Code Generation Update

Andy Edmunds
University of Southampton
ae2@ecs.soton.ac.uk

Code Generation with Tasking Event-B

- Tasking Event – B is
  - an extension to Event-B.
  - flow control language and annotations.

- Tasking Machines (1) map to task implementations.
- Shared Machines map to protected objects,
  - provide monitor-style protection.
- Environ Machines (2) map to tasks for simulation.

Code Generation with Tasking Event-B

- Tasking/Environ machines have 'Task Bodies'
  - to describe program flow.
  - which map to program statements.

- Program flow such as,
  - IF event1 ELSE event2 END
  - event1 ; event2

- Events 'populate' sequences, branches, loops, update statements, procedures, procedure calls.

Since the last RUDW

- Concluded industrial collaboration,
  - Improvements to translator.
  - Java interface for the Environment.
- Templates and code-injection.
- Event-B to FMI-C translation,
  - for use in co-simulation.
- Theory + Java Code for Implementable Sets and Functions.

Improvements to Translators

- Automatic flattening of invariants, and events.
- Automatic inference of typing annotations and parameter directions.
- … means fewer steps to generate code from an appropriately constructed model.

For Java integration with Event-B projects,

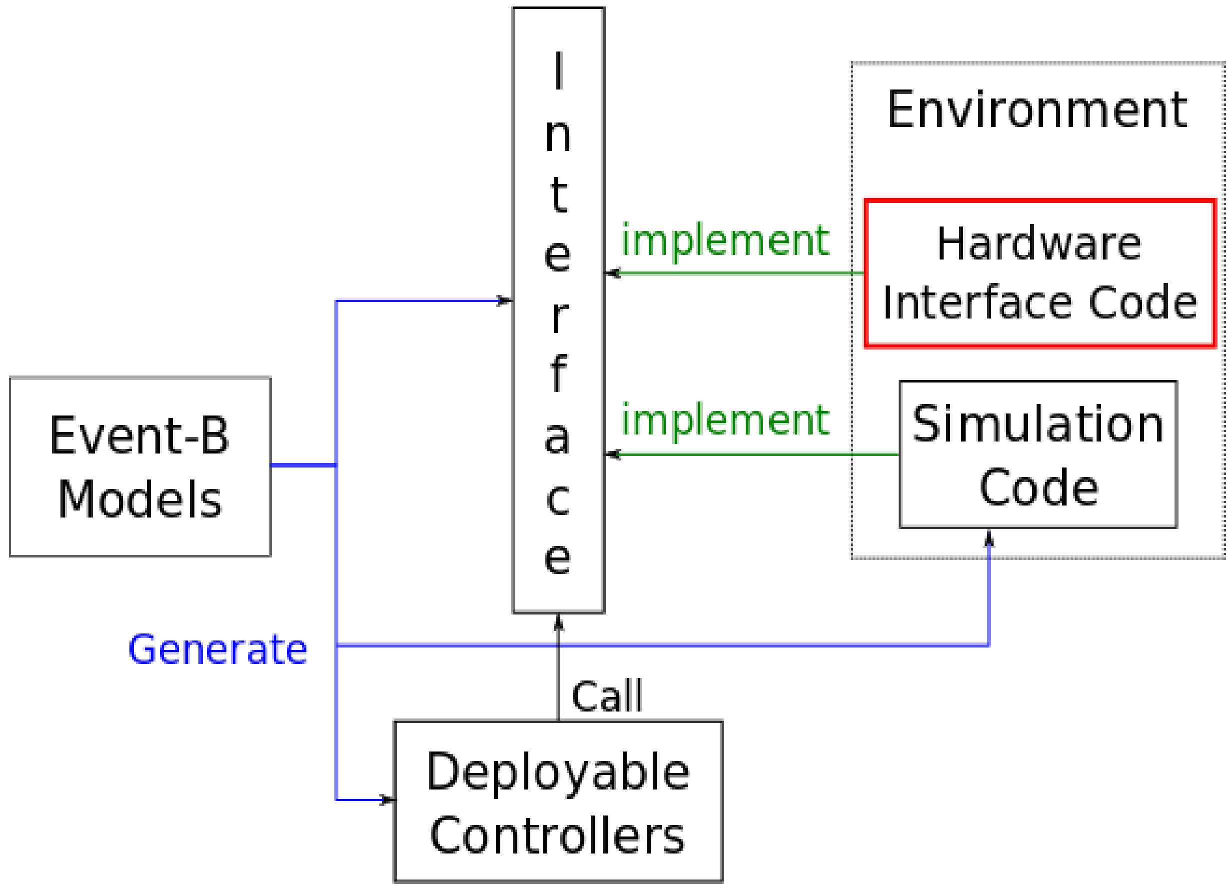
- To use Java Nature and Java Builder (JDT).

