Applying an Integrated Modelling Process to Run-time Management of Many-Core Systems

Asieh Salehi Fathabadi, Colin Snook, and Michael Butler

University of Southampton
{asf08r,cfs,mjb}@ecs.soton.ac.uk

Abstract. A *Run-Time Management* system for many-core architecture is aware of application requirements and able to save energy by sacrificing performance when it will have negligible impact on user experience. This paper outlines the application of a process for development of a run-time management system that integrates a range of modelling, validation, verification and generation tools at appropriate stages. We outline the models, process and tools we used to develop a temperature aware run-time management system for *Dynamic Voltage and Frequency Scaling* (DVFS) of a media display application. The *Event Refinement Structure* (ERS) approach is used to visualise the abstract level of the DVFS control. The *Model Decomposition* technique is used to tackle the complexity of the model. To model the process-oriented aspects of the system we used *iUML-B Statemachines*. We use several different visual animation tools, running them synchronously to exploit their different strengths, in order to demonstrate the model to stakeholders. In addition, a continuous model of the physical properties of the cores is simulated in conjunction with discrete simulation of the Event-B run-time management system. Finally executable code is generated automatically using the *Code Generation* plug-in. The main contribution of this paper is to demonstrate the complementarity of the tools and the ease of their integrated use through the Rodin platform.

Keywords: Many-core, Event-B, Formal methods, Run-time management, DVFS, Task allocation

1 Introduction

As electronic fabrication techniques approach the limit of atomic dimension, increases in performance can no longer be obtained from a single core with relative ease. Transistorised electronic devices gradually wear out due to physical phenomena such as electromigration and hot-carrier injection [1]. This effect is becoming more significant now that fabrication techniques are approaching the level of a few atoms. Interest in recent years, therefore, has increasingly focused on many core devices. Managing the use of a large collection of cores to achieve a given computing task with adequate performance in an energy efficient manner while minimising wear-out is a challenging problem, which is being tackled by the PRiME project [2] . A *Run-Time Management* system

that is aware of application requirements and able to save energy by sacrificing performance when it will have negligible impact on user experience is required. The run-time management system is also required to be aware of wear-out effects and minimise situations that accelerate them where possible. Furthermore we require such a system for disparate operating systems and hardware platforms.

Our approach is to developed formal models in Event-B [3] using the Rodin modelling platform [4] and plug-ins in an integrated formal development process, from requirements analysis through to code generation, so that we obtain a precise and correct specification from which we can generate variants and subsequently code for different platforms. Here we describe the models and modelling techniques we used as part of the formal development process to specify a temperature aware run-time management system for *Dynamic Voltage and Frequency Scaling* (DVFS) of a media display application. The run-time management system learns from the application when it can scale back voltage and frequency to save energy without missing too many frame deadlines. The run-time management system is also aware of core temperatures and controls thread scheduling of the operating system by setting thread-core affinity in order to avoid wear-out due to heating effects in the cores.

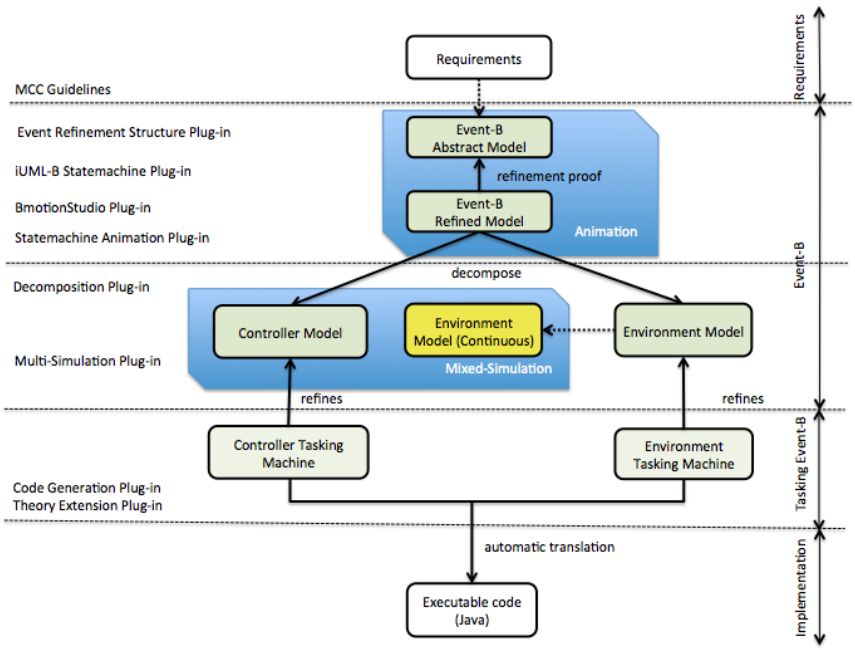

Fig. 1. Formal Design Towards Implementation

Figure 1 shows the complete formal development process indicating where different tools and methods are used. This figure should be referred to throughout the paper as different stages are discussed. The requirements are analysed using the *MCC* (Monitored, Controlled, Commanded) [5, 6] set of guidelines for cre-

