Issues in Mapping from UML Real-Time Profile to OSEK API *

Zonghua Gu, Shige Wang and Kang G. Shin
Real-Time Computing Laboratory
Department of Electrical Engineering and Computer Science
University of Michigan
Ann Arbor, MI 48109-2122, USA
zgu@eecs.umich.edu

Abstract

UML Profile for Schedulability, Performance and Time is designed to add standard real-time extensions to UML in order to facilitate real-time analysis capabilities, such as rate monotonic analysis, based on the profile. In this paper we explore issues involved in generating code for the OSEK API, which is a popular real-time operating systems standard in the automotive industry.

1 Introduction

As embedded real-time systems become more and more complex and mission- or safety-critical, the traditional development process of manual coding followed by extensive and lengthy testing is becoming inadequate. The overarching concern for an embedded system developer is no longer to optimize software at very low levels in order to squeeze every ounce of performance out of it,[1] but to ensure high-level system correctness, modularity and maintainability at the expense of some performance loss. In order to increase developer productivity, the abstraction level for software development has been raised from assembly language to modern programming languages such as C/C++ and ADA. There is a recent trend to raise the level of abstraction further to the model-level, and rely on automatic or semi-automatic code generators to produce code in a traditional programming language. Examples of this approach include Unified Modeling Language (UML) [11], Model-Driven Architecture (MDA) [10], and Model-Integrated Computing (MIC) [6].

*The work reported in this paper was supported in part by DARPA and ARO under contracts/grants F3615-00-1706 and DAAD19-01-1-0473, respectively.

[1]This may still be true for certain domains such as digital signal processing for mass-produced consumer products, where performance optimizations can result in large savings in hardware costs.

UML is a general-purpose modeling language originally designed for information systems. In order to use UML in the embedded real-time domain, it is necessary to customize the design notations with concepts specific to real-time systems. A *UML profile* is a specialized subset of UML that extends or specializes UML with mechanisms such as *stereotypes*, *tagged values*, and *constraints*. The UML Profile for Schedulability, Performance and Time [5] (called *RT-UML* in the following discussions) is defined to enhance UML with real-time modeling notations and facilitate development of analysis tools that work on the models. In this paper we mainly focus on the schedulability sub-profile. Therefore most of the steorotypes start with "SA", which stands for *Schedulability Anaysis*.

Using UML for real-time software design is not new. Currently, the major companies that produce UML tools for the embedded real-time market, such as Artisan Software [7], Rational [13], ILogix [8] and Telelogic [14], each have their own proprietary extensions to UML for modeling real-time concepts. The main purpose of RT-UML is to define a standard syntax for expressing real-time concepts, so that models created by different companies with different tools can be exchanged freely.

OSEK [12] is a popular standard for real-time operating systems (RTOSs) used in automotive control. The OSEK standard is defined with the stringent timing and resource constraints in the automotive domain in mind. It consists of three parts: *OSEK-OS* specifies the operating system, *OSEK-COM* specifies the communications mechanism, and *OSEK-NM* specifies the network management system. We mainly focus on OSEK-OS in this paper. It describes a static RTOS where all kernel objects such as tasks, counters, alarms, events, messages and resources are created at compile

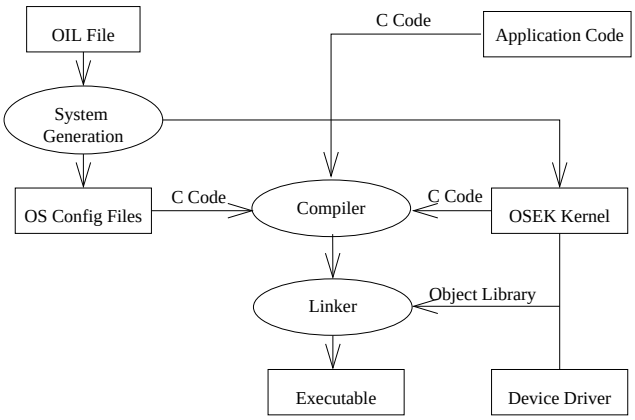

Figure 1: Development process with OSEK.

time. The OIL(OSEK Implementation Language) file is used to describe the kernel objects, and construct a customized kernel for the application, ensuring that only the necessary kernel objects and mechanisms are included in the kernel build. This minimizes the size and overhead of the RTOS as compared to the approach where all RTOS mechanisms are included regardless of whether the application needs them. Figure 1 shows the development process with OSEK. There is a clean separation between application functional code and OS configuration code thanks to the use of OIL files.