Some Items on the To-do List

But, we are still short of the goal, in terms of usability, and features.

- Validation and feedback.
- Translation of nested state-machines.
- Synchronization between events of a state-machine (Other than the current *between-cycles* approach).

New: A Java Interface for the Environment



Hardware Interface – most likely manually coded.
Simulation Code – can be auto-generated.

New: Templates and Code Injection

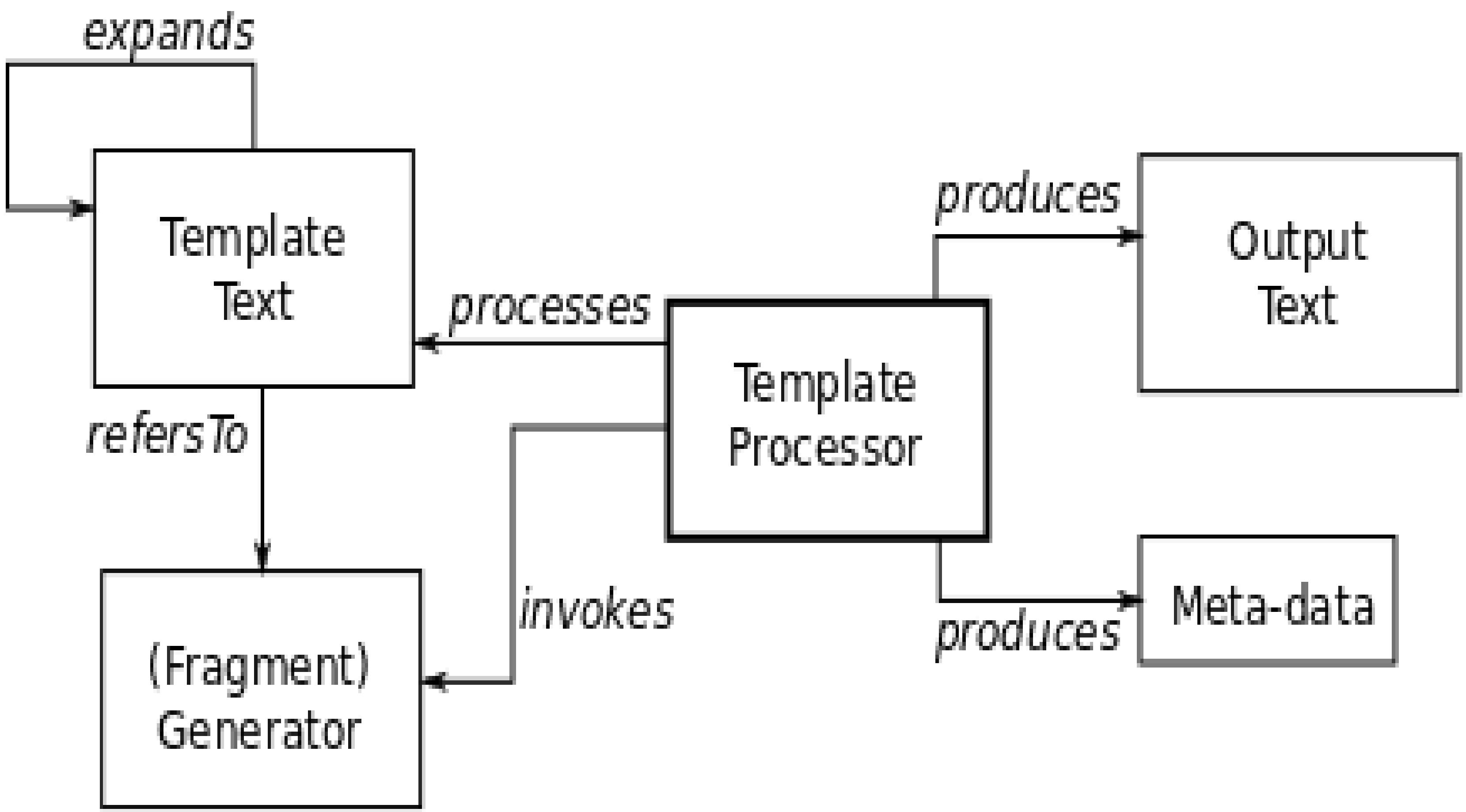
- Short paper in ABZ2014.
- Arose out of Thales' request to think about customisation for deploying on different targets.
- Boilerplate code with injection points.
- Injected code is generated from an Event-B model,
  - using a 'generator' extension point.

