Learning Material: Code Generation with Tasking Event-B

A. Edmunds

Electronics and Computer Science, University of Southampton

March 26, 2013
Event-B
- Background
- Overview of Event-B
- Composition / Decomposition

Implementation-Level Modelling
- Tasking Event-B
- The User Interface: Machine and Event Annotations

Adding New Types, and Translation Rules
- Translation Rules for Ada
- Example of Adding a New Type
Outline

1. Event-B
   - Background
     - Overview of Event-B
     - Composition / Decomposition

2. Implementation-Level Modelling
   - Tasking Event-B
   - The User Interface: Machine and Event Annotations

3. Adding New Types, and Translation Rules
   - Translation Rules for Ada
   - Example of Adding a New Type
Automatic Code Generation from Event-B To Ada, C, Java
- FMI C or OpenMP C.
- Java for concurrent shared memory systems.
- Modelling of Controllers / Protected, Shared Data and Environment.

Extensible Mathematical Language Translations:
- add new Types, and their Implementations.
What We Have...

- Automatic Code Generation from Event-B To Ada, C, Java
  - FMI C or OpenMP C.
  - Java for concurrent shared memory systems.
  - Modelling of Controllers / Protected, Shared Data and Environment.
- Extensible Mathematical Language Translations:
  - add new Types, and their Implementations.
From the EU funded RODIN, DEPLOY, and ADVANCE projects:

- [http://www.event-b.org/](http://www.event-b.org/)
- ... a unified tool-based framework for automated formal verification and simulation-based validation of cyber-physical systems.

Rodin Tools - A new not-for-profit company.
Outline

1. Event-B
   - Background
   - Overview of Event-B
   - Composition / Decomposition

2. Implementation-Level Modelling
   - Tasking Event-B
   - The User Interface: Machine and Event Annotations

3. Adding New Types, and Translation Rules
   - Translation Rules for Ada
   - Example of Adding a New Type
Event-B

- Based on Set-Theory + Predicate Logic + Arithmetic,
  - Tool Support, with Automatic and Interactive proof.
  - Refinement, for incremental development.

- Context Component.
  - Specify Sets, Constants, and Axioms.

- Machine Component.
  - Specify Variables, Invariants, and Events.

- Theory Component
  - Add new Types, Operators.
  - Add new Translation, Re-write Rules etc.
Based on Set-Theory + Predicate Logic + Arithmetic,
- Tool Support, with Automatic and Interactive proof.
- Refinement, for incremental development.

Context Component.
- Specify Sets, Constants, and Axioms.

Machine Component.
- Specify Variables, Invariants, and Events.

Theory Component
- Add new Types, Operators.
- Add new Translation, Re-write Rules etc.
Event-B

- Based on Set-Theory + Predicate Logic + Arithmetic,
  - Tool Support, with Automatic and Interactive proof.
  - Refinement, for incremental development.

- Context Component.
  - Specify Sets, Constants, and Axioms.

- Machine Component.
  - Specify Variables, Invariants, and Events.

- Theory Component
  - Add new Types, Operators.
  - Add new Translation, Re-write Rules etc.
Event-B

- Based on Set-Theory + Predicate Logic + Arithmetic,
  - Tool Support, with Automatic and Interactive proof.
  - Refinement, for incremental development.
- Context Component.
  - Specify Sets, Constants, and Axioms.
- Machine Component.
  - Specify Variables, Invariants, and Events.
- Theory Component
  - Add new Types, Operators.
  - Add new Translation, Re-write Rules etc.
... from the Heater Controller Example.

CONTEXT
   HC_CONTEXT
CONSTANTS
   Max
   Min
AXIOMS
   axm1 : Max = 45
   axm2 : Min = 5
   axm3 : Max ∈ ℤ
   axm4 : Min ∈ ℤ
END
MACHINE HCtrl_M0
SEES HC_CONTEXT
VARIABLES
hsc // heat source commanded
nha // no heat alarm
cttm2 // commanded target temp
...