ating an initial abstract Event-B model. The model is defined and refined using the *Event Refinement Structure* (ERS) approach [7, 8] and *iUML-B Statemachines* [9, 10] wherever appropriate. As well as formal refinement verification proof, the models are demonstrated to be valid using BMotionStudio [11] and iUML-B State-machine animation in tandem. The models are decomposed into controller and environment using *Model Decomposition* [12, 13]. To demonstrate the validity of the control of continuous environmental phenomena, a continuous model of the environment is created and co-simulated with the discrete Event-B controller using the multi-simulation plug-in [14]. Code can then be generated using the *Code Generation* plug-in [15] plug-in which has been enhanced for our purposes using the theory plug-in [16].

The contribution of this paper is to show how all of these different techniques and tools complement each other in a complete formal modelling process for a multi-core runtime management system. In section 2 we briefly describe the techniques and tools used. In section 3 we introduce the media decoder case study and the run-time management control of DVFS and temperature. In section 4 we describe the method of requirements analysis using the MCC guideline to obtain an initial abstract model. In section 5 we described our use of diagrammatic modelling notations. In section 6 we describe how we model the environment by decomposing the complete system model and/or by modelling in a continuous domain modelling tool. In section 7 we describe our use of visual animation and simulation tools to validate the models. In section 8 we describe how we generate an implementation from the refined models.

2 Background

Event-B [3] is a formal method for system-level modelling and analysis. Key features of Event-B are the use of set theory and first order logic as a modelling notation, the use of refinement to represent systems at different abstraction levels and the use of mathematical proof to verify correctness of models and consistency between refinement levels. The *Rodin* [4] platform is an Eclipse-based IDE for Event-B that provides effective support for modelling and mathematical proof.

We use Monitored, Controlled and Commanded phenomena (*MCC*) guideline [5, 6] to structure requirements. The MCC guideline facilitates formal modelling of a control system and help to formalising a set of informal requirements. Details are presented in section 4.

We use the *Event Refinement Structure* (ERS) approach [7, 8] to visualise and build the abstract and refined levels of the DVFS control as an Event-B model. ERS augments Event-B methodology with a diagrammatic notation for explicit representation of control flows and refinement relationships. Providing such diagrams aids understanding and analysing the control flow requirements and refinements without getting involved with the complexity of the mathematical formal language notation. Details are presented in section 5.2.

We use *Model Decomposition* [12, 13] to divide the DVFS control model into two sub-models: Controller and Environment. The controller sub-model consists of variables/events describing the softwaree layer properties whereas the envi-

ronment sub-model consists of variables/events describing the properties of the user and the hardware layer. Details are presented in section 6.1.

We use *iUML-B Statemachines* [9, 10] to model the thread scheduling process of the operating system under the influence of the run-time management system. State-machines provide excellent visualisation of mode-oriented problems and are animated for validation in synchronisation with BMotionStudio [11] visualisations of other parts of the model. Details are presented in section 5.3 (modelling) and 7.1 (animation).

We developed a continuous model of the thermal properties of a core depending on voltage and frequency using the Modellica [17] language. The continuous model is simulated in conjunction with ProB [18] simulation of the Event-B runtime management system. This is achieved via tools for mixed-simulation [14] which are under development in the ADVANCE project [19] . Details are presented in section 6.2 (modelling) and 7.2 (simulation).

Executable code was generated using the *Code Generation* plug-in [15]. The code generation feature provides support for the generation of code from refined Event-B models. To this end a multi-tasking approach has been added to the Event-B methodology. Tasks are modelled by an extension to Event-B, called *tasking machines* which are an extension of the existing Event-B machine component. The code generation plug-in provides the ability to translate to C and Java in addition to Ada source code. We adapted the code generation plug-in and used it to generate a Java implementation of the DVFS system. Details are presented in section 8.

3 The Case Study

Multimedia applications are expected to form a large portion of workload in general purpose PC and portable devices. The ever-increasing computation intensity of multimedia applications elevates the processor temperature and consequently impairs the reliability and performance of the system [20]. The run-time management system of the media decoder system scales the value of Voltage and Frequency (VF) for each frame of the media file. The VF value is scaled based on the speed of playing media requested by the operator and the type of the decoding frame. The run-time management system also learns from the number of CPU clock cycles taken to decode previous frames. The run-time management system aims to select an optimally minimal scaled VF value that provides adequate performance at minimal power consumption. The run-time management system also minimises thermal wear-out by selecting CPU cores that are well within the temperature thresholds for such effects. Unnecessary thermal cycling, which may also contribute to wear-out, is also avoided by selecting cores that are already warm when available.

