Software Architecture Synthesis for Retargetable Real-Time Embedded Systems*

Pai Chou and Gaetano Borriello
Department of Computer Science and Engineering, Box 352350
University of Washington, Seattle, WA 98195-2350 USA
{chou,gaetano}@cs.washington.edu

Abstract -- *Retargetability of embedded system descriptions not only enables better exploration of the design space and evaluation of cost/performance tradeoffs but also enhances design maintainability and adaptivity to new technologies. Unfortunately, the traditional boundary between run-time support and user-code encourages use of ad hoc architecture-specific features that lack the structure to permit automatic code synthesis for the satisfaction of timing constraints. This work proposes a specification style for control-dominated embedded systems that can be easily retargeted via automatic synthesis of the software architecture and run-time support. Unlike previous work, user-specified modes are an integral part of the run-time system and isolate architecture-specific details while scoping timing constraints to enable more efficient scheduling.*

1 Introduction

Retargetability is an important problem in embedded systems for several reasons. Most interesting tradeoffs are at the target architecture level, which has the greatest impact on the cost and performance. Retargetable designs will also be more maintainable and ready to take advantage of the latest technology. This is in contrast to today's heavy reliance on legacy code and components. Retargetable designs will also enable accurate evaluation of the architecture early in the design cycle, and save painful redesign efforts.

High-level programming languages and compilers have been invented to make programs *portable*, that is, make computations independent of the ISA. However, in system designs, just because the processor runs the code does not mean it will *behave* correctly. In other words, portability does not imply retargetability. The lack of retargetability comes from timing, device drivers, and hard coding system-specific assumptions. Retargetability is complicated by the fact that the behavior of many embedded systems, especially the "reactive" ones, can be quite complex and are intimately tied to architecture-specific elements and timing. Current solutions have not adequately addressed the retargetability of real-time reactive behaviors.

To capture complex reactive behavior, Esterel [BG91] and StateCharts [HLN+] rely on synchronous semantics to enable composition of parallel processes or state machines. An important assumption is that the required work in each time step can be done "fast enough" that schedulability is not a concern. These languages provide reactive constructs and rely on a host language such as C++ to express computation. Ada 95 [Ada95] provides a rich set of concurrent and real-time constructs so that it would serve well as both the host and the specification language. However, all of these languages rely on external run-time support for architecture specific facilities and for real-time.

Many designers use real-time kernels for embedded designs. Unfortunately, real-time kernels are not intended to be retargetable. They may provide device drivers but only for a fixed architecture, for example, by expecting the system to include a specific VME bus interface card. If the designers want to exploit different ways of connecting to devices, then they must rewrite the drivers manually for each specific configuration. It is a tedious, error prone task, and can easily conflict with assumptions made by the kernel, such as preemption.

Parameterized kernels [VLD96] attempt to enhance retargetability by dividing the kernel into separate scheduling and communication layers. The communication layer abstracts the architecture specific I/O facilities. The drivers for standard protocols are written once per ISA and stored in the library. The scheduler is preemptive, priority-driven, task-based, as assumed by Ada's Real-Time Annex. While this is adequate for data-dominated applications that have regular behaviors, it is unable to handle complex timing constraints often found in control-dominated systems. Instead of tuning code and delays, designers tune the priority assignments and design mostly by trial-and-error.

To overcome the limitations of process-based scheduling, schedulers that focus on *observable events* have been proposed. Schedulability can be enhanced by performing dependency analysis and apply code motion when the system is overconstrained by code-based constraints [GH95]. Frame Scheduling handles more general types of constraints verifies that constraints can be met even in the presence of interrupts [CTGC].

Though not yet mature enough for real-time, Java represents a new approach to retargetability through its virtual machine and standardized run-time API. Java was originally invented for embedded applications, and it has several attractive features, including built-in support for multithreading, synchronization, limited timing control, and exception handling. Its object orientation provides a good mechanism for declaring attributes for software synthesis without extending the syntax of the language. We also chose Java for building the CAD tool for easy distribution and documentation. It also enables better

* This work was supported by DARPA DAAH04-94-G-0272

presentation of examples of embedded systems as applets on web pages. Users can actually play with a model and get a feel of the behavior, instead of reading vague natural language descriptions or deciphering from HDL or C code.