INVARIANTS
typing_nha : nha ∈ BOOL
typing_hsc : hsc ∈ BOOL
typing_ota : cttm2 ∈ ℤ
...

EVENTS
INITIALISATION ≜
BEGIN
act3: hsc := FALSE
act4: nha := FALSE
act5: cttm2 :=∈ Z
...
END
Event-B - Events

TurnON_Heat_Source ≙
REFINES
TurnON_Heat_Source
WHEN
    // average temp less
grd1: avt < cttm2 // than commanded
    // value
THEN
    act1: hsc := TRUE // Turn heat source on
END

Based on guarded command: \( g \rightarrow a \)
- In Event-B, the guard \( g \) is an Event-B predicate;
- the action \( a \) is an Event-B expression.
Event-B - Event Parameters

Sense_Temperatures ≜
ANY t1 t2
WHERE grd1: t1 ∈ ℤ
grd2: t2 ∈ ℤ
THEN act1: stm1 := t1
act2: stm2 := t2
END

- The **ANY** construct admits parameters:
  - Parameters are typed in the Guard;
  - but may not be assigned to.
Outline

1. Event-B
   - Background
   - Overview of Event-B
   - Composition / Decomposition

2. Implementation-Level Modelling
   - Tasking Event-B
   - The User Interface: Machine and Event Annotations

3. Adding New Types, and Translation Rules
   - Translation Rules for Ada
   - Example of Adding a New Type
Distribute Variables Between Machines

Abstract Machine

Variables

M1
M2
M3
Events are Refactored.
- Synchronization $e_a \parallel e_b$ models an atomic subroutine call.
- The Composed Machine is a Refinement.
The Heater Controller Development

First Level Decomposition
- Heating_Ctrl_M0
  - Heating_Ctrl_M1
    - Environment
    - HCtrl_M0
      - Environment1
    - HCtrl_M1
  - First Level Decomposition
    - Second Level Decomposition
      - Shared Object
      - Temperature Ctrl Task
      - Heater Monitor Task
      - Display Update Task
Outline

1. Event-B
   - Background
   - Overview of Event-B
   - Composition / Decomposition

2. Implementation-Level Modelling
   - Tasking Event-B
   - The User Interface: Machine and Event Annotations

3. Adding New Types, and Translation Rules
   - Translation Rules for Ada
   - Example of Adding a New Type
Using ‘Annotated’ Event-B models - Tasking Event-B.

- Specify a task’s priority, and type (periodicity etc.) Formal modelling of time is in its early stages.
- A Machine’s Task-Body - formally describes the flow of execution,
- is the basis for refinement of the Abstract Development.
Using ‘Annotated’ Event-B models - Tasking Event-B.

Specify a task’s priority, and type (periodicity etc.) Formal modelling of time is in its early stages.

A Machine’s Task-Body - formally describes the flow of execution,

is the basis for refinement of the Abstract Development.
Using ‘Annotated’ Event-B models - Tasking Event-B.

Specify a task’s priority, and type (periodicity etc.) Formal modelling of time is in its early stages.

A Machine’s Task-Body - formally describes the flow of execution, is the basis for refinement of the Abstract Development.
Using ‘Annotated’ Event-B models - Tasking Event-B.

Specify a task’s priority, and type (periodicity etc.) Formal modelling of time is in its early stages.

A Machine’s Task-Body - formally describes the flow of execution,

is the basis for refinement of the Abstract Development.
Correspondence with Ada

- **AutoTask Machines**
  - map to Controller Task Implementations;
  - anonymous tasks declared in main.

- **Environ Machines**
  - map to Environment Tasks

- **Environment Tasks**
  - simulate the environment,
  - or, provide an interface to the environment
  - (to be explored in the Advance project)

- **Shared Machines**
  - map to Protected Objects in Ada
Correspondence with Ada

- **AutoTask Machines**
  - map to Controller Task Implementations;
  - anonymous tasks declared in main.

