Embedded Software Design Methodology Based on Formal Models of Computation

Soonhoi Ha(✉) and EunJin Jeong

Seoul National University, Seoul, Korea
{sha,chjej202}@snu.ac.kr
http://peace.snu.ac.kr

Abstract. The current practice of embedded software design resorts to test or simulation to verify the correctness of the design, which is very time-consuming and incapable of covering all cases. Existent software engineering techniques are not concerned about real-time performance and resource requirements that embedded systems should satisfy for correct operation. In this work, we propose a new methodology to design dependable software for embedded systems. The key idea of the proposed methodology is to define a universal execution model (UEM) of heterogeneous multiprocessor embedded systems and to design the software based on the UEM that hides the underlying system architecture from the programmer. UEM puts restrictions on how to communicate and synchronize tasks that conventional operating systems deal with. We define the UEM by extending well-known formal models such as Synchronous Dataflow (SDF) and finite state machine (FSM). There are several benefits to use formal models for software design. First, we can detect critical design errors such as deadlock and buffer overflow by static analysis of formal models. Second, we can estimate the resource requirement and real-time performance at compile time. Last, not the least, we can synthesize the target code from the UEM automatically minimizing the manual coding efforts. By preserving the semantics of the UEM, the synthesized code will be correct by construction. The key challenge lies in the expression capability of the proposed UEM. Preliminary experiments with several non-trivial applications prove the viability of the proposed methodology.

Keywords: SW design methodology · Universal execution model
SDF

1 Introduction

The application domain of embedded computers as special purpose computers is steadily increasing as virtually all things are becoming smart or intelligent. The complexity of embedded computers is also incessantly increasing as can be observed in automotive electronic systems, intelligent robots, medical devices, as well as mobile devices. Since everyday life will depend on embedded computers

M. Lohstroh et al. (Eds.): Lee Festschrift, LNCS 10760, pp. 306–325, 2018.
https://doi.org/10.1007/978-3-319-95246-8_18

extensively in the future, it is crucial to make them dependable to avoid serious damages caused by an error or failure of the. For instance, in 2013, the US Department of Justice issued a ruling that imposed a fine of 12 billion dollars to a car manufacturer for sudden acceleration incident that was caused by a software defect in the electronic control unit. Thus it is needless to say how significant is to increase the dependability of embedded computer systems, particularly dependability of the software.

The user of an embedded computer system anticipates that the system works correctly any time, even with the non-zero possibility of hardware component failure. Above all, the correctness of the software should be ensured under the normal operating condition. Unfortunately, it is a well-known fact that it is not possible to detect all errors of a sequential program even though extensive research efforts have been made to develop static analysis techniques to solve this problem. Ensuring functional correctness of parallel programs is much more difficult since the program may have non-deterministic behavior at run-time due to unpredictable access order to shared resources. To make matters worse, embedded systems impose extra constraints on memory space, energy budget, real-time performance, and so on. For real-time applications, we need to guarantee that the real-time constraints are satisfied under the worst-case scenario of the system behavior.

The current practice of software design resorts to test or simulation to verify the functional correctness. To improve the functional safety of automotive electronics systems, for instance, the ISO26262 standard defines how to perform unit test and integration test. However, verification by test or simulation is very time consuming and incapable of covering all possible behaviors. Thus there is non-zero possibility to face an unexpected software behavior at run-time that has not been visited in the test or simulation phase.

There have been various methodologies proposed in software engineering to increase the design productivity and maintainability of software, including structural programming, object-oriented programming, model-driven development, component-based development, and so on. Each methodology has its advantages and disadvantages depending on the application area and the hardware platform. Most of those approaches use a test-based or a simulation-based method to verify the correctness. Nay more, they do not consider the memory space constraints, energy budget, and real-time performance requirements that should be satisfied in the embedded system. It is the designer's responsibility to meet those constraints. A common way to satisfy the real-time performance is to over-design the system with a significant safety margin (at least 50% for instance) over the worst-case values measured in the test phase.

In short, how to verify the correctness of embedded software is still an open problem, a stronghold that could not be conquered by existent methodologies. In this work, we propose a new methodology to make an embedded software correct by construction by designing embedded applications with formal models of computation at the OS level. It is motivated by the observation that a parallel application consists of a set of tasks, or threads, at the operating system (OS) level regardless of the initial specification. A task, or thread, is a unit of mapping

and scheduling and there are various ways of communication and synchronization between tasks. In the proposed methodology, we define a set of rules on task synchronization and communication that tasks are enforced to follow. To be concrete, we make computation tasks follow a dataflow model where tasks communicate with each other through channels, disallowing implicit communication through shared variables. By using a restricted form of dataflow model such as Synchronous Data Flow (SDF) [1] for application specification, we can perform static analysis to check the possibility of deadlock or channel buffer overflow. Moreover, we can estimate the worst-case performance and resource requirement at compile-time.

The proposed methodology separates design and implementation of embedded software. The designed software based on formal models of computation is mostly independent of the hardware platform, except for a minimal set of platform-dependent tasks Parallelizing an application is easily performed by mapping tasks onto processors, and inter-processor interface code is automatically synthesized in the proposed methodology. By keeping the semantics of formal models, the implemented software is free from a class of errors that can be detected by static analysis performed at compile time. It is distinguished from the current practice of embedded software development that is tightly coupled with a given hardware platform. Parallelizing an application is performed manually considering the features of the given hardware platform. Since an embedded application is tailored to a specific platform, it is not easy to port an application to a different hardware platform.

The proposed methodology concerns about the execution of tasks at the OS level and above, assuming that each task is already verified and its execution profile is given a priori. It is complementary to the existing methodologies in that test-based verification or formal verification should be used to verify each task. Defining a set of rules on task synchronization and communication can be understood as defining a universal execution model[1] (UEM) for multi-processor embedded systems. Even though we aim to make the proposed execution model be *universal*, because the baseline model is a data flow model, it fits better for computation-oriented applications than database-oriented applications.