We have developed a way of organizing control-dominated real-time embedded systems and propose methods for synthesizing the software architecture. It centers on system *modes* as functions for mapping physical events to logical events and scoping timing constraints. Modes not only isolate target dependent features for synthesis, but also enables static scheduling for meeting reactive timing constraints. Additionally, scheduling observable events, rather than processes, guarantees the detailed timing assumptions of device drivers.

2 System Model

A retargetable software architecture should abstract away architecture-specific features that can potentially affect the program structure. It should have the following properties:

1. preserve timing assumptions of the device drivers at the fine-grained level
2. statically meet all timing constraints on observable events in complex, modal behavior

Most schedulers cannot handle highly complex reactive behavior, because they have no knowledge about modes, which are encoded as an arbitrary mixture of program counters and state variables throughout a program. Because this kind of specification can be very target specific, retargeting can force major restructuring of the behavioral description.

Our model centers around user-specified modes that become an integral part of the run-time system. Modes not only bridge the architecture elements and the behavior, but also define scopes of timing constraints for scheduling.

2.1 Modes

Reactive systems perform tasks in response to input events. A main feature that distinguishes these systems from data dominated ones, such as DSP, is their modal behavior. That is, a given event may trigger different reactions, depending on the internal state of the system. Until now, the term "mode" has been vaguely defined as something like a state in StateCharts, but is concerned more with the behavior rather than the topology of the graph.

Here we are proposing a more formal definition of modes in the context of reactive systems. A mode is a function that maps a physical event (possibly with associated values) to a logical event, each with a corresponding handler, as defined by the user. This allows a given physical event to take on different logical meanings by triggering an entirely different response, or even to be ignored entirely.

Each handler may also instruct the system to change mode. Note that there is a distinction between a transition back to the same mode and no transition. The former can be viewed as reentering the mode. Specifically, if the mode waits on a timer event, a self-transition effectively reinitializes the delay, while in the latter case the delay continues after handling an event without a transition.

Sequencing is a simple replacement of the current mapping with that of the target mode. A fork can be viewed as augmenting the mapping with additional (trigger, response) entries. It is a way of composing modes to form hierarchical modes, because it combines the mapping functions of the submodes. Each submode can make its own transitions independently of the other submodes. Different submodes may also transition to a joining mode that does not become effective until all of its predecessors have completed their transitions to it. A join is the inverse of a fork. Note that both sequencing and joining are done in the context of its "enabler" submode, rather than the composite mode. Finally, a disable kills other forked branches as well.

2.2 User Code

The user code defines the handlers to be invoked when an event is detected. The semantics of the handlers is *run-to-completion*, that is, once a handler starts executing, it cannot be suspended and context-switched out, though it may be preempted briefly for system interrupts. [GF96]

We call a specification *purely reactive* if it is invoked only to respond to events, and each response can complete before the next event needs to be handled. Another way of viewing the system is as a (program) state machine, where each state does nothing, but actions are executed only on state transition edges. In reality, this is shifting the responsibility of event detection to the run-time system. In object-oriented terms, a purely reactive object is a passive object.

Even though purely reactive objects are more restrictive, they can express a large class of interesting embedded systems. In the purest form, they cannot handle user code that still wants attention even in the absence of events. This can happen when the response to an event is a sustaining action that continues beyond subsequent events that need to be handled. Also, the system may be actively computing and generating events, such as a mobile robot that computes its course or a screen that displays messages according to a preprogrammed script. We can simulate active objects by treating clock ticks as a type of event.

2.3 Timing Constraints

Timing constraints in a pure specification are minimum or maximum separation requirements between the times of two observable events. Common classifications of events are periodic and sporadic. Periodic events are those that repeat, and sporadic ones do not always occur, though once they occur, they will not occur again for some minimum amount of time. In addition to events from the environment, the system may also generate events either actively or in response to them.

Constraints are specified on a pair of events, one of which must be generated by the system. A specified separation between two events from the environment is not a constraint, but a promise, though it is possible to derive constraints such as polling rates from promises. Our model considers three types of timing constraints: sequencing, rate, and response time [CWB94]. Sequencing is between two different events generated by the system, rate is the separation between successive occurrences of an event, and response time is between an input and a system event.

