Formal Modelling for Ada Implementations: Tasking Event-B

A. Edmunds[1], A. Rezazadeh and M.J. Butler

[1]`ae2@ecs.soton.ac.uk`

Department of Electronics and Computer Science
University of Southampton

Abstract. This paper describes a formal modelling approach, where Ada code is automatically generated from the modelling artefacts. We introduce an implementation-level specification, Tasking Event-B, which is an extension to Event-B. Event-B is a formal method, that can be used to model safety-, and business-critical systems. The work may be of interest to a section of the Ada community who are interested in applying formal modelling techniques in their development process, and automatically generating Ada code from the model. We describe a streamlined process, where the abstract modelling artefacts map easily to Ada language constructs. Initial modelling takes place at a high level of abstraction. We then use refinement, decomposition, and finally implementation-level annotations, to generate Ada code. We provide a brief introduction to Event-B, before illustrating the new approach using small examples taken from a larger case study.

1 Introduction

Event-B [1] 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 [2,3]. 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. We illustrate the approach using examples from a case study of an embedded Heater Controller, and we use Ada 1995 [4] as the target language. To be able to link Event-B artefacts to programming constructs we have devised an extension to Event-B called Tasking Event-B. Tasking Event-B concepts are directly influenced by Ada constructs. For instance, Ada tasks are modelled by AutoTask machines, and protected objects are modelled by shared machines, in Tasking Event-B.

We continue with section 1.1 in which we discuss our motivation. Section 2 provides a brief introduction to the Event-B approach. Section 3 provides an overview of the Tasking Event-B extension. In Section 4 we present more details of Tasking features and the translation to Ada. Section 5 describes how we can read/write directly to memory. Section 6 provides an overview of tooling issues, and Section 7 provides a summary and discussion.

1.1 Motivation

The Event-B method, and supporting tools [5], have been developed during the the EU DEPLOY [6] project. A number of the industrial partners, associated with the project, have been interested in the formal development of multi-tasking, embedded control systems. However, automatic generation from Event-B models, for these type of systems, was absent from the approach. We chose Ada as a basis for our approach, not only because of it's suitability for the application domain, but it also serves as a useful reference for our code generation constructs. Ada constructs match well with Event-B modelling elements, and this serves to simplify the translation to code. We do not, however, formally model all aspects of the implementation, e.g. time. We model the behaviour that relates to the control flow specified in the task bodies; for which we provide Event-B semantics. We developed a case study [7] of a Heating Controller to validate the code generation approach. The case study is an analogue of many embedded systems, where inputs from the environment are received and processed, and may have some effect in the environment caused by its outputs.

2 An Overview of Event-B

The Event-B method [1] was developed by J.R. Abrial, and uses set-theory, predicate logic and refinement to model discrete systems. The basic structural elements of Event-B models are contexts and machines. Contexts are used to describe the static aspects of a system, using sets and constants; the relationships between them are specified in the axioms clause. Machines are able to *see* Contexts; the content of a Context is visible and accessible to a machine. Machines are used to describe the dynamic aspects of a system, in the form of state variables, and guarded events, which update state. Safety properties are specified using the invariants clause. The invariants give rise to proof obligations, which are generated automatically by the tool; a large number of the proof obligations may be discharged without user intervention by auto-provers. Where auto-provers fail to discharge proof obligations, the user guides the interactive prover. They proceed by suggesting strategies, and sub-goals in the form of hypotheses, in the endeavour to complete the proof. Refinement is used to show that concrete models satisfy the safety properties of their abstract counterparts.

A fragment of an Event-B specification is shown in Fig. 1. The specification has variables, which are typed in the **invariant**. Invariants also describe desired safety properties. The event declares two parameters *tm1* and *tm2*. These are typed in the guard clause, following the **where** keyword. The third guard describes an enabling condition for the event. When the value of $avt < cttm2$ the event is enabled, and the updates described in the actions may take place. Actions may contain deterministic or non-deterministic assignments, or do-nothing (skip); but non-deterministic modelling constructs are removed by the time we reach the implementation level specification. In Fig. 1 the action assigns TRUE to the variable *hsc*.

```
machine HCtrl_M0
sees HC_CONTEXT
variables avt stm1 hsc cttm2...
invariants
  avt ∈ ℤ
  stm1 ∈ ℤ
    ...
```

```
event TurnON_Heat_Source
any tm1 tm2
where
  tm1 ∈ ℤ
  tm2 ∈ ℤ
  avt < cttm2
then
  hsc := TRUE
end
```

Fig. 1. Example of Textual Event-B

3 An Overview of Tasking Event-B

Tasking Event-B is an extension to Event-B, but includes some restrictions to ensure the code is implementable. An Event-B operational semantics underpins the extension. As a means of verifying consistency between tasking Event-B and higher-level abstractions, the Tasking Event-B can be translated to a standard Event-B representation. Then using the Rodin tool we can show that this generated model refines the abstract development.

