From Event-B Models to Code

Andrew Edmunds
School of Electronics and Computer Science,
University of Southampton, UK
ae2@ecs.soton.ac.uk

Abdolbaghi Rezazadeh
School of Electronics and Computer Science,
University of Southampton, UK
ra3@ecs.soton.ac.uk

Michael Butler
School of Electronics and Computer Science,
University of Southampton, UK
mjb@ecs.soton.ac.uk

Renato Silva
School of Electronics and Computer Science,
University of Southampton, UK
ras07r@ecs.soton.ac.uk

Issam Maamria
School of Electronics and Computer Science,
University of Southampton, UK
im06r@ecs.soton.ac.uk

Chris Lovell
School of Electronics and Computer Science,
University of Southampton, UK
cjl3@ecs.soton.ac.uk

Abstract

Event-B is a formal modelling approach, with good tool support, for use in safety-, and business-critical systems. The methodology supports hierarchical, structured specification, beginning with a high-level abstraction of the system. Detail is added in successive refinement steps, progressing towards implementation. Bridging the abstraction gap, between the modelling and implementation, is addressed in the work that we describe here. Since, at the implementation level, we are concerned with implementation specifics we have chosen to narrow the application domain to that of multi-tasking, embedded control systems. We show how, from an Event-B model, we can generate Ada and C code. We have developed a new methodology, and extended the existing Event-B tool to facilitate code generation. The approach has been formulated so that it integrates well with the existing Event-B methodology and tools. Event-B is extended with AutoTask, Shared, and Environ machines. We use these features to specify how elements should be translated to code, i.e. controller tasks, protected objects and environment tasks. Using refinement, decomposition, and the extension, we structure projects so that they are amenable to automatic code generation. This necessitates the introduction of a great deal of low-level detail into the specification; so models grow in size considerably. To ensure models remain tractable, we decompose the models (perhaps a number of times). Decomposition encourages modular reasoning, and makes proof more tractable. We make use of a Event-B Components to record the link between the decomposed model. To make the Code Generator extensible we use a combination of the Eclipse extension mechanism and a Theory component for specifying mathematical language translations. To validate the approach we undertook a case-study, which we use as a running example.

1 Introduction

Event-B [2] is a formal method that can be used in the rigorous development of software systems. It may be used in by industry for business-, and safety-critical systems; to increase confidence in the correctness of the system [34, 31]. In this paper we focus on the domain of multi-tasking, embedded control systems. Our interest is the application of techniques, and provision of tools, for modelling the systems, and generating code from the models. In previous work [16, 17, 18] we described an intermediate language for generating Java code [23] for multi-tasking implementations, using an object-oriented intermediate specification. During our investigations, we encountered difficulties due to the large semantic gap between Event-B and the intermediate specification language. We also found that models were intractable due to large size of the models. We found that our language introduced too much fine-grained atomicity; which gave rise to a large number of proof obligations.

To address the problems encountered in the previous work, and advance the code generation techniques, we developed a methodology which is more closely integrated with the existing Event-B tools, and approach. In order to achieve a small semantic gap we make just a few additions to the Event-B language. To address the problem of intractably large models we use decomposition [12, 35]. When we are ready to provide implementation details, we use the constructs introduced in the extension (we call it Tasking Event-B) to annotate the development. The annotations and Event-B models are then used to generate code. Tasking Event-B is under-pinned by Event-B operational semantics, so we can also generate a model of the implementation, and show that it is a refinement of the development. We have developed a demonstrator tool [40] to validate the approach; the tool integrates with the existing Rodin platform [33]. We illustrate the approach using a case study with an embedded Heater Controller, and we use of Ada [5] as the target programming language. Our approach, however, is equally applicable to other programming languages, such as C [29].

We continue with section 1.1 in which we discuss our motivation; and section 2 provides an overview of the existing Event-B approach. Section 3 describes how we model and implement tasks and protected objects. Section 4 describes our approach to generating controller tasks modelling and simulating the environment. In Section 5 we present a case study. Section 6 describes how we can read/write directly to memory. Sections 7 and 8 discuss some theoretical aspects of implementing the model. Section 9 is a brief résumé of the code generation step. Section 10 provides a summary and discussion.

1.1 Motivation

The development of the Event-B method, and supporting tools, has been undertaken during the EU DEPLOY [41] project. A number of the industrial partners are interested in the development of multi-tasking, embedded control systems, to which the Event-B method has been applied. The modelling activity is beneficial, leading to a precise definition of the system under development, and shows that various properties of the system hold true as the development proceeds. In order to obtain the full benefit of the Event-B approach, we believe that automatic code generation is useful; so began the work on the final development step, of generating code. The project's industrial partners use a number of programming languages to implement their systems, but we chose one language, Ada [5], as a starting point for code generation; primarily because it is a stable, well structured, well-defined language. However, our approach is not limited to Ada, but we consider it to be a good choice because it is typically used in safety-critical systems. There is also a version called SPARKAda [1], for high-integrity systems, which is amenable to formal verification. One of our aims is to support compliance with the Ravenscar profile [9]. This is in readiness for support, in the future, of the multi-tasking version of SPARKAda, called RavenSPARK [10]. Implementations conforming to Ravenscar remove many problems associated with concurrency, and make the development amenable to timing analysis, which is an important

consideration in real-time systems.

The work presented here is informed by our previous experience with code generation. Our aim was to find a solution that integrates well with Event-B, having identified Ada as our target language. We have added features that facilitate modelling at the implementation level, and provide the necessary details for code generation. In our current work, we do not formally model all aspects of an implementation, such as timing issues. We do, however, model the behaviour that relates to the control flow; this is specified in a new task body construct, and we provide Event-B semantics - so we generate an additional model of the implementation and show that it refines that abstract development.

To validate the approach, we developed a case study [21] of a Heating Controller. It is typical of many embedded systems; inputs from the environment are received and processed, and may have some effect in the environment caused by its outputs. We say the inputs to the system are sensed, outputs are actuations. We are interested in modelling sensing and actuating in the environment, and we find it useful to separate the system under development into three distinct partitions. In our approach, we distinguish between three kinds of entities in the implementation - we model controller tasks, environment tasks, and protected objects. The controller tasks interact with the protected objects and environment tasks, but there is no direct communication between controller tasks; this is done via the protected objects.

machine HCtrl_M1
sees HC_CONTEXT
refines HCtrl_M0
variables cttm cttm2 ...
invariants
  cttm $\in \mathbb{Z}$
  cttm2 $\in \mathbb{Z}$
    ...

event Get_Target_Temperature2
any tm
where
  tm $\in \mathbb{Z}$
  tm = cttm
then
  cttm2 := tm
end

Figure 1: Example of Textual Event-B

2 An Overview of Event-B

The Event-B method [2] is a descendant of the classical B-Method [3], and less directly, Z [37]. It is a set-theoretic approach to modelling discrete systems, using predicate logic, first order logic, and refinement. The motivation for developing Event-B, from the existing B-Method, was to simplify the task of modelling and proof, and increase the use of automation [24]. An Event-B development consists of a project containing contexts and machines. Contexts are used to describe the static aspects of a system. Sets and constants are specified, and axioms describe the relationships between them. The dynamic aspects of a system are specified in machines. Machines are able to *see* contexts, so that the contents of a context is visible, and can be used within the machine. Variables keep track of the values, and guarded events update state. The properties of the system are specified in the invariants clause. The invariants of a machine give rise to proof obligations, which are generated automatically by the tool. A large number of these proof obligations can be discharged quickly, by automatic proof tools. Where the automatic provers fail, the user has the opportunity to guide the proof interactively. The proof activity can proceed, with the user suggesting strategies to complete the proof.

2.1 An Event-B Model

A fragment of an Event-B specification is shown in Fig. 1. The specification has a number of variable declarations, which are typed in the **invariants** clause. Additional predicates are added to the invariants clause to describe the desired safety properties. The machine invariant is a conjunction of the list of predicates in the invariants clause. The *Get_Target_Temperature2* event declares one parameter *tm* in the **any** clause. The **where** clause contains the event guards, the first clause is a typing predicate for *tm*. The second constrains the value that *tm* can take. The guards describe enabling conditions for an event. Each clause is a predicate over the sets, constants, and variables of the system. All of the guard clauses must be true for an event to be enabled. If an event is enabled, the updates in its action may occur if the event is selected by the environment. Event actions are defined in the **then** clause; they consist of assignments to machine variables; the assignment may be simple, or non-deterministic. The clause in the example consists of a single, simple assignment, where the variable *cttm2* is assigned the value of the parameter *tm*.