Figure 1 shows the vertical software structure based on the UEM. The UEM is positioned on top of the OS layer, hiding the low-level details of the architecture from the application programmers. The UEM layer consists of three layers internally. The UEM execution engine serves the role of middleware that executes the UEM tasks on the target architecture, which is platform-dependent. To the application programmer, the UEM layer provides a set of APIs (application programming interface) for communication and synchronization between tasks. Thus, the application programmer can design an embedded software on top of the UEM without knowing the actual hardware platform on which the program runs. In the middle, a set of UEM tasks is generated from the application

[1] The term *universal* is not based on any formal proof but on our goal to make the model independent of underlying hardware platforms.

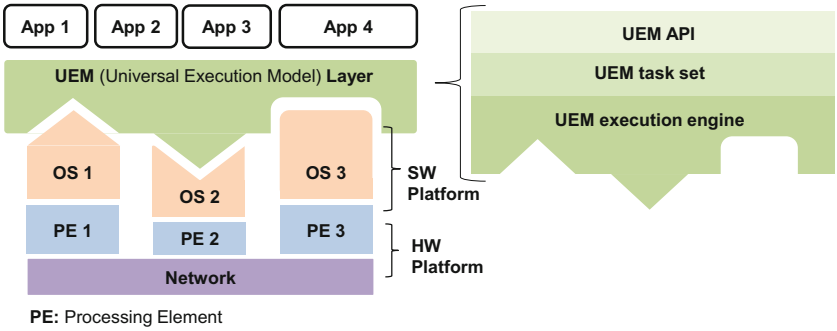


Fig. 1. Vertical software structure based on the UEM (universal execution model)

program. The UEM execution engine customized to a specific architecture aims to emulate the UEM efficiently.

The key challenge of the proposed methodology lies in the expression capability of the proposed UEM. Since the existent formal models of computation exhibit limitation on expression capabilities, several extensions have been made to the existent models in UEM. Preliminary experiments with several non-trivial applications prove the viability of the proposed methodology.

2 Dataflow Specification of an Application

In the proposed UEM, an application is specified by an extended synchronous dataflow (SDF) model. We first review the baseline SDF model and explain how the SDF model is extended.

2.1 Synchronous Data Flow

In the SDF model [1], an application is specified with a dataflow graph, $G(V, E)$, where V is a set of nodes and E is a set of arcs. A node $v \in V$ represents a function module, or a task, and an arc $e \in E$ is a FIFO channel between two tasks. Communication between two tasks is performed by explicit message passing via a FIFO channel. Figure 2(a) shows an example SDF graph where the number annotated on the arc indicates the number of data samples, called a sample rate, to produce or consume per task execution. If unspecified, the sample rate is 1 by default. The input sample rate and the output sample rate on an arc are represented as $cons(e)$ and $prod(e)$, respectively. In the SDF model, a task becomes executable only when all input arcs have no fewer samples than the specified sample rate in the associated arcs.

By comparing the input and the output sample rates on each arc, e, we can determine the relative execution rates between the source task, denoted by $src(e)$, and the destination task, denoted by $dest(e)$. For instance, the execution rate of task C should be twice higher than that of task A in Fig. 2(a), in order

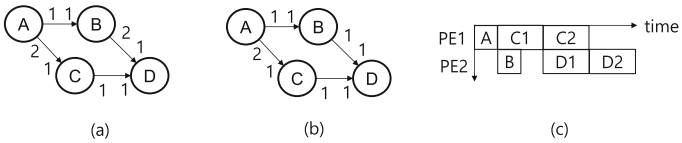


Fig. 2. (a) An example SDF graph with annotated sample rates on the arcs, (b) an inconsistent SDF graph that has a buffer overflow error, and (c) a mapping and scheduling result of the SDF graph onto two processing elements

to make the number of samples produced from the source task the same as the number of samples consumed by the destination task. This constraint can be formulated as the following equation, called balance equation: $prod(e) \times R(src(e)) = cons(e) \times R(dest(e))$ where $R(v)$ indicates the repetition counts of task v. An SDF graph is said to be consistent if we can find the repetition counts of all tasks to satisfy the balance equations of all arcs. Otherwise, the graph is called sample rate inconsistent, shortly inconsistent. The SDF graph shown in Fig. 2(b) is inconsistent, which may incur a buffer overflow error on arc AC. An iteration of an SDF graph is defined by the set of task executions with minimum repetition counts. The minimum repetition counts of tasks in the SDF graph of Fig. 2(a) are $R(A) = R(B) = 1$ and $R(C) = R(D) = 2$.

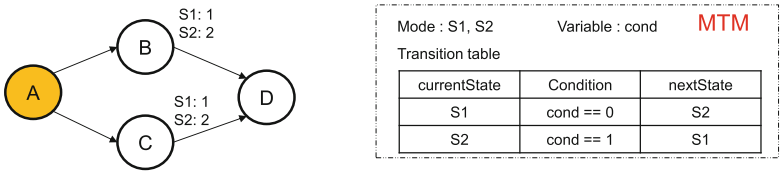
Since we can compute the minimum repetition counts of all tasks and the graph shows the dependency relationship between tasks, we can perform task scheduling at compile time, which is to determine where and in what order tasks will be executed on a given hardware platform. By constructing a static schedule of tasks at compile time, we can detect the critical software faults such as buffer overflow and deadlock. Figure 2(c) illustrates a parallel scheduling result by mapping tasks onto two processing elements. From the parallel scheduling result, we can estimate the buffer size and the real-time performance of the graph if the execution time of each task is bounded. Note that even though there may exist numerous schedules for a given application, determinism of the execution behavior is guaranteed, meaning that the execution result is independent of the schedule.

In summary, by using the SDF model, we can verify the satisfaction of real-time requirement and resource constraints with static scheduling. Moreover, we can detect buffer overflow and deadlock errors at compile time. While the SDF model has the aforementioned benefits from its static analyzability, it has a severe limitation to be used as a general model for behavior specification. It is not possible to specify the dynamic behavior of an application since the sample rate may not change dynamically. To overcome this limitation while preserving the static analyzability of the SDF model, several extensions have been proposed, including CSDF (cyclo-static dataflow) [2], SADF (scenario-aware dataflow) [3], and PSDF (parameterized SDF) [4]. In the proposed methodology, we use the FSM model in combination with the SDF model to express the dynamic behavior of an application at the task level.

