcknowledgedInit := pack
- @act2 Tstep := 2

end

**event Update**

where
- @grd1 CEvaluated = TRUE
- @grd2 TEvaluated = TRUE

then
- @act1 CEvaluated := FALSE
- @act2 TEvaluated := FALSE
- @act3 TCInit := TCInitprime
- @act4 TCAcknowledgedInit := TCAcknowledgedInitprime

end

*Update event updates the values of the communication variables and advances time when all processes have evaluated*
Timing Summary

• Specification refinement begins with an untimed model
• Refinement introduces sequences of temporal events
• Implementing SCCS semantics in the refined Event-B model enables synchronisation and communication between processes without race
  – Implements HDL cycle-based semantics
  – Enables HDL and Assertion generation from Event-B
Important Messages

• System assurance can be strengthened
  – using systematic processes and verified design

• Role of systematic requirements and safety analysis
  – Structures to focus the analysis
  – Path to formalisation

• Role of formal modelling and refinement:
  – increase understanding, decrease errors
  – manage complexity through multiple levels of abstraction

• Role of verification and tools:
  – improve quality of models (validation + verification)
  – make verification as automatic as possible, pin-pointing errors and even suggesting improvements

© Accellera Systems Initiative
Questions