- **Environ Machines**
  - map to Environment Tasks.

- **Environment Tasks**
  - simulate the environment,
  - or, provide an interface to the environment
  - (to be explored in the Advance project)

- **Shared Machines**
  - map to Protected Objects in Ada
Correspondence with Ada

- **AutoTask Machines**
  - map to Controller Task Implementations;
  - anonymous tasks declared in main.

- **Environ Machines**
  - map to Environment Tasks.

- **Environment Tasks**
  - simulate the environment,
  - or, provide an interface to the environment.
  - (to be explored in the Advance project)

- **Shared Machines**
  - map to Protected Objects in Ada.
Correspondence with Ada

- **AutoTask Machines**
  - map to Controller Task Implementations;
  - anonymous tasks declared in main.

- **Environ Machines**
  - map to Environment Tasks.

- **Environment Tasks**
  - simulate the environment,
  - or, provide an interface to the environment.
  - (to be explored in the Advance project)

- **Shared Machines**
  - map to Protected Objects in Ada.
Correspondence with Ada

- **Mapping of events**
  - depends on use in task body.
  - Some event guards and actions are ‘in-lined’.
  - Some events map to ‘subroutines’, and are called.
  - Guards
    - map to entry barriers,
    - or, looping/branching statements.
  - The code generator takes care of this.

- **Synchronizations:**
  - Tasking & Shared Machine = protected subprogram/entry
  - Tasking & Environ Machine = rendezvous
Correspondence with Ada

- Mapping of events
  - depends on use in task body.
    - Some event guards and actions are ‘in-lined’.
    - Some events map to ‘subroutines’, and are called.
    - Guards
      - map to entry barriers,
      - or, looping/branching statements.
    - The code generator takes care of this.

- Synchronizations:
  - Tasking & Shared Machine = protected subprogram/entry
  - Tasking & Environ Machine = rendezvous
Correspondence with Ada

- **Mapping of events**
  - depends on use in task body.
  - Some event guards and actions are ‘in-lined’.
  - Some events map to 'subroutines', and are called.
  - Guards
    - map to entry barriers,
    - or, looping/branching statements.
  - The code generator takes care of this.

- **Synchronizations:**
  - Tasking & Shared Machine = protected subprogram/entry
  - Tasking & Environ Machine = rendezvous
Correspondence with Ada

- Mapping of events
  - depends on use in task body.
  - Some event guards and actions are ‘in-lined’.
  - Some events map to ‘subroutines’, and are called.
- Guards
  - map to entry barriers,
  - or, looping/branching statements.
- The code generator takes care of this.
- Synchronizations:
  - Tasking & Shared Machine = protected subprogram/entry
  - Tasking & Environ Machine = rendezvous
Correspondence with Ada

- Mapping of events
  - depends on use in task body.
  - Some event guards and actions are ‘in-lined’.
  - Some events map to ‘subroutines’, and are called.
  - Guards
    - map to entry barriers,
    - or, looping/branching statements.

- The code generator takes care of this.

- Synchronizations:
  - Tasking & Shared Machine = protected subprogram/entry
  - Tasking & Environ Machine = rendezvous
Correspondence with Ada

• **Mapping of events**
  - depends on use in task body.
  - Some event guards and actions are ‘in-lined’.
  - Some events map to ‘subroutines’, and are *called*.
  - Guards
    - map to entry barriers,
    - or, looping/branching statements.
  - The code generator takes care of this.

• **Synchronizations:**
  - Tasking & Shared Machine = protected subprogram/entry
  - Tasking & Environment Machine = rendezvous
Correspondence with Ada

- **Mapping of events**
  - depends on use in task body.
  - Some event guards and actions are ‘in-lined’.
  - Some events map to ’subroutines’, and are called.
  - Guards
    - map to entry barriers,
    - or, looping/branching statements.
  - The code generator takes care of this.

