Learning Material: Code Generation with Tasking Event-B

A. Edmunds

Electronics and Computer Science, University of Southampton

March 26, 2013

Outline

Outline

What We Have...

- Automatic Code Generation from Event-B To Ada, C, Java
  - FMI C or OpenMP C.
  - Ada for Multi-Tasking Embedded Systems.
  - 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.
  - Ada for Multi-Tasking Embedded Systems.
  - Java for concurrent shared memory systems.
  - Modelling of Controllers / Protected, Shared Data and Environment.
- Extensible Mathematical Language Translations:
  - add new Types, and their Implementations.

Resources

- From the EU funded RODIN, DEPLOY, and ADVANCE projects:
  - http://www.event-b.org/
  - http://wiki.event-b.org/index.php/Main_Page
  - http://www.advance-ict.eu/
  - *. . . 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

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.

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, Rewrite 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 - Context

... 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
```

Event-B - Machines, Variables etc.

```
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

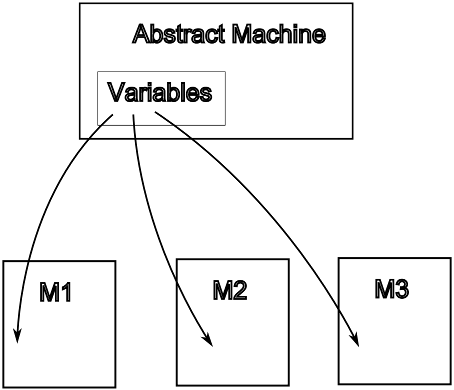
```
Sense_Temperatures   ≙
ANY t1 t2
WHERE grd1: t1 ∈ ℤ
      grd2: t2 ∈ ℤ
THEN act1:  stm1 ≔ t1
     act2:  stm2 ≔ t2
END
```

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

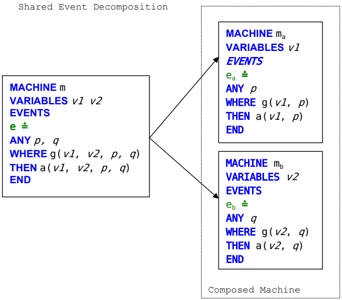
Outline

Decomposition

Distribute Variables Between Machines

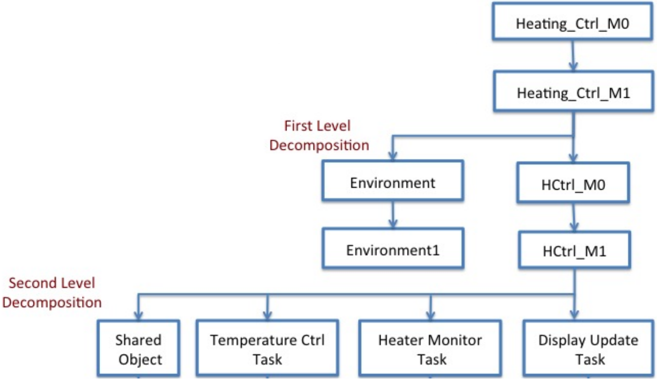


Automatic Decomposition



- Events are Refactored.
- Synchronization $e_a \parallel e_b$ models an atomic subroutine call.
- The Composed Machine is a Refinement.

The Heater Controller Development



Outline

Implementation Level Modelling

- 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.

Implementation Level Modelling

- 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.

Implementation Level Modelling

- 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.

Implementation Level Modelling

- 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 & 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

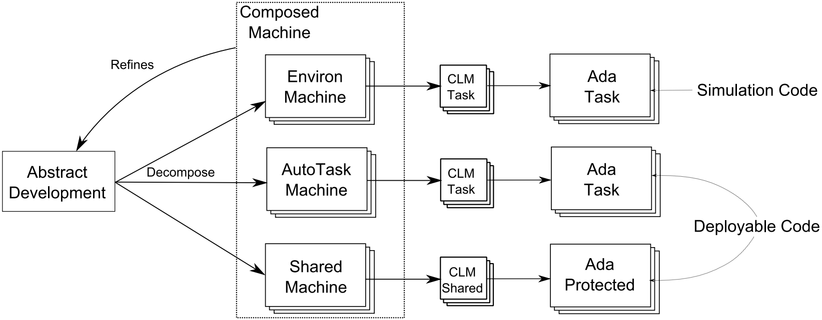
- Mapping of events
  - depends on use in task body.
  - Some event guards and actions are ‘in-lined’.
  - Some events map to ’subroutines’, and are *called*.
  - Guards
    - map to entry barriers,
    - or, looping/branching statements.
  - The code generator takes care of this.
- Synchronizations: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

The Common Language Model

The Common Language Meta-model is independent of the implementation; an abstraction based on Ada.

