

# An Experience Report on Event-B Code Generation, from an Industry-led Assessment of the Event-B Approach

A. Edmunds<sup>1</sup>, C. Snook<sup>1</sup>, K. Wiederaenders<sup>2</sup> and K. Reichl<sup>2</sup>

<sup>1</sup> University of Southampton, UK

<sup>2</sup> Thales Transportation Systems, Germany

**Abstract.** This report describes the experience of our participation in an industry-led assessment of Rodin/Event-B, and Tasking Event-B for code generation. The feasibility study involved the modelling of controller software, and the operating environment, of a simple fan controller. The aim of the study was to evaluate the complete Event-B methodology, from abstract specification to Java implementation. During the project various tool enhancements were completed, and a list of feature requests was created by the industrial partner. The list may serve to influence the direction of research, or the commercialisation of the tool. In the report we also draw some general conclusions, including how the collaboration brought to light some interesting differences between the goals of academia and industry.

## 1 Overview

Rodin is a tool platform for the rigorous specification of critical systems, using the Event-B approach [2]. The tool was developed in the RODIN project [10], and experience with industry was gained in the DEPLOY project [15]. However, code generation (specifically our version, which we call Tasking Event-B [7]) was not assessed by industry in any depth. The collaboration reported in this paper is the first such undertaking, that we are aware of, that seeks to evaluate the seamless development from abstract specification to Java implementation, using Tasking Event-B in an industrial setting. The tool enhancements, reported here, have been partly funded by the ADVANCE project [14], and in part by the industrial partner.

In the remainder of this section we present the industrial collaborator's motivation for the study, and give an overview of Event-B, Tasking Event-B and code generation. Where necessary, we illustrate the text with example fragments from the case-study. In Sect.2 we discuss translation of Tasking Event-B to code. In Sect. 3 we present more details of the case study. In Sect. 4 we provide a commentary on our experience; and in Sect. 5 we summarize the lessons learned.

### 1.1 Motivation for the Case-Study

At the beginning of the collaboration, a brief (2 week) period of Event-B training was provided, with additional consolidation time. The engineers involved were

very experienced railway domain engineers, so they were very interested to learn how they could apply the techniques in their domain. This led to the frequent use of practical demonstrations, followed by question and answer sessions, as a way to introduce a new topic. A case study was subsequently proposed, so that the industrial partner could assess the feasibility of using Event-B and Tasking Event-B in a larger project; by undertaking a small, but complete, development (from abstract specification to Java implementation). The industrial partner had an existing implementation, for comparison. They also wished to assess some of the other plug-ins, such as the ProB animator/model checker [8], and a related plug-in for model-based testing [4]. We will not report these results here.

The criteria for the case study was that it should be small enough to be completed in 3 months (whilst still being near the start of the Event-B learning curve). It was to cover as many aspects of Event-B development as possible. This included the use of Rodin platform, with proof, and refinement. Use of decomposition and composition [11]; iUML-B state-machines [13]; the theory plug-in for language specification [9]. We also needed to develop a Java interface to the environment; and use the Eclipse Java Development Tool-kit, so that the generated code could be run in the Eclipse environment [16] without further editing.

## 1.2 Event-B

The Event-B method [2] was developed by J.R. Abrial, and uses set-theory, predicate logic and mathematical constructs to model discrete systems. Event-B *machines* are used to describe dynamic properties of a system, and *contexts* the fixed properties. Properties (such as safety-properties) are described in a machine's *invariants* and a context's *axioms*. Machines are able to use contexts by adding them to the *sees* clause; then content of a Context is visible and accessible to a machine. Machines describe the dynamic aspects of a system using state variables, and guarded atomic *events*. Events can have non-deterministic parameters, which can be interpreted as a kind of local variable, or (as we do in code generation for Tasking Event-B) actual and formal parameters for methods and their calls. Proof obligations are generated by the tool automatically. They represent the conditions that should be satisfied, to demonstrate that the model is consistent with the specified properties. Discharging proof obligations (proving their truth) shows that the related properties hold. In many cases proof obligations are discharged by Rodin's automatic proof tools, but it is often necessary to perform interactive proof within Rodin. Interactive proof is undertaken by suggesting strategies, and sub-goals in the form of hypotheses.