2.2 Machine Decomposition and Event Synchronization

Decomposition is a technique that is commonly used to handle complexity. There are currently two styles of composition that can be used with Event-B. They are the shared variable [4], and shared event [12], styles. In our approach, we make use of shared event decomposition. During the decomposition process, we decide where the machine variables will reside in the decomposed model. In the example of Fig. 1

we put *cttm2* in one machine, and *cttm* in another. Decomposition gives rise to two machines; each with an event called *Get_Target_Temperature2*. The event guards and actions will differ though, so it is not just a simple copy. Events can be recomposed following decomposition, to restore the original structure. It can be shown that the recomposed machines refine the original abstract machine, and recomposed events refine their abstract counterparts. The pairs of events, arising from decomposition, are said to synchronize [11]. The implementation of synchronized events maps naturally to an atomic subroutine, and we use this in our translation to code.

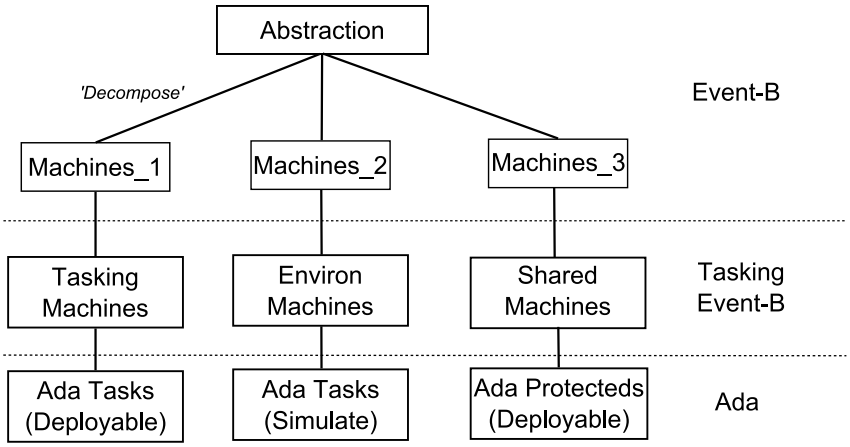

Figure 2: Linking Event-B to Code

3 Tasking Event-B: AutoTask and Shared Machines

The Tasking Event-B extension adds a small number of new constructs to Event-B, to facilitate the mapping of Event-B to code. By keeping the number of new constructs to a minimum, and by making use of existing editing tools, we have maintained a small semantic gap between Event-B and the Tasking Event-B specification. The Event-B decomposition approach described in [12, 35] is integral to our methodology. During decomposition, we identify the components of the system, and the interaction between them. We then introduce implementation specific constructs, which the translators use to guide code generation. The extended Event-B model also generates a model of the implementation.

Tasking Event-B extends Event-B by introducing the concepts of AutoTask, Shared, and Environ Machines. Each machine must be annotated with one of the following, to determine its implementation. AutoTask Machines are related to the concept of an Ada [39] task (but we are not restricted to Ada implementations). AutoTask Machines model *controller* tasks in the implementation, and can be viewed as an abstraction of an Ada [39] tasks (but we are not restricted to Ada implementations). Shared Machines are related to the concept of a protected resource, such as a monitor [27], or Ada protected object. Environ Machines model the environment tasks; these will be discussed in the next section. Figure 2 show the development artefacts, and how they relate to the generated code.

It is possible to generate a simple mapping from Event-B to code, for instance each event can map to a protected procedure. A scheduler can then be used to run the tasks according to a chosen schedule such as [38]. We take a different approach, where a task body is used to specify the schedule of activities within a task. We also specify the task life-cycle, such as periodicity of a repeating task, using annotations. These annotations are added to Event-B elements, and are used to guide the translation. The translation of an Event-B machine is characterised by one of the following annotations:

- *AutoTask Machine*, *Environ Machine Shared Machine*

AutoTasks are implemented by *controller* tasks, declared and elaborated in the *Main* procedure of the implementation. In Ada, a task declaration is the specification of a task's interface, and an elaboration is the task's implementation. Both parts may reside in the *Main* procedure. All of the tasks, generated from AutoTasks, are created when the program first loads, and automatically activated (made ready to run) by the run-time system; before the *Main* procedure body runs (hence the name AutoTasks). When we compile and build the implementation, we identify the *Main* procedure as the program entry point. In this paper, we only consider AutoTask machines, which are implemented as anonymous tasks of the

```
TaskBody ::=
   TaskBody ; TaskBody            // sequence
  | if EventSynch end             // branch
    [ else if EventSynch end ]
    [ else EventSynch end ]
  | do EventSynch od              // iteration
  | EventSynch                    // synchronization
  | Output                        // write to console


EventSync ::=
     Event ||_e Event
```

Figure 3: The Task Body Syntax

Main procedure. However, in the future we may wish to make use of Ada task types. These are used to create 'named' tasks, at compile time; thereby allowing re-use of machine specifications. However, whichever choice we make, fixing the number of tasks, before run-time, is a requirement for compliance with Ravenscar. Therefore, dynamic task allocation is prohibited.

3.1 Task Scheduling

The following extensions relate only to AutoTask and Environ machines, and allow a developer to provide implementation details, used by the code generators. Note, however, that timing aspects of periodic tasks, and task scheduling, are not modelled formally in the work that we describe here.

- *TaskType* and *Priority*

The *TaskType* construct is used to define the scheduling, cycle and lifetime of a task. i.e. one-shot, periodic or triggered. The period of a task is specified in milliseconds. *Priority* is an integer value; the task with the highest value priority takes precedence when being scheduled.

3.2 Flow Control

The controller tasks of a development are specified using a Tasking Event-B keyword; the machine is annotated with the *AutoTask* keyword. In the generated Ada code, this will be implemented as a controller task, as previously described. Each AutoTask machine has a *task body*. The task body contains flow control (algorithmic) constructs, which are described in the syntax shown in Fig. 3. The *sequence* construct is used for imposing an order on events. *branch* is choice between a number of events with mutually exclusive guards. *iteration* indicates that an event repeats until its guard is false. *EventSynch* is used to synchronize an event in an AutoTask machine, with an event in a Shared, or Environ machine. The *EventSync* construct defines an atomic update, between AutoTask and Shared machines; synchronization is implemented as an atomic procedure call. Between AutoTask and the Environ machines, synchronization may be implemented as an atomic *entry* call, or by direct access to memory. In Ada, a task *entry* is a kind of subroutine facilitating communication between tasks; it is part of the so-called rendezvous mechanism. It is important to note that, in our approach, controller tasks only communicate with each other through protected objects; however, controller tasks may communicate with environment tasks.

3.3 Events

Events can play one of several roles in the mapping to the implementation as follows,

procedureSynch	- the event is one of a pair of events modelling a parametrized procedure.
procedureDef	- the event (on its own) models a non-parametrized procedure.
branch	- the event models one part of a branch.
loop	- the event models conditional iteration update.

Events can play one of several roles in the mapping to the implementation. They can take part in event synchronization, parameterless procedure call, part of a branch, part of a loop. The Output construct is provided, to allow developers to output text to a console during simulation. We describe the events of an AutoTask as being *local* with respect to the AutoTask machine, and the events of a Shared machine as being *remote* with respect to the AutoTask machine. Events that are local to an AutoTask machine only update the AutoTask machine state; conversely, events that are remote only update the state of the Shared machine. Synchronised events share parameters to model communication between controller tasks and protected objects, and controller tasks and environment tasks. An Event-B development typically begins with a high-level abstraction of the system; and we refine, and decompose the models, until we are ready to use Tasking Event-B to describe the implementation. Figure. 2 shows how a development may proceed from abstract model to implementation. The abstract development may be refined, and then decomposed into separate components. In the Tasking Event-B stage, some of the machines are defined as AutoTask Machines; these will be implemented as Ada tasks and used for deployment. Some are defined as Shared Machines; these will be implemented as Ada protected objects and also used for deployment. The remainder are defined as Environ Machines; these will be implemented as the Ada tasks which can be used to simulate the environment.

3.4 Event Synchronization

An example of an AutoTask machine with synchronization, and other annotations, is shown in Fig. 4. The *EventSync* construct models parameter passing between controller tasks and protected objects. The *EventSynch* construct is represented in concrete syntax by the $\|_e$ operator. It can be seen in the figure in a labelled clause in the task body. The label $tc4$ can be translated to a program counter value, as part of the Event-B underlying semantics. This can then be used in invariant clauses to specify properties relating to specific events. In our discussion the labels help to clarify the link between specification and translated code, since we translate them to code comments.

In Tasking Event-B we match pairs of Event-B parameters in order of declaration, and map them to parameters in procedure declarations or calls. We know that controller tasks call protected objects' procedures, so we determine that AutoTask events have actual parameters, and Shared machine events have formal parameters. Fig. 5 shows the *SOGet_Target_Temperature2* event of the Shared machine, it synchronizes with the *TCGet_Target_Temperature2* event of the task shown in Fig. 4.