- **Synchronizations:**
  - Tasking & Shared Machine = protected subprogram/entry
  - Tasking & Environ Machine = rendezvous.
Correspondence with Ada

- Mapping of events
  - depends on use in task body.
  - Some event guards and actions are ‘in-lined’.
  - Some events map to ‘subroutines’, and are called.
- Guards
  - map to entry barriers,
  - or, looping/branching statements.
- The code generator takes care of this.

- Synchronizations:
  - Tasking & Shared Machine = protected subprogram/entry.
  - Tasking & Environ Machine = rendezvous.
Correspondence with Ada

- Mapping of events
  - depends on use in task body.
  - Some event guards and actions are ‘in-lined’.
  - Some events map to ’subroutines’, and are called.
  - Guards
    - map to entry barriers,
    - or, looping/branching statements.
  - The code generator takes care of this.

- Synchronizations:
  - Tasking & Shared Machine = protected subprogram/entry.
  - Tasking & Environ Machine = rendezvous.
The Common Language Meta-model is independent of the implementation; an abstraction based on Ada.
Outline

1. Event-B
   - Background
   - Overview of Event-B
   - Composition / Decomposition

2. Implementation-Level Modelling
   - Tasking Event-B
   - The User Interface: Machine and Event Annotations

3. Adding New Types, and Translation Rules
   - Translation Rules for Ada
   - Example of Adding a New Type
UI - Specifying a Task Body

Integrated with

- Machine Editor.

```plaintext
Get_Target_Temperature();
Sense_PressIncrease_Target_Temperature;
if Raise_Target_Temperature
else Raise_Target_Temperature Blocked;
Sense_PressDecrease_Target_Temperature;
if Lower_Target_Temperature
else Lower_Target_Temperature Blocked;
Set_Target_Temperature;
Display_Target_Temperature
```
UI - Events

- Synchronized Events

- Parameter Directions.

- Typing.

```
Get_Target_Temperature1 ⊑
   COMBINES EVENT
   Shared Object IMPL.Get Target Temperature1 ||
   Display_Update_Task_IMPL.Get_Target_Temperature1
   REFINES
   Get_Target_Temperature1

Get_Target_Temperature1 ⊑
   REFINES
   Get_Target_Temperature1
   ANY
   in tm
   WHERE
   grd1 : tm ∈ Z  TYPING
   THEN
   act1 : ctm1 := tm
   END
```
Generating Code

Event-B Implementations-Level Modelling
Adding New Types, and Translation Rules

Tasking Event-B
The User Interface: Machine and Event Annotations

Learning Material: Code Generation with Tasking Event-B
Outline

1. Event-B
   - Background
   - Overview of Event-B
   - Composition / Decomposition

2. Implementation-Level Modelling
   - Tasking Event-B
   - The User Interface: Machine and Event Annotations

3. Adding New Types, and Translation Rules
   - Translation Rules for Ada
   - Example of Adding a New Type
Using Mathematical Extensions

THEORY AdaRules

TRANSLATOR Ada

Metavariables: \( a \in \mathbb{Z}, \ b \in \mathbb{Z}, \ c \in \mathbb{Q}, \ d \in \mathbb{Q} \)

Translator Rules

\[
\begin{align*}
\text{trns2}: & \quad a - b \mapsto a - b \\
\text{trns9}: & \quad c = d \mapsto c = d \\
\text{trns19}: & \quad a \neq b \mapsto a \neq b \\
\text{trns21}: & \quad a \ mod \ b \mapsto a \ mod \ b \\
\text{trns22}: & \quad \neg c \mapsto \neg(c) \\
\text{trns23}: & \quad c \lor d \mapsto (c) \lor (d) \\
\text{trns24}: & \quad c \land d \mapsto (c) \land (d) \\
\text{trns25}: & \quad c \Rightarrow d \mapsto \neg(c) \lor (d)
\end{align*}
\]