During the development, before the tasking Event-B stage, we use model decomposition to tackle complexity. The Rodin tool supports different approaches to decomposition; here we use shared event decomposition [8,9]. In section 4.1 we provide a more detailed picture of how an abstract model is decomposed into its sub-models. This decomposition approach results in a partitioning of the system whereby variables are distributed over decomposed machines. A machine has access to variables of another machine using pairs of synchronized events. Synchronized events allow machines to communicate using parameters; they model atomic access to variables residing in another machine. This synchronization approach is described in more detail in [10]. In order to keep track of the synchronizations, the Rodin tool produces a composition component [11] during the decomposition process.

In our approach controllers can be comprised of a number of tasks and, rather than allowing direct communication between controller tasks, we use a shared machine to encapsulate the shared data. This means that synchronizations are taking place between tasking machines and the shared machine. This structure is illustrated in Fig. 2, which describes the relationships between the components of an Event-B development, tasking Event-B and the generated Ada code. In the Tasking Event-B layer, machines are identified as AutoTask, Environ, or Shared. AutoTask Machines model *controller* tasks in the implementation level, and are implemented using Ada tasks. Shared Machines model encapsulated shared objects, and are implemented by protected objects. Environ Machines model the environment, and are implemented using Ada tasks.

An example of an AutoTask Machine, from our case study [7,12], is shown in Fig. 3. It is a descendant of the fragment shown in Fig. 1, following a number of refinement and decomposition steps. The machine of Fig. 3 is an implementation

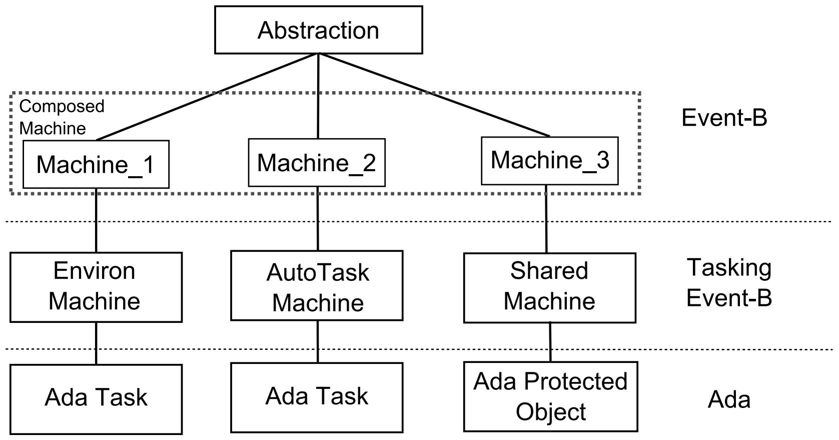


Fig. 2. Heating Controller Artefacts

level refinement, as indicated by the *autotask* annotation. As a convention, we prefix *event* names of the *environ machine* with **EN**, the *shared machine* with **SO**, and *temperature controller* with **TC**. We specify some tasking features such as the task type (e.g. periodic, triggered, one-shot, and repeating); the priority, and the task body. A main feature of the tasking level specification is the task body; this is used to specify flow control aspects of the task, with respect to the events that already reside in the machine. The task body may contain clauses, such as sequence, loop, and branch; and uses a programming-style syntax, e.g. **;**, **do**, **if** and event names. Notice that there is no explicit use of a synchronization construct in the task body, we only refer to events that are local to the tasking machine. The code generator tool uses the composition component, mentioned previously, to find any synchronizations (if they exist) and then generates the implementations. Synchronizations between AutoTasks and the Shared machine are implemented using protected procedures. Synchronizations between AutoTask and Environ machines are implemented as rendezvous, or direct memory access as required. A machine's events can model local (wrt the machine) state updates, subprograms, or branching and looping constructs. As indicated earlier, protected procedure calls are modelled using synchronized events in the task body; this is used when two machines communicate. If an event just updates local state, then updates are mapped to assignment clauses in the target, rather than incurring the overhead of a subprogram call. An *Output* construct is provided to allow text output to a console during simulation.