The modes scope the timing constraints. In event detection, polling loops may be rate constrained, and the body of the loop and handlers may be under sequencing constraints. Some events may be specified to be ignored in a particular mode and do not need to be scheduled at all.

2.4 Time-Triggered I/O

Time-triggered I/O is used to guarantee both the detailed timing assumptions of device drivers and meeting inter-driver timing constraints. That is, time-triggered I/O's are scheduled on clock interrupts only. This enables more accurate timing on observable events than delay statements. By disallowing nested interrupts, intra-driver timing can be guaranteed.

3 Retargetable Software Architecture

Our proposed software architecture is divided between user code and run-time system, but the interesting issues are the separation of responsibilities between the two parts. The run-time system to be synthesized is divided into several parts: device drivers, event management, mode management, and the real-time engine. (See also slide 5)

3.1 Mode Manager

The mode manager is divided into two distinct tasks: mapping physical events to handlers for dispatch and changing modes. When the event manager detects an event, the mode manager looks up the corresponding handler if one is defined. Certain events may be "filtered out" because the system may be in a mode that ignores those events. Otherwise, the mode manager maps the event to a handler, which runs to completion before returning control to the mode manager.

After the handler completes execution, if there is a mode transition to perform, the mode manager reconfigures the event manager to reflect the current sensitivity list. To construct the new mapping, the mode manager applies one of the transition operators, which can be sequencing, fork, join, and disable.

3.2 Event Manager

The event manager detects incoming events and schedules outgoing events. Event detection involves invoking the polling methods or enabling an interrupt by calling a driver routine. On detecting an event, control is passed to the mode manager to map to the handler. Also, output requests are received by the event manager for scheduling . It informs the real-time engine to update the polling list, disables the interrupts for the exiting mode, and enables the interrupts for the new mode.

3.3 Real-Time Engine

The real-time engine dispatches the scheduled events at the right times. It accepts requests from the event manager for scheduling I/O. The primitives are in the form of scheduling an I/O operation some number of time units relative to a reference event, which can be the previous iteration in the case of a polling loop, or relative to the trigger in the case of a response. The timing is statically determined by the scheduler.

The real-time engine maintains a calendar for the scheduled events, and it must be able to dynamically reconfigure the calendar as a result of mode transitions. In particular, if a polling loop is no longer active then it may need to be removed from the calendar. In the case of composite modes, a transitioning submode does not affect events schedules for other parallel submodes that do not undergo transitions.

4 Using the Java Language

Current proposals for adapting Java to real-time applications focus their effort on several problems, including standardizing the API, real-time garbage collection, and compilation vs. interpretation tradeoffs. While these are important problems, these still assume process-based preemptive scheduling. To achieve retargetability, we believe it is necessary to augment the existing model with event-based scheduling.

4.1 Timing Control

Java provides several standard library methods for timing control. The `sleep(`*milliseconds*`)` call is a delay, and `join(`*thread, milliseconds*`)` waits for another thread to terminate, or kills it after the specified time. These are not adequate for implementing the real-time engine described earlier, which needs a delay relative to a reference point. Unlike Ada 95, Java has no "delay-until" a 64-bit absolute point in time. Even though Java API allows reading of absolute times, attempts to use a relative-delay on the difference between two absolute times will not always work in the presence of preemptive task scheduling. This is because if context switching happens between computing the relative delay and the delay call, then the delay amount will be too long.

Both JavaTime [You96] and Real-Time Java [Nils96] take the process-based approach to solving the delay problem. JavaTime requires periodic tasks to define rate constraints and supply a timeout handler. It is built on existing mechanisms and is platform independent. Real-Time Java proposes sporadic and spontaneous in addition to cyclic (periodic) tasks. It assumes platform-specific support.

Ideally, user code should avoid using imperative delay calls whenever possible for retargetability reasons. The same delay values do not necessarily yield the same timing behavior on different architectures, because the code execution time varies. Instead, timing should be expressed in terms of constraints between events, as described in the next subsection. Synthesis will determine the delays by solving the constraints and implement them as a static schedule.