2.2 Dynamic Behavior Specification

In case an application has a finite number of different behaviors, called modes of operation, the behavior of each mode is expressed by an SDF graph, and mode transitions are specified by a tabular specification of an FSM, called Mode Transition Machine (MTM) [5]. It is similar to FSM-SADF [3]. An MTM describes the mode transition rules for the SDF graph, defined as a tuple {*Modes, Variables, Transitions*} where *Modes* and *Variables* represent a set of modes and a set of mode variables respectively, and *Transitions* is a set of transitions that consists of the current mode, a Boolean function of conditions, and the next mode. A Boolean function of the transition condition is defined by a simple comparison operation between a mode variable and a value.

An example of MTM-SDF specification is shown in Fig. 3 in which an application has two modes of operation, S1 and S2. The input and output sample rates of a task may vary, depending on the mode. In this example, the MTM is quite simple since it needs to distinguish two modes of operation by a single mode variable. Since the granularity of a task is large and the dynamic behavior inside a task is not visible in the UEM, an MTM is not complex in general. At compile time, the SDF graph is scheduled separately for each mode of operation. We assume that all modes share the same initial buffer states. Then, mode change can be made at the iteration boundary safely without any inconsistency of buffer states between modes.



currentState	Condition	nextState
S1	cond == 0	S2
S2	cond == 1	S1

Fig. 3. Extended SDF graph with an MTM with 2 modes

Mode transition is enabled by setting the mode variable. There are two ways of setting the mode variable. It can be set by a hidden supervisor, which will be explained in the next section. Alternatively, it can be set by a designated task. A stream-based application usually starts with parsing a header information that determines the mode of operation, followed by processing a stream of data. In this case, the SDF task that parses the header information is designated as a special task that may change the mode variable. In the example of Fig. 3, task A can be designated as the special task that determines the mode of operation.

When mode transition occurs, the SDF schedule is changed accordingly. If the mode change is enabled by the hidden supervisor, it is activated at the iteration boundary of the SDF graph. If it is enabled by a designated task, mode change occurs right after the task finishes its execution. For consistency of operation, in this case, the schedules of all modes should have the same task schedule before the designated task. In case the designated task is the first task in the SDF schedule, this restriction is satisfied easily.

2.3 Library Task

In the dataflow specification, use of shared variables among tasks is not allowed since the access order to a shared variable may vary depending on the execution order of tasks and the application behavior will be non-deterministic. In many embedded applications, however, it is popular to use a global data structure that is shared among tasks. In the UEM, another extension is made to the SDF graph by introducing a special type of task, called library task, to allow the use of shared resources in the SDF model [6].

A library task is a mappable object that defines a set of service functions to a shared resource among tasks. Figure 4 shows an SDF graph that consists of three normal SDF tasks ($T1$ - $T3$) and two library tasks ($L1$ - $L2$). For connection with a library task, we introduce new types of ports, library master port and library slave port that are represented by a red circle and a blue square, respectively in the figure. An arc between a library master port and a library slave port is not a data channel, but represents a client-server relation. A library task plays the role of a server with a single slave port that can be connected to multiple masters that request the predefined services of the library task. Unlike a normal SDF task, a library task is not invoked by input data but by a function call inside an SDF task; it is a passive object.

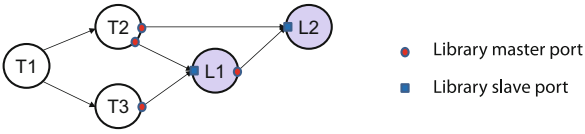


Fig. 4. Extended SDF graph with library tasks (Color figure online)

There are several use cases of library task depending on the kind of a shared resource. A library task can be used as a monitor that handles the access conflict to the shared variables at a high level. If a shared resource is a hardware device, the library task is a thread-safe device driver that provides a set of Application Programming Interfaces (APIs) to access the device. In a server-client application, the server task can be specified by a library task that may be shared by multiple clients. Another use case of a library task is to make a vertically layered software structure by providing a set of APIs of the software layer below the application layer. Figure 4 depicts three layers of software structure.

In case multiple masters access a shared variable that a library task manages, it is unavoidable that the return value of a library function depends on the execution order of the master tasks, which is anathema to any deterministic model. By the use of a library task, however, we explicitly specify the possibility of such non-determinism. In case the library task has no state or returns the same value to the master tasks regardless of the calling order, the library task is classified as deterministic. Otherwise, the developer should be aware that the

library task does not guarantee deterministic behavior in the sense that the return value to a master task depends on the scheduling order of master tasks. Nonetheless, the same behavior can be repeated if the same scheduling order is followed since the SDF model allows us to construct a static schedule of tasks. Then the application behavior becomes deterministic if the static schedule of tasks is followed at run-time.

2.4 Loop Structure (SDF/L)

A compute-intensive application usually spends most of its execution time in loop structures and how to parallelize them is the main challenge for accelerating the application. Even though dataflow models, including the SDF model, are good at exploiting the task-level parallelism of an application, it is difficult to exploit the parallelism of loop structures since they are not explicitly specified in existent dataflow models. In SDF, a loop structure is implicitly expressed by sample rate changes as illustrated in Fig. 2(a). Among many possible schedules, a looped schedule $AB2(CD)$ can be constructed. In case 2 executions of (CD) can be parallelized with 2 output samples from A and B, a user may want to construct a parallel schedule as illustrated in Fig. 5(b). However, identifying such a loop structure and parallelizing it is not easy because existent parallel scheduling techniques usually aim to exploit task-level parallelism only.

