This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/LES.2019.2955316, IEEE Embedded Systems Letters

Verifying Cross-layer Interactions through Formal Model-based Assertion Generation

Asieh Salehi Fathabadi*, Mohammadsadegh Dalvandi†, Michael Butler* and Bashir M. Al-Hashimi*
* University of Southampton † University of Surrey

***Abstract*—Cross-layer runtime management (RTM) frameworks for embedded systems provide a set of standard APIs for communication between different system layers (i.e. RTM, applications and device) and simplify the development process by abstracting these layers. Integration of independently developed components of the system is an error-prone process that requires careful verification. In this paper, we propose a formal approach to integration testing through automatic generation of runtime assertions in order to test the implementation of the APIs. Our approach involves a formal model of the APIs, developed using the Event-B formal method which is automatically translated to a set of assertions and embedded in the existing implementation of APIs. The embedded assertions are used at runtime to check the correctness of the integration.** [1]

I. Introduction

Runtime Management (RTM) is used to optimise and trade-off between performance and power/energy efficiency. However, the complexity of embedded platforms together with the highly dynamic nature of modern applications make the development of RTM for managing hardware platforms and applications at runtime difficult. One possible approach to reduce the complexity of RTM development is the introduction of abstract layers for hardware (or *device layer*), application and RTM and providing a set of APIs (Application Programming Interface) for cross-layer interactions. Recent work [1]–[4] proposes systematic cross-layer frameworks managing interactions between applications, RTM and devices which can facilitate effective runtime software. In such frameworks, the RTM manages monitored and controlled values between the layers, explores trade-offs between performance and power, and optimizes energy consumption while maintaining the required performance.

The abstraction provided by such frameworks allows the development of the system in three independent layers, i.e. application, RTM, and device. Although this approach can reduce the development complexity, it introduces a new challenge regarding correctness in integration of the higher layers and the lower layers. While each of the layers may have been independently tested and verified, the interactions between different layers needs to be checked carefully to ensure the overall correctness of the system. Consistence interactions between layers can be challenging, since the layers can be developed and verified individually by difference sources and techniques. Violation of correct interacting behaviours like flawless control flows and valid value range settings can cause the overall system faulty.

[1]This work was supported by the EPSRC PRiME Project (EP/K034448/1), www.prime-project.org.

As an example, consider a device control (e.g. frequency) whose value is set by an RTM which uses an approximate computing technique to calculate the control value. Since the RTM is developed independently from the other layers and so does not have full knowledge of the device layer, it may try to set the control to a value outside the allowed range (e.g. a frequency that is not supported by the device). This paper addresses this problem by introducing a formal approach for runtime verification of the system to improve the confidence in interactions between layers, either by improving system understanding, or by checking conformance to specifications. Our proposed approach takes advantage of formal modelling and verification of the framework at an abstract level and automatic assertion generation from the verified model to perform runtime integration testing. The reason that we need runtime assertion checking is that we might not control the development of different components of the system that use the framework APIs so we cannot apply static verification to them. To validate our approach we have formally modelled and verified some properties of the PRiME Runtime Framework APIs [4] and generated the necessary assertions for testing the integration of different layers of the framework at runtime for the existing implementation. Our approach enabled us to identify a number of inconsistencies in the implementation of a video decoder application that was using the PRiME Runtime Framework APIs. The formal modelling of the framework also led to discovery of an inconsistency between the framework specification and its implementation.

Our integration testing approach consists of three stages: 1) formal modelling of the framework APIs in Event-B language, 2) formal verification of consistency properties (including order of API calls and value bounds) 3) automatic generation of assertions from the verified model and integration of assertions within the existing C++ implementation of the APIs.

The rest of this paper is structured as follows: Section II provides an overview of cross-layer RTM frameworks and briefly discusses the PRiME Runtime Framework. Section III introduces the Event-B formal modelling language briefly. Details of the verified formal model of the framework APIs are presented in Section IV. Section V presents the generation of assertions. Section VI evaluates the approach and finally Section VII concludes the paper.