We wish to make a local copy of the Shared machine target temperature $cttm$, in the AutoTask machine, using the assignment $cttm2 := cttm$. However, in preparation for decomposition, we have ensured that the two variables do not appear in a single action; ultimately, *cttm2* and *cttm* will reside in different machines, so the information will need to be passed as a parameter. For this reason we have specified the event as shown in Fig. 1. There is an action $cttm2 := tm$ with guard $tm = cttm$. The decomposition of this event can be seen in Figs. 4 and 5 with the variables in separate machines.

The event parameters play a role in implementation; they map to the actual and formal parameters in the code. We assist the translator by adding annotations to the parameters, but in theory these could be derived automatically from the machine structure. We retain this as an open issue, for further investigation. During translation, an event's formal parameters will be mapped to formal parameters in a

machine Temp_Ctrl_TaskImpl
is autoTask
refines Temp_Ctrl_Task
variables cttm2, ...
tasktype periodic(250)
priority 5
taskbody is
...
tc4: TCGet_Target_Temperature2
$\|_e$ SOGet_Target_Temperature2;
...

event TCGet_Target_Temperature2
is procedureSynch
refines TCGet_Target_Temperature2
any
actualIn tm
where
tm $\in \mathbb{Z}$
then
cttm2 := tm
end

Figure 4: Example AutoTask Machine

machine SharedObject1Impl
is sharedMachine
refines SharedObject1
variables cttm ...
invariants ...

event SOGet_Target_Temperature2
is procedureSynch
refines SOGet_Target_Temperature2
any
formalOut tm
where
tm $\in \mathbb{Z}$
tm = cttm
then
skip
end
...

Figure 5: Example Shared Machine

protected object procedure parameter declaration. An event's actual parameters will map to the actual parameters of a call, wherever an *EventSynch* refers to the event in a task body. In Fig. 4 we see that *tm* is marked as an *actualIn* parameter, and in Fig. 5 *tm* is marked as a *formalOut* parameter. For translation to code, we the match parameters in order of declaration (rather than the names). The *actualIn tm* parameter of the AutoTask event becomes an actual parameter in a call; and the *formalOut tm* parameter of the Shared machine event becomes a formal parameter in a procedure declaration. The *EventSynch* construct may also be used with just a single event, in which case it is implemented as a procedure call with no parameters. A fragment of the generated code is shown in Fig. 6. It shows the Ada code from the *SOGet_Target_Temperature2* event, and the call to the protected object *s* with the actual parameter *cttm2*. The actual parameter, used in the call, is determined by its appearance on the left-hand-side of the assignment expression in the *TCGet_Target_Temperature2* event.

4 Tasking Event-B: AutoTask and Environ Machines

The previous section introduced the concept of *AutoTask Machines* and *Shared Machines*. We now introduce modelling of the environment, using *Environ Machines*. Controller tasks in the system do not communicate with each other; we prohibit inter-task communication because we wish to support

```
s: Shared_Object1Impl;
...
task body Temp_Ctrl_Task1Impl is
  ...
  s.SOGet_Target_Temperature2(cttm2);
  ...
end Temp_Ctrl_Task1Impl;
```

```
protected body SharedObject1Impl is
  procedure SOGet_Target_Temperature2
                        (tm: out Integer) is
  begin
    tm := cttm;
end;
```

Figure 6: Ada Implementation of a Synchronization

event Sense_Temperatures
  any t1 t2
  where
    $t1 \in \mathbb{Z}$
    $t2 \in \mathbb{Z}$
    t1 = ts1
    t2 = ts2
  then
    $stm1 := t1$
    $stm2 := t2$
end

Figure 7: An Abstract Sensing Event

implementations that conform to the Ravenscar profile [9, 10]. This profile prohibits the use of the task entries, which are required for inter-task communication. However, we consider that we can relax this restriction, for the *Environ Machines*. This is feasible since the model of the environment will be mapped to simulation code. The remaining deployable code can still be made to conform to the Ravenscar profile, if required. The *Environ Machine* is implemented as an Ada task, and communicates with the deployable tasks using Ada's rendezvous mechanism of *entries*. In this paper, we describe the implementations generated from the Environ machine as environment tasks, and the remainder as controller tasks.

When modelling embedded control systems it is useful to distinguish between the values we are measuring and setting in the environment, and their internal representation in the controllers. In this paper we describe the values in the environment as monitored or controlled variables (for sensing and actuating respectively). In the controller tasks, the corresponding stored values are described as sensed and actuated variables.

In a development we use an Environ machine to model the environment. The separation between controller components and environment components is done during decomposition 2.2. It is achieved by placing the monitored/controlled variables in one or more machines; and placing the sensed/actuated variables in others. After decomposition, communication between machines is modelled by the synchronization of the events. The implementation of the Environ machine will be somewhat different to that of the Shared machine, seen in the last section. The implementation of the latter uses protected procedures as opposed to task entries. Nevertheless, the annotations are very similar; we now introduce annotations for *sensing* and *actuating* roles. These are added as alternative options to *procedureSynch*, or *procedureDef*, and so on.

event TCSense_Temperatures $\triangleq$
refines TCSense_Temperatures
sensing
begin any
  actualIn t1
  actualIn t2
when
  t1 $\in \mathbb{Z}$
  t2 $\in \mathbb{Z}$
then
  stm1 := t1
  stm2 := t2
end

event ENSense_Temperatures $\triangleq$
refines ENSense_Temperatures
sensing
begin any
  formalOut t1
  formalOut t2
when
  t1 $\in \mathbb{Z}$
  t2 $\in \mathbb{Z}$
  t1 = ts1
  t2 = ts2
then
  skip
end

Figure 8: Synchronization of a Sensing Event

sensing - the event is one of a pair of events modelling sensing.
actuating - the event is one of a pair modelling actuation.

The pair of events arising from decomposition (one in the AutoTask machine, the other in the Environ machine) will be annotated to indicate participation in either sensing or actuating roles.

An abstract sensing event can be seen in Fig. 7. It is the first refinement from the case study presented in Section 5; the model has yet to be decomposed, therefore it has no tasking annotations since these are added as the last step of the development. Sensed variable $stm1$ keeps track of the monitored temperature $ts1$ through the assignment $stm1 := t1$, and guard $t1 = ts1$ and so on. Note that the guards $t1 = ts1$ and $t2 = ts2$ link the parameters to the monitored variables. Through decomposition, we obtain the synchronizing events in different machines.

The *Sense_Temperatures* event is decomposed into two events, one in each of the machines. In preparation for decomposition we named the AutoTask machine *Temp_Ctrl_Task1*, and the Environ machine was named *Envir1*. In a subsequent refinement the events were renamed to *TCSense_Temperatures* (in the AutoTask machine) and *ENSense_Temperatures* (in the Environ machine). These events can be seen in Fig. 8, which show the decomposed events with their Tasking Event-B annotations. The sensed variables *stm1* and *stm2* reside in the AutoTask machine, and can be seen on the left-hand side of the assignment in the *TCSense_Temperatures* event (on the left side of the figure); and monitored variables *ts1* and *ts2* reside in the in the Environ machine, and appear in the guard of the *ENSense_Temperatures* event. In Event-B decomposition the order of parameter declaration is not important, synchronization is the same regardless of parameter order. However, in Tasking Event-B we introduce a further constraint on parameter declaration to assist with translation. In translating to code we match parameters in order of declaration. Using this constraint simplifies the translation process; see Sects. 8 and 8.2 for more details of the mapping from synchronized events to the implementation.

5 Case Study

This section describes an Event-B development of a simple heating controller. This case study covers the entire development process; starting from a system specification and ending with implementable Ada

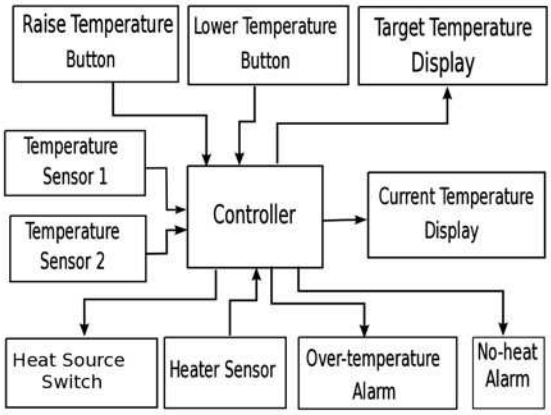

Figure 9: Heating Controller Architecture

code. The methodology that we propose here is key to optimising the development process, since the abstract development must be structured in a way that is appropriate for code generation. The development process starts with an abstract specification followed by two successive refinements. We then decompose the model into two separate sub-models, one representing the environment, and the other representing the remainder of the system. The refinement process continues after the first decomposition in order to arrive at a concrete level suitable for implementation. This allows us to derive the appropriate machine structure, needed by the code generation plug-in. The second decomposition gives rise to a number of AutoTask machines, and a Shared machine to manage the protected shared data. The Shared machine synchronizes with the AutoTask machines to model the communication between them. We illustrate how, using the code generation plug-in, a concurrent Ada implementation is generated. The overall aim of this case study is to put in practice the recommended methodological aspects of Event-B, particularly those aspects of modelling concerned with decomposition and code generation.