The run-time management system operates in conjunction with an operating system by constraining or instructing the operating system in order to achieve the additional management features. Although we focus, here, on one type of multimedia application, the run-time management system should eventually be more generic and handle multiple types of application running simultaneously

on a many-core platform. This is reflected in our modelling of the more generic layers of run-time management system and operating system.

4 Requirements and Analysis of the Case Study

Guidelines can be used to facilitate the transition between an informal requirements and its formal representation. Influenced by Parnas four-variable [21], a guideline to model a control system using its monitored, commanded and controlled (*MCC*) phenomena is proposed in [5, 6]. A phenomenon can be of type variable or event. The definitions of MCC variable and event phenomena are given below:

- *Monitored phenomena*: Monitored variables whose values are determined by the environment. Environment events update monitored variables.
- *Controlled phenomena*: Controlled variables representing phenomena in the environment whose values are set by the controller. Control events update controlled variables.
- *Commanded phenomena*: Commanded variables whose values are determined by the user and that influence controlled phenomena. Commanded events are user requests to modify commanded variables.

We have used the MCC guideline to structure requirements of the media decoder case study. The PRiME architecture structures a many core system into four layers: user, application, system software and hardware (Fig. 2). Fig. 3 and Fig. 4 illustrate how the MCC phenomena map to the PRiME layers. In Fig. 2, from top to bottom, the interaction between the user layer and the application layer is categorised as commanded phenomena. We propose a new phenomena, called *task characterization*. Task characterization phenomena are determined by the application layer and are fixed during run time. The controlled phenomena values are set by the software layer, and finally the monitored values are determined by the hardware.

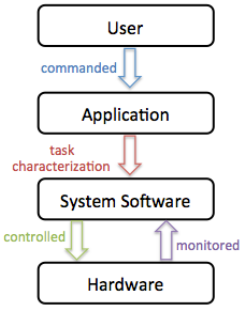

Fig. 2. PRiME Layers Illustration

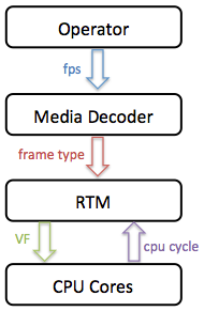

Fig. 3. DVFS Aspects of Case Study

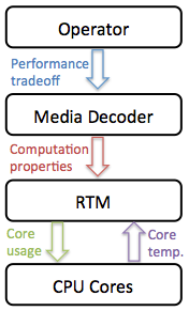

Fig. 4. Thermal Aspects of Case Study

Fig. 3 presents the PRiME layers and corresponding requirements phenomena for the media decoder DVFS aspects of the case study. Here the user of the system is an operator who attempts to open the media file and request the speed of playing media. The application is the decoder. The software layer corresponds to the run-time management, and the hardware layer corresponds to the CPU cores. The definition of the MCC phenomena are as below:

- *Commanded*: Frame Per Second (*fps*) property whose value is determined by the operator.
- *Task Characterization*: Frame type property whose value is determined by the decoder application.
- *Controlled*: Voltage and Frequency (*VF*) property representing the value of VF to be set in the hardware; VF value is set by the run-time management system.
- *Monitored*: CPU cycles number whose value is determined by the core.

These phenomena are specified as variables and events in the Event-B model. For example *fps* property is modelled as a variable and an event (*set_fps* introduced in Section 5.2) to set its value. The ordering between these phenomena (from top to bottom) are specified as invariants and event guards in the Event-B model (details in Section 5.2).

Fig. 4 presents the PRiME layers and corresponding requirements phenomena for the temperature control aspects of the case study. The definition of the MCC phenomena are as below:

- *Commanded*: Performance tradeoff is a property which the operator can adjust to control the balance between performance and wear-out
- *Task Characterization*: Computation properties (multi-threading) whose value is determined by the decoder application.
- *Controlled*: Core usage property representing the allocation of threads to CPU cores controlled by the run-time management system.
- *Monitored*: Core temperature measured for each CPU core.

5 Modelling

5.1 The Event-B Formal Method

Event-B [3] evolved from the B-Method [22] and adopts the more flexible systems-oriented refinement of Action systems [23]. New events can be introduced in refinements or old ones split into different cases as more complex data structures reveal more detailed behaviour. Event-B modelling and verification proof is supported by the Rodin platform [4]. Different design and modelling techniques, such as UML-B, model decomposition, ERS etc., have been developed and integrated into the Rodin platform as plug-in extensions which enrich the Event-B modelling process. The primary aim of Event-B is to validate that we are building the right system. Since validation is inherently a human decision it is important to build models which clearly show the important properties of a system. Event-B's method of verified refinement allows us to make abstractions so

that important properties can be modelled and validated without being obscured by detail. Verified refinement allows us to then add more details in simple stages in the knowledge that the validated properties are maintained. We use visual animation tools to validate abstract levels of the model and mixed-simulation at more refined levels.