Type Rules

\[
\begin{align*}
\text{typeTrns1}: & \quad \mathbb{Z} \mapsto \text{Integer} \\
\text{typeTrns2}: & \quad \mathbb{BOOL} \mapsto \text{boolean}
\end{align*}
\]
Outline

1. Event-B
   - Background
   - Overview of Event-B
   - Composition / Decomposition

2. Implementation-Level Modelling
   - Tasking Event-B
   - The User Interface: Machine and Event Annotations

3. Adding New Types, and Translation Rules
   - Translation Rules for Ada
   - Example of Adding a New Type
THEORY Array

TYPE PARAMETERS T

OPERATORS

•array : array(s : P(T))
  direct definition
  \( \text{array}(s : \mathbb{P}(T)) \equiv \{ \, n, f \cdot n \in \mathbb{N} \land f \in 0 \cdot (n-1) \rightarrow s \mid f \} \)

•arrayN : arrayN(n : \mathbb{Z}, s : \mathbb{P}(T))
  well-definedness condition \( n \in \mathbb{N} \land \text{finite}(s) \)
  direct definition
  \( \text{arrayN}(n : \mathbb{Z}, s : \mathbb{P}(T)) \equiv \{ \, a \mid a \in \text{array}(s) \land \text{card}(s) = n \} \)
Theory: Translation Rules for Arrays

• update : update(a : $\mathbb{Z} \leftrightarrow T$, i : $\mathbb{Z}$, x : T)

• lookup : lookup(a : $\mathbb{Z} \leftrightarrow T$, i : $\mathbb{Z}$)

• newArray : newArray(n : $\mathbb{Z}$, x : T)

TRANSLATOR Ada

Metavariabes s ∈ $\mathcal{P}(T)$, n ∈ $\mathbb{Z}$, a ∈ $\mathbb{Z} \leftrightarrow T$, i ∈ $\mathbb{Z}$, x ∈ T

Translator Rules

trns1 : lookup(a,i) ⇒ a(i)
trns2 : a = update(a,i,x) ⇒ a(i) := x
trns3 : newArray(n,x) ⇒ (others ⇒ x)

Type Rules

typeTrns1 : arrayN(n,s) ⇒ array (0..n-1) of s
Theory: Applying the Rules for Arrays

Event-B:

\[
\begin{align*}
\text{Invariants} & \quad \text{cbuf} \in \text{arrayN}(\text{maxbuf}, \mathbb{Z}) \\
\text{Initialisation} & \quad \text{cbuf} := \text{newArray}(\text{maxbuf}, \theta)
\end{align*}
\]

\[\downarrow\]

type rule: \quad \text{arrayN}(n,s) \mapsto \text{array}(\theta..n-1) \text{ of } s

c\_constructor: \quad \text{newArray}(n,x) \mapsto (\text{others} \Rightarrow x)
\]

\[\downarrow\]

Ada:

```
type cbuf\_array is array (0..maxbuf-1) of Integer;
cbuf : cbuf\_array := (others => 0);
```
Wrapping Up

- Tasking Event-B guides code generation.
- Event-B modelling artefacts correspond to:
  - Ada Tasks - Protected Objects;
  - Java threads - monitors.
  - C which uses PThread library.
- The Common Language Meta-model is an abstraction of commonly used programming constructs.
- Tasking Event-B has: AutoTask, Environ and Shared machines
  - AutoTask Machines have a Task-body to specify flow of control.
  - The Tasking Language has sequence, branch and loop constructs.
We make use of the tool-driven decomposition approach, to structure the development.

- This allows us to partition the system in a modular fashion, reflecting implementation constructs.
- Decomposition is also the mechanism for breaking up complex systems to make modelling and proof more tractable.

We have data-type and operator extensibility.

Target Language specification is extensible.