5.1 Overview of the System

Figure 9 shows an overview of the Heating Controller and related components. The controller in the centre of the diagram communicates with the components in the environment using input and output parameters. At the top there are two buttons that allow the user to increase/decrease Target Temperature. The target temperature periodically will be sent by the controller to the related display to be shown to the outside world. The controller uses two Temperature Sensors to poll the environment temperature. The average of the values read from these two sensors is calculated and displayed by the controller on the Current Temperature Display. If the current temperature is lower than the target temperature, the controller will turn on the heater source using Heat Source Switch, otherwise this switch will be turned off by the controller. The status of the heater itself also will be monitored through the Heater Sensor. If due to some faults the heater is not working properly, the controller will activate either of Over-temperature Alarm or No-heat Alarm.

5.2 System Development

Before discussing Event-B models the Heater Controller in more detail, we provide an overview of the development approach in Fig. 10. Starting from the top level of the diagram, it shows the most abstract model of the system, or as it is known - the system specification. At the top level we specify the system's main functionality. This includes events for modelling increase/decrease target temperature,

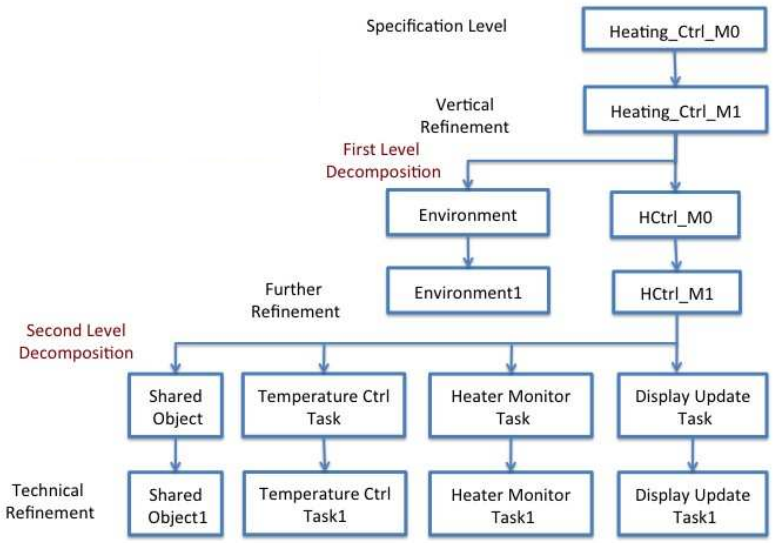

Figure 10: The Development Approach

polling the environment temperature using the two available sensors, calculating the current temperature, turning on/off the heat source, and activating/deactivating the two different alarms. Events at this level are deliberately simple, to keep the specification clear and concise. Event ordering and iteration are introduced in the later stages. In the first refinement we introduce sensing and actuation, which are considered to be design steps. Sensing events model the polling of the state of the increase/decrease buttons, the temperature sensors, and the heater sensor. Actuating events model the updates of target, and current temperature displays. The also model actuation occurring as a result of controller decisions, such as turning the heat on/off, and activating the various alarms.

As illustrated in Fig. 10 we decompose our model in two stages. In the first stage we separate the controller, the part of the system that should be implemented, from its surrounding environment. The second stage of decomposition is concerned with the tasking structure of the implementable system. The structure of a particular development is determined by the developer, based on their understanding of the required implementation.

5.2.1 First Decomposition

In Event-B, we usually start modelling by specifying the whole system as a closed system. This includes the system that we are intending to develop and its surrounding environment. Therefore when our model of the system becomes large and complex it is reasonable to separate the controller from its environment. Hence in the first stage of decomposition, we partition our model into those two parts, as shown in Fig. 11. The environment machine models all of the external parts of the system that the controller interacts with. These include buttons, sensors, actuators and their related behaviours. In addition to modelling updates in the components, the machines modelling the environment and controller, also model the synchronisation between the two.

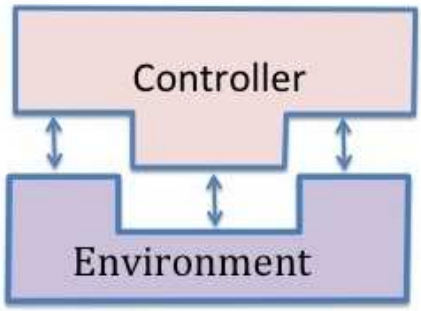

Figure 11: Decomposition - Stage 1

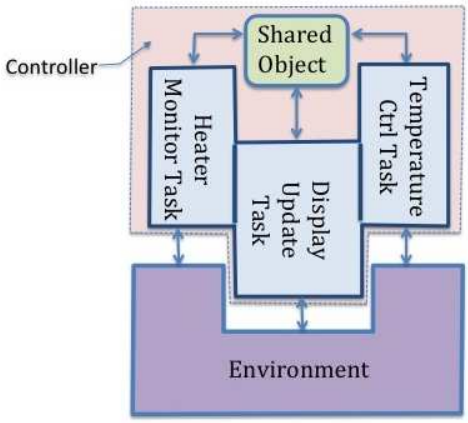

Figure 12: Decomposition - Stage 2

5.2.2 Second Decomposition

In this, the second stage of decomposition, we decompose the controller into three different tasks which interact with each other through a protected object. This structure is presented in Fig. 12. Note that individual tasks are able to communicate with the environment directly, but do not communicate with each other. In practice, the number of tasks, and the way that we distribute different events over these tasks is determined by various factors. Other properties of tasks, such as their priorities; life-cycle (periodic, triggered etc); length of period in milliseconds, if applicable, are defined later using Tasking Event-B.

5.3 Tasking Event-B

When we are ready to start adding implementation level details to the model we use Tasking Event-B annotations; entering the *Tasking Event-B* modelling stage shown in Fig. 2. We do not present the whole case study specification here, but we use extracts to illustrate the main issues. In the following discussion we use the temperature control task event *Temp_Ctrl_Task1Impl* from Fig. 4, as an example. Our main aim is to highlight how we use Tasking Event-B to facilitate the task's interaction with the environment, but we also discuss the task's behaviour in a more general sense. The first step is to use annotations to identify the machines, as one of AutoTask, Environ or Shared machines. When we have identified machines as either Environ, or AutoTask, then we add a task body specification. The task body is used to constrain the Event-B model, in such a way that it can be implemented using commonly available programming constructs, such as sequence, branch and subroutine calls. The generated code can be viewed as an implementation of a schedule of events. Fig. 13 shows the *Temp_Ctrl_Task1Impl* task body. The clauses of the task body are labelled, to assist with translation, and the following list describes the behaviour of the events that they refer to. The task body gives rise to the Ada task body of Fig. 14.

The development proceeds by adding annotations to events; we saw annotated sensing events in Fig. 8. The *sensing* keyword is used with the *TCSense_Temperatures* and *ENSense_Temperatures* events. This indicates that the events model polling of the environment; the *actuating* keyword is similar, except that it indicates that events update values is the environment. The clause $tc1$ in the task body, shown in Fig. 13, gives rise to the following Ada program statement:

$$\text{Envir1Impl.ENSense\_Temperatures(stm1, stm2);} \qquad --tc1$$

tc1 - a sensing synchronization that models polling of the $ts1$ and $ts2$ temperature sensors, as shown in Fig. 8.
tc2 - an update that calculates, and stores locally, the average of the two temperature values.
tc3 - an actuating synchronization that updates ctd, the displayed temperature.
tc4 - a procedure synchronization that gets the target temperature from the protected object.
tc5 - branching choice: set the heater on or off flag in the task.
tc6 - a procedure synchronization that sets the heat source active flag in the protected object.
tc7 - an actuating synchronization that updates *ahsa*, the activate heat source flag.
tc8 - a branching choice: set activate overheat alarm flag in the task.
tc9 - an actuating synchronization that updates *aota*, the activate overheat alarm flag.

tc1: TCSense_Temperatures $\|_e$ ENSense_Temperatures;
tc2: TCCalculate_Average_Temperature;
tc3: TCDisplay_Current_Temperature $\|_e$ ENDisplay_Current_Temperature;
tc4: TCGet_Target_Temperature2 $\|_e$ SOGet_Target_Temperature2;
tc5: **if** TCTurnON_Heat_Source **end if**
else TCTurnOFF_Heat_Source **end else**;
tc6: TCSet_Heat_Source_State $\|_e$ SOSet_Heat_Source_State;
tc7: TCActuate_Heat_Source $\|_e$ ENActuate_Heat_Source;
tc8: **if** TCSwitchOn_OverHeat_Alarm **end if**
else TCSwitchOff_OverHeat_Alarm **end else**;
tc9: TCActuate_OverHeat_Alarm $\|_e$ ENActuate_OverHeat_Alarm

Figure 13: The Temp_Ctrl_Task1Impl Task Body

Envir1Impl is the name of the environment task, and *ENSense_Temperatures* is the name of the task entry. The sensed variables $stm1$ and $stm2$, appear in the actions of $TCSense\_Temperatures$. The action models assignment of return values to the variables. In the translation to Ada the AutoTask event parameters is replaced by machine variables, and the variables are passed as actual parameters in an entry call. The entry is implemented as an Ada *accept* statement, see Fig. 15, in the $Envir1Impl$. In the body, statements assign the temperature values $ts1$ and $ts2$ to the formal parameters $t1$ and $t2$. The variables $ts1$ and $ts2$ were added to the Event-B model, in a refinement step, to represent the monitored values in the environment. The monitored variables appear in the guard of the *ENSense_Temperatures* event. See Fig: 8. With Ada's pass-by-result parameter semantics, the assignment statement $t1 := ts1$ ensures $stm1$ is assigned the value of $ts1$ upon return. More details of the translation of synchronized events can be found in Section 8.

