System Level Specification and Software Synthesis of Multimedia Embedded Systems: PeaCE Approach[1]

Soonhoi Ha
sha@snu.ac.kr
Seoul National University, Korea

Dohyung Kim
dhkim@ucsd.edu
University of California, San Diego, U.S.A.

ABSTRACT
In this paper we present a hardware/software codesign flow in which embedded software code is automatically generated from system level specification of multitasking multimedia embedded system, both for simulation and implementation. In the proposed system design methodology, the system behavior is specified with a heterogeneous mixture of formal models of computation: a dataflow model to specify the internal behavior of a signal processing task, an FSM model to specify a control task. At the top level, we introduce a novel task-level specification model to represent diverse task execution semantics and communication protocols. The generated software has a layered structure using virtual OS APIs and OS wrapper implementations to make it reconfigurable for multiple target platforms. Experiments with a Divx player example prove the viability of the proposed technique.

Keywords
System level specification, HW/SW codesign, software synthesis, data flow model, embedded software

I. INTRODUCTION
As development of implementation technology is not slowed down, design of multimedia embedded systems becomes more challenging due to the increasing system complexity as well as relentless time-to-market pressure. Conventional practice of designing an embedded system performs algorithm design, hardware design, and software design separately and sequentially. Past design experiences and some performance profiling techniques are resorted to determine the hardware architecture. Then we manually partition the system behavior to the processing components and perform software development and hardware prototyping concurrently. After a hardware prototype is made, developed software codes are downloaded into the processors and verified. Since the expected growth rate of design productivity in this conventional design method is far below that of system complexity, hardware/software (HW/SW) co-design has emerged as a new design methodology in the past ten years.

As a systematic system-level design methodology, HW/SW co-design methodology no longer separates hardware design and software design. To determine the hardware architecture, we explore a wide range of feasible architectures and evaluate each one by estimating the expected performance after considering all subsequent design decisions such as hardware/software partitioning and software implementation. A systematic design space exploration needs separate specification of functions and architectures. An explicit mapping step maps parts of functional specification to architecture building blocks. After mapping is completed, we estimate the expected performance and may trigger further iterations of architecture specification and mapping until we find an optimal architecture. After an optimal architecture is found and mapping is decided, more accurate performance verification is needed before final hardware implementation. Thus hardware/software co-design includes various design problems including system specification, hardware/software partitioning, performance estimation, hardware/software co-verification, and system synthesis. A co-design environment is a software tool that facilitates capabilities to solve those design problems. Figure 1 illustrates a generic flow of hardware/software codesign procedure.

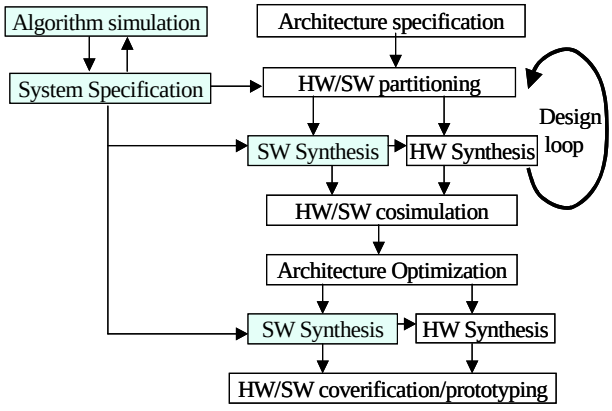


Figure 1 Hardware/Software codesign procedure

[1] This paper is an extended version of conference papers [1]and [2].

HW/SW co-design process starts with system specification problem: how to specify the system behavior. While a sequential "C" code description has been probably most preferred for functional simulation in conventional design procedure, it is not adequate for initial system specification in HW/SW co-design. Since a system is implemented as a collection of concurrent components such as programmable processors, ASICs, and discrete hardware components, models of computations that express the concurrency naturally are preferred for initial specification. In the proposed codesign methodology, we start with a behavioral level specification of the entire system using block diagram representation for functional simulation and static analysis of some system properties. Then, an optimal architecture is searched for through a systematic design space exploration procedure and system behaviors are partitioned and mapped to the processing components of the target architecture. Finally, C codes for software components and RTL level HDL codes for hardware components are automatically synthesized from the behavioral level specification. *In this paper, we focus on software synthesis from the behavioral level specification.*

As a test vehicle for system level specification, Figure 2 shows a multi-mode multimedia terminal (MMMT) system. An embedded system is called multi-mode when it supports multiple applications by dynamically reconfiguring the system functionality.

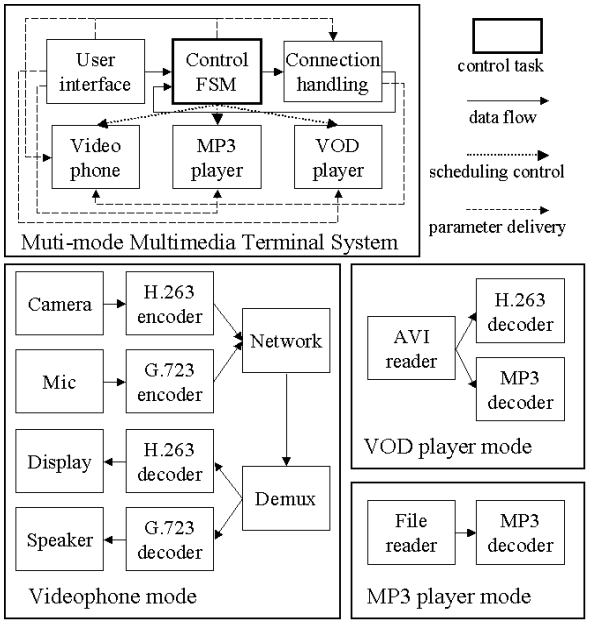


Figure 2 Multimode Multitasking Multimedia Terminal (MMMT) example

At the top level, three hierarchical nodes are defined to specify each mode of operation: a videophone, a VOD (Divx) player, and an MP3 player. Three other tasks are also specified at the top level for user interface, connection handling, and task execution control. And, each mode of operation consists of multiple tasks as shown in the figure. An example operation scenario is as follows; (1) A user determines the mode of operation with arguments, for instance the videophone mode with a host address name. The user interface task reads and delivers this information to the control task and to the connection handling task; (2) The control task now activates the mode by activating all the tasks that compose the selected mode after deactivating the previous mode; (3) The connection handling task takes care of network management such as connection establishment and message exchange; (4) The control task listens to the status signal from the current mode of operation and stops all the component tasks when termination or exception signal is received. We distinguish data, control, and parameter flows between tasks in Figure 2 while only data flow arcs are visible in the real implementation.