5.2 Event Refinement Structure Approach

The Event Refinement Structure (ERS) approach [7, 8] is used to visualise and build the control flows of the abstract level and refinement levels of the DVFS control Event-B model. In Event-B, control flows are implicitly modelled by laboriously adding control variables, and corresponding guards and actions in order to specify the sequencing of events. ERS provides an explicit visual representation of control flows which is used to automatically generate these control variables, guards and actions. In addition, ERS provides explicit representation of the refinement relationships between an abstract event and the refining concrete events. ERS augments Event-B with a diagrammatic notation that is capable of explicit representation of control flow requirements and the corresponding refinement structure. The rest of the requirements are modelled directly in Event-B which allows full expressiveness. Providing such diagrams aids understanding and analysing the control flow requirements in a more direct way than the plain Event-B representation. The ERS diagram of the DVFS control is illustrated in Fig. 5.

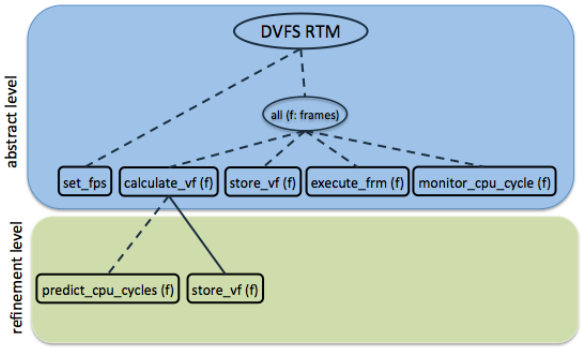

Fig. 5. ERS Diagram, Abstract Level of DVFS Model

The root oval contains the name of the system, DVFS RTM, Dynamic Voltage and Frequency System Run Time Management. Considering the abstract layer (blue region), the leaves are read from left to right, indicating the ordering between them. First the *set_fps* event executes followed by execution of four events for all of the frames of the decoding media file. The "*all*" replicator, appearing in the oval, indicates (potentially parallel) execution of its children for all of the instances of parameter "*f*" of type "*frames*" . For each frame, first the optimal value of the value of the VF is calculated (*calculate_vf* event), second the VF value is stored (*store_vf* event), then the frame is executed in the core

(*execute_frm* event) and finally the number of taken CPU cycles is monitored (*monitor_cpu_cycle* event). In the refinement level (green region), *calculate_vf* is decomposed into two sub-events, *predict_cpu_cycles* and *select_vf*. The ERS diagram explicitly illustrates that the effect achieved by *calculate_vf* at the abstract level is realised at the refined level by occurrence of *predict_cpu_cycles* followed by *select_vf*. The solid line indicates that *select_vf* event refines *calculate_vf* event while the dashed line indicates that *predict_cpu_cycles* event is a new event which refines skip.

5.3 iUML-B State-machines

iUML-B State-machines provide a mode-oriented control over a sequence of events. Hierarchical State-machines are added to the Event-B model and contribute (i.e. automatically generate) guards and actions to existing events in order to represent the transition source and target. Additional guards and actions may be added to transitions and invariants may be placed within states. We used state-machines to visualise the affinity-restricted thread scheduling of the temperature control operating system (Fig. 6). The state-machine shows that a thread is initially *PREMPTED* and may then *start* with a new time-slot (time-slot is modelled in the underlying Event-B and not shown in the diagram). While *RUNNING*, the thread can *progress* which consumes time-slot until it runs out (*timeup*) and becomes *EXPIRED*. If it has completed it will then *exit*. If not it returns to *PREMPTED* via *prempt* and must then *resume* (or *resumeCold*) with a new time-slot when a core is available. When resuming, the scheduler chooses a hot core (*resume*) in preference to a cold one (*resumeCold*) in order to avoid thermal cycling which increases wear-out. The alternative *suspend-SUSPENDED-activate* path represents a thread becoming blocked while waiting for resources to become available. The superstate *CURRENT*, consisting of *RUNNING* and *EXPIRED*, represents the conditions when the thread is allocated to a cpu core. The state-machine is 'lifted' to the set of current threads so that each thread has an independent state. During animation, (section 7) example instances are used to illustrate the behaviour of the model and appear as tokens indicating the current mode of each thread. The affinity restrictions are in the form of additional transition guards that restrict the selection of a CPU core when the *start*, *resume* or *resumeCold* transitions are taken. Refinement is performed by adding new state-machines inside old states. For example the states *RUNNING* and *EXPIRED* and transitions *timeup* and *progress* would not have existed in the abstract version of Fig 6.