A fragment of an Event-B specification is shown in Fig. 1, with more shown in Appendix A. Variables are introduced in the variables clause, and typed in the **invariant**. Invariants also describe desired safety properties. An initialisation event occurs before all others, Fig. 2. The initial state can be specified using deterministic or non-deterministic assignments,  $:=$  or  $:\in$  resp. The *read* event refines some abstract *read* event, and declares a parameter *p1*, which is used in the **where** clause. Other event guards may also appear here. After initialisation,

any enabled event with all its guards true may occur. State updates are specified in the **then** (action) clause using deterministic or non-deterministic assignments, or the event may do nothing (*skip*). Event-B also incorporates diagrammatic means of specification. One such approach is that of state-machine diagrams [13]. Fig. 3 shows a state-machine diagram, which is a preview of the one used in our case-study.

```

MACHINE FanCtrlMachine
SEES FanCtrlMachine_implicitContext, FanCtrlContext
VARIABLES C_tempPSU, C_tempAmbient, ...
INVARIANTS
  C_tempPSU  $\in \mathbb{Z} \wedge$  C_tempAmbient  $\in \mathbb{Z} \wedge$  C_fanSpeed  $\in \mathbb{Z} \dots \wedge$ 
  (Controller = Normal  $\Rightarrow$ 
    ((C_tempPSU  $\geq$  HWM_PSU)  $\vee$  (C_tempAmbient  $\geq$  HWM_AMBIENT)
       $\Rightarrow$  C_fanOn = TRUE))  $\wedge$ 
    (Controller = Normal  $\Rightarrow \dots$ 

```

**Fig. 1.** Example of Event-B with Invariants

Refinement is the process of adding detail to a development. A refinement machine can introduce new variables, invariants, and events. New and existing events can modify new variables, but there are restrictions on how existing variables are modified. Consistency in the relationship between an abstract machine and its refinements is maintained by discharging the automatically generated proof obligations. An example of a refined machine can be seen in Appendix B.

Shared Event-B decomposition [3,12] is a technique that we use to handle complexity; we are able to split a single machine specification into several. We begin with partitioning variables into machines. After decomposition the events that refer to them are shared between machines. The shared events are said to synchronize: i.e. only enabled when both events are enabled. Fig. 4 shows a diagram, where event  $e$  is decomposed, into  $e_a$  and  $e_b$ . The events can be recomposed; in [7] we describe the synchronization of two events as being equivalent to a single, merged, atomic event. Each event consists of guards and actions, and  $e$  can be represented as a guarded command:  $g \rightarrow a$ . The composition, using this notation, is  $g \rightarrow a \triangleq g_a \wedge g_b \rightarrow a_a \parallel a_b$ . In this case  $g_a$  is the guard of  $e_a$  and so on; in this situation we use the parallel action operator  $\parallel$  where the operands are actions. Event Synchronization uses shared parameters to facilitate communication between machines. A composed machine construct is generated [11], that keeps track of the artefacts after decomposition. As we move towards implementation, the decomposed artefacts reflect entities in the implementation; which assists with code generation.

### 1.3 Tasking Event-B

Tasking Event-B incorporates an implementation level specification language, which, when added to a machine, describes how to translate an Event-B ma-

INITIALISATION:	write:
THEN	REFINES write
C_tempPSU : $\in \mathbb{Z}$	ANY p1
C_tempAmbient : $\in \mathbb{Z}$	WHERE
...	p1 = C_fanOn
END	THEN
	E_MAX6650_fanOn := p1
	END

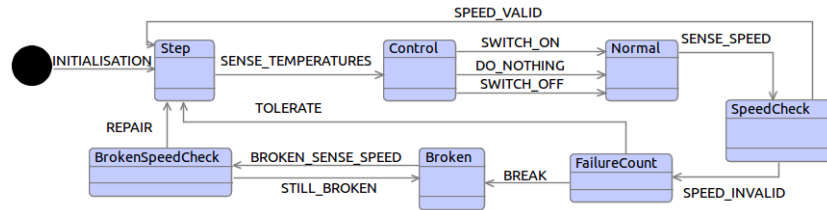
**Fig. 2.** Example Event-B Events

chine into an implementation. Annotations are added to machines and contexts. These provide additional, implementation-level details, to guide the code generator. In this way machines can be implemented as task/thread-like constructs; shared, monitor-like constructs; or provide simulations of the environment. These annotations can be translated back into to Event-B, to provide a model of the implementation, and are also translated to their related constructs in an implementation. The machine *Type* annotations are *AutoTask*, *Shared* and *Environ* respectively. In embedded systems, AutoTask Machines typically model *controller* tasks (of the implementation).