Behavioral level specification of such an MMMT system is very challenging with the following key requirements;

(1) Tasks of MMMT system have diverse execution semantics: data-driven, event-driven, and time-driven. The connection handling task is waked up in the event-driven fashion while the decoder tasks are usually time-driven for the constant rate of output production. Some encoder tasks are data-driven, triggered by the arrival of input data stream from the predecessor task.

(2) There are various kinds of interactions between tasks with different synchronization requirements. Scheduling control information from the control task to a mode is asynchronous while data flow between tasks should be delivered in a synchronous fashion. Parameter delivery is also asynchronous.

(3) A single task may need to be partitioned into multiple processing components to meet the timing constraint though this case is not considered in this paper. For example, the motion estimation block of H.263 encoder task should be mapped to a hardware block. Therefore functional decomposition of the inside of a task is desirable for design space exploration.

In this paper, we propose a system level specification for MMMT system using a composition of diverse models of computation, satisfying those requirements. It uses dataflow model and FSM model to specify the internal behavior of a signal processing task and a control task respectively. Dataflow and FSM models are chosen for internal specification of tasks because they have well-understood formal semantics and extensive research results on automatic software synthesis. There are several benefits of using formal models for specification as observed by many researchers [3]: (1) Static analysis of the specification allows one to verify the correctness of the functional specification. (2) The specification model can be

refined to an implementation to ease the system validation by the principle of "correct-by-construction". (3) A formal specification model is not biased to any specific implementation method, so allows one to explore the wider design space. (4) It represents the system behavior unambiguously so that collaboration and design maintenance can be accomplished easily.

Nonetheless using formal models of computation has not gain wide acceptance mainly because of limited expression power and inefficient synthesis results. We overcome the problems of limited expression capability and of inefficiency by extending the existent dataflow and FSM models.

At the top level, we devise a novel task-level specification model. This task model allows the user to represent diverse task execution semantics and communication protocols between tasks. We observe that the data size and the data rate of an input port may be unknown before task execution. Such dynamic rate of input consumption should be supported in the task model.

In this paper, we demonstrate the viability of the proposed approach by showing that the software implementation of the complex MMMT system is fully automated from the behavioral-level specification and the code quality is satisfactory though there is still much room for further optimization.

To our best knowledge, it is the first demonstration of fully automated system level design of a complicated embedded system design from behavioral level specification thus can be used for comparative study with other system-level design work: how easy to represent the MMMT system and how good is the automatically implemented system. This is the main contribution of this work. Moreover, we have developed an architecture independent code synthesis framework by using virtual operating system APIs and task wrappers.

The rest of paper is organized as follows. Section 2 briefly describes the related works. Section 3 explains the inside details of system specification of the example MMMT system. Software synthesis from the system level specification is discussed in section 4. And we show experimental results with the MMMT example and a real Divx player implementation in section 5 and draw conclusions in sections 6.

II. RELATED WORK

Researches about system level specification can be divided into two groups. One group uses formal models of computation for the sake of easy design validation by static analysis and automatic code synthesis. SDL [4] and CFSM [5] models are devised for control-oriented reactive system specification and they are both extended from FSM model. They are not appropriate to describe signal processing algorithms. ACFSM [6], TinyGALS [7], and Kahn process network [8] models have been proposed to describe both control and signal processing tasks with a single computation model. But it is doubtful whether a single computation model can satisfy all the requirements of system-level specification for a complex embedded system like the example MMMT system.

Ptolemy II [9] and El Greco [10] use hierarchical compositions of diverse models of computation by arbitrary nesting. Even though they provide strong capability of system specification and functional simulation, no result on the automatic software (or hardware) synthesis from the behavior level specification has been reported. In fact, our proposed approach is similar to them in that it also uses the idea of composing diverse computation models in a single framework. It is distinguished, however, that it uses different mechanism of model interaction and it provides the automatic model refinement capability to real software and hardware implementation

There are some other approaches of using separate representation of control and computation tasks. STATEMATE [11] and FunState [12] are two examples of this category. In their approach, the central FSM model describes the entire system behavior while task models do not possess formal semantics. Since task executions are controlled by the FSM model, complicated task scheduling should be described manually in the FSM model.

Because formal semantics restrict their communication and execution rules, formal models of computation usually have limitation in their expression power as well as synthesis capability. It should be very challenging to specify and synthesize the example MMMT system in those approaches of using formal models of computation.

Therefore, the other group concentrates on system level description using the existent program language or extending it to specify concurrent processes and communication via channels. SystemC [13] and SpecC [14] are well-known two examples of system description language. They usually specify a system as a network of function blocks but without enforcing any formal semantics on the task execution and the inter-block communication. In this approach, the system level design steps such as functional simulation, design space exploration, and code generation, are independently performed. And, performance analysis and system validation are performed by simulation. Thus, we lose all advantages of using formal models of computation. In fact, using these languages and using formal models of computation is not contradictory. Therefore, using formal models of computation on top of these languages is recently pursued [13].

The proposed approach extends the existent formal models of computation to greatly enhance the expression power and synthesis capability while not losing analytical

properties of formal models. Thus we retain all the advantages of using formal models.

On the other hand, several research efforts have been made recently to generate embedded software automatically from system level specification. They are distinguished from each other by the specification model itself and target platform of the generated code.

Timed Multitasking(TM) [15] model is an event-triggered real-time programming model in which a task, called an actor in the TM model, is triggered by external events or data arrivals from other tasks. The TM model is similar to ours in that an actor communicates with other actors only through ports: it means that no shared resource is accessed during the execution of tasks. The key feature of TM model is that the timing properties of tasks can be made deterministic. By controlling the time at which outputs are produced and triggering tasks with new events, both the start and the stop time of each task are effectively controlled. But this model does not support non-blocking IO of input ports, which is needed for MP3 decoder task since the size of input data stream is unknown a priori. On the other hand, our task model supports both blocking and non-blocking IOs.