In the final, deployable system, inter-task communication can be 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 [13]. We may relax this restriction, for environment tasks, to simulate the environment.

```
machine Temp_Ctrl_TaskImpl
is autoTask
refines Temp_Ctrl_Task
variables avt, cttm2, hsc, . . .

tasktype periodic(250)
priority 5
taskbody is
  . . .
  TCGet_Target_Temperature2;
   - - ||e SOGet_Target_Temperature2
  if TCTurnON_Heat_Source
  else TCTurnOFF_Heat_Source;
  . . .
```

```
event TCTurnOn_Heat_Source
refines TurnOn_Heat_Source
when
  avt < cttm2
then
  hsc := TRUE
end

event TCGet_Target_Temperature2
refines Get_Target_Temperature2
any tm
where tm ∈ ℤ
then cttm2 := tm
end
```

Fig. 3. An Fragment of an AutoTask Machine

```
event SOGet_Target_Temperature2
refines Get_Target_Temperature2
any tm
where tm ∈ ℤ
      tm = cttm
then skip
end
```

Fig. 4. A Synchronizing Event in the Shared Object

4 Case Study

This section makes use of model and code fragments from a case study [7] to illustrate the translation from Tasking Event-B to Ada. We begin with some background to the case study, introducing the variables of the model. We will look at just one controller task, the temperature controller, which polls two temperature sensor values $ts1$ and $ts2$, in the environment. Their average value avt is calculated and displayed. If the average temperature is lower than the target temperature $cttm2$, the controller will turn on the heater source using Heat Source Switch $hsc := TRUE$, otherwise this switch will be turned off by the controller, $hsc := FALSE$. The status of the heater itself is monitored, and has an over-temperature ota alarm.

The development process starts with an abstract specification, followed by two successive refinements. We then decompose the model into two parts, 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.

4.1 Event-B Development

At the top we show the most abstract model of the system where we specify the system's main functionality, such as modelling the increase/decrease of the target temperature, polling of the temperature sensors, calculation of the average temperature, and activation of the heat source and alarms. In the first refinement we introduce sensing and actuation. Sensing events model 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. We also model actuation occurring as a result of controller decisions, such as turning the heat on/off, and activating the various alarms. We decompose our model in two stages; we first separate the controller subsystem, the part of the system that should be implemented, from its surrounding environment. In the second stage we decompose the controller subsystem; we identify three controller tasks, and a protected object. The structure of the decomposition is visible in the diagram in Fig. 5. Following decomposition we add an additional refinement, the Tasking Event-B Layer. This refinement layer is used for our implementation level specification. It is necessary to use refinement here, since the automatically generated files (from the decomposition tool) cannot be modified.

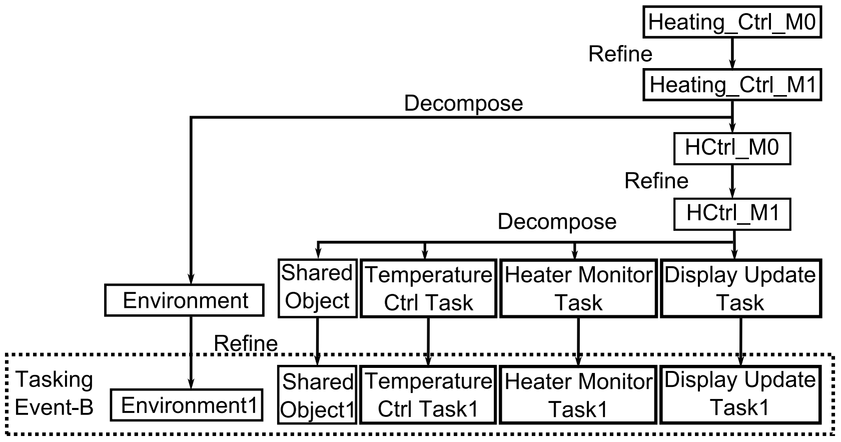


Fig. 5. The Development Approach

4.2 Guiding Code Generation with Tasking Event-B

In the previous step we decomposed into five machines; one modelling the environment, one modelling each of the three controller tasks, and one modelling the protected object. We should now add the Tasking Event-B annotations to guide code generation. The first step is to use annotations to identify the machines as being an AutoTask, Environ or Shared machine. With Environ, or

AutoTask machines, we also 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 programming constructs, such as sequence, branch and subprogram calls. The generated code is viewed as an implementation of a schedule of events.