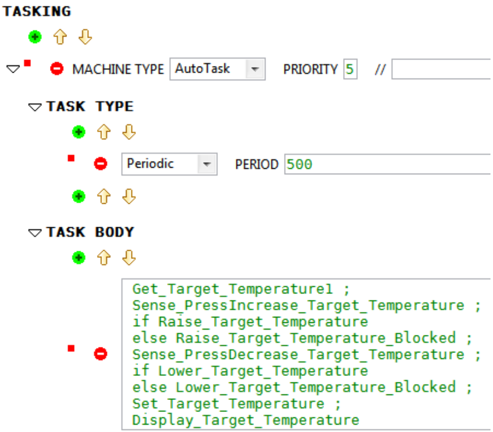


Outline

UI - Specifying a Task Body

Integrated with

- Machine Editor.



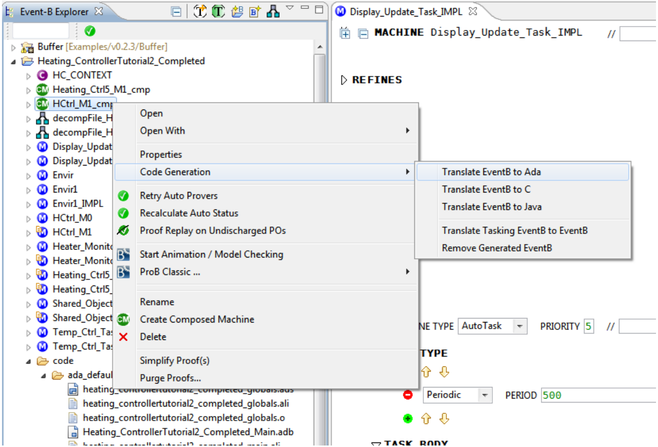
UI - Events

- 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  :  cttm1 ≔ tm
END
```

Generating Code



Outline

Using Mathematical Extensions

```
THEORY AdaRules
TRANSLATOR Ada
Metavariables • a ∈ ℤ, b ∈ ℤ, c ∈ Q, d ∈ Q
Translator Rules
     ...
     trns2:   a − b ↦ a - b
     trns9:   c = d ↦  c = d
     trns19:  a ≠ b ↦  a /= b
     trns21:   a mod b   ↦  a mod b
     trns22:   ¬$c    ↦   not($c)
     trns23:   $c ∨ $d  ↦   ($c) or ($d)
     trns24:   $c ∧ $d  ↦   ($c) and ($d)
     trns25:   $c ⇒ $d  ↦ not($c) or ($d)
Type Rules
     typeTrns1:   ℤ     ↦ Integer
     typeTrns2:   BOOL  ↦ boolean
```

Outline

Adding Arrays

THEORY Array
TYPE PARAMETERS T
OPERATORS

- **array** : $array(s : \mathbb{P}(T))$
  direct definition
  $array(s : \mathbb{P}(T)) \mathrel{\hat=} \{ n, f \cdot n \in \mathbb{N} \land f \in 0 \cdot\cdot (n-1) \rightarrow s \mid f \}$

- **arrayN** : $arrayN(n : \mathbb{Z}, s : \mathbb{P}(T))$
  well-definedness condition n ∈ ℕ ∧ finite(s)
  direct definition
  $arrayN(n : \mathbb{Z}, s : \mathbb{P}(T)) \mathrel{\hat=} \{ a \mid a \in array(s) \land card(s) = n \}$

Theory: Translation Rules for Arrays

```
•update     :  update(a : ℤ↔T, i : ℤ, x : T)
 …
•lookup  :  lookup(a : ℤ↔T, i : ℤ)
 …
•newArray   :  newArray(n : ℤ, x : T)
 …

TRANSLATOR Ada
Metavariables s ∈ ℙ(T), n ∈ ℤ, a ∈ ℤ↔T, i ∈ 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:
          Invariants cbuf ∈ arrayN(maxbuf,ℤ)
          Initialisation cbuf ≔ newArray(maxbuf,0)
```

⇩

type rule :	arrayN(n,s)	↦	array (0..n-1) of s
constructor :	newArray(n,x)	↦	(others => x)
	ℤ	↦	Integer

⇩

```
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 constucts.
- 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.

Wrapping Up

- 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.