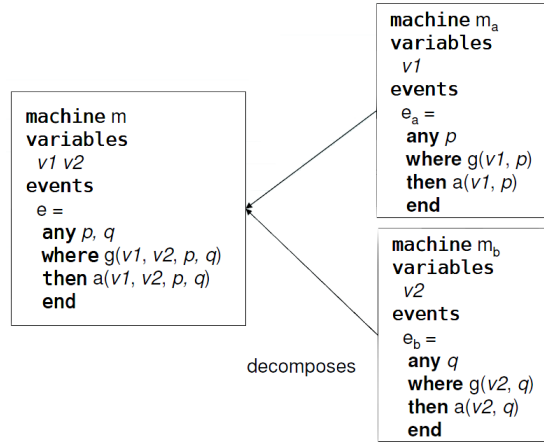
The AutoTask, shown in Fig. 5, allows us to specify a task type; which can be either periodic, single-shot, triggered, or repeating. In the figure we see a value, in milliseconds, for the period. In a *task-body* we can specify an ordering of events. We can use sequential, branching and looping operators. In the example, we state that the *write* event takes place, followed by the *read* event. Both synchronize with their counterpart events in the environment. The *output* clauses provides feedback via the console during a simulation. In an implementation, if there is a state-machine associated with an AutoTask machine, it is evaluated before the task-body.

## 2 Translation to Code

Event-B and Tasking Event-B provide the information used to generate code. There are code generators for translations to Java, Ada and OpenMP C, which make use of the theory plug-in [9]. The theory plug-in allows us to specify a



**Fig. 3.** Fan Controller Event-B State-machine Diagram



**Fig. 4.** Shared Event Decomposition

```

TASKING MACHINE TYPE AutoTask
TASK TYPE Periodic PERIOD 100
TASK BODY
  write ;
  read ;
  output C_fanSpeed: C_fanSpeed ;
  output C_fanOn: C_fanOn
END
  
```

**Fig. 5.** A Task Body

mapping between Event-B mathematical operators, and their implementation counterparts, for each target language. We then use a pattern matching feature, where the translation is performed. AutoTask, shared, and environ machine translations, introduced in the last section, are hard-coded. In this report we focus on Java translations, at the request of our industrial collaborators.

There is a formal definition of the relationship between Event-B with Tasking Event-B, and implementation constructs; details are given in [7]. But, an intuitive explanation is to describe the implementation as a refinement of the formal model. For the task body clause, in Tasking Event-B, sequences map to sequences of program statements. The branching statement *IF*  $e_a$  *ELSE*  $e_b$ , with events  $e_a$  and  $e_b$  map to a conditional statement where the event guards become branch conditions, and event assignment actions become assignment statements. The code fragment of Fig. 6 shows the implementation of the fan-controller task, the details are shown in context in Appendix D. The task body is implemented in the *run* method of a Java thread. The method has code for forcing the thread to sleep, if the period has not been reached. The scheduling of threads in this way was considered sufficient for the purposes of this feasibility study. The state-machine code is in the *ControllerstateMachine* method. After this is invoked, the

```

public void run(){
    while(true)
    { ...
        ((ControllerNewImpl)MainEntry.
            getTask(''ControllerNewImpl ''')).ControllerstateMachine();
        ((EnvNewImpl)MainEntry.getTask(''EnvNewImpl ''')).write(C_fanOn);
        if ((Controller != Normal))
        {
            Pointer<Integer> C_fanSpeedPointer = new Pointer<Integer>();
            Pointer<Integer> C_tempAmbientPointer = new Pointer<Integer>();
            Pointer<Integer> C_tempPSUPointer = new Pointer<Integer>();
            ((EnvNewImpl)MainEntry.getTask( ''EnvNewImpl '''))
                .read(C_fanSpeedPointer, C_tempAmbientPointer,
                    C_tempPSUPointer);
            C_fanSpeed = C_fanSpeedPointer.value;
            C_tempAmbient = C_tempAmbientPointer.value;
            C_tempPSU = C_tempPSUPointer.value;
        }
        System.out.println(''C_fanSpeed: '' + C_fanSpeed);
        System.out.println(''C_fanOn: '' + C_fanOn);
        ...
    }
}