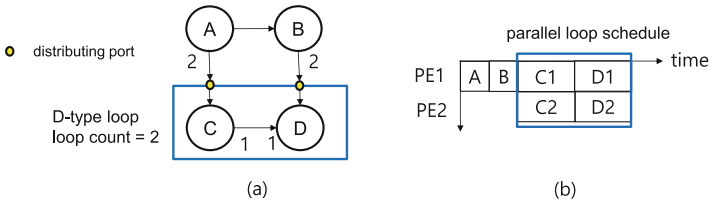


Fig. 5. SDF graph with a loop structure

Recently, we proposed a novel extension to specify a loop structure as a super node to make the SDF graph hierarchical [7]. The extended SDF graph with loop structures is called an SDF/L graph. Figure 5(a) is the SDF/L graph representation of the application of Fig. 2(a).

In the SDF/L model, two types of loop structures are distinguished, data loop (D-type) and convergent loop (C-type), and two types of input ports, distributing port and broadcasting port. In a D-type loop (data loop), each iteration of the loop consumes new input data from each distributing input port. The number of iterations is determined by the sample rate change of the associated input channel. The loop structure of Fig. 5(a) is a D-type loop.

On the other hand, Fig. 6 shows an SDF graph that has a C-type loop. For a C-type loop. The C-type loop has two attributes, loop_count and exit_flag. The former is the maximum iteration count and the second is set by a designated

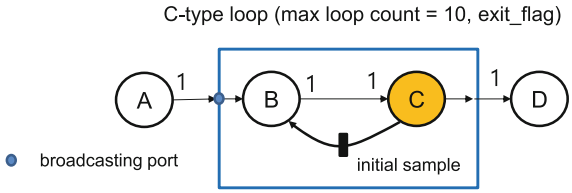


Fig. 6. SDF graph with a C-type loop structure

task, task C in this figure. The number of iteration is dynamically decided by the result of computation that will set the exit_flag. All input ports of a C-type loop should be broadcasting ports from which input samples are reused in all iterations of the loop; the sample rate of the output connection is equal to the sample rate of the input connection.

In summary, in UEM the SDF model is extended to express dynamic behavior with an MTM, to allow the use of shared resources with a library task, and to explicitly specify the loop structures hierarchically. Refer to the corresponding references for more detailed explanation of each extension. Note that these extensions preserve the static analyzability of the SDF model. We perform static scheduling for each mode of operation. In the SDF/L model, static scheduling can be performed hierarchically from the bottom layer. A loop structure is encapsulated as a regular SDF task at the upper layer.

3 Universal Execution Model (UEM)

Figure 7 shows the overall software architecture of the UEM that is layered hierarchically. Each application that is specified by an aforementioned extended SDF model is encapsulated as a dataflow process at the upper layer. We can group a set of dataflow processes in case whose execution states are inter-related. For each application group, a control process is defined in the dynamic behavior of an application group is specified formally by an FSM (finite state machine).

Figure 8(a) represents a multi-mode multimedia terminal (MMMT) application group that contains 8 dataflow processes and 1 control process. Among 8 dataflow processes, 4 processes with pink color have internal dataflow graphs while the other 4 processes with yellow color are single sequential tasks. This application group has the following 4 different modes of operation: Menu, Video player, MP3 player, and Video phone. The *UserInput* task receives a user input to select a mode of operation and sends it to the control process. Based on the selected model, the control process enables a set of applications that run concurrently to serve the mode of operation.

Figure 8(b) shows the FSM specified inside the control process The FSM consists of 4 states that correspond to the modes of operation. In the Video play mode, it enables 2 dataflow processes, H264 decoder and MP3 player. During execution, the mode transition may occur from the Video play mode to Video phone mode if a call is received from the Interrupt task. Then, the control process

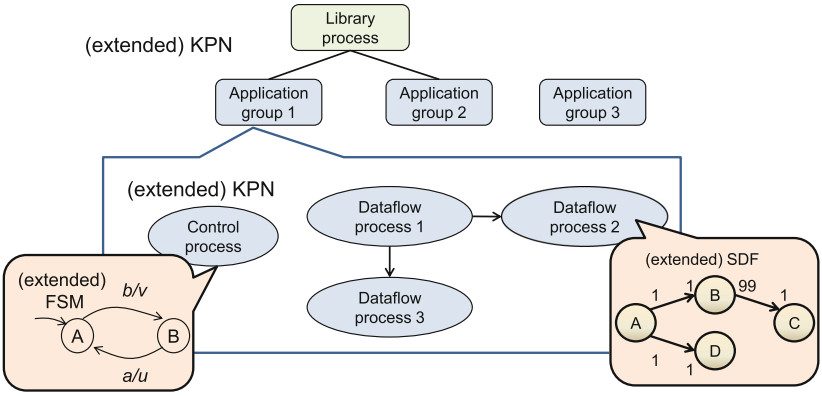


Fig. 7. Software architecture of the UEM

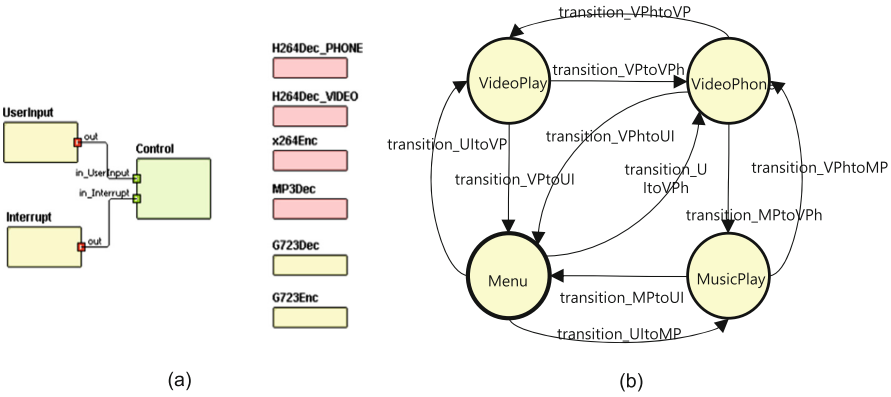


Fig. 8. A multi-mode multimedia terminal application group (a) specification, and (b) the control task specification (Color figure online)

suspends the dataflow tasks of the Video play mode and enables 4 dataflow processes, H264 decoder, x264 encoder, G723 Decoder, and G723 Encoder. After the call is completed, it resumes the suspended dataflow processes of the Video play mode. We can perform model checking to verify the behavior of the control task satisfies the specification at compile time.