TinyGALS[7] is a programming model for event-driven embedded systems in which a set of modules communicates asynchronously through message passing. Within each module, components communicate via synchronous method calls. This approach is similar to our approach in that it has separate communication mechanisms for inter- and intra-task communication. But this model supports only event-driven semantics for task initiation while our model allows various types of execution semantics. Since the model is originally proposed for wireless sensor network, the generated code does not assume operating system support of module communication and global variable accesses.

SoCOS[16] is a SystemC-based design environment. The system behavior is specified with communicating processes and a process is triggered by time, not by events, which is different from our task model. Communication between processes is achieved through ports and synchronization is implemented by channel similarly to ours. From the specification, SystemC code is generated for simulation. But for the implementation code, SystemC code definition of each model is translated to RTOS code automatically identifying the statements to be translated. On the other hand, in the proposed technique, we do not translate the task code but redefining the virtual OS APIs that are used in the task code.

III. SYSTEM LEVEL SPECIFICATION

In this section, we review the proposed system level specification with the MMMT example. It is a heterogeneous mixture of models of computation based on the Ptolemy work [17]. At the top level we use a task-level specification model where each block represents a task. Each task model may have internal specification model to specify the internal behavior; we use a dataflow model for signal processing tasks and an FSM model for the control task.

III.1 Extended Dataflow Model

For internal task specification, we extended the existent formal models of computation to greatly enhance the expression power and synthesis capability while not losing analytical properties of formal models. Thus we retain all advantages of using formal models.

In a hierarchical dataflow program graph, a node, or a block, represents a function that transforms input data streams into output streams. An arc represents a channel that carries streams of data samples from the source node to the destination node. The number of samples produced (or consumed) per node firing is called the output (or the input) sample rate of the node. In case the number of samples consumed or produced on each arc is statically determined and can be any integer, the graph is called a synchronous dataflow graph (SDF) [18] that is widely adopted in numerous DSP design environments.

But, the SDF model has a couple of serious limitations to represent multimedia applications: First, the SDF model can not express the data dependent node executions such as "if-then-else" and "for" constructs. Second, the efficiency of the synthesized code is noticeably worse than the hand-optimized code in terms of memory requirements. Therefore, we make two extensions to the SDF model to overcome these problems, which are fractional rate dataflow (FRDF) for efficient code generation minimizing buffer size and synchronous piggybacked dataflow (SPDF) to express the global states and data dependent execution. For detailed discussion on the extended models, refer to [19] for FRDF and [20] for SPDF.

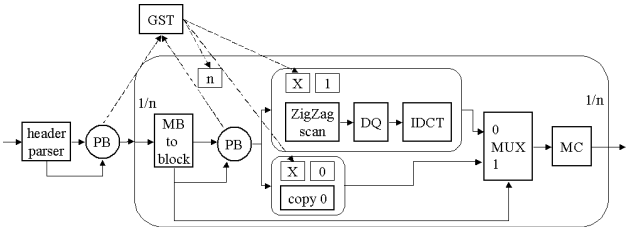


Figure 3 Simplified H.263 decoder specification using extended SDF model (SPDF)

Figure 3 shows a simplified H.263 decoder specification using the SDPF model to express data dependent behavior. The SPDF model has a special block, called piggyback (PB) block. It is a sole writer to a global variable that can be read by multiple downstream blocks. And those downstream blocks can make data dependent execution decision based on the global variable. When the header parser block reads one frame data, it divides the frame into

a number of macroblocks depending on the frame size. Therefore the decoding body is placed in a data dependent iteration loop in the generated code. Such a data dependent iteration is specified using a piggyback block (PB). The piggyback block puts the iteration number, *n*, into a global structure, or the global state table (GST) in the figure. The loop body is encapsulated in a hierarchical node and the data rate at the boundary ports is specified as a fractional number *1/n* meaning that *n* executions of this node consumes *1* data sample (one frame in this example). Inside the "for" construct, "if-then-else" construct specifies a conditional execution. When the current frame is the same as the previous frame, the computation for zigzag scan, dequantization (DQ) and inverse discrete cosine transform (IDCT) can be skipped. The condition variable is put into another global state entry by the inside PB block. The "if" body and the "else" body are conditionally executed by referring to the condition variable in the GST. Figure 4 shows the generated code structure from Figure 3.

```
header_parser();
for (i=0; i<n; i++) {
    MB_to_block();
    if (X==1) { ZigZag_scan(); DQ(); IDCT(); }
    else if (X==0) copy 0;
    MC();
}
```

Figure 4 Code template generated from Figure 3

III.2 Extended FSM Model

The pure FSM representation suffers from *state explosion problem* to represent a system with concurrent modules or memory. To avoid this problem, we devise an extended FSM model, called flexible FSM (fFSM), to express the state transition behavior of a control task [21]. The proposed fFSM model supports concurrency, hierarchy, and internal event mechanism as Harel's Statechart does. Unlike Statechart, however, we forbid inter-hierarchy transition to make it more modular and compositional

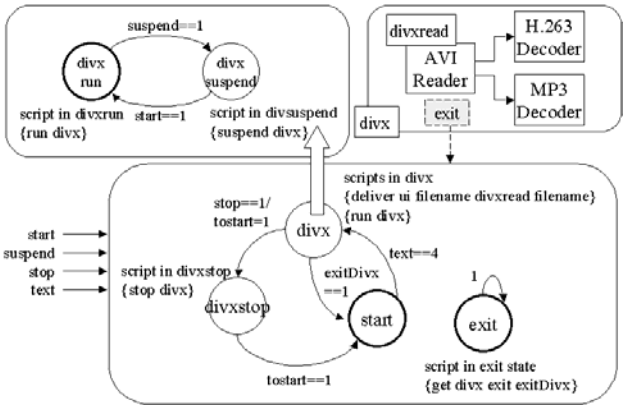


Figure 5 Control task specification based on the fFSM model in Divx mode

Figure 5shows an example FSM to control the VOD(divx) mode, which consists of two concurrent FSMs and one hierarchical FSM and gets four inputs from a user interface task. Figure 5 also describes the task model of the divx mode Similar to the statechart, a state may have action scripts to define the control interactions between the FSM and the computation tasks. The action scripts are executed only when the state transition occurs. A bolded circle represents the default state of the FSM.