In the discussion that follows, we use the temperature control task event *Temp_Ctrl_TaskImpl* from Fig. 3, as an example. We describe how we use Tasking Event-B to specify implementation details, that is, how the controller interacts with the environment. The full task body of the *Temp_Ctrl_TaskImpl* AutoTask is shown in Fig. 6, it includes a brief description of the activities performed. In (1) the temperature controller uses the TCSense_Temperatures event; in (2) the average temperature is calculated, and so on. The comment identifies the synchronizing event, which is presented here for clarity. The task body gives rise to the Ada code show in Fig. 7. We will look at the translated code in more detail, later in the section.

```
TCSense_Temperatures;                      - -(1)(||e ENSense_Temperatures)
TCCalculate_Average_Temperature;           - -(2)
TCDisplay_Current_Temperature;             - -(3)(||e ENDisplay_Current_Temperature)
TCGet_Target_Temperature2;                 - -(4)(||e SOGet_Target_Temperature2)
if TCTurnON_Heat_Source end                - -(5)
  else TCTurnOFF_Heat_Source end;
TCSet_Heat_Source_State;                   - -(6)(||e SOSet_Heat_Source_State)
TCActuate_Heat_Source;                     - -(7)(||e ENActuate_Heat_Source)
if TCSwitchOn_OverHeat_Alarm end           - -(8)
  else TCSwitchOff_OverHeat_Alarm end;
TCActuate_OverHeat_Alarm;                  - -(9)(||e ENActuate_OverHeat_Alarm)

    - -(1) poll the ts1 and ts2 temperature sensors.
    - -(2) calculate the average temperature.
    - -(3) update ctd, the displayed temperature.
    - -(4) get the target temperature from the protected object.
    - -(5) branching choice: set the heater on or off flag in the task.
    - -(6) set the heat source active flag in the protected object.
    - -(7) update ahsa, the activate heat source flag.
    - -(8) a branching choice: set activate overheat alarm flag in the task.
    - -(9) update aota, the activate overheat alarm flag.
```

Fig. 6. The Temp_Ctrl_TaskImpl Task Body

The development proceeds by adding annotations to events. In Fig. 8 we see the *sensing* annotation being used to indicate that an event is used in a sensing role. The *sensing* keyword is used with both 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. Now, returning to the translated

```
task body Temp_Ctrl_TaskImpl is
  . . .
  procedure TCCalculate_Average_Temperature is
  begin
    avt := ((stm1 + stm2) / 2);
  end;
  begin
  . . .
  Envir1Impl.ENSense_Temperatures(stm1, stm2);                    -- (1)
  TCCalculate_Average_Temperature;                                -- (2)
  Envir1Impl.ENDisplay_Current_Temperature(avt);                  -- (3)
  shared_object1implInst.SOGet_Target_Temperature2(cttm2);        -- (4)
  if(avt < cttm2) then                                            -- (5)
    hsc := TRUE;
  else
    hsc := FALSE;
  end if;
  shared_object1implInst.SOSet_Heat_Source_State(hsc);            -- (6)
  Envir1Impl.ENActuate_Heat_Source(hsc);                          -- (7)
  if(avt > Max) then                                              -- (8)
    ota := TRUE;
  else
    ota := FALSE;
  end if;
  Envir1Impl.ENActuate_OverHeat_Alarm(ota);                       -- (9)
  . . .
end Temp_Ctrl_TaskImpl;
```

Fig. 7. Implementation of Temp_Ctrl_TaskImpl Task Body

```
event TCSense_Temperatures is sensing
refines TCSense_Temperatures
any t1 t2
when
  t1 ∈ ℤ
  t2 ∈ ℤ
then
  stm1 := t1
  stm2 := t2
end
```

```
event ENSense_Temperatures is sensing
refines ENSense_Temperatures
any t1 t2
when
  t1 ∈ ℤ
  t2 ∈ ℤ
  t1 = ts1
  t2 = ts2
then
  skip
end
```