In case there are multiple application groups, we can add another layer as shown in Fig. 7. At the top level, each application group is represented by an extended KPN. A process in the KPN (Kahn process network) [8] model allows only blocking read, which makes the KPN determinate, meaning that the execution result is independent of the execution order of processes. To allow shared resources among KPN processes, we extend the KPN with library processes, similarly to the extended SDF model with library tasks.

3.1 Dataflow Task Code Template

In the proposed methodology, a programmer is supposed to specify the system behavior following the software architecture of Fig. 7, starting from dataflow specification of tasks at the bottom layer. As a unit of mapping and scheduling, a dataflow task is a sequential program that should be written with the UEM APIs, based on the coding guidelines defined in the UEM. Figure 9 shows the task code template that consists of three sections, TASK_INIT, TASK_GO, and TASK_WRAPUP. In the current implementation, it is assumed that the task is written in C programming language that is most popular for embedded SW design.

```
TASK_INIT {/* task initialization code */
        port_in = PORT_INITIALIZE(TASK_ID, "in");
        port_out = PORT_INITIALIZE(TASK_ID, "out");}
TASK_GO {
        MQ_RECEIVE(port_in, ...)
        /* main body of the task */
        MQ_SEND(port_out, ...) }
TASK_WRAPUP { /* task wrapup code */ }
```

Fig. 9. A code template of a dataflow task that uses generic APIs for communication

The TASK_INIT section contains the code that will be executed in the initialization stage of the task such as initialization of internal variables and data structures associated with ports. The TASK_GO section is the main body of the task that will be executed repeatedly when it is scheduled by the operating system. The TASK_WRAPUP section is executed just before the task is terminated.

In the TASK_GO section, the task reads the input data from its input ports, perform computation, and sends the output data to the output ports. To make it independent of the hardware architecture and FIFO channel implementation, generic APIs are defined for communication via ports and port initialization as shown in Table 1.

The UEM assumes that there is a hidden supervisor that manages the tasks. We define a set of services that a task can request to the supervisor using a special API, SYS_REQ. The first argument of this API is the service name. Remind that a task may vary its internal behavior depending on the mode of operation in the extended SDF model as explained in the previous section. To change its internal definition, a task can ask the supervisor of what is the current mode; Mode = SYS_REQ(GET_CURRENT_MODE_NAME). A designated task can set the mode by using the following API; SYS_REQ(SET_MTM_PARAM, task_name, var_name, value).

Table 1. UEM application programming interfaces

Task type	API	Description
Common	PORT_INITIALIZE(task_id, port_name)	Initialize a port
	MQ_RECEIVE(channel_id, buffer, buffer_length)	Read data from the FIFO-type channel
	MQ_SEND(channel_id, buffer, data_length)	Write data to the FIFO-type channel
	MQ_AVAILABLE(channel_id)	Check if there is data in the input FIFO
	BUF_RECEIVE(channel_id, buffer, buffer_length)	Read data from the buffer-type channel
	BUF_SEND(channel_id, buffer, data_length)	Write data to the buffer-type channel
	SYS_REQ(service_name, arguments)	Request a service to the hidden supervisor. The first argument of the API designates the service name
Dataflow	LIBCALL(master_port, function_name, function_arguments)	Call a library function from the library task connected through the library master port
Library	LIBFUNC(return_type, function_name, function_arguments)	Define a library function

3.2 Control Task Code Template

A control task is supposed to specify its internal behavior with an FSM. The FSM code template is defined as shown in Fig. 10, which can be automatically generated from the graphic FSM editor in our design environment. In each state, the programmer may use SYS_REQ API to define the control action, which is similar to action scripts of the statechart in STATEMATE [9].

The control services that a control task can request to the supervisor are listed in Fig. 11. The first category is to control the execution status of an application

```
while(1){
   MQ_AVAILABLE(all_ports);                          // 1-1. Check the existence of a new event
   SYS_REQ(CHECK_TASK_STATE, "task_name", …); // 1-2. Check the termination of a task
   if(available) MQ_RECEIVE(selected port);          // 2. read the new event
   if(some event or task state is triggered)   break;    // 3. Break a loop to make transition
}
switch( current_state ) {
   case ID_STATE_S1:
      if(selected port==1 && input data==2) {          // 4. check the transition condition
         current_state = ID_STATE_S2;
         SYS_REQ(SET_PARAM_INT, "task_name", "param_name", data, 0, 0);
      }                                                // 5. send the control message through the system port
      break;
   case ID_STATE_S2: { }
   case ID_STATE_S3: { }
    ….
}
```

Fig. 10. An example code template of a control task in UEM

and the second category is to change or monitor a specific parameter of an application. The third category is defined to specify the timing requirements of the system explicitly.

Category	APIs	Description
Execution Status Control	SYS_REQ(RUN_TASK, task_name);	Run the task
	SYS_REQ(STOP_TASK, task_name);	Terminate the task
	SYS_REQ(SUSPEND_TASK, task_name);	Suspend the task
	SYS_REQ(RESUME_TASK, task_name);	Resume the task
	status=SYS_REQ(CHECK_TASK_STATE, task_name);	Get the current state of the task
Parameter Control	p_value = SYS_REQ(GET_PARAM_INT/FLOAT, task_name, param_name);	Get the value of a task parameter
	SYS_REQ(SET_PARAM_INT/FLOAT, task_name, param_name);	Change a value of a task parameter
Timing Control	SYS_REQ(SET_THROUGHPUT, task_name, thr_val);	Set throughput requirement
	SYS_REQ(SET_DEADLINE, src_task, dst_task, lat_val);	Set deadline requirement to the task chain (src_task to dst_task)

Fig. 11. Control actions that a control task can request to the supervisor

3.3 Library Task Code Template

Figure 12 illustrates code templates associated with a library task. A library task has two separate files associated: a library header file and a library code file. The library header file declares the library functions, while the library code file defines the function bodies. The prototype of a library function is defined by a directive, LIBFUNC(), that will be translated into a regular function definition automatically by the CIC translator.