Even though OSEK is designed targeting the automotive domain, it is suitable for any resource-constrained environment. The major concepts in OSEK include:

TASK Real-time schedulable entity in the RTOS with a priority. Two types of tasks are defined: a *basic task* can assume states *running*, *ready* or *suspended*, while an *extended task* has an additional state *waiting* where it suspends its execution waiting for an event to occur.

RESOURCE Passive entity protected by *Priority Ceiling Protocol*(PCP), typically shared among several tasks.

COUNTER and ALARM A *counter* counts timer ticks or other recurring signals such as angle-based encoder interrupts. An *alarm* is associated with a counter and is triggered whenever the counter reaches a certain value. A periodic task can be defined with a cyclic alarm associated with a counter that is driven by the system timer, and an aperiodic task can be defined with an alarm associated with a counter driven by some external interrupt.

EVENT Synchronization mechanism between extended tasks. An extended task can suspend itself waiting for an event sent by another task (extended or basic) or an interrupt service routine. When the event is received, the task transitions from *waiting* state to *ready* state.

ISR Three categories of ISR are defined: *Category 1* ISR is not allowed to call any OSEK API, and is independent of the OS; *category 2* ISR is allowed to call OSEK API functions freely; *category 3* ISR is divided into 2 sections, where API calling is allowed in the first section but not in the second.

RT-UML	OSEK
≪SAResource≫	Resource.
SAResource. SACapacity	Maximum number of tasks that can access a shared resource simultaneously. Always one.
SAResource. SAAccessControl	Priority Ceiling Protocol. OSEK does not implement other protocols specified in RT-UML such as Priority Inheritance.
≪GRMAquire≫	API call GetResource()
GRMAquire. isBlocking	Boolean parameter that defines if the requesting task should block if the requested resource is not available. Always true.
≪GRMRelease≫	API call ReleaseResource()
≪SASchedulable≫	Basic Task. The OSEK concept of an extended task is not present in RT-UML.
≪SATrigger≫	Alarm. Declared in the OIL file, and set with API calls SetRelAlarm() and SetAbsAlarm(), to set an alarm with either relative or absolute expiration time.
≪SAPriority≫	Task priority.

Table 1: Mapping from RT UML concepts to OSEK concepts, either entities in the OIL files, or OSEK API calls. This is intended to be a small sample rather than a comprehensive definition.

In order to achieve seamless integration of modeling and software development, it is desirable to have automated mechanisms for generation of application code from models. In our context, we would like to map from RT-UML concepts into the OSEK API. Since RT-UML is mainly a collection of notations for expressing application timing behavior and is orthogonal to the functional behavior, we do not consider generation of functional code from UML models, which has been covered adequately by many commercial UML tools. Instead,

we focus on the mapping of concurrency, synchronization and real-time constructs found in RT-UML.

Table 1 shows some example mappings. Most concepts in RT-UML have intuitive mappings into OSEK entities, either in the OIL file, or in the C-based OSEK API. However, some concepts in RT-UML do not have direct correspondences in OSEK, mainly because they were designed for different purposes. OSEK is an implementation-level API designed for software *development* on an embedded target, while RT-UML is mainly designed for software *modeling* and *real-time analysis*. Therefore, some annotations in RT-UML such as worst-case execution time, absolute and relative deadlines are absent in OSEK. Note that in OSEK, task deadline is always implicitly defined the same as its period. That is, if a task is invoked again while it is still executing, it is deemed a deadline violation, also called timing overrun, and error handling mechanisms in the OSEK RTOS can detect and report this situation.