```

**Fig. 6.** Code Generated from the Fan Controller Task-body

result (the command to turn the fan on/off) is written to the environment using the *write* method call. This is followed by a *read* method call, where the latest environment values are read. The environment implementation (not presented here) simulates a temperature rise when the fan is off, and is lowered when the fan is on. A fragment of the state-machine implementation is shown in Fig. 7 to give a flavour of the generated code.

### 3 Details of the Case Study

The case study is based on a fan-controller, used in a railway, track-side cabinet cooling system. Its function is to monitor the temperature from a number of sensors, and turn the fan on (above a high water mark temperature) or off (below a low water-mark temperature). There are two temperature sensors, one for the ambient temperature in the cabinet, and one for the PSU temperature, each with different water marks. The abstract specification describes a number of properties derived from a requirements document. An example property is shown below, which describes the condition required for the controller to signal that the fan should be turned on. It does this by setting the *C\_fanOn* variable to *TRUE*. Other variables such as *Controller* keep track of the state-machine state; *C\_tempPSU* the PSU temperature, and *C\_tempAmbient* the ambient temperature are the controller's copies of the values sensed in the environment. Sensing from the environment is done using an event called *read*, and both temperature values are read in one event, in our model. The controller can be in one of

```

public void ControllerstateMachine(){
    switch(Controller){
        case Step:
            Controller = Control;
            break;
        case Control:
            if (((C_tempPSU >= HWM_PSU) || (C_tempAmbient >= HWM_AMBIENT)))
                && (!((C_tempPSU <= LWM_PSU) && (C_tempAmbient <= LWM_AMBIENT)))
            {
                Controller = Normal;
            }
            else if (((C_tempPSU >= HWM_PSU) || (C_tempAmbient >= HWM_AMBIENT)))
            {
                Controller = Normal;
                C_fanOn = true;
            }
            else if (((C_tempPSU <= LWM_PSU) && (C_tempAmbient <= LWM_AMBIENT)))
            {
                Controller = Normal;
                C_fanOn = false;
            }
            break;
        ...
    } ...
}

```

**Fig. 7.** Controller State-machine Implementation

number of states, as described by the state-machine shown in fig. 3.

$$\begin{aligned}
 Controller = Normal \Rightarrow ((C\_tempPSU \geq HWM\_PSU) \vee \\
 (C\_tempAmbient \geq HWM\_AMBIENT) \Rightarrow C\_fanOn = TRUE)
 \end{aligned}$$

The property above specifies that, when in the normal state, if the PSU temperature is greater than the PSU high water mark (HWM\_PSU), or ambient temperature is greater than the ambient high water mark (HWM\_ambient) then the controller can signal to turn on the fan ( $C\_fanOn = TRUE$ ).

We now turn our attention to refinement, composition and decomposition. As we refine our way towards an implementation, any large scale development will need to be broken down into manageable sections. Composition and decomposition [11] plug-ins allow us to do this. In our case-study we decompose the system into controller and environment. A description of the methodological steps follows,

1. Abstract specification; with non-deterministically assigned input from the environment. A state-machine diagram describes controller behaviour.
2. Model changed, so that the input values are deterministically read, directly from the environment.

3. Add parameters to events; this models value transfer across the interface with environment, and facilitates decomposition.
4. Decompose the model into environment and controller.
5. Refine generated models, and add Tasking Event-B annotations.
6. Generate Java controller with state-machine implementation, and environment simulation code.