5.4 ERS versus State-machines

ERS and State-machines both provide visual modelling tools for adding event sequence restrictions to Event-B. We use ERS for explicit representation of iterative flow (e.g., all) and for representing *event* decomposition in refinement. We choose state-machines for modelling cyclical modal behaviour and for representing *state* decomposition. Since these visual models are alternatives for representing the same thing (i.e. event sequencing) they are not both used for the

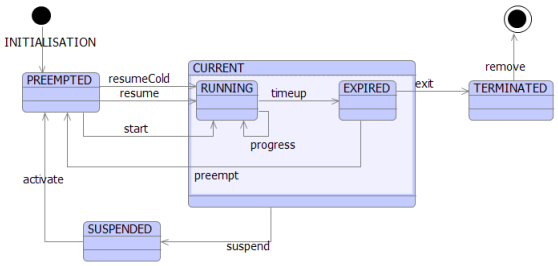

Fig. 6. iUML-B State-machine model of Thread Scheduling

same events but can be used for different sets of events within the same Event-B model. This is achieved by simply referring to different events in the respective diagrams. In our case study, we have used ERS to show the progressive event refinement of the DFVS processes and state-machines for the hierarchical modal behaviour of the thread scheduling. Another approach would be to use activity diagrams as explored by Dygham et al. [24], however, there is currently no tool support for activity diagrams in Rodin.

6 Modelling the Environment

6.1 Model Decomposition

Model decomposition pre-dated Event-B and is found in action systems [25]. Model decomposition in Event-B [12, 13], is used to manage the complexity and increase the modularity of an Event-B model. The idea of model decomposition is to divide it into components that can be refined independently while ensuring that if the components were re-composed they would constitute a valid refinement of the original model. The components interact through synchronisation over shared events. We applied the model decomposition technique to the DVFS control model dividing it into two sub-models: Controller and Environment. The controller sub-model consists of variables/events describing the SW layer properties whereas the environment sub-model consists of variables/events describing the properties of the user and HW layers. We have also used the definition of the model decomposition for code generation purpose. By using model decomposition, we introduce the controller and environment actions to the code generation process. Details of the code generation application are provided in section 8.

The *predict_cpu_cycles*, *select_vf*, *store_vf* and *monitor_cpu_cycle* events are the actions of the operating system (SW layer) and are included in the controller sub-model. There are two shared events between the controller and the environment which appear in both sub-models: *set_fps* and *execute_frm*. The *set_fps* event is the action of reading the fps value from the environment (user layer) and setting its value in a controller variable. The *execute_frm* event models execution of the frame in the core; it sets the environment variable recording the CPU cycle number taken for execution of the frames; this variable is read later by the controller (by execution of *monitor_cpu_cycle* event).

6.2 Continuous Models of the Environment

While Event-B is very suitable for modelling the discrete state-event based behaviour of a system, it is sometimes important to model the physical continuous behaviour of an environment. For the Temperature Control system we modelled the thermal characteristics of CPU cores in Modellica [17], Fig. 7. The left side of the model (blue connectors) calculates the amount of power being consumed. This is the sum of the static power which is proportional to the voltage and the dynamic power which is proportional to frequency and square of voltage. The power then determines the amount of heat flowing into the right side (red connectors). The thermal model consists of the heat capacitance of the core and a thermal conductor to the cooling system (which we assume is a fixed flow to ambient for now). The model has a boolean input *Active* which is controlled by the run-time management system (Event-B) model to represent when the core is running a thread (i.e. in state *RUNNING* of Fig. 6) on that core, and an output, *temp* which is returned to the run-time management system and used to influence decisions about which cores to use. Currently the model is an approximation which needs further development and more detailed comparison with empirical measurements. In particular, the cooling component is over simplified and the coefficients are chosen to give a typical response. However this is sufficient for our purpose which is to illustrate how the controller responds to and controls a typical environmental phenomena.

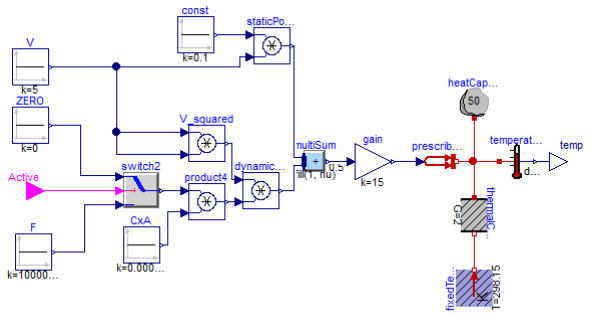

Fig. 7. Core Temperature modelled in Modellica

6.3 Decomposition versus Continuous models

We started from an abstract model of the system that includes the application, run-time manager, operating system and the hardware. We refine this model in steps to add details and then, at a suitable point, decompose it into two subsystems representing the software and the hardware. We consider the hardware to be an environment for the software subsystem. Following this process, we obtain a discrete event model of the environment that is a verified refinement of the abstract system model. This is an advantage when the controlled and measured phenomena of the environment can be adequately modelled by discrete events.

However, for continuous physical properties such as temperature, the verification relies on the validity of the discrete abstractions. Using a continuous model enables us to validate how the run-time management system works with these physical properties (as will be described in section 7.2) in order to improve our confidence in the discrete representation of them that the Event-B model uses. A further step would be to model the continuous properties in Event-B using recent extensions for continuous domain modelling [26] which would allow verifications of dynamic properties within Event-B. This would be beneficial when the validity of discrete approximations is less clear and can not be reliably determined simply by simulation.