2 An Example Application Scenario in Automotive Control

Figure 2 shows an application scenario in automotive engine control. The system consists of 2 periodic tasks and 1 interrupt-triggered task accessing a shared data area protected by Priority Ceiling Protocol. It consists of both Electronic Throttle Control(ETC) and Air-Fuel Ratio (AFR) Control. With ETC, the usual mechanical linkage between the gas pedal and the throttle plate is eliminated, and the throttle is actuated by a DC motor. AFR controls the fuel injector timing so that the ratio of fuel to air must not deviate more that 0.1 from the stoichiometric air-to-fuel ratio of 14.64. Fuel is injected into the intake port area of the engine cylinders once every thermodynamic cycle (once every two engine revolutions). The injectors are set up to deliver a pulse of fuel whose duration corresponds to a fuel amount (mass) which achieves a desired air to fuel ratio of the charge in the cylinder. The timing of the pulse with respect to the cylinder intake valve opening is also important for reasons of proper mixing of fuel and air so that ignition of the charge is reliable.

Both ETC and AFR are triggered periodically, with different periods dictated by application characteristics. Both tasks read from a shared data area called *SFP-Data* (SFP stands for *Scaled Fuel Parameters*) that gets updated by an *SFPCalculation* task, which is triggered by the crankshaft and camshaft pulse signals aperiodically. The task calculates the appropriate duration of each injector pulse as well as the engine angle to start each injector pulse. The corresponding OSEK OIL file is shown in the appendix.

3 Implementation Approach

We are aware of some implementations of the UML RT Profile in commercial UML tools [7], and we believe it is in the interest of UML tool vendors to implement a code generator from RT-UML to OSEK in order to facilitate adoption of RT-UML in the automotive industry. For demonstration purposes, we propose to implement a prototype code generator based on a generic meta-modeling tool called Generic Modeling Environment(GME) [3] from Vanderbilt University. GME is a configurable toolset for creating domain-specific modeling and program synthesis environments through a *meta-model* that specifies the modeling paradigm of the application domain. The *meta-model* captures all the syntactic, semantic and presentation information regarding the application domain, and defines the family of models that can be created using the resulting modeling environment. It contains descriptions of the entities, attributes, and relationships that are available in the modeling environment, and the constraints that define what modeling constructs are legal. *Model transformation* is defined as transforming a model conforming to its meta-model A to another model conforming to a different meta-model B. It is often termed *semantic translation* since it changes model semantics in the process of transformation, which is generally a more difficult problem than *syntactic translation* such as file format conversion from postscript to PDF, where document syntax is changed while semantics remains the same. A *model interpreter* performs model transformation by traversing model structure based on meta-model definitions, and generates another model by mapping entities in one model to the other. A code generator from RT-UML to OSEK can be viewed as a model interpreter that transforms from RT-UML to OSEK API. Conceptually the process works as follows: we first construct a meta-model for RT-UML based on the OMG document, then synthesize a domain-specific modeling environment for RT-UML using GME. The model interpreter is written in C++. It traverses the RT-UML model structure based on a set of APIs generated from the RT-UML meta-model, and generates OSEK entities in the OIL file whenever it encounters a corresponding RT-UML entity. We can either explicitly define a meta-model for the OSEK API in the form of UML class diagrams, or implicitly embed this knowledge within the model interpreter.