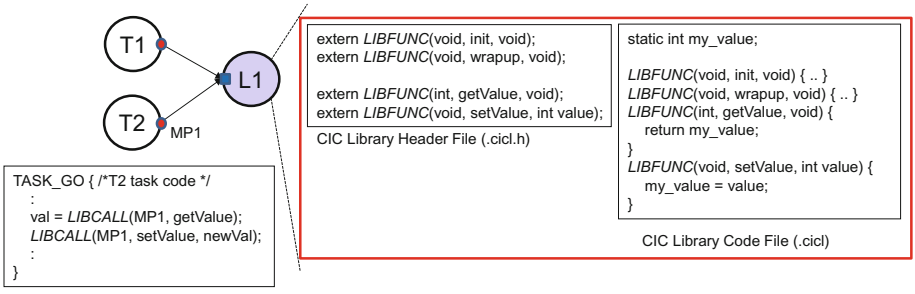


Fig. 12. Code templates associated with a library task

A library task defines init and wrapup functions like a normal SDF task for initialization and finalization of the library task. A caller task uses LIBCALL directive to call a library function as shown in Fig. 12. The first parameter of LIBCALL() is the name of the library master port, the second is the function name, and the others are the arguments. If the function has a return value, it can be taken from the LIBCALL invocation. Note that pointers may not be used for arguments and return values to make the SDF graph portable to a variety of target architectures. For shared address space architectures, however, the

developer may use pointers for efficient implementation, giving up portability. A library task may have a persistent internal state, simply called a state. Then the access to the state should be protected by synchronization primitives, Lock() and Unlock() to avoid data race problems.

4 Automatic Code Generation

Based on the mapping and the scheduling result, we can generate the target code automatically from the UEM assuming that the task code inside each node of a SDF graph and the control task is given and correct. It remains as a future work to check the correctness of each task.

We can synthesize the communication and interface code between tasks as well as the scheduling code automatically. Since the HW/SW interface code and the task synchronization code are particularly error-prone, automatic synthesis of those codes will alleviate the burden of the programmer significantly. Moreover, by keeping the SDF semantics, the synthesized code is guaranteed to be free of buffer overflow and deadlock error. Then, functional verification of embedded software can be performed by verifying the functional correctness of each task only. Since each task is a sequential code, we can use the state-of-the-art verification techniques of a sequential code, which is complementary to the proposed methodology.

Another benefit of automatic code generation from the UEM is that we can add extra software modules to enhance the reliability or the safety of the software. Even though the software is designed based on the UEM without consideration of any possibility of hardware failure, we apply fault-tolerant techniques to insert extra codes to the generated target code, while satisfying the real-time requirements and the resource constraints.

If the efficiency of the automatically generated code is much worse than the manually optimized code, people may prefer manual coding even with the higher risk of error to automatic code synthesis because embedded systems are usually cost-sensitive. Since the internal code of each task is assumed to be optimized, the overhead will be associated with inter-task communication if exists. For efficient code generation, we may use several techniques that have been developed to minimize the buffer size when constructing a static schedule of an SDF graph [10,11].

Note that code generation is specific to the system architecture that runs the application. Then a key challenge in the proposed methodology is how difficult is to make the UEM compiler to synthesize the software automatically for a given architecture. If the difficulty of making the UEM compiler is higher than that of developing the software manually for a given architecture, the proposed technique will be of no use. For UEM compilation, following the well-established procedure of traditional compilation, we separate the platform-independent part and the platform-dependent part of UEM compilation. By pre-defining the HW-specific interface as software component libraries, we simplify the platform-dependent part maximally. From our experience, we expect that it will take less than a month to make a UEM compiler for a new hardware platform.

5 Preliminary Experiments

The proposed methodology has been applied to the development of a parallel embedded software design framework, called HOPES [12]. Specification and parallel scheduling of the MMMT application group in Fig. 8(a) can be found in [12]. The reference also presents the scheduling and mapping result of a lane detection algorithm for a CPU-GPU heterogeneous system. In this section, we present two more examples that use library tasks.

5.1 A Cryptographic System

Figure 13 shows the captured screen of HOPES that specifies a cryptographic system following the UEM software architecture. The pink task represents two dataflow processes that have an extended SDF graph inside as displayed in the figure. Two tasks, *Encryption* and *Decryption*, call library functions inside to request the service of a library task, *CrytographyLibrary* that provides a set of service functions for cryptography. The *Control* task activates the *Sender* task if it receives a user input and the *Sender* task packs the input data, encrypts the packed message, and transfers the encrypted message. If the control task is triggered by an incoming message, it decrypts, unpacks and displays it.

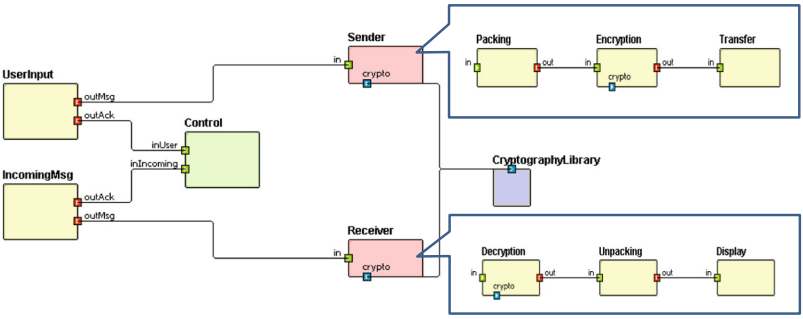


Fig. 13. A cryptographic system example (Color figure online)

5.2 Cooperating Robots

Figure 14 specifies a multi-robot system where multiple robots accomplish a mission, sharing the information. The mission in this experiment is to find all color papers scattered on the floor whose boundary is marked by black tapes. While each robot searches color papers in the region independently, the found papers are reported to the library task to avoid the redundant labor of robots. When all papers are found, the robots go back to the initial position. Each robot performs a group of applications that are depicted in the figure. An application group consists of 8 tasks: 3 sensor tasks, 4 actuator tasks, and 1 control task.