6 Writing Directly to Memory Locations

The discussion, so far, has focussed on a simulation of the environment whereby the task communicates with the environment using an entry call. In Ada this is implemented as an entry call to the environment task. It may be the case, however, that the developer can specify some memory locations to read from, and write to, directly. Our approach allows developers to annotate event parameters with the address information. We have introduced the $addr$ annotation, with which we define an addressed variable. It is used in conjunction with the existing $actualIn$. $actualOut$, $formalIn$, and $formalOut$ annotations. With $addr$ we can specify a memory location and its number base. In the Tasking Event-B specification, on

```
task body Temp_Ctrl_Task1Impl is
  ...
  procedure TCCalculate_Average_Temperature is
  begin
    avt := ((stm1 + stm2) / 2);
  end;
  begin
  ...
  Envir1Impl.ENSense_Temperatures(stm1, stm2);                  -- tc1
  TCCalculate_Average_Temperature;                              -- tc2
  Envir1Impl.ENDisplay_Current_Temperature(avt);                -- tc3
  shared_object1implInst.SOGet_Target_Temperature2(cttm2);      -- tc4
  if(avt < cttm2) then                                          -- tc5
    hsc := TRUE;
  else
    hsc := FALSE;
  end if;
  shared_object1implInst.SOSet_Heat_Source_State(hsc);          -- tc6
  Envir1Impl.ENActuate_Heat_Source(hsc);                        -- tc7
  if(avt > Max) then                                            -- tc8
    ota := TRUE;
  else
    ota := FALSE;
  end if;
  Envir1Impl.ENActuate_OverHeat_Alarm(ota);                     -- tc9
  ...
end Temp_Ctrl_Task1Impl;
```

Figure 14: Implementation of Temp_Ctrl_Task1Impl Task Body

```
accept ENSense_Temperatures(t1: out Integer; t2: out Integer) do
  t1 := ts1;
  t2 := ts2;
end ENSense_Temperatures;
```

Figure 15: Implementation of ENSense_Temperatures

the left of Fig. 16, the parameter $t1$ is given the address $ef14$ in base 16. We can see, on the right, the generated Ada code. The parameter $t1$ has been mapped to the integer variable declaration t1: Integer. The address of the variable has been set using the following statement, the **pragma** Atomic(t1) statement is used to indicate that any access to $t1$ must occur atomically:

```
for t1'Address use System'To_Address(16#ef14#);
pragma Atomic(t1);
```

In Ada this is known as an *address clause*. In the Ada *TCSense_Temperatures* procedure, also shown in Fig. 16, the variable $t1$ appears on the right-hand side of the assignment. When the statement is executed, the value is read from the memory location accessed by $t1$, and assigned to $stm1$. If we were generating

C code, we would declare a pointer to integer type,

int* t1 = (**int***) 0xef14

and use an assignment stm1 = *t1 . We can see that this approach differs from that of the entry-call approach shown earlier; the controller task updates its value without a call to an external, environment entry. Now, since there is no call to an entry (which is atomic) the environment must be responsible for ensuring that sensing events with multiple read actions, and actuating events with multiple write actions, are performed atomically. When using addressed variables, in Tasking Event-B, we can discard the environment tasks, and simply use the controller tasks and protected objects in the deployment. In this case we need not consider the effects of atomicity further. However, we may wish to use the environment model for other purposes, in which case we need to ensure that atomic access is achieved in the code.

In the initial stage of development we are able to simulate interaction with the environment using entry calls to the environment task. It may be the case that later in the development, we choose to read from, and write to, memory directly. To do this we simply add the address information to the relevant variables. We can then choose to continue using the environment simulation, which now writes to memory also; or we may have in mind to use another simulator, or perhaps use the code for deployment. If we choose to use environment simulation using addressed variables, we add address information to the monitored variables in the environment. The translator then generates address clauses, and omits the environment entries. The environment task procedures then update monitored variables, in memory, directly.

```
event TCSense_Temperatures ≙
refines TCSense_Temperatures
sensing
begin any
  actualIn  addr(16,ef14) t1
  actualIn  addr(16,ef18) t2
when
  t1 ∈ ℤ
  t2 ∈ ℤ
then
  stm1 := t1
  stm2 := t2
end
```

```
task body Temp_Ctrl_Task1Impl is
  stm1 : Integer := 0;
  stm2 : Integer := 0;
  ...
  procedure TCSense_Temperatures is
    t1 : Integer;
    for t1'Address
      use System'To_Address(16#ef14#);
    pragma Atomic(t1);
    t2 : Integer;
    for t2'Address
      use System'To_Address(16#ef18#);
    pragma Atomic(t2);
  begin
    stm1 := t1;
    stm2 := t2;
  end;
  ...
begin
  loop
    delay until nextTime;
    TCSense_Temperatures;
    ...
  end loop;
end Temp_Ctrl_Task1Impl;
```

Figure 16: Addressed Variables: Specification and Implementation

7 Implementing the TaskBody Construct

We now look at the mapping of the flow control constructs to a programming language abstraction. The abstraction appears in the text as a kind of pseudo-code, and its potential mapping to a programming language, such as Ada, should be obvious. The pseudo-code makes use of Event-B guards and actions, and it represents an intermediate translation step during code generation. The intermediate step is not visible to users, though, so we will say no more about it here.

The task body of an AutoTask machine contains *Control* constructs, such as sequence, branch and iteration; the translation of which is mostly straight forward. Control constructs may contain events, and it will be useful, here, to remind ourselves of some terminology, since we make use of it in the remainder of the section. Local events are local to an AutoTask machine, they can only update the task's state; conversely remote events belong to any Environ, or Shared machine, and update that machine's state. Synchronized events' parameters model communication between AutoTask, and Shared machines. In [19] we described the synchronization of the two events using the definition 1 based on the guarded command language [15]. The left-hand side defines the synchronization $\|_e$ operator, it relates the decomposition (on the left-hand side) to the composed event (on the right-hand side).

$$g_l \rightarrow a_l \,\|_e\, g_r \rightarrow a_r \triangleq g_l \wedge g_r \rightarrow a_l \,\|\, a_r \tag{1}$$

We now wish to work with the machine variables, and event parameters, so we extend the notation to allow the guards and actions to range over ordered sets of parameters, P and Q, of the events of local and remote machines respectively; and sets of variables V and W of the local and remote machines. We can then re-write (1) as (2).

$$g_l(P,V) \rightarrow a_l(P,V) \,\|_e\, g_r(Q,W) \rightarrow a_r(Q,W) \triangleq g_l(P,V) \wedge g_r(Q,W) \rightarrow a_l(P,V) \,\|\, a_r(Q,W) \tag{2}$$

We make use of the parameter definitions, in the translation of the EventSynch construct, shown in Fig. 17. The actual parameters of the procedure calls are $p \in P$, the formal parameters used in the subroutines are $q \in Q$. In Fig. 17 we describe how the control constructs, events, and their guards and actions, relate to the programming constructs. The right-hand side of the table shows the implementation of the control constructs, but the guards and actions are left as Event-B predicates and expression, to be translated at a later stage. More details of the translation, to code, of the synchronization constructs are given in Sect. 8.