4 Related Work

Alan Moore[4] described an extension to UML RT Profile in order to model the OSEK kernel in the form of an *OSEK Sub-profile*. The intention is to stress-test the RT Profile, and to facilitate tool integration

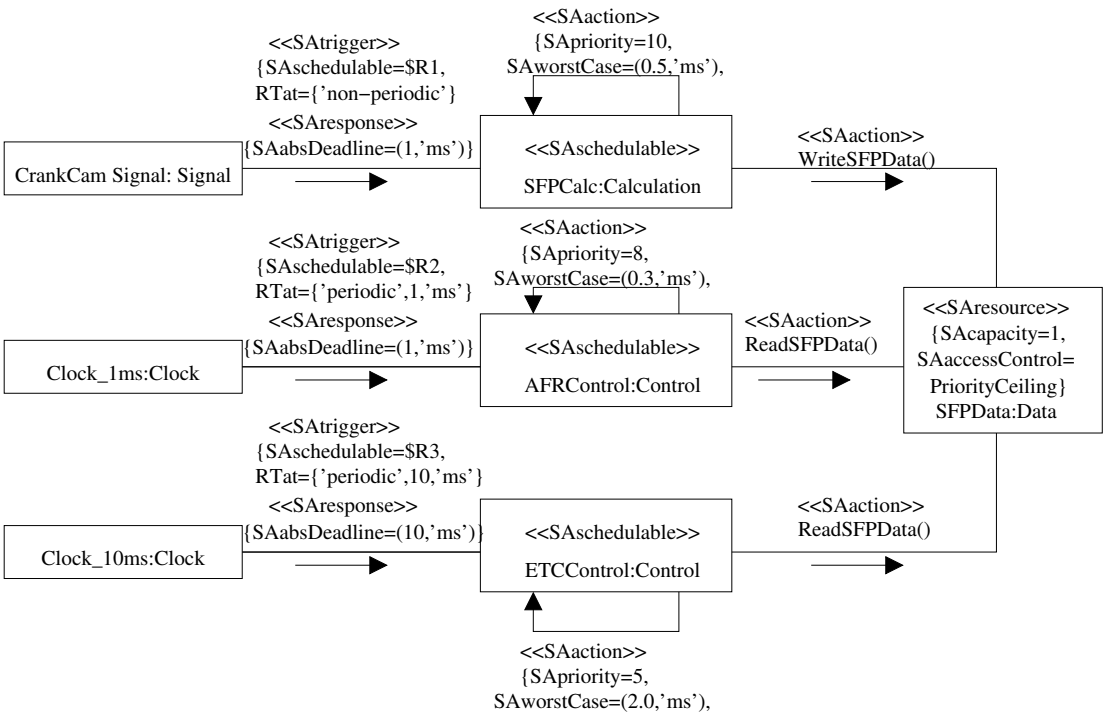

Figure 2: An application scenario in automotive engine control. SFP stands for *Scaled Fuel Parameters*; AFR stands for *Air Fuel Ratio* control; ETC stands for *Electronic Throttle Control*.

between UML modeling tools and OSEK-based analysis tools. The sub-profile defines stereotypes to represent OSEK-specific concepts such as Task, ISR, Alarm, Counter, Resource, etc. This approach takes advantage of the UML profile mechanism and allows accurate modeling of OSEK concepts. It would be straightforward to develop a code generator based on the OSEK sub-profile, which can be implemented within a UML modeling tool such as Artisan Studio [7]. However, it may take years to achieve standardization of the proposed sub-profile, so this approach is likely to remain a vendor-specific solution instead of a standard solution. Instead of proposing a separate OSEK sub-profile, we propose a more pragmatic approach of mapping the existing real-time profile to OSEK API. However there are also drawbacks to our approach. Even though significant overlaps exist between RT-UML and OSEK, many concepts exist in OSEK but are missing in RT-UML, and vice versa, due to different design purposes. For example, RT-UML does not have suitable definitions for the OSEK concepts of alarms, counters, ISRs. Even though similar concepts such as clock and timer exist in the RT-UML Real-Time Profile, their semantics is not an exact match for the corresponding OSEK concepts. Therefore, the code generator would have to infer from the UML model and insert these objects into the generated OIL file and C code.

Some tool vendors, such as Telelogic [14], ILogix [8], DSpace [2] and Mathworks [9], have developed automated code generators for modeling tools such as SDL, Statemate and Matlab/Simulink that can generate OSEK-compliant code. However we are not aware of code generators for UML that targets the OSEK API. We believe this should be a worthwhile endeavor, since UML is widely used in the automotive body electronics domain, and UML RT Profile should be an ideal candidate modeling notation.