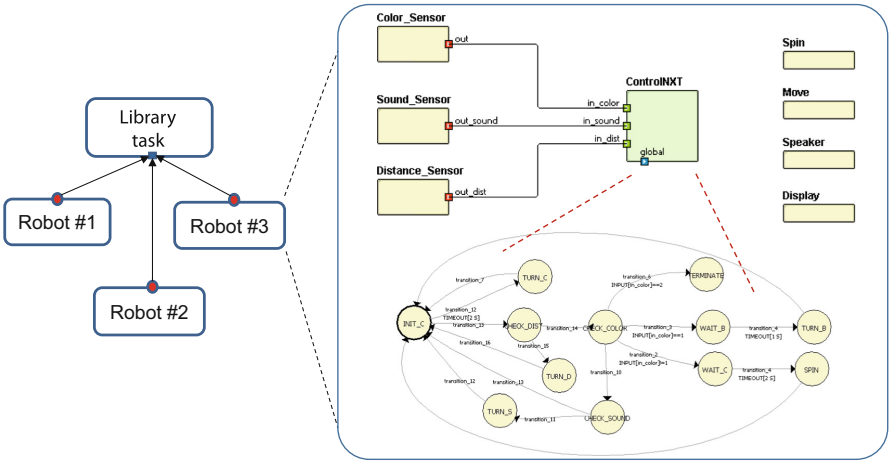


Fig. 14. A cooperative robots example (Color figure online)

In this experiment, three different types of robots used: *TI Evalbot*, *NXT LEGO*, and *iRobot Create*. The robots have different hardware platforms and operating systems as shown in Fig. 15. The library task is mapped to *iRobot Create* that is most powerful. The figure also shows the distribution of code size. In addition to the task code given by the user, the scheduler code, data structure, and communication codes are automatically generated. In this example, the task code takes about 21.9% of the total code size on average. Since the coding error probability is known to be dependent on the code size in general, it can be said that automatic code synthesis increases the design productivity of this control oriented application.

	TI Evalbot	NXT LEGO	IRobot Create
OS	uC-OS III	NXT-OSEK	Linux (Ubuntu)
Proc.	LM3S9B92 (80MHz)	Atmel® 32-bit ARM (48MHz)	Intel i3-4000M (2.4GHz)
Mem.	96KB	64KB	4GB
Total code size	2721	3301	2996
Given task code	580	604	787
Scheduler code	1171	928	931
Other code	970	1,769	1,278

Fig. 15. Robot hardware specs and synthesized code size

6 Related Work

Defining a universal execution model is not a unique idea of the proposed methodology. Several executions models have been proposed in various application domains, to make the software independent of the hardware platform and so portable to different types of architecture easily. A good example is the AUTOSAR (AUTomotive Open Software Architecture) methodology that defines the open and standardized software architecture for automotive electronic control units [13]. AUTOSAR defines a set of APIs assuming that the software components communicate with each other through virtual sockets. Thus, software developer can design software using the APIs, without knowledge of the underlying hardware platform, which is the same as the proposed technique that provides the UEM APIs to the programmer. After mapping of software components onto the ECUs is determined, the runtime environment supervises the execution of software components and communication between them. Since the AUTOSAR is not based on formal models of computation, however, this methodology resorts to test-based methods for software verification.

The proposed methodology has been evolved from a hardware/software codesign methodology where the behavior specification of a system is made separately from architecture specification. In this codesign methodology, formal models of computations are widely for behavior specification since they make it easy to explore the wide design space of architecture configuration and mapping of the application to the processing elements. In case the hardware platform is given, the design space is reduced to find an optimal mapping of the application and the software code for each processing element is automatically generated based on the mapping and scheduling decision. In other words, the HW/SW codesign methodology becomes an embedded SW design methodology if the hardware platform is fixed.

Nonetheless, the proposed methodology differs from conventional model-based codesign environments such as Daedalus [14] and DAL [15]. While they use the KPN model for behavior specification, the UEM model is defined as the execution model at the operating system level, combining three different models of computation. Its model composition rule is different from that of Ptolemy [16] which allows hierarchical composition of models without limitation on the depth of hierarchy and the kinds of models. Last, not the least, a major difference between the codesign methodology and the software design methodology is the granularity of the atomic actor. The atomic actor is as large as a function module that can be implemented as a hardware component in the codesign methodology, while it is larger in the software design methodology as a sequential task that is a unit of mapping and scheduling at the operating system level.

7 Conclusion

For the design of embedded software, we have to ensure not only the functional correctness but also satisfaction of several constraints on real time performance

and resource limitation. In this work, a novel methodology to make an embedded software correct by construction is proposed by designing embedded applications with formal models of computation. Unlike the conventional model-based design, formal models of computation are applied to the software architecture of tasks that are mapped and scheduled by the operating systems. Thus, the proposed software architecture can be understood as a universal execution model (UEM) of underlying hardware platforms. We define the UEM by *extending* well-known formal models, Synchronous Dataflow (SDF) for the computation parts of the system and finite state machine (FSM) for the control structure of the system. At the top level, an extended KPN (Kahn process network) is used to define the interaction between applications. To be concrete, the SDF model is extended to specify dynamic behavior by combining a FSM model, called MTM (mode transition machine), to allow the use of shared resources by defining a new type of task, called library tasks, and to express loop structures explicitly by defining a loop super node to make the SDF model hierarchical.