Fig. 8. Synchronization of a Sensing Event

code, arising from clause (1) of Fig. 6. It results in the following Ada program statement:

```
Envir1Impl.ENSense_Temperatures(stm1, stm2);
```

Envir1Impl is the name of the environment task, and *ENSense_Temperatures* is the name of the task entry. The entry call implements a pair of synchronized events. In the most abstract model (not shown in this paper) $stm1$ keeps track of the sensed temperature, $ts1$, using an assignment $stm1 := ts1$. In the decomposition, the two temperature sensing events synchronize to achieve the same result; this is implemented as an entry call. We now describe the relationship between the implementation and the model. The variable $stm1$, appears in the action of the $TCSense\_Temperatures$ event, see Fig: 8. In the translation, the event parameter, $t1$ is replaced by the variable, $stm1$, and passed as an actual parameter in the entry call. The entry is implemented as an Ada *accept* statement in the $Envir1Impl$ task, see Fig. 9. The monitored variable $ts1$ appears in the guard of the $ENSense\_Temperatures$ event, of Fig: 8, and translates to an **out** parameter in the entry signature. Note that in this case the event guard is translated to an assignment in the implementation. When returning from the entry call, the value held by the *out* parameter is assigned to the actual parameter; that is, $stm1 := t1$, in our example. Since $t1 = ts1$ we have $stm1 := ts1$, as required.

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

Fig. 9. Implementation of ENSense_Temperatures

The translation of events of shared machines is similar, except that we implement the machines as protected objects with procedures. The translation of synchronized events is otherwise the same, with respect to the mapping of event parameters to subroutine parameters.

5 Writing Directly to Memory Locations

So far we have described an approach which facilitates interation with the environment using rendezvous. However, we also provide an alternative approach, where the developer specifies some memory locations to read from, and write to. We provide a feature which allows developers to annotate event parameters with address information; using the *addr* keyword. Use of the *addr* address keyword is shown in Fig. 10. We specify a memory location, and its number base. In the example, $t1$ is given the address $ef14$ in base 16. We can see, on the right of the figure, 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. In the $TCSense\_Temperatures$ procedure implementation of Fig. 10, 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 $stm1$. This approach does differ from the entry approach described in the previous section. Entry calls are atomic, whereas we are using non-atomic statements. For this reason the environment must be responsible for ensuring that the implementation of sensing events with multiple read actions, and actuating events with multiple write actions, are performed atomically (we do not envisage mixing sensing and actuating in a single event).