Becker [1] described mapping from UML RT Profile to the Real-Time Java API (RTSJ). Since Java is an object-oriented language, it is a natural fit for code generation from UML, while OSEK is based on C instead of C++ due to tight resource constraints. RT-Java offers a much richer set of concurrency and real-time API calls than OSEK, which must live under much tighter resource constraints than RT-Java. For example, RT-Java offers sophisticated memory management facilities with a real-time garbage collector, while OSEK does not allow dynamic memory allocation, and forces all system objects to be pre-allocated at system startup. RT-Java evolved from Java running on the desktop and

mainly targets hand-held devices such as PDAs and cell phones, which often have ample processing power running heavyweight OSs such as Windows CE, instead of "under-the-hood", deeply embedded systems with severe resource constraints targeted by OSEK. Note that traditionally embedded software development in automotive control does not even use a RTOS. It was relatively recent that usage of a RTOS becomes commonplace. Therefore, it is understandable that OSEK designers are very conscious of resource requirements of the RTOS, and tradeoff efficiency against features.

References

[1] L.B. Becker, R.H. Holtz, and C.E. Pereira. On mapping rt-uml specifications to rt-java api: bridging the gap. In *Proceedings of Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, pages 348 –355, 2002.

[2] Lutz Koster, Thomas Thomsen, and Ralf Stracke. Connecting simulink to osek: Automatic code generation for real-time operating systems with targetlink. In *Proceedings of SAE Congress*, 2001.

[3] Akos Ledeczi, Miklos Maroti, Arpad Bakay, and Gabor Karsai. The generic modeling environment. In *Proceedings of the IEEE International Workshop on Intelligent Signal Processing*, May 2001.

[4] Alan Moore. Extending the uml rt profile to support the osek infrastructure. In *Proceedings of Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, pages 341–347, 2002.

[5] OMG. Uml profile for schedulability, performance and time. Technical report, Object Management Group, 2003.

[6] Janos Sztipanovits and Gabor Karsai. Model-integrated computing. *IEEE Computer*, 30(4):110–111, April 1997.

[7] Artisan Software website. http://www.artisansw.com.

[8] ILogix website. http://www.ilogix.com.

[9] Mathworks website. http://www.mathworks.com.

[10] OMG MDA website. www.omg.org/mda.

[11] OMG UML website. www.omg.org/uml.

[12] OSEK website. http://www.osek-vdx.com.

[13] Rational website. http://www.rational.com.

[14] Telelogic website. http://www.telelogic.com.

A OSEK Files for the Example Scenario in Figure 2

The OIL file defines the alarms used to trigger tasks, but does not specify the period for a periodic alarm. The way to have a task triggered periodically is to explicitly set alarms by using the *SetRelAlarm* API call in the C code. For example, in order to set the *RunAFRControlAlarm* to trigger *AFRControl* task every 1 ms, we need to call

```
SetRelAlarm(RunAFRControlAlarm /*AlarmID*/,
 0/*Offset*/,1/*Period*/);
```

When the system starts up, the TASK *InitAlarms* is first invoked which sets up alarms for all the periodic tasks. At runtime, the interrupt service routine *CrankCamSignalISR* is triggered by external interrupts from crankshaft and camshaft, and increments the counter *CRANKCAM_COUNTER*, which in turn sets the alarm *RunSFPCalcAlarm* that triggers the aperiodic task *SFPCalcTask*.