There are several benefits to use formal models for software design. First, we can detect critical design errors such as deadlock and buffer overflow by static analysis of formal models. Second, we can estimate the resource requirement and real-time performance at compile time. Last, not the least, we can synthesize the target code from the UEM automatically minimizing the manual coding efforts. By preserving the semantics of the UEM, the synthesized code will be correct by construction. The key challenge lies in the expression capability of the proposed UEM. Preliminary experiments with several non-trivial applications prove the viability of the proposed methodology.

8 Epilogue

I am very grateful that I was involved in the development of Ptolemy [16] from its birth during my doctoral study. Under the supervision of Prof. E.A. Lee, I developed and implemented several models of computations and their hierarchical structure. Naturally, I became an advocate of formal models of computation and their mixture for system specification and simulation. After joining the faculty of SNU (Seoul National University, Korea), I switched my gear to the HW/SW codesign of embedded systems and had developed a HW/SW codesign environment, called PeaCE (Ptolemy extension as a Codesign Environment) [17] for the first 12 years. As the name implies, the baseline of the environment is Ptolemy classic. Since our aim was to synthesize the system automatically from the behavior specification, we had to restrict the use of formal models and their composition. Our choice was to use SDF and FSM models since they offer good static analyzability and the refinement path from specification to implementation is well established. To overcome the limitation of expression capability, we have proposed several extensions to those models. The viability of the proposed approach was proven with a design of a simple smartphone application.

As the number of processors integrated into a chip increases and platform based design becomes popular, parallelizing software becomes more challenging

than partitioning an application into hardware and software. Since the PeaCE environment was not well engineered from the start, graduate students had difficulty of maintaining the environment. So we decided to develop a new design environment, HOPES [18], from scratch, focusing on the development of parallel embedded software based on the formal models of computation, keeping the spirit of Ptolemy and PeaCE. Since a hardware component can be regarded as a special processing element that can perform a designated task only, the HOPES environment can be used as a HW/SW codesign environment. By increasing the granularity of a task, it is easier to use formal models for behavior specification. Another 12 years have passed. We are now renovating the HOPES environment. Our goal is to make the HOPES environment as a software engineering tool that can be adopted in the industries.

Acknowledgments. This research was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIP) (No. NRF-2016R1A2B3012662). The ICT at Seoul National University provided research facilities for this study.

References

1. Lee, E.A., Messerschmitt, D.G.: Synchronous data flow. Proc. IEEE **75**, 1235–1245 (1987)
2. Bilsen, G., Engels, N., Lauwereins, R., Peperstraete, J.: Cyclo-static dataflow. IEEE Trans. Signal Process. **44**, 397–408 (1996)
3. Stuijk, S., Geilen, M., Theelen, B.D., Basten, T.: Scenario-aware dataflow: modeling, analysis and implementation of dynamic applications. In: Proceedings of International Conference on Embedded Computer Systems: Architecture, Modeling, and Simulation, vol. 72, pp. 404–411 (2011)
4. Bhattacharya, B., Bhattacharyya, S.: Parameterized dataflow modeling for DSP systems. IEEE Trans. Signal Process. **49**, 2408–2421 (2001)
5. Jung, H., Lee, C., Kang, S., Kim, S., Oh, H., Ha, S.: Dynamic behavior specification and dynamic mapping for real-time embedded systems: HOPES approach. ACM Trans. Embed. Comput. Syst. **13**, 135:1–135:26 (2014)
6. Park, H., Jung, H., Oh, H., Ha, S.: Library support in an actor-based parallel programming platform. IEEE Trans. Ind. Inform. **7**, 340–353 (2011)
7. Hong, H., Oh, H., Ha, S.: Hierarchical dataflow modeling of iterative applications. In: Proceedings of Design Automation Conference, vol. 39 (2017)
8. Kahn, G.: The semantics of a simple language for parallel processing. In: Proceedings of the IFIP Congress (1974)
9. Harel, D., Naamad, A.: The STATEMATE semantics of statecharts. ACM Trans. Softw. Eng. Methodol. **5**, 293–333 (1996)
10. Shin, T., Oh, H., Ha, S.: Minimizing buffer requirements for throughput constrained parallel execution of synchronous dataflow graph. In: Proceedings of Asia and South Pacific Design Automation Conference (2012)
11. Oh, H., Ha, S.: Memory-optimized software synthesis from dataflow program graphs with large size data samples. EURASIP J. Appl. Signal Process. **2003**, 514–529 (2003)

12. Ha, S., Jung, H.: HOPES: programming platform approach for embedded systems design. In: Ha, S., Teich, J., et al. (eds.) Handbook of Hardware/Software Codesign, pp. 951–981. Springer, Dordrecht (2017). https://doi.org/10.1007/978-94-017-7267-9_1
13. Pelz, G., Oehler, P., Fourgeau, E., Grimm, C.: Automotive system design and AUTOSAR. In: Boulet, P. (ed.) Advances in Design and Specification Languages for SoCs, pp. 293–305. Springer, Boston (2005). https://doi.org/10.1007/0-387-26151-6_21
14. Nikolov, H., et al.: Daedalus: toward composable multimedia MP-SoC design. In: Proceedings of Design Automation Conference, pp. 574–579 (2008)
15. Schor, L., Bacivarov, I., Rai, D., Yang, H., Kang, S.-H., Thiele, L.: Scenario-based design flow for mapping streaming applications onto on-chip many-core systems. In: Proceedings of CASES, pp. 71–80 (2012)
16. Buck, J.T., Ha, S., Lee, E.A., Messerschmitt, D.G.: Ptolemy: a framework for simulating and prototyping heterogenous systems. Int. J. Comput. Simul. **4**, 155–182 (1994)
17. Ha, S., Kim, S., Lee, C., Yi, Y., Kwon, S., Joo, Y.: PeaCE: a hardware-software codesign environment for multimedia embedded systems. ACM Trans. Des. Autom. Electron. Syst. **12**, 24:1–24:25 (2007)
18. Kwon, S., Kim, Y., Jeun, W., Ha, S., Paek, Y.: A retargetable parallel programming framework for MPSoC. ACM Trans. Des. Autom. Electron. Syst. **13**, 39:1–39:18 (2008)