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

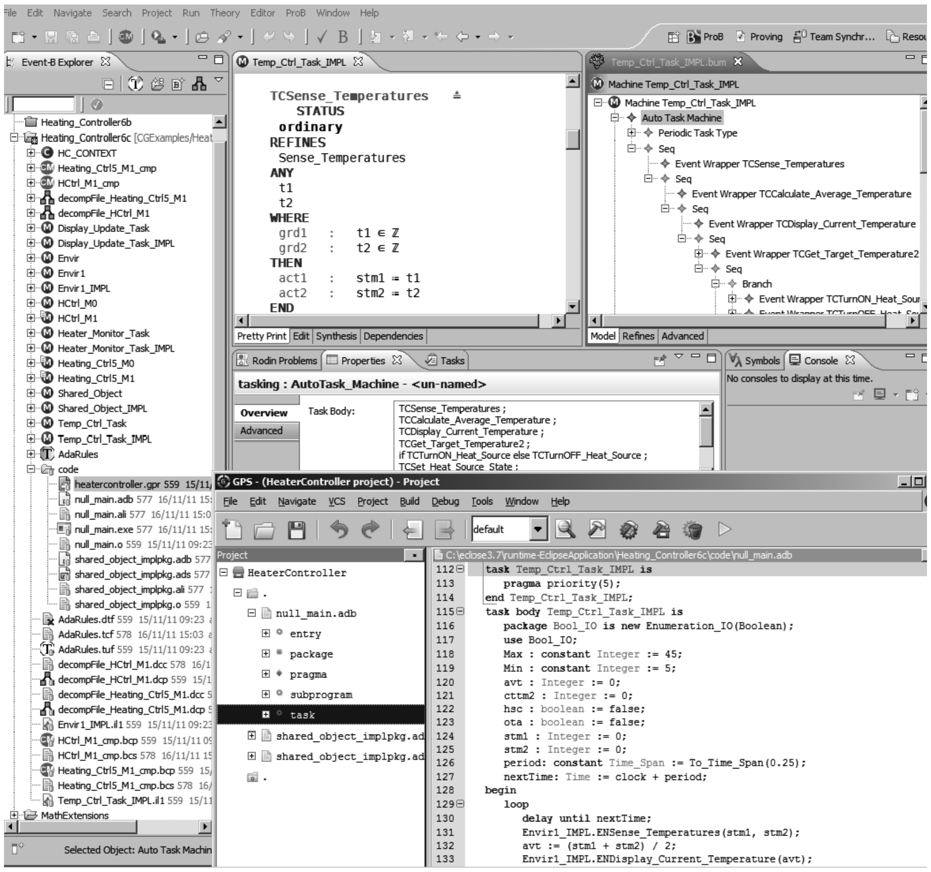
```
task body Temp_Ctrl_TaskImpl is
  stm1 : Integer := 0;
  stm2 : Integer := 0;
  ...
  procedure TCSense_Temperatures is
    t1 : Integer;
    for t1'Address
      use System'To_Address(16#ef14#);
    pragma Atomic(t1);
    t2 : Integer;
    for t2'Address
      use System'To_Address(16#ef18#);
 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

Fig. 10. Addressed Variables: Specification and Implementation

Using a combination of the approaches described in this paper, we can simulate interaction with the environment in the early stages of development, using entry calls.. Later in the development we can choose to read from, and write to, memory directly. To do this we simply add the address information to the relevant variables. We also have the option of the environment simulation reading from, and writing to memory.

6 Tooling

The Rodin tool [5], based on the Eclipse Platform [14], is a complete development environment for Event-B. We have extended the methodology and tools to add implementation level specification, using Tasking Event-B. Tasking Event-B and the code generators are fully integrated into the Rodin toolset, see Fig 11 . When a development is ready for translation to code we have a simple pop-up



Temp_Ctrl_Task_IMPL
Machine Temp_Ctrl_Task_IMPL
Auto Task Machine
Periodic Task Type
Seq
Event Wrapper TCSense_Temperatures
Event Wrapper TCCalculate_Average_Temperature
Event Wrapper TCDisplay_Current_Temperature
Event Wrapper TCGet_Target_Temperature2
Branch
Event Wrapper TCTurnON_Heat_Sour
Pretty Print
Edit
Synthesis
Dependencies
Model
Refines
Advanced
Rodin Problems
Properties
Tasks
No consoles to display at this time.
GPS - (HeaterController project) - Project
HeaterController
null_main.adb
entry
package
pragma
subprogram
task
Selected Object: Auto Task Machin

```
TCSense_Temperatures
   STATUS
 ordinary
REFINES
 Sense_Temperatures
ANY
 t1
 t2
WHERE
 grd1   :   t1 ∈ ℤ
 grd2   :   t2 ∈ ℤ
THEN
 act1   :   stm1 ≔ t1
 act2   :   stm2 ≔ t2
END
```

tasking : AutoTask_Machine - <un-named>

Task Body:

```
TCSense_Temperatures ;
TCCalculate_Average_Temperature ;
TCDisplay_Current_Temperature ;
TCGet_Target_Temperature2 ;
if TCTurnON_Heat_Source else TCTurnOFF_Heat_Source ;
TCSet_Heat_Source_State ;
```

C:\eclipse3.7\runtime-EclipseApplication\Heating_Controller6c\code\null_main.adb