Below is definition of tasks in the C source code.

```
TASK(InitAlarms) {
  /* Initialize Model */
  osek_mrate_initialize(1);

  /* AFRControlAlarm runs every 1ms */
  SetRelAlarm(RunAFRControlAlarm /*AlarmID*/,
    0/*Offset*/,1/*Period*/);

  /* ETCControlAlarm runs every 10ms */
  SetRelAlarm(RunETCControlAlarm /*AlarmID*/,
    0/*Offset*/,10/*Period*/);

  TerminateTask();
}

ISR(CrankCamSignalISR) {
  /*Increment CRANKCAM_COUNTER*/
}

TASK(SFPCalcTask) {
  GetResource(SFPData);
  /* Perform computation */
  /* Write output variables */
  ReleaseResource(SFPData);
  TerminateTask();
}

TASK(AFRControlTask) {
  GetResource(SFPData);
  /* Read input variables */
```

```
  /* Perform computation */
  ReleaseResource(SFPData);
  TerminateTask();
}

TASK(ETCControlTask) {
  GetResource(SFPData);
  /* Read input variables */
  /* Perform computation */
  ReleaseResource(SFPData);
  TerminateTask();
}
```

Below is the OIL file.

```
CPU MPC555{
   /*************************************/
   /*             Tasks                 */
   /*************************************/
   TASK SFPCalcTask {
        TYPE = BASIC;
        SCHEDULE = NON;
        PRIORITY = 10;
        ACTIVATION = 1;
        AUTOSTART = FALSE;
        STACKSIZE = 4096;
        SCHEDULE_CALL = FALSE;
        RESOURCE = SFPData;
   };

    TASK AFRControlTask {
        TYPE = BASIC;
        SCHEDULE = NON;
        PRIORITY = 8;
        ACTIVATION = 1;
        AUTOSTART = FALSE;
        STACKSIZE = 4096;
        SCHEDULE_CALL = FALSE;
        RESOURCE = SFPData;
   };

   TASK ETCControlTask {
        TYPE = BASIC;
        SCHEDULE = NON;
        PRIORITY = 5;
        ACTIVATION = 1;
        AUTOSTART = FALSE;
        STACKSIZE = 4096;
        SCHEDULE_CALL = FALSE;
        RESOURCE = SFPData;
   };

   /*************************************/
   /* Must be highest priority in the system.*/
   /*************************************/
   TASK InitAlarms {
           TYPE = BASIC;
           SCHEDULE = FULL;
           PRIORITY = 16;
           ACTIVATION = 1;
           AUTOSTART = TRUE;
           STACKSIZE = 128;
           SCHEDULE_CALL = FALSE;
   };

   /*************************************/
   /*             Alarms                */
   /*************************************/
   ALARM RunSFPCalcAlarm {
        COUNTER = CRANKCAM_COUNTER;
        TASK = SFPCalcTask;
        ACTION = ACTIVATETASK;
   };

   ALARM RunAFRControlAlarm {
        COUNTER = SYSTEM_TIMER;
        TASK = AFRControlTask;
        ACTION = ACTIVATETASK;
   };

   ALARM RunETCControlAlarm {
        COUNTER = SYSTEM_TIMER;
        TASK = ETCControlTask;
        ACTION = ACTIVATETASK;
   };

   /*************************************/
   /* ISRs */
   /*************************************/
   ISR CrankCamSignalISR {
        CATEGORY = 2;
   };

   /*************************************/
   /*             Resources             */
   /*************************************/
   RESOURCE SFPData {
        /*Put application-specific attributes here. */
   }

   /*************************************/
   /*             Counters              */
   /*************************************/
   COUNTER SYSTEM_TIMER {
        MAXALLOWEDVALUE = 65535;
        TICKSPERBASE = 1;
        MINCYCLE = 1;
   };

   COUNTER CRANKCAM_COUNTER {
        MAXALLOWEDVALUE = 65535;
   };

   /*************************************/
   /*             O/S                   */
```

```
   /************************************/
   OS OSEK_OS {
        CC = AUTO;
        STATUS = STANDARD;
        SCHEDULE = AUTO;
        SYSTEMSTACKSIZE = 16000;
        StartupHook = TRUE;
        ErrorHook = FALSE;
        ShutdownHook = FALSE;
        PreTaskHook = FALSE;
        PostTaskHook = FALSE;
        WINDVIEW_SUPPORT = FALSE;
        RTA_SUPPORT = FALSE;
        STACK_FILL_DIAGNOSTIC = FALSE;
};
```