An unconstrained specification and implementation may lead to undesirable behaviour. This may occur in a naive implementation of an event; say the event is translated into a subroutine with a conditional critical region [25] which can block the caller if the condition is false. In this case a task, if the task is both caller, and target of the call, may block itself. This is not desirable since the task state is not visible externally and the task would remain blocked. We therefore prohibit the use of a local guard g_l. This can be seen in the redefinition of the synchronization operator (3). Here, we have omitted the details of parameters and variables for clarity. The omission of the local guard is relevant in the translation of the *EventSynch* construct in Fig. 17.

$$a_l \,\|_e\, g_r \rightarrow a_r \triangleq g_r \rightarrow a_l \,\|\, a_r \tag{3}$$

The *Branch* and *Loop* constructs are subject to the restriction that guards are defined for the local event only, that is g_l. There is no remote guard, since the generated code would require a protected procedure call for each branching clause. We decided to keep the implementation as simple as possible, and use just local guards which are not subject to interference. This restriction is summarised in (4) which shows a simple branch as an example. It makes use of the alternative choice operator, $[]$, where each guard

Control	$< Control >^T$
$Control1$; $Control2$	$< Control1 >^T$; $< Control2 >^T$

`EventSynch` $e_l \parallel_e e_r$
where $e_l \parallel_e e_r =$
$a_l \parallel_e g_r \rightarrow a_r$
	a_l`(); call target.`e_r`(`$p_1 \ldots p_n$`)`
	Add to task:
	`subroutine` e_l`(){` a_l`; }`
	Add to Protected Object:
	`subroutine` e_r`(`$q_1 \ldots q_n$`)when(`g_r`){` a_r`; }`

L`: DO` $e_l \parallel_e e_r$ `OD`
where:
$e_l \parallel_e e_r = g_l \rightarrow a_l \parallel_e a_r$
	Add to task body:
	`while(`g_l`){`
	a_l`; call` e_r`();`
	`}`
	Add to Protected Object:
	`subroutine` e_r`( ){` a_r`; }`

L`: IF` $e_{1l} \parallel_e e_{1r}$ `ENDIF`
`[ ELSEIF` $e_{il} \parallel_e e_{ir}$ `ENDELSEIF]`..
`[ ELSE` $e_{nl} \parallel_e e_{nr}$ `ENDELSE]`
 	
$i \in 1..n$
where:
$e_{il} \parallel_e e_{ir} = g_{il} \rightarrow a_{il} \parallel_e a_{ir}$
	Add to task body:
	`[if(` g_{1l} `){` *body* `}]`
	`[elseif(`g_{il}`){` *body* `}]`..
	`[else{` *body* `}]`
	
	body $\triangleq$
	a_{il}`; call` e_{ir}`();`
	
	and in Protected:
	`subroutine` e_{ir}`(){` a_{ir} `}`

Figure 17: Flow Control Implementation

is disjoint, and in order to ensure that the branches cover all possible outcomes, the conjunction should form a tautology. Our tool should produce proof obligations to ensure this, but is not yet implemented. Similar restrictions are applied to the loop construct, thus our loop definition has the same form as the simple branch.

$$g_l \rightarrow a_l \parallel_e a_r \; [] \; \neg g_l \rightarrow SKIP \tag{4}$$

7.1 Parallel to Sequential semantics

An Event-B action with a number of assignments is a parallel composition of those assignments. Translation to Ada involves serialising the assignments. Parallel composition does not readily translate to its serial form without the introduction of some auxiliary variables to hold the values of machine variables on entry to the procedure. Take the parallel assignment $x := y \parallel z := x$. If $x = 1$ initially, and $y = 0$, then we should have $z = 1$ after the update. If we were simply to serialize this $x := y$; $z := x$, then we would have $z = 0$. To remedy this we introduce auxiliary variables for any variable appearing on the left-hand side of an assignment, and also on the right-hand side of some other assignment. We then replace variables appearing on the right-hand side of assignments with the auxiliary variable. Since x appears on the

Construct	Parameter Mapping
AutoTask machine	actual
Shared Machine	formal
Action	in
Guard	out

Figure 18: Parameter Mappings

left-hand side and right-hand side of the clauses, we introduce an auxiliary variable *ini_x*, and translate the action to $ini\_x := x \; ; \; x := y \; ; \; z := ini\_x$

8 Implementing Synchronized Events

Following the decomposition step, we identify how machines will be implemented. They can be annotated to identify them as AutoTask, Shared, or Environ Machines. In the discussion that follows we show how the events of these machines relate to synchronized communication using a subroutine-style call. We then look at how synchronized events relate to implementations using direct memory access.

8.1 ProcedureSynch Events

In this section we describe how *procedureSynch* events map to the implementation. In Tasking Event-B synchronizations are atomic, and this should be enforced in the implementation. To enforce this we map the pair of events to a single procedure in a protected object. The protected object is Ada's mutual exclusion mechanism.

Event Parameters are paired (using order of declaration) between the synchronized machines. We show how (using synchronization) the Event-B parameters, guards and actions provide implementations of update actions; provide formal input and output parameters for the procedure, provide actual input and output parameters for use by the caller of the procedure. Table 18 shows how the relationships. For instance, parameters of tasks give rise to actual parameters of the procedure call. Parameters that appear in event guards (other than their typing predicates) map to out parameters etc.

To assist with our explanation we remind ourselves of the description of parameters and variables. The local event has an ordered set of parameters P, and a set of variables V. The remote event has an ordered set of Parameters Q, and set of variables W, and we can write the synchronization as follows,

$$g_l(P,V) \rightarrow a_l(P,V) \parallel_e g_r(Q,W) \rightarrow a_r(Q,W)$$

We use p to denote an event parameter of a local machine, and q to denote an event parameter in a remote machine; where $p \in P$, and $q \in Q$ from our definition. In a synchronization we match parameter pairs in order of declaration; and from the semantics of decomposition/composition we say that $p = q$. We denote variables as v and w of local and remote machines respectively; where $v \in V$ and $w \in W$. In our current work we only consider variables of integer and Boolean type. In our implementations these map to Ada elementary types; parameter passing for elementary types, for *in* parameters, has call-by-value semantics; and for *out* parameters call-by-result semantics. With call-by-value, the actual parameter value is assigned to the formal parameter, $q := v$, on procedure entry. With call-by-result, the formal parameter value is assigned to the actual parameter, $v := q$, on return from the procedure.

In an abstract procedureSynch event we have assignment actions $v := w$; we will now show that our code implements this assignment, following translation using decomposed machines. During decomposition v and w are put into different machines. We introduce parameters, prior to decomposition, to

facilitate this. In a refinement we introduce a parameter p to the development. Then the action $v := p$, with guard $p = w$, refines $v := w$. Decomposition then takes place; it gives rise to an AutoTask machine (local) event, with action $v := p$; this synchronizes with a Shared machine (remote) event with a skip action, and guard $q = w$. From event composition semantics we know that $p = q$. We now wish to show that our code implements $v := w$, when translated from the decomposed events. In our description of the code we use the same variable names, primed. So v translates to v'. From Table 18 we know that parameter p in the AutoTask machine is an *actualIn* parameter, and in the Shared machine q is a *formalOut* parameter. The translation gives rise to a procedure, in the protected object. The procedure body contains the statement $q' := w'$, where q' is a formal *out* parameter. The caller of the procedure supplies the actual parameter v', so (following Ada's call-by-result semantics) on return from the call, v' is assigned the value of q'; that is, $v' := q'$. We know that $q' := w'$, from the procedure body, and the return assignment $v' := q'$ follows the procedure body; therefore $v' := w'$ as required by the specification.

We now look at an assignment in an abstract procedureSynch event, $w := v$; we show that our code implements this assignment, following translation using decomposed machines. During decomposition v and w are again put into different machines, and again, we introduce parameters, prior to decomposition. We introduce a parameter p to the development. Then the action $w := p$, with guard $p = v$, refines $w := v$. Decomposition gives rise to an AutoTask machine event with a *skip* action and guard $p = v$; this synchronizes with a Shared machine event, with action $w := q$. From Table 18 we know that parameter p in the AutoTask machine is an *actualOut* parameter, and in the Shared machine q is a *formalIn* parameter. The translation gives rise to a procedure, in the protected object. It has the statement $w' := q'$, where q' is a formal *in* parameter. The caller of the procedure supplies the actual parameter v', so (following Ada's call-by-value semantics) on entry to the procedure, q' is assigned the value of v'. That is, $q' := v'$; and then $w' := q'$, in the procedure body; therefore $w' := v'$, as required.

8.2 Sensing and Actuating Events

In this section we describe how synchronized sensing and actuating events map to code implementing the Ada rendezvous mechanism; for communication between the controller and environment tasks. In the environment simulation, the sensing/actuating events map to an entry in the environment task, and a call in the controller task. The entry is an Ada specification, implemented by an accept clause. In the environment's accept clause, the monitored variable values can be assigned to the sensed variables of the controller task (sensing); or controlled variables in the environment task can be assigned the values of actuated variables of the controller task (actuating). An event is either actuating or sensing, but not both. Event Parameters are paired (using order of declaration) between the synchronized machines. In the case of sensing, we show how (using synchronization) the Event-B guards and actions are used to implement assignment of values in the environment to variables in the AutoTask Machine. In the case of actuation we show how they relate to the setting of variables in the environment. In the discussion that follows we denote parameters as p and q, as in the previous subsection; and we denote variables as v and w. In this section we provide a slightly different interpretation, however, since variables v represent the sensed/actuated variables of the AutoTask machine (local). Variables w are the monitored/controlled variables in the environment (remote).