During the project we developed the tools and approach, to a point where code was generated, and run. The project progressed sufficiently for the feasibility study to be completed. We will discuss some of the issues that arose during project in the sections that follow.

## 4 Experiences

The discussion covers some general, and some code generation specific difficulties, that came to light throughout the duration of the project. The difficulties were usually overcome by fixing bugs, or making compromises, usually in the form of workarounds. A detailed record was kept of issues to be addressed. A sizeable compromise was made when scoping the model. In order to have a tractable problem to solve within the available time, we carefully selected a small number of requirements from a pre-prepared requirements document. The industrial partner's main goal was to assess as many methodological and tooling aspects as possible.

### 4.1 General Issues

During the project set-up we uncovered plug-in compatibility problems, where separately developed plug-ins do not work with each other. Reasons for this include a rapidly evolving code-base, with updated or removed features; and conflicting dependencies. The problem was addressed by installing only the required plug-ins, and then working to resolve any problems with these. Many bugs were fixed, usability issues addressed, and small features added during this time. The Rodin core team were able to fix some bugs (unrelated to code generation, but revealed during the project activities) and even the team responsible for the feasibility study contributed code to Rodin core development.

An issue that quickly became clear in the early stages of the project, was that the industrial partner was not aware of the 'nature' of the tool deliverables arising from our academic plug-in projects. Note that the Rodin platform core is developed and maintained by a commercial organisation, so we are not referring to that here. In academia, tools are often developed to validate new approaches and facilitate further research. This does not necessarily satisfy the needs of industry in terms of having all the required features, and level of maintenance and support. An example of this was that the code generator for Java had not been maintained to work with new plug-in releases. Another shortcoming was that of decomposition involving state-machines. The decomposition process



does not reproduce any state-machines in the decomposed machines. We had to provide a manual work-around, by copying the abstract version of the state-machine to the decomposed machine; and then correcting internal references (in the underlying xml file) using a text editor. This could not be done easily by the industrial partners as it requires a working knowledge of the underlying Rodin tool. In the context of this feasibility study, this kind of work-around was acceptable, although the industrial users were unable to do this themselves, in the time available. A request to add this functionality has been added to the list of feature requests.

During the learning phase, the industrial partners had difficulty in understanding the relationship between shared event decomposition, and their implementations. This was due to the fact that it is not immediately obvious how decomposed events synchronize in a task body, and how this relates to a procedure call, and its definition. The solution was to work through the example a number of times to gain familiarity with the methodology. This was typical of the success of the interactive sessions, where we would present some theory, and demonstrate it in practice, with a question and answer session. The next step was for the industrial users to gain a full understanding by using the feature under close supervision.

## 4.2 Code Generation Issues

We performed an in-depth assessment of the code generation approach; this uncovered some bugs and usability issues. As mentioned above, the Java code generator had not been maintained, and did not work due to changes in plugins on which it depended. It is also the case that the Java code generator was immature, since in previous work we had focussed our efforts on generating Ada code. These are ‘back-end’ code generator issues, we did work to fix many of the problems. There is also a back-end code generator, which translates to an intermediate model, and for this we had to make a compromise. For instance, we do not translate from nested state-machines, therefore nested state-machines had to be flattened into a single state-machine. Translation of nested state-machines has been added to the list of feature requests.

We also required the use of a very specific pattern of state-machine execution, where inputs are received from the environment, the state-machine evaluated, and results output. A feature request has been added to permit I/O during the state-machine execution cycle. We were also able to address some usability issues during the project; much of the annotation of Tasking Event-B was automated, in a flattening process, where all the otherwise-hidden invariants, and event details are made available at the lowest level, and annotated. Annotations are also derived from the models for typing and parameter direction annotations (incoming/outgoing parameters). Recall that the annotations guide the code generator.

At the end of the project we were also able to undertake some research, to develop some ideas about generating interfaces (to the environment), and implementation of non-deterministic assignment.

### 4.3 The Industrial Assessment

As we mentioned in the introduction, the case study was undertaken to assess the feasibility of using Event-B and Tasking Event-B in a larger project. The feature requests, described below, indicate that there is further work to be done before the whole tool-chain is ready for use. There is ongoing interest from the industrial partner in the use of Event-B, and the assessment gave rise to further investigations; the results of which have not been disclosed to us.