Consider the following scenario to see how the proposed FSM model works. If a user specifies a file name and starts a VOD mode, the FSM input port *text* will have a new value *4*. Then a state transition occurs to the *divx* state. At the same time, the scripts in the *divx* state are processed sequentially. The first script delivers the *filename* state in the user interface task to the *filename* state in the *divxread* task. The next starts a group of tasks in the *divx* mode. When the control FSM is at the *divx* state, the hierarchical FSM as shown in upper side of Figure 4 becomes active and changes the status of the *divx* mode by user controls. If a user puts the stop command to the FSM, the state changes to the *divxstop* state and a script stops the *divx* mode. The internal event *tostart* invokes another state transition to the *start* state. If one of tasks in the *divx* mode meets a completion condition or an error, it sets the *exit* state in the *divx* mode by calling the exception handling routine, and it terminate the *divx* mode and triggers the control FSM. In the *exit* state, the transition is triggered whenever the control FSM is called and the script in the *exit* state signals the internal event *exitDivx*. Then the state transition occurs from the *divx* state to the *start* state. As illustrated in this example scenario, scripts specify the control interactions between the FSM task and the dataflow tasks. Table 1 describes the supported script languages. More formal treatment of fFSM model can be found in [21].

Table 1 Script language of fFSM model

Script	Actions

run *n_name*	run *n_name* [task or mode]
suspend *n_name*	suspend *n_name* [task or mode]
stop *n_name*	stop *n_name* [task or mode]
set *n_name n_value X*	set *n_value* state with value *X* in *n_name* task
get *n_name n_value n_event*	get value of *n_value* state in *n_name* task and set it to *n_event* event in FSM
deliver *n_src n_srcval n_dst n_dstval*	deliver value of the *n_srcval* state in *n_src* task to *n_dstval* state in *n_dst* task

III.3 Task-level Specification

Tasks in the MMMT system have diverse activation conditions and port semantics that should be clearly specified in the task-level specification model at the top-level. For this purpose, a novel task-level specification model, shortly task-model, is proposed. In the proposed task model, synchronization and communication between tasks are performed at only ports as message boundary. Each task accesses shared resources explicitly through ports and internal states are not directly accessible outside the task. While this task model is somewhat restricted compared with the general task model assumed in an operating system, it is free from priority inversion problem and race condition. So task scheduling and synchronization between tasks are greatly simplified. Note that data flow and FSM models satisfy these restrictions.

Table 2 summarizes the classification of supported task types and port properties. We support three types of tasks depending on the triggering condition: periodic tasks triggered by time, sporadic tasks triggered by external IO, and function tasks triggered by data. And we also define various port semantics by combining port type, data size, and data rate. For example, the data port semantic for SDF model is classified as a static-rate static-size queue and the data port semantic for FSM model as a dynamic-rate static-size queue. Especially, we support dynamic-rate variable-size port semantics to express the condition that the sample rate is determined at runtime.

Table 2 Classification of task types and port properties

Task type	Periodic	Triggered by time
	Sporadic	Triggered by external IO
	Function	Triggered by data
Port type	Queue	Destructive read/non-destructive write
	Buffer	Nondestructive read/destructive write
Data size	Static	Size is determined at compile time
	Variable	Size is determined at run time
Data rate	Static	Rate is determined at compile time
	Dynamic	Rate is determined at run time

Remind that the internal behavior of a task is represented as a dataflow model or FSM model. On the other hand, the task itself has diverse execution semantics. So we provide the mechanism to connect the internal SDF and FSM models to the outer task-model. A task wrapper is created at the boundary of hierarchical block(or a task block) that translates the outer execution semantics to internal execution semantics. It also contains a semantic translator for each port of the internal model.

For example, if a dataflow task is defined as a periodic task with a static-size static-rate buffer port, arrival of input data is ignored while the arrived data is stored in the port buffer. Instead the current values stored in the input port buffers are delivered to the inside at periodic wake-up.

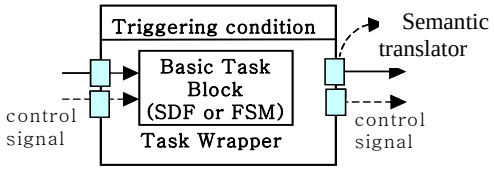


Figure 6 Task wrapper

Figure 7 demonstrates the sequence of task wrapper creation using the example of Figure 5. (1) Initially, sporadic and periodic tasks are notified by the user. (2) The task wrapper defines port semantics for each data port of SPDF and fFSM models. In this example, data ports between the user interface task and the control FSM task are defined as dynamic-rate static-size buffer-type. (3) It appends control ports to the SPDF or fFSM task by analyzing the scripts in the fFSM model and also defines a port semantic for each type of control port. In Figure 5 (b), two control connections are created between the control task and the divx mode to exchange the control command and the status signal. (4) At last, it appends an additional control port associated with the task type. Through this port, the supervisory task gives commands to change the execution status of the task. We summarize the port semantics for the task wrapper in Table 3.

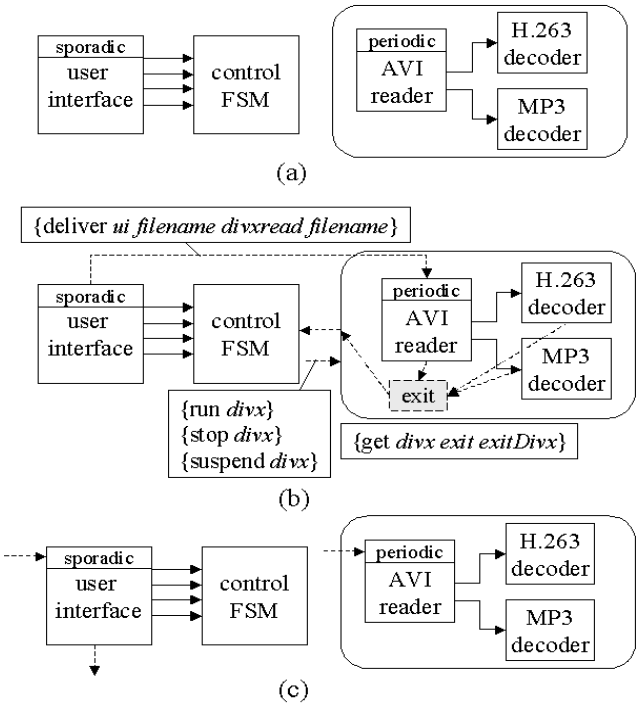


Figure 7 (a) Task level specification of the Divx mode to which control ports are appended by (b) control path analysis and (c) task types