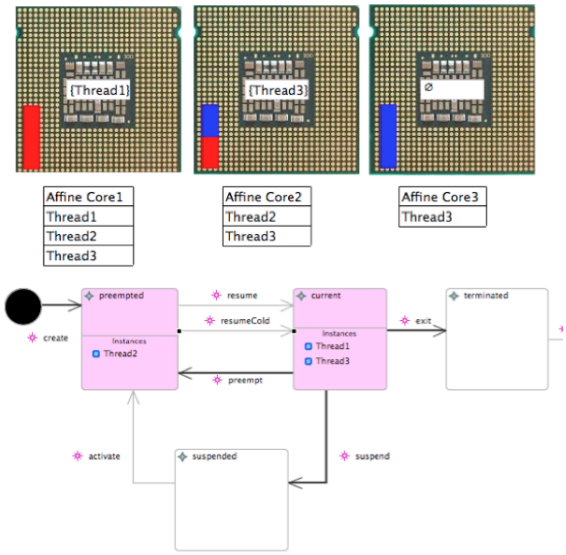

Fig. 8. Animating the model using BMotionStudio and State-machine Animation

7 Validation

7.1 BMotion Studio and Statemachine Animation

In order to validate our models we performed visual animations to demonstrate them to stakeholders. The ProB [18] model checker provides an animation facility which is the basis for two visual animation tools. BMotionStudio [11] is used to provide a graphical visualisation of three cores with threads being run on them and the resultant core temperatures increasing and decreasing. The iUML-B state-machine plug-in includes an animation tool to highlight active states and enabled transitions showing which threads are in each state at any point in time. The two visualisation work together on the same ProB animation, complementing each others view of the underlying model.

7.2 Mixed-simulation

To validate the response of the system to the continuous model of core temperature, we linked the Event-B run-time management system model with the continuous Modellica model using the co-simulation plug-in [14] developed in the ADVANCE project [19]. Fig. 9 shows the temperature (red line) of a single core heating rapidly when a thread becomes active (blue line goes low) and cooling when no thread is active. Currently the model only supports co-simulation of a single core and we are unable to demonstrate the temperature response to thread-core allocation for multiple cores. However, we are currently addressing this and co-simulation has the potential to allow us to demonstrate the validity of the run-time management system model in a simulation that appears close to reality, before going on to generate the implementation.

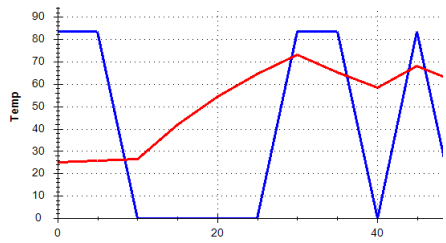

Fig. 9. Mixed-Simulation of Continuous Thermal Model and Event-B Discrete Run-time Management System

8 Code Generation

Code generation is an important part of the formal engineering tool chain that enables support for development from high-level models down to executable implementations. In the Event-B methodology, the code generation feature [15] provides support for the generation of code, for real-time embedded control systems, from refined Event-B models. To this end a multi-tasking approach, which is conceptually similar to that of the Ada tasking model, is designed and added to the Event-B methodology. Tasks are modelled by an extension to Event-B, called tasking machines which are an extension of the existing Event-B machine component. The code generation plug-in supports translation to C, Java or Ada source code. In the DVFS run-time management system are currently using the Java code generation option.

The theory plug-in [16] allows mathematical extensions to be added to an Event-B development. The code generation plug-in uses a theory component to define new data types, and their translation into target programming language data types [27]. In the DVFS run-time management system model, we specify the relation between a frame and the corresponding assigned VF as a function data type:

$frm\_vf \in FRAMES \rightarrow VF$

The function data type is not supported by the current release of the code generation plug-in. We extended the code generation plug-in to support functions by adding a new theory component to define a function data type and a translation rule to the target data type (a Java hash map). Fig. 10 presents the definition of the implementable function (*pfunImp*) with arguments for the types of the domain D and range R. *newpFunImp* is a constructor for an empty *pfunImp*.

```
THEORY
  FunctionImpl
TYPE PARAMETERS
  D
  R
OPERATORS

  •pfunImpl  :  pfunImpl(d : ℙ(D), r : ℙ(R))

  direct definition
   pfunImpl(d : ℙ(D), r : ℙ(R)) ≔ d⇸r

  •newpFunImpl  :  newpFunImpl(f : D ⇸ R)

  direct definition
   newpFunImpl(f : D ⇸ R) ≔ ø⦂ℙ(D × R)
```