**Feature Requests** The following code generation features were requested and constitute, potential, future work. This list is not exhaustive, and merely illustrates that there is much to do before our code generation tools are in the context provided by the feasibility study.

1. Templates for configuring different implementation targets.
2. Generation of Java Interfaces
3. Introduce an Eclipse model for composed machine refinement.
4. Code generation for nested state-machines.
5. Hide Tasking Event-B in non-tasking machines.
6. Provide static checking of tasking machines
7. Introduce state-machine decomposition and in-cycle I/O.
8. Reflect Event-B composition structure in the generated code.
9. Add Java-JML Generator, to assist with manual coding effort.

Since completion of the collaboration, we have been able to make progress on items 1, 2 and 3, with a paper arising from the work on templates [6]. As it stands today, the remainder are not seen as critical, in research terms, since it just involves coding. The issues will be addressed on an as-needed basis, limited by availability of resources. Further work on item 6 is perhaps most important for users of the tool; good feedback of errors in the model is vital for the sake of usability. From an industrial user's perspective, assessing the criticality of the features depends on what the end-user requires of the tool. For instance, item 4 may be important to those who use state-machines, but not every industrial user will use them.

## 5 What we learned from the collaboration

We found that, during the early learning-phase, interactive demonstrations with questions/answer sessions were more effective than slide-based presentations. We found that close collaboration (face-to-face) was essential for speedy transfer of ideas, between all parties; a point highlighted in a recent UK government report [1] (Page 38 - note 120). The work was undertaken as a consultancy service by ECS Partners [5]. As academic tool-developers, we gained an understanding of the specific needs of our partner. It was apparent that the goals of academia and industry are somewhat different, in academia we tend to develop tools to validate new approaches, and facilitate further research. Industry relies

on efficient, stable, well maintained software. In that respect, finding mutually beneficial goals is key to making progress. We were able to drive forward development of our tool, and the interaction has provided the basis for new research. The industrial partner was able to begin evaluating the readiness for using the tool in a production situation.

To establish a sound basis for our development, it was necessary to find a minimal, robust set of plug-ins, with the desired functionality. To satisfy a commercial organisations' expectations, more co-ordination may be required to ensure compatibility between plug-ins. For most problems that we encountered, bug fixes were quickly provided by Rodin, and plug-in, developers. Occasionally we had to compromise where quick fixes were not possible.

A useful artefact was produced, i.e. the list of features requests from the industrial partner. The feature requests is a public (on sourceforge) record of where the shortcomings are in the tool. It can be viewed as a mix of research opportunities, and also requirements for the tool's commercialisation.

In conclusion, we gained valuable insight into the area where research shifts towards commercialisation of a tool. We saw the need for a robust, stable, extensible tool-set; where usability and productivity is of prime importance. How we improve the transfer of technology at the boundary between research and commercialisation remains an open question; but we feel we have a greater understanding of what can be done in the Event-B field.

## References

1. House of Commons Science and Technology Committee Report - Bridging the valley of death: improving the commercialisation of research, March 2013.
2. J. R. Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 2010.
3. M. Butler. Decomposition Structures for Event-B. In *Integrated Formal Methods iFM2009, Springer, LNCS 5423*, volume LNCS. Springer, February 2009.
4. I. Dinca, F. Ipate, L. Mierla, and A. Stefanescu. Learn and Test for Event-B - A Rodin Plugin. In John Derrick, John A. Fitzgerald, Stefania Gnesi, Sarfraz Khurshid, Michael Leuschel, Steve Reeves, and Elvinia Riccobene, editors, *ABZ*, volume 7316 of *Lecture Notes in Computer Science*, pages 361–364. Springer, 2012.
5. ECS Partners. ECS Consulting Services. Available at <http://www.ecspartners.ecs.soton.ac.uk/>.
6. A. Edmunds. Templates for Event-B Code Generation. In *4th International ABZ 2014 Conference*, 2014.
7. A. Edmunds and M. Butler. Tasking Event-B: An Extension to Event-B for Generating Concurrent Code. In *PLACES 2011*, February 2011.
8. M. Leuschel and M. Butler. ProB: A Model Checker for B. In *Proceedings of Formal Methods Europe 2003*, 2003.
9. I. Maamria, M. Butler, A. Edmunds, and A. Rezazadeh. On an Extensible Rule-based Prover for Event-B. In *ABZ2010*, February 2010.
10. RODIN Project. at <http://rodin.cs.ncl.ac.uk>.
11. R. Silva and M. Butler. Shared Event Composition/Decomposition in Event-B. In *FMCO Formal Methods for Components and Objects*, November 2010. Event Dates: 29 November - 1 December 2010.