Table 3 Port semantic definition for the task wrapper

Port type	Port semantic
Data port of SDF model	Static-rate static-size queue
Data port of FSM model	Dynamic-rate static-size buffer
Scheduling control port	Static-rate static-size queue
Parameter delivery port	Static-rate static-size buffer
Exception handling port	Dynamic-rate static-size queue
Control port for periodic task	Static-rate static-size queue
Control port for sporadic task	Static-rate static-size queue

During the task-level specification of the MMMT system, we meet a need to specify a new port semantics with the dynamic-rate variable-size (DRVS) port type in the dataflow task. In the MP3 decoder task, the required data size of one encoded frame cannot be determined until the MP3 decoder task starts decoding and detects it at runtime. In Figure 8, the sample rate x is determined at runtime by the MP3 decoder itself. No existent model of computation can specify this port semantics to our knowledge. But, the proposed task model allows dynamic-size ports to specify this case.

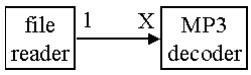


Figure 8 Dynamic rate variable size port type

Note that port semantics and task semantics only define the behavior not the implementation of the task model. In the next section, we explain how to automate software implementation from the task model.

IV. Software Synthesis

Depending on the design step, simulation or implementation, we synthesize different versions of software code from the same task-model while the inside of a dataflow or control task is unchanged. Such reconfiguration is achieved by a layered software structure as shown in Figure 9. The task structure as shown in Figure 9(b) is automatically generated from the task wrapper for each task. It is a general task code structure that can be tailored to different software implementations. Task types and port properties from the task-model are defined in the structure. A main task code is separated into four methods to support dynamic reconfiguration; preinit() is called once when the system starts, init() initializes internal variables when a task is initialized, go() contains the main task body and wrapup() is called when the system ends. In the generated code of each task, virtual OS APIs are used to access ports. The virtual OS APIs are implemented by the OS wrapper according to the port semantics and task scheduling policy.

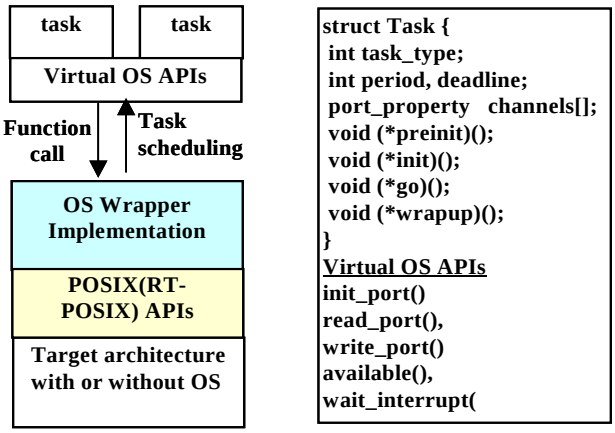


Figure 9 (a) Layered software structure, (b) task structure and virtual OS APIs

While the OS wrapper can be implemented with diverse methods, we use POSIX standard library[22] that allows writing portable real-time applications. We implement concurrent executions of tasks with pthread library, especially with RT-POSIX APIs for accurate timing services. Figure 10 shows a simplified implementation of the OS wrapper. Each task is created as a separated thread

by the OS wrapper. The scheduling control task receives control signals from the FSM task and determines the status of the tasks. Scheduling action is performed by the *scheduler* task that emulates the scheduling policy in the target operating system. Note that division of the scheduling control task and the scheduler task is not obligatory in the OS wrapper implementation.

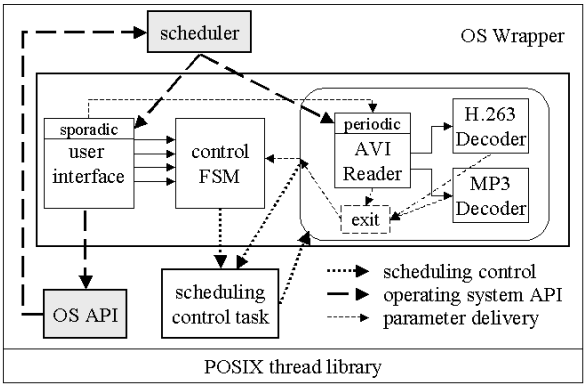


Figure 10 Software structure of the OS wrapper using Pthread library

Figure 11 represents the pseudo codes of the main and the scheduler functions generated by the OS wrapper showing the difference between for simulation and for implementation.

```
main {
   initialize architecture, channels, tasks
   create_task //only create, wait for execution
   scheduler(); //scheduling of task executions
  for(number of task) task_wrapup();
  control_task_wrapup();  //for control task
}
```

```
scheduler {
   start time = 0//start from virtual clock 0
   for( ;; ) {
   find_executable tasks
   find next task among executable tasks with
   smallest next_time
   execute the task
   wait for ending of the task execution
}
```

(a)

```
main {
  initialize architecture, channels, tasks
  Initialize control task
  for( number of task) Task_Start( task_info[i])
  for( number of task)Task_Stop( task_info[i])
  control_task_wrapup();  //for control task
}
```

```
Task_start {
If( receive resume signal from control task)
      resume_task()
else { task->init(), sem->init()
       pthread_create(task->pid, NULL,
                      &task_main, &task_id ) }
}

Task_stop {
If(receive suspend signal from FSM)
     suspend_task()
else { //wait for thread termination
     pthread_join( task.pid, (void **)&status )
    //destroy_task()
    task.wrapup();}
}
```

(b)

Figure 11 Main and scheduler code for (a) simulation and (b) implementation

In the simulation code, the main function first creates tasks, and then calls the scheduler function that handles task scheduling. The scheduler is an event driven scheduler that

executes tasks sequentially in the order of simulated time of triggering events. The simulated time is managed by the virtual clock that is initially set to zero in the beginning of the scheduler. Thus the simulation code emulates concurrent execution of multiple tasks on the host machine.