Fig. 10. Theory Extension for the New Function Data Type

```
TRANSLATOR
  Java
  Metavariables
   • f ∈ ℙ(D×R)
   • d ∈ D
   • r ∈ R
  Translator Rules
   newpFunImplRule   :
     newpFunImpl(ø⦂ℙ(D×R)) ⟾
       new HashMapImpl<D,R>()
```

Fig. 11. Translation of Function Data Type

In the controller refinement, we first refine the Event-B definition of *frm_vf* as a function, to instead use the new function data type:

$frm\_vf \in pfunImp(FRAMES, VF)$;

with initialisation:

$frm\_vf := newpFunImpl(\varnothing \oftype \mathbb{P}(FRAMES \times VF))$

The translation of the function data-type to the target of the Java *HashMap* data type is presented in Fig. 11. Here is the generated Java code for the *frm_vf* variable:

```
HashMapImpl<FRAMES,VF>  frm_vf = new HashMapImpl<FRAMES,VF>();
```

In the controller tasking machine, we specify the task body, where we define the flow of control in the controller sub-model as below; and the control flow in the Java codes is managed using threads.

set_fps;
calculate_vf;
store_vf;
execute_frm;
monitor_cpu_cycle

To date, we have executed the generated code on a simulation of the hardware. We are currently working on porting the code to run on real hardware in order to evaluate the performance of the generated code.

9 Conclusion and Future Work

We have shown how various Rodin plug-ins can be used with the basic Event-B modelling and verification platform to form a complete formal development

process including requirements analysis, model development using diagrammatic modelling editors, model refinement, decomposition, validation, mixed-simulation and code generation. Several authors have discussed using formal methods within an existing development process (e.g. [28],[29]). However these works are quite dated and did not have the benefit of extensive tool support. As far as we know there is no comparable work on how all of the various Rodin centred tools and plug-ins can be utilised to form a new development process.

We have not stressed verification in this paper. It is an inherent feature of the Event-B refinement method where proof obligations are automatically generated and discharged within the Rodin platform ensuring that the models are well-formed and consistent. All of our refinements were fully verified using the Atelier-B automatic provers available for Rodin. Here we focus on the use of diagrammatic modelling aids which make models easier to construct and understand and visual validation tools which allow us to observe the behaviour of the model so that we can demonstrate that it behaves as we desire. Having produced a correct and useful model we use code generation to obtain a high-quality implementation from it. In summary, we use the following Rodin plug-ins to provide tool-support for a useable model-based formal development process.

- MCC provides a systematic way of structuring the requirements and of constructing a formal model from the requirements for a control system.
- ERS provides explicit representation of replication of flow (e.g., all) and for representing event decomposition.
- iUML-B state-machines provide explicit representation of cyclical modal behaviour and for representing state decomposition.
- Animation with BMotionStudio and iUML-B state-machines allow for easy validation by stakeholders who do not have Event-B expertise.
- Co-simulation supports validation the the discrete controller model behaves correctly in conjunction with a continuous model of the environment.
- Code generation provides an automated way of generating multi-tasking code with the potential for easy targeting of different languages and architectures.
- Rodin provers support formal verification of correctness of design with respect to a (discrete) model of the system.

The temperature control model contains of 4 levels of refinement, including 9 variables in total and 13 events in the last refinement. All of the proof obligation (118 in total) are discharged automatically by the Rodin prover. The DVFS control model contains 2 levels of refinement, including 9 variables and 7 events in the last refinement. 13 out of 38 proof obligations are discharged automatically; the rest are discharged manually due to introducing the mathematical extension using the theory plugin. The new defined operators cause manual effort to discharge proof obligation generated for the sub-component after the model decomposition.

We think our process of formal design and integrated modelling can be scaled and promoted in industry. MCC provides guidance to overcome the difficult task of constructing an abstract model from the requirements. Diagrammatic editors like iUML-B and ERS provide a high-level visualisation of the models and automate some of the lower-level Event-B infrastructure making the construction of

models more intuitive for engineers. Visual animation and simulation techniques bridge the semantic gap between mathematical models and real-world problem domains making the models accessible to stakeholders for validation. Generating code from a verified model reduces code and testing effort to offset the resources put into the modelling process. One self-criticism is that, although they can be used along side each other, in some areas the plug-in tools would benefit from better integration. For example, the ERS and iUML-B diagramming plugins could be integrated so that they use the same common diagram framework and generation mechanisms. Similarly, the animation tools could be integrated into a common visualisation.

One of the motivations for our approach is that we can produce model variants at the lower levels of refinement so that we can generate different implementations for different platforms. For example, we could provide alternative thread scheduling for different operating systems. In future work we will investigate these variants. Further work is also needed in order to use code generation to different target languages such as C. In future work, we will also perform experiments running the generated code on different many-core hardware platforms such as XEON Phi etc. in order to fully evaluate the benefits of the run-time management system.

Acknowledgement

This work is funded by the FP7 ADVANCE Project, www.advance-ict.eu, and the EPSRC PRiME Project, www.prime-project.org.