```
task Temp_Ctrl_Task_IMPL is
   pragma priority(5);
end Temp_Ctrl_Task_IMPL;
task body Temp_Ctrl_Task_IMPL is
   package Bool_IO is new Enumeration_IO(Boolean);
   use Bool_IO;
   Max : constant Integer := 45;
   Min : constant Integer := 5;
   avt : Integer := 0;
   cttm2 : Integer := 0;
   hsc : boolean := false;
   ota : boolean := false;
   stm1 : Integer := 0;
   stm2 : Integer := 0;
   period: constant Time_Span := To_Time_Span(0.25);
   nextTime: Time := clock + period;
begin
   loop
      delay until nextTime;
      Envir1_IMPL.ENSense_Temperatures(stm1, stm2);
      avt := (stm1 + stm2) / 2;
      Envir1_IMPL.ENDisplay_Current_Temperature(avt);
```

Fig. 11. Code Generation Tools

menu with translation options. The code generators use the Tasking Event-B model, and a two-step process, see Fig. 12. The first step generates a Common

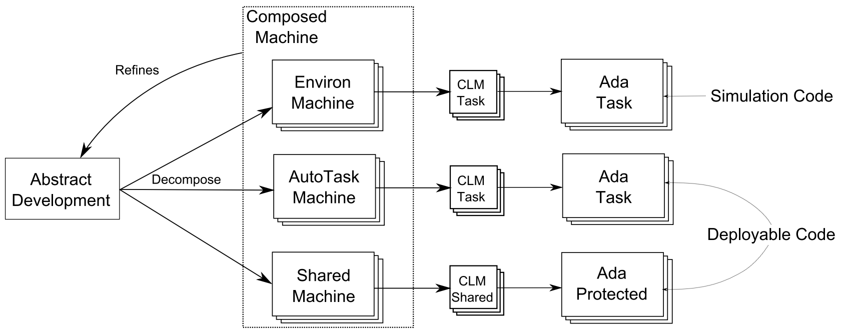


Fig. 12. Two-Step Code Generation

Language Model (CLM); the CLM is an abstraction of commonly used software constructs. The abstract tasks and shared objects of the CLM are then used in the translation to Ada. The Ada translator generates the main procedure file, and specification and body files, in a directory ready for compilation. We have been successfully compiling and executing the generated code, using the GPS tool from AdaCore [15]. The only additional effort has been the creation of the project file; this may also be automated in the future.

7 Conclusions

In this paper we have described our methodology and tools for linking Event-B, through the use of the Tasking Event-B extension, to Ada code. We relate the Event-B modelling artefacts to their Ada counterparts; and, using the case study, we explain the relationship between the modelling abstraction and implementation in more detail. We have explained how Event-B is augmented with Tasking Event-B annotations, these are used to guide the code generator to produce code. For example, annotations identify the role of the machines in the implementation; a machine may be an AutoTask machine, Environ machine or Shared machine. AutoTask and Environ machines have a task body in which we are able to specify flow of control. This is done through the use of the sequence, branch and loop constructs. We make use of the tool-driven decomposition approach, to structure the development. This allows us to partition the system in a modular fashion, reflecting Ada implementation constructs. Decomposition is also the mechanism for breaking up complex systems to make modelling and proof more tractable. As part of the specification we indicate which of the events take part in sensing and actuating roles; we describe the relationship between event parameters, and their role in the implementation of sensing and actuating events. We extend the sensing and actuating features to allow specification of direct reads from, and writes to memory.

In a wider context, the work we have undertaken is to improve the approach for modelling of, and providing implementations for, multi-tasking embedded control systems. In this sense the case study can be seen as representative of the style of interactions, using sensing and actuation, in a domain where controllers are continuously monitoring and reacting to the environment. This work provides a basis for future developments in our sphere of interest, and will continue in the Advance project [16], and others. The Tasking Event-B control flow language has been given Event-B semantics, although we do not formalize every aspect of Tasking Event-B, such as modelling timing, or priority. With regard to modelling time, several projects are under way, investigating timing related issues [17]. We can use the Tasking Event-B model to generate an Event-B model of the implementation, using the Event-B semantics. We can show that this model refines the abstract development, thus showing that the properties of the abstract development are satisfied.

7.1 Related Work