II. Cross-Layer Runtime Framework

A cross-layer runtime framework introduces abstraction between different layers of a system. A number of frameworks

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/LES.2019.2955316, IEEE Embedded Systems Letters

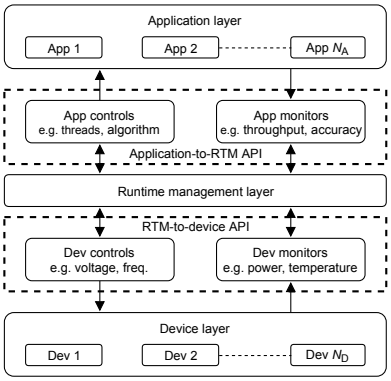

Fig. 1. Cross-layer framework and API.

that offer different levels of abstraction exist [1]–[4]. These link two or three layers of a system composed of application, runtime management and device layers (Figure 1). The application layer can include any application that uses the runtime API, the device layer includes the hardware platform and any low-level drivers, and runtime management level includes the RTM software. Application-to-RTM and RTM-to-Device APIs connect the layers to enable the exposure of adjustable controls (also known as knobs) and observable monitors to the runtime layer from the application and device layers. Controls are parameters in the applications and devices that can be adjusted at runtime, such as the degree of parallelism in an application or frequency at the device level. Monitors represent any property of the application that characterises its performance, such as a throughput measured in frames-per-second (fps). In the device, monitors can include physical properties of interest such as power consumption or architectural properties such as core utilization.

The PRiME Runtime Framework [4] is a cross-layer framework that provides solutions for application and platform agnostic runtime management. The framework achieves abstraction by separating the system into three layers (i.e. application, RTM, and device) and provides a set of APIs facilitating the interactions between different layers using cross-layer controls (knobs) and monitors. Controls and monitors have associated bounds with them in the form of minima and maxima. A bound represents a range of allowed or desired value for controls and monitors, respectively. The framework and APIs are built into a software library, with functions to perform all control and monitoring interactions.

III. Event-B Formal Method

Event-B is a formal modelling language for system level modelling based on set theory and first order predicate logic. The language is designed to target a set of different domains including distributed and embedded systems. A model in Event-B consists of two parts: a *context* which is the static part of the model (constants and types) and a *machine* which is the dynamic part of the model (variables and events). A model is specified using variables, invariants and events. An event models state change in the system and comprises a number of guards (conditions) and actions (assignments). An event is executed only if all its guards hold. An invariant is a predicate stating a condition on the state of the model. All events should preserve all model invariants. An event has the following of: E $\triangleq$ **any** *t* **when** *P(t,v)* **then** *S(t,v)* **end**, where *E* is the name of the event, *t* represents event parameters and *P(t,v)* and *S(t,v)* denote the guards and actions of the event, respectively. Event-B supports refinement. Refinement is a correctness-preserving stepwise process which starts from an abstract level and continues towards a concrete level by introducing new details to the model. Event-B is supported by the Rodin [6] platform. Rodin is an Eclipse-based IDE that provides effective support for refinement and mathematical proof of Event-B models. In our approach, we have used Event-B to model the PRiME Framework APIs and verify its different consistency properties. The following section, outlines the modelling of the framework and its verification.

IV. Formal Modelling and Verification of the Framework

To illustrate our approach, we present the modelling and verification of some of the framework properties in this section. Due to space limitation, we have simplified the formal model of the framework and omitted a lot of detail in this paper. Two of the most important properties of the framework, amongst others, are correct ordering between different interactions and monitor/control value boundaries. For instance, an application's control can be registered with the framework only if the application itself is registered. This property can be specified in the model using the following invariant: $app\_ctrl\_reg \in app\_reg \leftrightarrow APP\_CTRL$. In this invariant, app_reg is a variable denoting the set of all registered applications. app_ctrl_reg is a relation between registered applications (app_reg) and their registered controls and APP_CTRL is the type of application controls. The following event models the registration of an application control:

Event app_ctrl_manage
any a, ac **where**
grd1: $a \in app\_reg$
grd2: $ac \in APP\_CTRL$ **then**
act1: $app\_ctrl\_reg := app\_ctrl\_reg \cup \{a \mapsto ac\}$ **End**

Parameters a and ac denote an application and a control, respectively. The first guard specifies that the application should be registered. If application a is registered and ac is an application control, then the execution of the above event will add a new application-control pair ($a \mapsto ac$) to the *app_ctrl_reg* relation. If we remove the first guard, then a can be an unregistered application, hence the execution of the event may result in violation of the invariant. In addition to the correct ordering between different operations in the framework, it is important to ensure that the value of controls remain in the allowed range and different parties do not set an out of bound value. Each control and monitor has an associated pair of minimum and maximum value with it. This is specified for application controls using the following two total functions:

$app\_ctrl\_min \in APP\_CTRL \rightarrow \mathbb{Z}$
$app\_ctrl\_max \in APP\_CTRL \rightarrow \mathbb{Z}$

The following invariant states the aforementioned property about the value of controls formally:

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/LES.2019.2955316, IEEE Embedded Systems Letters

$$\forall\, c \,.\, c \in ran(app\_ctrl\_reg) \implies app\_ctrl\_value(c) \geq app\_ctrl\_min(c) \;\wedge\; app\_ctrl\_value(c) \leq app\_ctrl\_max(c)$$

where $app\_ctrl\_value(c)$ returns the value of control c. Setting the value of a control by RTM is modelled as follows:

Event $RTM\_app\_ctrl\_set$
any $a, ac, value$ **where**
grd1: $a \in app\_reg$
grd2: $ac \in APP\_CTRL$
grd3: $a \mapsto ac \in app\_ctrl\_reg$
grd4: $value \in \mathbb{Z}$
grd5: $value \geq app\_ctrl\_min(ac)$
grd6: $value \leq app\_ctrl\_max(ac)$ **then**
act1: $app\_ctrl\_value(ac) := value$ **End**

Guards of the above event, in particular $grd5$ and $grd6$, guarantee that the value of a control cannot be set to a value outside of the allowed range, hence the operation does not violate the invariant.

The consistency of the model is verified using theorem proving with the Rodin tool. The verification requirements are expressed in a number of automatically generated *Proof Obligations* (POs). For our model of RTM APIs, the tool generated 116 POs, where 97% of them were discharged automatically and the rest (3 POs) were discharged through interactive proof.

V. Automatic Assertion Generation

We have developed a proof-of-concept tool to automate the generation of C++ assertions from Event-B formal models. The tool, illustrated in Figure 2, is an extension of our existing contract generation tool [7] and is implemented as a Rodin plug-in. The assertion generation tool translates an Event-B event to a C++ function where the function implements an assertion. If an assertion is violated, an exception is thrown with a message about the violation to guide the developer.

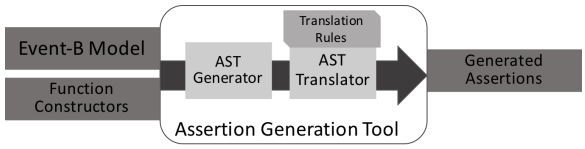

Fig. 2. Tool Overview

In translating a high-level Event-B event to a low level C++ implementation, we address the data abstraction gap between the model and the implementation. Although the model of an API is constructed independently from their actual implementations, implementation level variables refine model-level variables. The relationship between high- and low-level variables are made explicit by the modeller prior to translation. The tool extends an Event-B machine with a new construct called a *function constructor* to allow the modeller to define the relation of each event and its variables with its implementation in the final code. For instance, recall event *RTM_app_ctrl_set* from the previous section. A function constructor should be defined as follows:

RTM_app_ctrl_set (a:T1, app_reg:T2, ac:T3, value:T4, min:T5, max:T6) = {RTM_app_ctrl_set}

where $T1 - T6$ are the types of variables in the low-level C++ implementation and *RTM_app_ctrl_set* is the name of the corresponding Event-B event. The tool translates an event by using a pattern-matching algorithm and finding suitable translation rules for the event. We have defined a number of patterns for event guards. Each pattern has a corresponding translation rule for transforming a guard matching the pattern to C++ code. For instance, guard *grd3* of event *RTM_app_ctrl_set*, which specifies that the control should be registered with the application before setting the value of the control, is translated to a function returning true if the control is registered with the application and false if it is not.