In a sensing implementation we want the environment to perform the assignment $v := w$. The value of the monitored variable w in the environment is assigned to the sensed variable v in the AutoTask machine. During decomposition v and w are put into different machines. We introduce parameters, prior to decomposition, to facilitate this. In a refinement we introduce a parameter p to the development. As a result of this, the action $v := p$, with guard $p = w$, refines $v := w$. Decomposition then takes place, and we obtain an event in the AutoTask machine, with an action $v := p$, and a synchronizing event in the Environ machine, with a guard $q = w$. Translation gives rise to an *accept* statement $q' := w'$, where

q' is a formal *out* parameter. So (following Ada's call-by-result semantics) on return from the call, v' is assigned the value of q'; that is, $v' := q'$. In the accept statement $q' := w'$ is followed by $v' := q'$. Therefore we have $v' := w'$, as required.

In an actuating implementation we want the environment to perform the assignment $w := v$. The controlled variable w in the environment is assigned the value of the actuated variable v in the AutoTask machine. In Ada, the statement $w' := q'$ is implemented in the body of an environ task *accept* statement. This is mapped from the Environ machine's synchronized action $w := q$. Here q is a *formalIn* parameter. On entry to the *accept* body, the formal *in* parameter q' is assigned the value of the actual parameter v'. So, $q' := v'$. From the assignment on entry to the *accept* body $q' := v'$ and then the *accept* bodies statement $w' := q'$, we can see that $w' := v'$, as required.

In Tasking Event-B sensing/actuating synchronizations are atomic, and this is enforced in the implementation by Ada's rendezvous mechanism in the implementation.

8.3 Addressed Variables

Tasking Event-B addressed variables are annotated machine variables of the Environ machine; or annotated event parameters (which are local variables) of AutoTask machine sensing, and actuating events. Parameters p of AutoTask machine events map to *local variables* p' in the implementation (they do not map to subroutine parameters). Machine variables w of an Environ machine, map to *variables* w' in the implementation. Each parameter p, of an AutoTask machine's *sensing* events, is paired with a variable w in the Environ machine. They share a common address loc. We annotate the declaration of w as follows,

$$addr(b,\ loc)\ w\ ,$$

where b is the base, and loc is the memory location.

Now consider the case where a task senses the value of a monitored variable w from the environment. At the most abstract level, this involves making an assignment $v := w$, where v is a sensed variable in the task. Following decomposition we obtain synchronized events, with the action $v := p$ in the AutoTask event. In Tasking Event-B we annotate the parameter p with the location loc as follows,

$$addr(b,\ loc)\ p\ ,$$

where b is the base, and loc is the memory location. Therefore, the addresses of p and w are the same. The address declaration translates to Ada code as a variable declaration statement $p' : Integer$, followed by an address clause,

for p''Address **use** System'To_Address(b#loc#).

Note the use of the ' character for accessing Ada attributes. In the *for* clause, p''Address, accesses the Address attribute of the variable p'. The Address attribute is set to the address object constructed in the *use* clause. The translation also generates a procedure declaration, in the task; from the AutoTask machine action $v := p$ we obtain the procedure body $v' := p'$. Remembering that p' refers to the value held at location loc, we have $p' = w'$. Then the assignment $v' := p'$ equates to $v' := w'$, as required by the specification.

We now consider actuating of a controlled variable w in the environment. At the most abstract level, this involves making an assignment $w := v$, where v is an actuated variable in the task. Following decomposition we obtain synchronized events, with the guard $p = v$ in the AutoTask event. In the Tasking Event-B we indicate that we want to read from loc by specifying an address annotation for parameter p, as above. The generated variable declaration of p', and the address clause is also the same as the above.

The translation generates a procedure declaration, in the task; from the AutoTask machine guard $p = v$, we obtain the procedure body $p' := v'$. Remembering that p' refers to the value held at location *loc*, we have $p' = w'$. Then the assignment $p' := v'$ equates to $w' := v'$, as required by the specification.
– A Question about atomicity.
In a specification, the synchronizing events atomically set, and read from, variables in the environment. So, an event that writes to two controlled variables in the environment $w1 := v1 \parallel w2 := v2$ has a guarantee that if $v1 = v2$ then $w1 = w2$. In the approaches presented up until now the protected procedure and task entries ensure tha this is the case. However in the implementation accessing memory directly it is implemented in a serial fashion $w1' := v1'; w2 := v2$. In this case there is no guarantee that $w1 = w2$ if $v1 = v2$. We therefore need to restrict sensing and actuating to a single action per event to maintain the refinement relation.

9 Generating and Executing Code

Code is generated from the Tasking Event-B model in a two-step process. The first step generates an common language model (CLM), from which a number of target implementations could be generated. The CLM is an abstraction of commonly used software constructs. Translation to a specific target language takes place in the second step. Our current tool generates a pretty print of Ada code, which we copied and placed in appropriate files in an Ada development environment. We successfully compiled and ran the code with no further changes to the automatically generated code. The environment simulation, that we specified, manipulated changes to the monitored variables; changes were reflected in the environment's controlled variables after the heater controller tasks had reacted. The variable values in the environment are written to a console, using the *Output* construct, to provide feedback.

10 Conclusions

The main focus of the work that we report here is modelling of, and providing implementations for, multi-tasking embedded control systems. We initially model the system using Event-B, but to generate implementations in the coding style that we desire, we have the need for features that do not exist in Event-B. We add various modelling constructs, in an extension to Event-B, that are used to model implementation-level details, including the controller's interaction with the environment. We make use of the extensions to guide code generation. Our contribution is a methodology, and tool support, for facilitating this. We have validated the method and tools using a case study, which provides a basis for future developments in our sphere of interest.

10.1 discussion

In our previous work [16, 17, 18], we found that models quickly became intractable; the models became very large and generated a large number of proof obligations. To address this problem we use the decomposition approach of [12, 35]. After decomposition, the models can be refined further, and when we are ready to provide implementation details, we use the constructs introduced in the extension. The annotations and Event-B models are then used to generate code. In previous work we described an object-oriented intermediate language that we used, to guide code generation. However, we encountered difficulties due to the large semantic gap between Event-B and the intermediate specification language. In the work presented here, the methodology maintains a small semantic gap, we achieve this by adding only a minimal number of constructs to the Event-B language. The implementation of a machine, for instance, can be either a controller task, environment task, or protected object. We apply a single annotation to the machine, which describes the implementation. In the case of the tasks, we add a task body annotation describing the event ordering; which includes branching and iteration constructs. Individual events can map to a number programming constructs; their implementation is defined by their use in the task body, and by any additional annotations.

The work presented here, also includes a solution for modelling low-level interaction with the environment; that is, sensing and actuating. We take into consideration that, in the final deployable system, inter-task communication is prohibited. The main driver for this restriction is that we wish to generate safe multi-tasking code which is compliant with the Ravenscar subset of Ada [10]. However, we are able to relax this restriction for environment tasks, since our primary task is to simulate the environment. The sensing and actuating events give rise to two styles of implementation: one using entry calls, and one using addressed variables. In the entry-call style, calls are a way of updating sensed variables in a controller task, and controlled variables in the environment task. The entry is implemented in the environment, and called from the controller task. In the Ada translator we generate entries in the environment task specification, and these are implemented by **accept** clauses in the task body. We may find, in future work, that this approach has a role to play when implementing the interaction between tasks and external driver APIs. The second style, addressed variables, associates address information with each of the parameters of a controller task's sensing or actuating events. In our case study, a parameter with address and base, is translated to a variable which is declared using Ada representation address clauses. Other languages, like C, would reference the address using a pointer. When the variable is used in an expression, the specified memory address is read from, or written to.

The case study of Section 5 was proposed in the course our Deploy project [41]. The models arising from this activity are available at [20], and form part of a tutorial available at [21]. It is here that one can find a listing of all the generated code. The contribution made in this paper is a general solution, to a particular problem highlighted by the initial work on the case study. Namely, that the restriction on inter-task communication forced us to adopt a particular modelling style, and this did not reflect the

way that systems are implemented in industry. We removed the restriction on inter-task communication, making the the environment machine a special case. We introduced new event annotations, sensing and actuating; and a new parameter annotation, addressed variables. These new features use the existing synchronization approach, arising from model decomposition, to model interaction with the environment. The annotations guide the translator to produce the appropriate code.

The case study was used as to validate the approach. We have shown that it is possible to use Tasking Event-B to model a system, and its environment, in such a way that leads to generated code of use to industry. The annotations facilitate automatic generation of deployable controller code, and a simulation of the environment. The partitioning of the tasks in the controller and the environment, using Event-B decomposition, is quite intuitive. The semantics of event synchronization allow us to model the interaction between the decomposed tasks of the controller and environment.