12. R. Silva, C. Pascal, T.S. Hoang, and M. Butler. Decomposition Tool for Event-B. *Software: Practice and Experience*, 2010.
13. C. Snook. Event-B Statemachines. at [http://wiki.event-b.org/index.php/Event-B\\_Statemachines](http://wiki.event-b.org/index.php/Event-B_Statemachines), 2011.
14. The Advance Project Team. Advanced Design and Verification Environment for Cyber-physical System Engineering. Available at <http://www.advance-ict.eu>.
15. The DEPLOY Project Team. Project Website. at <http://www.deploy-project.eu/>.
16. The Eclipse Foundation. Eclipse - an Open Development Platform. Available at <http://www.eclipse.org/>.

## A Abstract Specification (abridged)

**MACHINE** FanCtrlMachine

**SEES** FanCtrlMachine.implicitContext, FanCtrlContext

**VARIABLES** Controller, C\_tempPSU, C\_tempAmbient, ...

**INVARIANTS**

$C\_tempPSU \in \mathbb{Z} \wedge C\_tempAmbient \in \mathbb{Z} \wedge C\_fanSpeed \in \mathbb{Z} \dots \wedge$   
 $(Controller = Normal \Rightarrow$   
 $((C\_tempPSU \geq HWM\_PSU) \vee (C\_tempAmbient \geq HWM\_AMBIENT)$   
 $\Rightarrow C\_fanOn = TRUE)) \wedge$   
 $(Controller = Normal \Rightarrow$   
 $((C\_tempPSU \leq LWM\_PSU \wedge C\_tempAmbient \leq LWM\_AMBIENT) \Rightarrow$   
 $C\_fanOn = FALSE))$

**EVENTS**

INITIALISATION:

THEN

init\_tempPSU:  $C\_tempPSU : \in \mathbb{Z}$

init\_tempAmbient:  $C\_tempAmbient : \in \mathbb{Z}$

...

SENSE\_TEMPERATURES:

WHERE

Controller = Step

THEN

Controller := Control

END

DO\_NOTHING:

WHERE

$\neg((C\_tempPSU \geq HWM\_PSU) \vee (C\_tempAmbient \geq HWM\_AMBIENT))$

Controller = Control

$\neg((C\_tempPSU \leq LWM\_PSU) \wedge (C\_tempAmbient \leq LWM\_AMBIENT))$

THEN

Controller := Normal

END

...

read:

WHERE

Controller  $\neq$  Normal

THEN

C\_fanSpeed :  $\in \mathbb{Z}$

C\_tempAmbient :  $\in \mathbb{Z}$

C\_tempPSU :  $\in \mathbb{Z}$

END

**END**

## B Refinement For Decomposition (abridged)

```
MACHINE FanCtrlMachineGC2
REFINES FanCtrlMachineGC
...
VARIABLES E_LM75_tempPSU, E_LM75_tempAmbient, E_MAX6650_fanOn ...
EVENTS
  SENSE_TEMPERATURES:
  REFINES SENSE_TEMPERATURES
  WHERE
    Controller = Step
  THEN
    Controller := Control
  END

  read:
  REFINES read
  ANY p1, p2, p3
  WHERE
    p1 = E_MAX6650_fanSpeed
    p2 = E_LM75_tempAmbient
    p3 = E_LM75_tempPSU
    Controller ≠ Normal
  THEN
    C_fanSpeed := p1
    C_tempAmbient := p2
    C_tempPSU := p3
  END

  write:
  REFINES write
  ANY p1
  WHERE
    p1 = C_fanOn
  THEN
    E_MAX6650_fanOn := p1
  END
...
END
```