References

1. Oboril, F., Tahoori, M.: Extratime: Modeling and analysis of wearout due to transistor aging at microarchitecture-level. In: Dependable Systems and Networks (DSN), 2012 42nd Annual IEEE/IFIP International Conference on. (June 2012) 1–12
2. PRiME: Power-efficient, Reliable, Many-core Embedded systems. `http://www.prime-project.org`
3. Abrial, J.R.: Modeling in Event-B - System and Software Engineering. Cambridge University Press (2010)
4. Abrial, J.R., Butler, M., Hallerstede, S., Hoang, T., Mehta, F., Voisin, L.: Rodin: an open toolset for modelling and reasoning in Event-B. International Journal on Software Tools for Technology Transfer **12**(6) (2010) 447–466
5. Yeganefard, S., Butler, M.: Structuring Functional Requirements of Control Systems to Facilitate Refinement-based Formalisation. ECEASST **46** (2011)
6. Yeganefard, S., Butler, M.: Control Systems: Phenomena and Structuring Functional Requirement Documents. In: ICECCS. (2012) 39–48
7. Butler, M.: Decomposition Structures for Event-B. In Leuschel, M., Wehrheim, H., eds.: Integrated Formal Methods. Volume 5423 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2009) 20–38
8. Fathabadi, A.S., Butler, M., Rezazadeh, A.: A Systematic Approach to Atomicity Decomposition in Event-B. In: SEFM. (2012) 78–93

9. Snook, C.: Modelling Control Process and Control Mode with Synchronising Orthogonal Statemachines. In: B2011, Limerick. (2011)
10. Savicks, V., Snook, C.: A Framework for Diagrammatic Modelling Extensions in Rodin. In: Rodin Workshop 2012, Fontainbleau. (2012)
11. Ladenberger, L., Bendisposto, J., Leuschel, M.: Visualising Event-B Models with B-Motion Studio. In: FMICS'2009. Lecture Notes in Computer Science, Verlag (2009) 202–204
12. Silva, R., Pascal, C., Hoang, T.S., Butler, M.: Decomposition tool for Event-B. Softw., Pract. Exper. **41**(2) (2011) 199–208
13. Hoang, T.S., Iliasov, A., Silva, R., Wei, W.: A Survey on Event-B Decomposition. ECEASST **46** (2011)
14. Savicks, V., Butler, M., Bendisposto, J., Colley, J.: Co-simulation of Event-B and Continuous Models in Rodin. In: Rodin Workshop 2013, Turku. (2012)
15. Edmunds, A., Butler, M.: Tasking Event-B: An Extension to Event-B for Generating Concurrent Code. In: PLACES. (2011)
16. Butler, M., Maamria, I.: Practical Theory Extension in Event-B. In: Theories of Programming and Formal Methods. (2013) 67–81
17. Fritzson, P., Engelson, V.: ModelicaA unified object-oriented language for system modeling and simulation. In: ECOOP98Object-Oriented Programming. Springer (1998) 67–90
18. Leuschel, M., Butler, M.: ProB: A Model Checker for B. In: FME 2003: FORMAL METHODS, LNCS 2805, Springer-Verlag (2003) 855–874
19. ADVANCE: Advanced Design and Verification Environment for Cyber-physical System Engineering . `http://www.advance-ict.eu/`
20. Ge, Y., Qiu, Q.: Dynamic thermal management for multimedia applications using machine learning. In: DAC. (2011) 95–100
21. Parnas, D.L., Madey, J.: Functional Documents for Computer Systems. Science of Computer Programming **25** (1995) 41–61
22. Abrial, J.R.: The B-book - assigning programs to meanings. Cambridge University Press (2005)
23. Back, R.J., Kurki-Suonio, R.: Distributed cooperation with action systems. ACM Trans. Program. Lang. Syst. **10**(4) (1988) 513–554
24. Dghaym, D., Butler, M., Fathabadi, A.S.: Evaluation of Graphical Control Flow Management Approaches for Event-B Modelling. ECEASST **66** (2013)
25. Back, R.J.: Refinement calculus, part ii: Parallel and reactive programs. In: REX Workshop. (1989) 67–93
26. Abrial, J.R., Su, W., Zhu, H.: Formalizing Hybrid Systems with Event-B. In: ABZ. (2012) 178–193
27. Edmunds, A., Butler, M., Maamria, I., Silva, R., Lovell, C.: Event-B Code Generation: Type Extension with Theories. In: Abstract State Machines, Alloy, B, VDM, and Z. Volume 7316 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2012) 365–368
28. Fraser, M.D., Kumar, K., Vaishnavi, V.K.: Strategies for incorporating formal specifications in software development. Commun. ACM **37**(10) (October 1994) 74–86
29. Kemmerer, R.: Integrating formal methods into the development process. Software, IEEE **7**(5) (Sept 1990) 37–50