There are a number of avenues that may be explored in future work. For instance multiple Environ machines could be used; that is one machine for each device in the environment. The task entries may provide the basis for a link with device driver APIs. It would also be useful to explore interaction with a modelling tool such as Simulink.

10.2 Related Work

The closest comparable work is that of Classical-B's code generation approach [3] using B0 [13]. B0 consists of concrete programming constructs, these map to programming constructs in target programming languages. B0 can be translated to Ada, but there is no support for concurrency. The Event-B and Classical-B approaches differ in many respects. For example, the Event-B approach supports modelling of the controller, and environment together; Classical-B is aimed more at modelling software systems in a modular fashion, and the generated code is sequential, and does not consider concurrency. In the work presented here, we have added the ability to provide an Ada implementation of the deployable code, and an environment simulation. Code generation of B to embedded systems was carried out in [7], where the implementation results in sequential code. Some consideration is given, in [38], to the use of an Event-B-like syntax for analysis of multi-tasking programs. In comparison, we use the task body for scheduling, rather than taking a purely interrupt driven approach; we have yet to incorporate modelling of interrupts in Tasking Event-B.

VDM++ [14] may be used to generate code, it is an object-oriented extension to VDM-SL formal specification language. It has been used to model real-time systems, see [42]. The paper describes a controller and environment model similar to our own. They go on to define an abstract operational semantics, which is quite general, and has additional features when compared to our work. They model time, and asynchronous communication. We do not address these issues in our work since the research is ongoing.

Scade [6] is an industrial tool for formally modelling embedded systems. It provides a graphical approach to specification, and has a certified code generator. It has a similar control flow approach to that of UML-B statemachines [36]. However, we have not investigated code generation from UML-B to Ada.

In [22] Karsai et al. describe some common elements of modelling approaches for embedded software development. The work that we present here is comparable in some ways, since we can view our approach as an example of the Model-Integrated Computing [32] that they describe. Their approach suggests development is based on models and generation, which ours is; and that the approaches have multiple views which ours has (for example UML-B [36]). They also advocate an extensible approach: the Rodin tool has a built-in extension mechanism, and our tools are based on a extensible meta-models. The approach differs in that they focus on hardware architecture and signal-flow aspects of the design, and use a Finite State Automata [26] approach to formalize the dynamic aspects of the system. Event-B

is a state-based approach, but does have analysis such as ProB [30] to address dynamic aspects.

References

[1] SPARKAda. Available at http://www.praxis-his.com/sparkada/index.asp.

[2] J. R. Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 2010.

[3] J.R. Abrial. *The B Book - Assigning Programs to Meanings*. Cambridge University Press, 1996.

[4] J.R. Abrial and S. Hallerstede. Refinement, Decomposition, and Instantiation of Discrete Models: Application to Event-B. *Fundam. Inform.*, 77(1-2):1–28, 2007.

[5] J. Barnes. *Programming in Ada 2005*. International Computer Science Series. Addison Wesley, 2006.

[6] G. Berry. Synchronous Design and Verification of Critical Embedded Systems Using SCADE and Esterel. In Stefan Leue and Pedro Merino, editors, *FMICS*, volume 4916 of *Lecture Notes in Computer Science*, page 2. Springer, 2007.

[7] D. Bert, S. Boulmé, M. Potet, A. Requet, and L. Voisin. Adaptable Translator of B Specifications to Embedded C Programs. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, *FME*, volume 2805 of *Lecture Notes in Computer Science*, pages 94–113. Springer, 2003.

[8] Egon Börger, Michael J. Butler, Jonathan P. Bowen, and Paul Boca, editors. *Abstract State Machines, B and Z, First International Conference, ABZ 2008, London, UK, September 16-18, 2008. Proceedings*, volume 5238 of *Lecture Notes in Computer Science*. Springer, 2008.

[9] A. Burns. The Ravenscar Profile. *Ada Lett.*, XIX:49–52, December 1999.

[10] A. Burns, B. Dobbing, and T. Vardanega. Guide for the use of the Ada Ravenscar Profile in high integrity systems. *Ada Lett.*, XXIV:1–74, June 2004.

[11] M. Butler. Synchronisation-based Decomposition for Event-B. In: RODIN Deliv- erable D19 Intermediate report on methodology. Technical report, University of Southampton, 2006.

[12] M. Butler. Decomposition Structures for Event-B. In *Integrated Formal Methods iFM2009, Springer, LNCS 5423*, volume LNCS. Springer, February 2009.

[13] ClearSy System Engineering. *The B Language Reference Manual*, version 4.6 edition.

[14] CSK Systems Corporation. The VDM++ Language Manual.

[15] E.W. Dijkstra. Guarded Commands, Non-determinacy and Formal Derivation of Programs. *Commun. ACM*, 18(8):453–457, 1975.

[16] A. Edmunds. *Providing Concurrent Implementations for Event-B Developments*. PhD thesis, University of Southampton, March 2010.

[17] A. Edmunds and M. Butler. Linking Event-B and Concurrent Object-Oriented Programs. In *Refine 2008 - International Refinement Workshop*, May 2008.

[18] A. Edmunds and M. Butler. Tool Support for Event-B Code Generation, 2010.

[19] A. Edmunds and M. Butler. Tasking Event-B: An Extension to Event-B for Generating Concurrent Code. In *PLACES 2011*, February 2011.

[20] A. Edmunds and A. Rezazedah. Event-B Project Archives: Tasking Event-B Tutorial. University of Southampton. at http://deploy-eprints.ecs.soton.ac.uk/304/.

[21] A. Edmunds and A. Rezazedah. Event-B Wiki: Development of a Heating Controller System. at http://wiki.event-b.org/index.php/Development_of_a_Heating_Controller_System.

[22] G.Karsai, J. Sztipanovits, Á. Lédeczi, and T. Bapty. Model-integrated Development of Embedded Software. *Proceedings of the IEEE*, 91(1):145–164, 2003.

[23] J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java Language Specification - Third Edition*. Addison-Wesley, 2004.

[24] S. Hallerstede. Justifications for the Event-B Modelling Notation. In Julliand and Kouchnarenko [28], pages 49–63.

[25] P.B. Hansen. Structured multiprogramming. *Commun. ACM*, 15:574–578, July 1972.

[26] T.A. Henzinger. The Theory of Hybrid Automata. In *LICS*, pages 278–292, 1996.

[27] C. A. R. Hoare. Monitors: An Operating System Structuring Concept. *Commun. ACM*, 17(10):549–557, 1974.

[28] J. Julliand and O. Kouchnarenko, editors. *B 2007: Formal Specification and Development in B, 7th International Conference of B Users, Besançon, France, January 17-19, 2007, Proceedings*, volume 4355 of *Lecture Notes in Computer Science*. Springer, 2006.

[29] B.W. Kernighan and D. Ritchie. *The C Programming Language, Second Edition*. Prentice-Hall, 1988.

[30] M. Leuschel and M. Butler. ProB: A Model Checker for B. In *Proceedings of Formal Methods Europe 2003*, 2003.

[31] Christophe Metayer and Mathieu Clabaut. Dir 41 case study. In Börger et al. [8], page 357.

[32] S. Neema and G. Karsai. Software for Automotive Systems: Model-Integrated Computing. In Manfred Broy, Ingolf H. Krüger, and Michael Meisinger, editors, *ASWSD*, volume 4147 of *Lecture Notes in Computer Science*, pages 116–136. Springer, 2004.

[33] RODIN Project. at http://rodin.cs.ncl.ac.uk.

[34] A.G. Russo. Formal Methods in Industry: The State of Practice of Formal Methods in South America and Far East, September 2009.

[35] R. Silva, C. Pascal, T.S. Hoang, and M. Butler. Decomposition Tool for Event-B. *Software: Practice and Experience*, 2010.

[36] C.F. Snook and M.J. Butler. UML-B: A Plug-in for the Event-B Tool Set. In Börger et al. [8], page 344.

[37] J. M. Spivey. *The Z notation: A Reference Manual*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[38] Bill Stoddart, Dominique Cansell, and Frank Zeyda. Modelling and proof analysis of interrupt driven scheduling. In Julliand and Kouchnarenko [28], pages 155–170.

[39] T.S. Taft, R.A. Tucker, R.L. Brukardt, and E. Ploedereder, editors. *Consolidated Ada reference manual: language and standard libraries*. Springer-Verlag New York, Inc., New York, NY, USA, 2002.

[40] The Deploy Code Generation Wiki. at http://wiki.event-b.org/index.php/Code_Generation_Activity.

[41] The DEPLOY Project Team. Project Website. at http://www.deploy-project.eu/.

[42] M. Verhoef, P. Gorm Larsen, and J. Hooman. Modeling and Validating Distributed Embedded Real-Time Systems with VDM++. In J. Misra, T. Nipkow, and E. Sekerinski, editors, *FM*, volume 4085 of *Lecture Notes in Computer Science*, pages 147–162. Springer, 2006.