For the aforementioned event, for example, the assertion function is generated based on the event guards and the above function constructor:

```
void assert_RTM_app_ctrl_set(T1 a, T2 app_reg, T3 ac,
                             T4 value, T5 min, T6 max)
{
  if(!isMember(a, app_reg))
      throw std::runtime_error("Registration violation");
  if(value < min)
      throw std::runtime_error("Boundary violation");
  if(value > max)
      throw std::runtime_error("Boundary violation");
}
```

To transform Event-B operators to code (eg. *grd1* to *isMember*), a set of predefined translation rules is provided as part of the tool. The typing information is also provided by the user through the function constructors. It is important that the modeller specifies the concrete variable types correctly. This requires the modeller to have a full understanding of the low-level implementation. For instance, in the framework implementation, the type of variable *a* is a user-defined type *pid_t* and *app_reg* is a vector of type *pid_t* [2]. This typing relationship should be made explicit. In the above generated assertion function code, $isMember(a, app\_reg)$ is a function which returns true if *a* is in *app_reg* or false otherwise. Generated assertions should then be injected into the implementation manually. First, the corresponding implementation of abstract events should be identified in the code. Then the assertion function generated from each event should be called right before the corresponding implementation of the abstract event. A catch mechanism should also be implemented as part of APIs implementation so that exceptions do not cause disruption in the overall execution. Below is the integration of assertions into the implementation:

```
...
try{

        assert_RTM_app_ctrl_set(a,app_reg,ac,value,min,max);
        //Block of code implementing RTM_app_ctrl_set
        ...
}
catch(std::exception &err)
{
        //Catch mechanism
}
...
```

The formal properties of our approach are: - *sufficient guards*: we formally verify that the guards are strong enough to guarantee that the invariants are preserved in the model; and - *assertion violation*: violation of a generated assertion within the code identifies an incorrect program state which may violate the model invariants.

[2]The full generated code and Event-B model is available at http://dalvandi.github.io/esl/. This can be compiled alongside the PRiME framework code at https://github.com/PRiME-project/PRiME-Framework.

VI. Evaluation

As mentioned before, the first stage of our approach is formal modelling and verification of the cross-layer framework APIs. Our modelling of the PRiME framework was divided among four refinement levels which helped us to manage the complexity of the model through gradual development. Formal models are usually simpler than executable implementation and are easier to ensure their consistency and correctness through formal verification. By assessing the case study requirements, we identified 18 consistency properties within the interactions between layers; our tool automatically generated 18 corresponding assertion functions which were then manually embedded to the existing implementation of the framework. The collection of correctness properties can further expanded by experimenting with new application and/or devices. In this paper, we ensure the correctness of API flows and value boundaries.

We have experimented with our approach by testing the integration of two applications (a video decoder and a Jacobi application), one RTM (Q-Learning) and a device (Odroid XU3) through the PRiME framework APIs [9]. Our approach enabled us to discover several inconsistencies in how the implementation of the video decoder application called the cross-layer API. The video decoder application has a specific monitor which returns the decode time of each decoded frame. If the developer forgets to register the monitor with the framework initially and tries to set/read the value of the monitor, then an assertion will be violated and an exception will be thrown ("Registration violation" assertion from the previous section). Likewise, if the value of the monitor is out of bound (i.e. less than lower bound or greater than upper bound) then second or third assertions will be violated and an exception will be thrown ("Boundary violation" assertion)[3].