On the other hand, in real implementation, the main function needs to perform is to start and stop the tasks. Then each task is executed for itself according to the task type and the scheduling policy of the operating system run on the target platform. The main function first creates the control task generated from the fFSM model, and then creates the signal processing tasks(or threads) by calling Task_Start function and executing the “else” part in the function body. Task_Start function is also called inside the control task to resume the suspended task. By calling Task_Stop function, the main task waits until all signal processing tasks are terminated.

When each task is started, it executes task_main() as shown in Figure 12. It implements the execution semantics of a task. In case of a periodic task, with given the period parameter from the task model, it executes sleep operation until the next wake-up time after execution. While the scenario is logically simple, we are more concerned about the time resolution whether it is accurate enough for real-time processing. Precise periodic activation cannot be achieved with sleep() operation. So for more accurate execution of periodic task, real-time POSIX APIs such as

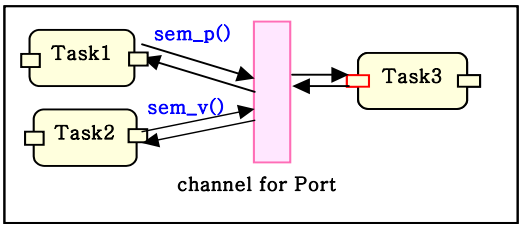
clock_gettime()and nanosleep() are used. clock_gettime() provides the real-time clock whose value cannot be changed explicitly.

```
task_main(void *task_id) {
 if( periodic task ) {
   clock_gettime(CLOCK_REALTIME,  &ts_start)
   task->next_time = ts_start;
    for( ;; ){
     clock_gettime(CLOCK_REALTIME,  &ts_before_exec)
     Task_Go();
     clock_gettime(CLOCK_REALTIME,  &ts_after_exec)
     task->next_time += task->period
     task->exec_time = ts_after_exec – ts_before_exec
     ts_wait = task->next_time – task->exec_time
      nanosleep(&ts_wait, NULL);
    }
 } else {
    for(;;) Task_Go();
 }
}
```

Figure 12 Implementation of task_main()

But these APIs does not guarantee complete real-time property since there is a possibility of preemption between two API calls, clock_gettime() and nanosleep(). To solve this problem, real-time POSIX library contains other APIs with higher time resolution such as clock_nanosleep(), pthread_cond_timedwait(). However these APIs have not been implemented in the operating systems we have used in our experiments. Currently we could achieve reasonable results in the Divx example though we implemented periodic task by using clock_gettime() and nanosleep() operations. It is because all tasks in our application have equal priority and we reduce the possibility of such unlucky preemption by the help of operating system. Data-driven tasks execute task_go() without special condition.

Another issue in impmementation is communication and synchronization between tasks. Communication between tasks is performed only through ports. Figure 13(a) illustrates a simplified scheme of the synchronization mechanism of port communication. Using semaphore operations, sem_p() and sem_v(), exclusive access to the shared channel between tasks is guaranteed. And it is realized in the definition of read_port() and write_port() definition in the OS wrapper. For accessing ports, read_port() and write_port() is called in task_go() function. Since the proposed task model supports dynamic rate ports, a task may be blocked due to the absence of data at the input port. In our Divx player example, the number of samples to be consumed at each invocation of MP3 decoder is not known a priori. Therefore, we support blocking read in read_port() definition as shown in Figure 13 (b). If the data size stored in the channel is less than the requested size, the task is READ_BLOCKED. And, before it leaves the port, it should release a task blocked for writing at the port, and then, finally release the port.



(a)

```
read_port( channel id, data, size ) {
   sem_p( semaphore of the channel )
   while( data size of the channel less than size ) {
        sem_v( semaphore of the channel )
        status of the read_task is READ_BLOCKED;
        sem_p( semaphore of the read_task )
        sem_p( semaphore of the channel )
   }
   if( channel is WRITE_BLOCKED)
      sem_v( sem of the source task of the channel)
   sem_v( semaphore of the channel )
   return size;
}
```

(b)

Figure 13 (a) Channel access model and (b) implementation of read_port()API

V. EXPERIMENTS

We implemented the proposed technique in the PeaCE codesign environment [22].

V.1 MMMT Example

Figure 14 shows the system level specification of MMMT example in PeaCE environment. Left bottom schematic of Figure 14 shows the top view of MMMT system with three operational modes (videophone, MP3 player and VOD player). Each mode has hierarchical structures and consists of multiple tasks. A FSM task controls dynamic scheduling of modes (and the tasks inside) and delivers to the tasks parameters received from the user interface task. And two connection handling tasks for videophone are connected to the control task. We automatically synthesized the MMMT terminal in a C code and downloaded it into a PC and a Compaq iPAQ. The user interface task creates a control widget using Qt window manager on Linux. We obtained frame rates 1 for the PDA and 10 for the PC implementation. This experiment confirms the viability of the proposed specification method both for complex system modeling and for automatic software synthesis for behavioral simulation.