The closest comparable work is that of Classical-B's code generation approach [18] using B0 [19]. 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. Code generation of B to embedded systems was carried out in [20], where the implementation results in sequential code. Some consideration is given, in [21], to the use of an Event-B-like syntax for analysis of multi-tasking programs. By 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++ [22] 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 [23]. The paper describes a controller and environment model similar to our own. They define an abstract operational semantics to describe additional modelling features, whereas we use Event-B semantics. They model time, and asynchronous communication, whereas we do not address these issues in the work presented here. However, the specification of timing properties is of great interest to us; and work has been done to address the issue in Event-B such as [24], or more recently [17]. Scade [25] 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 [26].

References

1. Abrial, J.R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press (2010)
2. Russo, A.: Formal Methods in Industry: The State of Practice of Formal Methods in South America and Far East (2009)
3. Metayer, C., Clabaut, M.: Dir 41 case study. [27] 357
4. Taft, T., Tucker, R., Brukardt, R., Ploedereder, E., eds.: Consolidated Ada reference manual: language and standard libraries. Springer-Verlag New York, Inc., New York, NY, USA (2002)

5. RODIN Project. (at http://rodin.cs.ncl.ac.uk)
6. The DEPLOY Project Team: Project Website. (at http://www.deploy-project.eu/)
7. Edmunds, A., Rezazedah, A.: Event-B Wiki: Development of a Heating Controller System. (at http://wiki.event-b.org/index.php/Development_of_a_Heating_Controller_System)
8. Butler, M.: Decomposition Structures for Event-B. In: Integrated Formal Methods iFM2009, Springer, LNCS 5423. Volume LNCS., Springer (2009)
9. Silva, R., Pascal, C., Hoang, T., Butler, M.: Decomposition Tool for Event-B. Software: Practice and Experience (2010)
10. Edmunds, A., Butler, M.: Tasking Event-B: An Extension to Event-B for Generating Concurrent Code. In: PLACES 2011. (2011)
11. Silva, R.: Towards the Composition of Specifications in Event-B. In: B 2011. (2011)
12. Edmunds, A., Rezazedah, A.: Event-B Project Archives: Tasking Event-B Tutorial. University of Southampton. (at http://deploy-eprints.ecs.soton.ac.uk/304/)
13. Burns, A., Dobbing, B., Vardanega, T.: Guide for the use of the Ada Ravenscar Profile in high integrity systems. Ada Lett. **XXIV** (2004) 1–74
14. The Eclipse Project: Eclipse - an Open Development Platform. (Available at http://www.eclipse.org/)
15. AdaCore: GNAT Programming Studio. (Available at http://www.adacore.com/home/)
16. The Advance Project Team: The Advance Project. (Available at http://www.advance-ict.eu)
17. Sarshogh, M., Butler, M.: Specification and Refinement of Discrete Timing Properties in Event-B. In: AVoCS 2011. (2011)
18. Abrial, J.: The B Book - Assigning Programs to Meanings. Cambridge University Press (1996)
19. ClearSy System Engineering: The B Language Reference Manual. (Version 4.6 edn.)
20. Bert, D., Boulmé, S., Potet, M., Requet, A., Voisin, L.: Adaptable Translator of B Specifications to Embedded C Programs. In Araki, K., Gnesi, S., Mandrioli, D., eds.: FME. Volume 2805 of Lecture Notes in Computer Science., Springer (2003) 94–113
21. Stoddart, W., Cansell, D., Zeyda, F.: Modelling and Proof Analysis of Interrupt Driven Scheduling. In Julliand, J., Kouchnarenko, O., eds.: B. Volume 4355 of Lecture Notes in Computer Science., Springer (2007) 155–170
22. CSK Systems Corporation: (The VDM++ Language Manual)
23. Verhoef, M., Larsen, P.G., Hooman, J.: Modeling and Validating Distributed Embedded Real-Time Systems with VDM++. In Misra, J., Nipkow, T., Sekerinski, E., eds.: FM. Volume 4085 of Lecture Notes in Computer Science., Springer (2006) 147–162
24. Degerlund, F., Grnblom, R., Sere, K.: Code Generation and Scheduling of Event-B Models (2011)
25. Berry, G.: Synchronous Design and Verification of Critical Embedded Systems Using SCADE and Esterel. In Leue, S., Merino, P., eds.: FMICS. Volume 4916 of Lecture Notes in Computer Science., Springer (2007) 2
26. Snook, C., Butler, M.: UML-B: A Plug-in for the Event-B Tool Set. [27] 344
27. Börger, E., Butler, M.J., Bowen, J.P., Boca, P., eds.: Abstract State Machines, B and Z, First International Conference, ABZ 2008, London, UK, September 16-18, 2008. Proceedings. In Börger, E., Butler, M.J., Bowen, J.P., Boca, P., eds.: ABZ. Volume 5238 of Lecture Notes in Computer Science., Springer (2008)