An Example Template

```
//## <addToHeader>
fmiStatus fmiInitializeSlave(fmiComponent c,
        fmiReal tStart, fmiBoolean StopTimeDefined,
        fmiReal tStop) {
    ModelInstance* comp = (ModelInstance*) c;
    //## <initialisationsList>
    //## <stateMachineProgramCounterIni>
    return fmiOK;
}
```

Tags are 'processed'.
Other lines are output verbatim.

Template Processing



New: Event-B to C, for Co-simulation

- For Advance EU FP7, uses FMI.
- The objective is to,
  - test the generated code in a simulation of its environment.
  - improve performance of simulation.
- Master and Slaves communicate through API.
- Slaves are FMUs.

Event-B to C, for Co-simulation

- The master is cyclic; slaves are initialized,

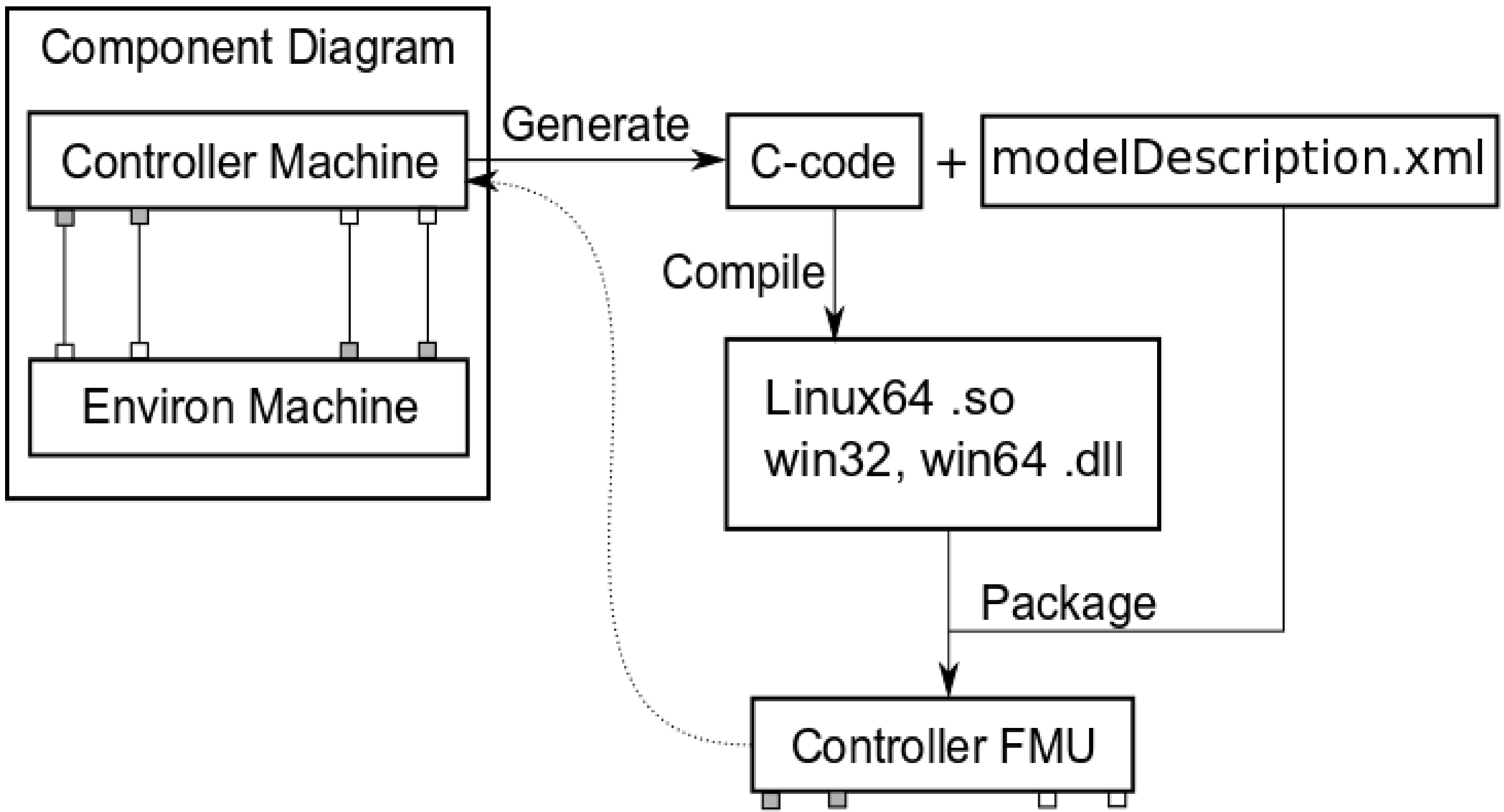
  … then master does simulate-update cycle.