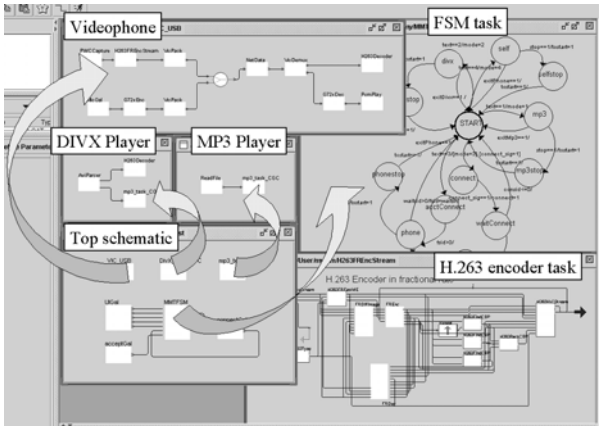


Figure 14 MMMT specification in PeaCE codesign environment

We also implement a performance profiler during simulation. Because the task scheduler executes each task sequentially, we can measure the execution time and counts of a task. Furthermore, we augment checkpoint functions at the block boundaries of dataflow model so that the block execution time and iteration count also can be profiled. Figure 15 is a performance profile result for the videophone mode, which shows execution times and iteration counts of tasks and blocks in the decreasing order, whose portion is higher than 3%. From this information, we could find out the performance bottleneck in the implemented code.

```
=============================================================================
task            execTime (%)            RunCnt  GoCnt   taskName
-----------------------------------------------------------------------------
[04]            9.900097 (56.590664)    181     180     H263FREncStreamII12
    [05]        3.256959 (32.898253)    17820           H263FRFastMEII4
    [36]        0.527065 (5.323837)     71280           FixDCTBlockII16
    [39]        0.514875 (5.200707)     71280           H263QII3
    [49]        0.324623 (3.278988)     17820           H263FRReconII67
-----------------------------------------------------------------------------
[03]            3.103285 (17.738913)    180     180     PWCCaptureII10
    [00]        3.102410 (99.971804)    360             PWCGrabII0
-----------------------------------------------------------------------------
[05]            2.046990 (11.700948)    181     180     H263DecoderII15
    [16]        0.292478 (14.288199)    180             H263ReconII55
    [17]        0.206490 (10.087494)    180             H263DisplayFrameII28
    [02]        0.188219 (9.194915)     0               H263FrameDecodeII0
    [14]        0.187688 (9.168975)     71280           InvZigzagBlockII11
    [12]        0.185724 (9.073029)     71280           FixCBPIDCTBlockII22
    [13]        0.171259 (8.366382)     71280           H263DeQII6
-----------------------------------------------------------------------------
[06]            1.292385 (7.387496)     278     0       G72xEncII17
    [00]        1.292381 (99.999690)    279             pt_CGCReceiveM0
-----------------------------------------------------------------------------
[07]            1.114046 (6.368079)     278     0       G72xDecII20
    [00]        1.114039 (99.999372)    279             pt_CGCReceiveM0
=============================================================================
 execTime       17.494223
schedTime       0.343062 (1.923286)
totalTime       17.837285
```

Figure 15 Profile result of videophone on PC Linux

V.2 Divx Player Example

In this subsection, we show the performance result of the automatically generated software of real-time implementation of the Divx player subsystem of the MMMT example. We are currently working on the real-time implementation of the entire MMMT system. We compare two platforms, Linux kernel 2.6.7 and eCOS_1.3.1. Linux is run on a Pentium processor and eCOS is transplanted on ADS1.2 as an ARMulator simulating ARM922T processor.

At first, we measure the code size overhead of the OS wrapper implementation as shown in Table 4. On the Linux platform, we measured the binary image size of the OS wrapper implementation and that of the total application. The OS wrapper implementation takes about 27KB. Since the most part of this overhead is independent of the number of tasks, the overhead gets less significant as the application size grows. And the overhead for real-time code and for simulation code is comparable in its size. On the eCOS platform, the total application includes the OS wrapper and the POSIX support library as well as eCOS image. The overhead of the OS wrapper implementation is similar to the Linux case in its size. Since we do not need POSIX library for simulation, the last column (Without POSIX) corresponds to the simulation code.

Table 4 Code size overhead of OS wrapper implementation

	Linux		ECOS	
	Real-time code	Simula-tion code	With POSIX	Without POSIX
OSWrapper/ Total(KB)	27 / 170	25 / 165	23/ 217	22/209

Next, we compared the real-time performance of the generated code by jitter measurements in two different platforms under the same timing properties such as period and frequency of task execution. Jitter is measured by the time difference between the schedule time when the task must wake up and the time on which it actually wakes up. Interrupt dispatch latency affects the jitter. As can be seen in Table 5, jitter in the eCOS platform is much smaller than that in the Linux platform. It is because eCOS platform has small and deterministic interrupt dispatch latency. In the table, T[0] represents the AviParser task, T[1] MP3 decoder task. H263Decoder task is not represented in Table 5 because the H263Decoder task is implemented as a functional task that can be executed immediately after receiving the data samples from the AviParser task.

Another experiment is made to compare the jitters in two different versions of Linux kernels: kernel 2.4.20 and kernel 2.6.7. In the table, Task[0] and Task[1] are those of above experiment. As can be seen in Table 6, the generated software experiences much smaller jitter rate in kernel 2.6.7 than in kernel 2.4.20. Since kernel 2.6.7 is designed for embedded applications, it has improved kernel latency. It is a preemptable kernel that has shorter critical sections and context switch time by an efficient scheduler. However the performance of Linux platform is still not comparable to eCOS platform.

Table 5 Comparison of jitter in Linux and eCOS

	Linux		eCOS	
Task	T[0]	T[1]	T[0]	T[1]
Period	120	100	120	100
Max_Jitter	1.932	3.362	0.356	0.424
Avg_Jitter	1.434	1.453	0.123	0.090

Table 6 Comparison of jitter in different Linux kernels

	Kernel 2.4.20-8		Kernel 2.6.7	
Task	T[0]	T[1]	T[0]	T[1]
Period	90	75	90	75
Max_Jitter	22.713	40.035	1.966	1.721
Avg_Jitter	14.978	2.337	1.011.	0.982
ExecTime	14.050356		14.041118	

VI. CONCLUSION

In this paper we presented a hardware/software design flow in which embedded software code is generated from system level specification of multi-tasking embedded systems, both for simulation and implementation. We specified the system behavior with three computation models where a dataflow model and a FSM model is used to specify the internal behavior of signal processing and control tasks respectively while a task model specifies the task execution semantics and inter-task communications. Task wrapper translates the task-model semantics and the internal task semantics.

The generated software from the specification has a layered structure using virtual OS APIs and OS wrapper implementations to make it re-configurable for multiple target platforms. We implemented the OS wrapper that represents concurrency of tasks, synchronization between tasks, and real-time features by using POSIX standards. With a Divx play example, we compared the real-time performance between two different platforms: a Linux platform on a Pentium processor and a eCOS platform on a ARM922T processor.