## C Tasking Event-BMachine (abridged)

```
MACHINE ControllerNewImpl
REFINES ControllerNew
VARIABLES C_tempPSU, C_tempAmbient, C_fanOn, ...
INVARIANTS ...

EVENTS
  SENSE_TEMPERATURES:
  REFINES SENSE_TEMPERATURES
  WHERE
    Controller = Step not theorem TYPING NonTyping
  THEN
    Controller := Control
  END

  read:
  REFINES read
  ANY
    p1 DIRECTION In
    p2 DIRECTION In
    p3 DIRECTION In
  WHERE
```

```

p3 ∈ ℤ          TYPING Typing
p2 ∈ ℤ          TYPING Typing
p1 ∈ ℤ          TYPING Typing
Controller ≠ Normal TYPING Typing
THEN
  C.fanSpeed := p1
  C.tempAmbient := p2
  C.tempPSU := p3
END

write:
REFINES write
ANY
  p1 DIRECTION Out
WHERE
  p1 ∈ BOOL      TYPING NonTyping
  p1 = C.fanOn    TYPING NonTyping
END

```

```

TASKING MACHINE TYPE AutoTask
TASK TYPE Periodic PERIOD 100
TASK BODY
  write ;
  read ;
  output C.fanSpeed: C.fanSpeed ;
  output C.fanOn: C.fanOn
END

```

## D Controller - Fragment of Java Implementation

```

public class ControllerNewImpl implements Runnable {
    protected Controller.STATES Controller = Step;
    protected int C.tempPSU = 0;
    protected int C.tempAmbient = 0;
    protected int C.fanSpeed = 3000;
    protected boolean C.fanOn = true;
    protected int failureCount = 0;
    protected int priority = 5;

    public void run(){
        while(true)
        {
            long THREAD_START_TIME = System.currentTimeMillis();
            ((ControllerNewImpl)MainEntry.
                getTask("ControllerNewImpl ")).ControllerstateMachine();
            ((EnvNewImpl)MainEntry.getTask("EnvNewImpl ")).write(C.fanOn);
            if ((Controller != Normal))
            {
                Pointer<Integer> C.fanSpeedPointer = new Pointer<Integer>();
                Pointer<Integer> C.tempAmbientPointer = new Pointer<Integer>();
                Pointer<Integer> C.tempPSUPointer = new Pointer<Integer>();
                ((EnvNewImpl)MainEntry.getTask("EnvNewImpl "))
                    .read(C.fanSpeedPointer, C.tempAmbientPointer, C.tempPSUPointer);
                C.fanSpeed = C.fanSpeedPointer.value;
                C.tempAmbient = C.tempAmbientPointer.value;
                C.tempPSU = C.tempPSUPointer.value;
            }
            System.out.println("C.fanSpeed: " + C.fanSpeed);
            System.out.println("C.fanOn: " + C.fanOn);
            long THREAD_END_TIME = System.currentTimeMillis();
            long THREAD_TIME_TAKEN = THREAD_END_TIME - THREAD_START_TIME;
            try{ Thread.sleep(Math.max(100 -- THREAD_TIME_TAKEN,0));
                ...
            }
        }
    }
}

```

```
}
```

```
public void ControllerstateMachine(){  
    switch(Controller){  
        case Step:  
            Controller = Control;  
            break;  
        case Control:  
            if (((C.tempPSU >= HWM.PSU) || (C.tempAmbient >= HWM.AMBIENT)))  
                && (!((C.tempPSU <= LWM.PSU) && (C.tempAmbient <= LWM.AMBIENT))))  
            {  
                Controller = Normal;  
            }  
            else if (((C.tempPSU >= HWM.PSU) || (C.tempAmbient >= HWM.AMBIENT)))  
            {  
                Controller = Normal;  
                C_fanOn = true;  
            }  
            else if (((C.tempPSU <= LWM.PSU) && (C.tempAmbient <= LWM.AMBIENT)))  
            {  
                Controller = Normal;  
                C_fanOn = false; }  
            break;...
```