In addition, we artificially introduced different errors to both the RTM and the applications and were able to detect all of them through the generated assertions. More interestingly, formal modelling of the framework and embedding the assertions into the existing implementation led us to find a bug in the implementation of the APIs which remained undiscovered in previous tests.

VII. Conclusion

One of the most important parts of a cross-layer runtime framework is its APIs. The APIs provide a standard way for cross-layer communications and this makes them extremely important in developing correct and reliable runtime management software. Inconsistencies in API calls in embedded system software can result in incorrect behaviour at runtime which can be difficult to discover and resolve. Since different components at different layers may be developed independently of each other by third party developers, it is essential that integration testing is performed to ensure that different software components are using the framework in a correct and consistent way. In this paper we introduced a formal-based approach for integration testing and validated our approach by applying it to the PRiME framework. Our use of Event-B allowed us to specify and reason about important properties of the PRiME framework at an abstract level. After proving the consistency of the framework APIs at a high level, we automatically generated a number of assertions specifying different consistency properties of the framework from the verified model. We embedded assertions into the existing implementation of the framework and checked their validity at runtime to ensure that different software components involved do not violate these assertions.

There has been little research reported on API runtime verification and to the best of our knowledge none on high-level modelling and testing of cross-layer API interactions in embedded systems. Hallé et al in [10] introduced an approach for runtime verification of a Web service API. They used a formal language to define properties of the API and employed model checking to check for potential violations. Spinellis and Louridas in [11] proposed a framework for static verification of API calls as a complementary tool to runtime verification. In our previous work [12] an approach was proposed toward automatic generation of the RTM system; this includes modelling and verifying the RTM system independent of its interactions with application and device layers. In this paper, however, we proposed an approach to model and verify the interactions (between RTM, application and device) within a cross-layer framework using its APIs.

Our current tool generates assertions automatically, however, embedding the generated assertions into the existing implementation is done manually. Automatic injection of assertions into the code will be addressed as future work. In our experience, modelling using a formal notation such as Event-B has great potential for modelling and verification of embedded software. In addition to integration testing, assertions can be used for other formal verification techniques such as static code analysis and model checking. In future, we would like to apply previous work on generating correct-by-construction implementations [8] from Event-B models to embedded systems, including cross-layer management.

[3]A demonstration video is available here: https://dalvandi.github.io/esl/

References

[1] Hoffmann, Henry, et al. "Application heartbeats: a generic interface for specifying program performance and goals in autonomous computing environments." ACM, 2010.

[2] Hoffmann, Henry, et al. "Dynamic knobs for responsive power-aware computing." ACM SIGPLAN Notices. Vol. 46. No. 3. ACM, 2011.

[3] Gadioli, Davide, et al. "Application autotuning to support runtime adaptivity in multicore architectures." SAMOS, IEEE, 2015.

[4] Bragg, G. M., et al: "An Application- and Platform-agnostic Control and Monitoring Framework for Multicore Systems." PEC, 2018.

[5] Abrial, Jean-Raymond. "Modeling in Event-B - System and Software Engineering." Cambridge University Press, 2010.

[6] Abrial, Jean-Raymond, et al: Rodin: an open toolset for modelling and reasoning in Event-B. STTT 12, 6, 447-466, 2010.

[7] Dalvandi, Mohammadsadegh, et al. "From Event-B models to Dafny code contracts." Springer, Cham, 2015.

[8] Dalvandi, Mohammadsadegh, et al. "Verifiable Code Generation from Scheduled Event-B Models." ABZ, Springer, Cham, 2018.

[9] Fathabadi, Asieh Salehi, et al. "A model-based framework for software portability and verification in embedded power management systems." Journal of Systems Architecture, 2018.

[10] Hallé, Sylvain, et al. "Runtime verification of web service interface contracts." Computer 43.3, 2010.

[11] Spinellis, Diomidis, and Panagiotis Louridas. "A framework for the static verification of API calls." Journal of Systems and Software, 2007.

[12] Fathabadi, Asieh Salehi, et al. "Towards automatic code generation of run-time power management for embedded systems using formal methods." MCSoC, IEEE, 2015.