While software synthesis for behavioral simulation is supported in the current PeaCE environment, fFSM model is not supported in the hardware/software codesign flow yet. We are now extending the environment to support the fFS M model and its interaction with the SPDF model.

VII. ACKNOWLEDGMENTS

This work was supported by National Research Laboratory Program (Grant No. M1-0104-00-0015) and IT leading R&D Support Project funded by Korean MIC. The ICT and ISRC at Seoul National University provided research facilities for this study. We also give thanks to all CAP laboratory members for great efforts to implement PeaCE codesign environment, especially to Kiseun Kwon and Youngmin Yi for experiments of this paper.

VIII. REFERENCES

[1] Dohyung Kim, Minyoung Kim and Soonhoi Ha, "A Case Study of System Level Specification and Software Synthesis of Multi-mode Multimedia Terminal", Workshop on Embedded Systems for Real-Time Multimedia (ESTIMedia), Newport Beach, CA, USA Oct 2003

[2] Kiseun Kwon, Youngmin Yi, Dohyung Kim, Soonhoi Ha, "Embedded Software Generation from System Level Specification for Multi-Tasking Embedded Systems", ASP-DAC'05 Jan 18-21 2005

[3] S. Edwards, L. Lavagno, E. A .Lee, and A. Sangiovanni-Vincentelli, "Design of Embedded Systems: Formal Models, Validation, and Synthesis," IEEE Proceedings, March 1997, pp.366-390.

[4] M. Ashour, F. Khendek and T. Le-Ngoc, "Formal Description of Real-time Systems using SDL", 6th Int.Conf.on Real-Time Computing Systems and Applications, Dec., 1999.

[5] F. Balarin, et. al., "Hardware-Software Co-Design of Embedded Systems: The Polis Approach," Kluwer Academic Press, June 1997.

[6] Marco Sgroi, Luciano Lavagno, Alberto Sangiovanni-Vincentelli, "Formal Models for Embedded System Design", IEEE Design & Test of Computers, 17, 2, 14-27, June, 2000.

[7] E. Cheong, J.Liebman, J. Liu, and F. Zhao. "TinyGALS: A programming model for event-driven embedded systems.", In Proceedings of the Elighteenth Annual ACM Symposium on Applied Computing, Pages 698-704, March 2003

[8] H. J. H. N. Kenter et. al., "Designing Digital Video Systems: Modeling and Scheduling," Codesign Symposium (CODES), May 1999.

[9] Girault, B. Lee, and E. A. Lee, `Hierarchical Finite State Machines with Multiple Concurrency Models," IEEE Transactions On Computer-aided Design Of Integrated Circuits And Systems, Vol. 18, No. 6, June 1999.

[10] J. Buck, R. Vaidyanathan, "Heterogeneous modeling and simulation of embedded systems in El Greco", Proceedings of the eighth international workshop on Hardware/software codesign May 2000.

[11] D. Harel, H. Lachover , A. Naamad , A. Pnueli , M. Politi , R. Sherman , a. Shtul-Trauring., "STATEMATE: a working environment for the development of complex reactive systems", 10th

International Conference on Software Engineering April 1988

[12] K. Strehl, L. Thiele, M. Gries, D. Ziegenbein, R. Ernst, J. Teich, “FunState-an internal design representation for codesign “, Very Large Scale Integration (VLSI) Systems, IEEE Transactions on , Volume: 9 Issue: 4 , Aug. 2001, Page(s): 524 –544

[13] T. Groetker, et. al., “System Design with SystemC”, Kluwer Academic, Norwell, Mass., 2002

[14] D. Gajski et al., “SpecC Specification Language and Methodology,” Kluwer Academic, Norwell, Mass., 2000.

[15] Jie Liu Lee, EA Palo Alto Res, “Timed multitasking for real-time embedded software”, Control Systems Magazine, IEEE Publication, Vol.23, Feb 2003, pp.65-75,

[16] Dirk Desmet , D. Verkest , Hugo De Man, “Operating system based software generation for systems-on-chip” Proceedings of the 37th conference on Design automation, p.396-401, June 05-09, 2000, LA, U.S.A.

[17] Ptolemy. http://ptolemy.eecs.berkeley.edu/

[18] E. Lee, D. Messerschmitt, “Synchoronous data flow”. Proceedings of IEEE, Vol. 75, No. 9, pp. 1235-1245, 1987.

[19] Hyunok Oh and Soonhoi Ha, "Fractional rate dataflow model and efficient code synthesis for multimedia applications", ACM SIGPLAN Notice Vol. 37 July 2002, pp 12-17.

[20] Chanik Park, Jaewoong Chung and Soonhoi Ha, "Extended Synchronous Dataflow for Efficient DSP System Prototyping", Design Automation for Embedded Systems, Kluwer Academic Publishers Vol. 3 March 2002, pp 295-322.

[21] Dohyung Kim, Soonhoi Ha, "Static Analysis and Automatic Code Synthesis of flexible FSM Model", ASP-DAC 2005 Jan 18-21 2005

[22] POSIX.13 (1998). IEEE Std. 1003.13-1998. Information Technology –Standardized Application Environment Profile-POSIX real-time Application Support(AEP). The Institute of Electrical and Electronics Engineers, 1998

[23] http://peace.snu.ac.kr/research/peace

Soonhoi Ha

Soonhoi Ha is currently a professor in the School of Computer Science and Engineering at Seoul National University. From 1993 to 1994, he worked for Hyundai Electronics Industries Corporation. He received his Bachelors (1985) and Masters (1987) in Electronics Engineering from Seoul National University, and PhD (1992) degrees in Electrical Engineering and Computer Science from University of California, Berkeley. He has worked on Ptolemy project and he is now leading the PeaCE project. His research interests include hardware-software codesign, design methodology for embedded systems. He is a member of IEEE Computer Society.

sha@snu.ac.kr, 019-223-8382

Dohyung Kim

Dohyung Kim received the B.S. and M.S. degrees in computer engineering from Seoul National University, Seoul, Korea in 1997 and 1999 respectively and the Ph.D. in electrical engineering and computer science from the same university in 2004. He is currently a post-doctoral researcher at University of California, San Diego. His primary research interests are various aspects of MPSoC platform including system level specification and accurate evaluation of performance and power consumption.

dhkim@ucsd.edu