- We can generate an FMU from an Event-B machine and component diagram.

- We can replace the machine, with the FMU, in the component diagram, and simulate/test with that.

- We can also import FMUs into other simulators.

FMUs from Machines



Current Status

- Event-B to C code generator – Working.

- FMU packager – Working.

- Examples pass FMU checkers/simulators
  - Win32
  - Linux64

- Component simulation – not yet working.

- Dymola Simulator import – not yet working.

NEW: Implementable Sets and Functions

- Translate Sets and Functions to code.
  - Uses the Theory plug-in.
- Depends on target language API,
  - Java HashSet and HashMap.
  - Function - domain elements map to keys,
  - Range elements map to values.
- Still *experimental*.
- Used in PRIME project.

Implementable Sets

```
THEORY
   SetImpl
TYPE PARAMETERS
   T
OPERATORS

   •setImpl  :  setImpl(t : T)   EXPRESSION  PREFIX

   direct definition
    setImpl(t : T) ≙ ℙ(T)

   •newSet  :  newSet(t : ℙ(T))   EXPRESSION  PREFIX

   direct definition
    newSet(t : ℙ(T)) ≙ ∅⦂ℙ(T)

   •newEnum  :  newEnum(t : T)   EXPRESSION  PREFIX

   direct definition
    newEnum(t : T) ≙ ℙ(T)
   •singleton  :  singleton(a : T)   EXPRESSION  PREFIX
   direct definition
    singleton(a : T) ≙ {a}
   •setUnion  :  setUnion(a : ℙ(T), b : ℙ(T))
   direct definition
    setUnion(a : ℙ(T), b : ℙ(T)) ≙ a ∪ b
```

Translation Rules

TRANSLATOR

Java

Metavariables

- a ∈ ℙ(T)
- b ∈ ℙ(T)
- t ∈ ℙ(T)
- s ∈ T

Translator Rules

```
IntegerType     :   Z  ⟾ Integer
unionRule     :   setUnion(a,b) ⟾ a.union(b)
intersectRule     :   setIntersection(a,b) ⟾ a.intersect(b)
subtractRule     :   setSubtract(a,b) ⟾ a.subtract(b)
newSetRule     :   newSet(ø⦂ℙ(t)) ⟾ new SetImpl<t>()
setReduceRule     :   setReduce(a) ⟾ a.getFirst()
singletonRule     :   singleton(s) ∪ a ⟾ a.setUnion(s)
newInstanceRule1     :   newInst(T) ⟾ new T()
newInstanceRule2     :   newInst2(t,s) ⟾ new t(s)
```

Type Rules

```
typeTrns2     :   Z  ⟾ Integer
typeTrns1     :   setImpl(T) ⟾ SetImpl<T>
typeTrns3     :   newType(T) ⟾ T
```

Java Set Implementation

```java
package setImpls_java;

import java.util.HashSet;

public class SetImpl<E> extends HashSet<E> {

    /**
    private static final long serialVersionUID = 26389

    public SetImpl<E> union(SetImpl<E> otherSet) {
        addAll(otherSet);
        return this;
    }

    public SetImpl<E> intersect(SetImpl<E> otherSet) {
        retainAll(otherSet);
        return this;
    }

    public SetImpl<E> subtract(SetImpl<E> otherSet) {
        removeAll(otherSet);
        return this;
    }

    public E getFirst() {
        Iterator<E> iter = iterator();
        if(iter.hasNext()) return iter.next();
        else return null;
    }

    public SetImpl<E> setUnion(E element) {
        add(element);
        return this;
    }

}
```

Questions:

How to improve plug-in development when much of it is *engineering*, not research?

- Academia v Industry: bridging the gap?
- Providing a platform to 'sell' Event-B?

Day-to-day,

- Keeping up with (communicating) changes?
- Compatibility issues?
- ...