4.2 Declarative Semantics

Software synthesis requires that certain attributes be expresses declaratively for retargetability, including timing and modes. Previously, new syntactic extensions to the host language or external annotation languages were invented for specifying these non-functional attributes. As an object-oriented language, Java classes and their interface mechanism have been used for specifying some of these attributes declaratively without extending the syntax of the language. An `interface` in Java is similar to an abstract class in that it defines the required methods, and that all classes conforming to the same interface are type compatible, but it is not their base class. We define interfaces that require user-defined classes to supply the attributes, which will be used by the synthesis tool.

5 Software Architecture Synthesis

The software architecture can be synthesized from the following inputs: a system (hardware) architecture model, the modes of the system, and user code organized in objects. The architecture model instantiates the devices and interconnection schemes for device driver synthesis, which can be done with [COB92,COB95]. The compilation is done in two phases: one to compile and run the CAD tool for static scheduling and generate code for the customized run-time system, and the second to build the custom OS with the user code.

5.1 Input Data Structures

The user describes modes by supplying the mapping function from triggers to responses. Triggers and Responses are containers for the triggering and response actions and their attributes, including timing constraints.

A Trigger defines how an event is detected, which may be fixed by a specific protocol, derived during static scheduling, or user specified. Each trigger has a flag that specifies whether it is detected by polling, interrupt, or call-back, and is linked to the device driver library. Timing information such as polling rates and execution times are also defined as attributes of the Trigger.

A Response, similarly, is also a container of both an executable part and a declarative part. It contains a handler that invokes the appropriate user-defined methods, which returns a mode operator object for the transition. Execution times and constraints are also declared as object attributes.

A mode operator has a field indicating what type of transition to perform: sequencing, fork, join, disable, or no transition. All except for the last requires a destination mode parameter.

5.2 Synthesis Procedure

The steps in the synthesis procedure are mode manager synthesis, execution time estimation and timing analysis, event manager synthesis, scheduling, and real-time engine synthesis.

To synthesize the mode manager, the algorithm first follows the transition links, builds the complete mode graph, and checks for consistency. Next, Java code is generated by parameterizing a template file with the mapping functions.

Timing estimation returns the execution time bounds of each piece of code on the target platform. The code segments include the user-defined handlers, mapping functions, and the various device driver calls customized to the target architecture. Timing analysis uses the execution time estimation and traverses program paths to determine the time separation between events. This will be used determine the methods for event detection.

To synthesize the event manager, the triggers of each composite mode are extracted and generated as either interrupt enable or static scheduling for polling. The polling loops and all the I/O's generated by user code are added to a constraint graph for scheduling later. The event manager code is generated by customizing a template with the scheduling calls for the device drivers.

The constraint graphs built during event manager synthesis are annotated with the execution times and input to the static scheduler [CB94]. The corresponding run-time engine is generated by linking the schedule tables with a small set of target-specific primitives extracted from the library.

6 Example

We have written several examples of reactive real-time systems in the proposed style. Here we illustrate the concepts with a simple mobile robot. (See also slide 11)

The robot has four inputs: a bumper, a sonar, an arm connected to a switch, and a button on its head. The robot also has two separate motors controlling two wheels. By running the two motors at different speeds, the robot can turn in different directions. The physical input events come from the sensors. The bumper generates a bump event with no value, though the bumper state can be polled. The sonar generates a pong event. The arm and the head switches both generate a press event.

The robot controller can handle the following logical events: foward, backward, turn, pause, resume, and reset. Forward sets both wheels in forward motion, and backward sets them in reverse. Turn, for simplicity, always means turning left, by setting the left wheel in reverse and the right wheel in forward motion. Pause will stop the wheels, and resume will continue the motion before the pause. Finally, reset will reinitialize the robot.

The required behavior is that the events are prioritized in the following order: reset, pause/resume, bumper, and sonar. That is, when a higher priority event is being handled, the lower priority ones are ignored completely. In our model, this means changing mode to map those low-priority events to no response. For example, when the bumper is pressed, the sonar is completely ignored. The event manager will disable pulsing the sonar, and this is an important feature because the sonar consumes a significant amount of power.

The behavior can be modeled well with hierarchical modes. On startup, the initialization handler sets the wheels in forward motion, and enters the main mode. The main mode is composed of two parallel branches. One mode maps the reset event to a handler that does no work and makes a disable transition back to initialization. The other forked branch contains a nested fork, where one branch maps the arm button to the pause handler, and the other branch is yet another fork that maps bumper and sonar events. The pause handler will suspend the bumper and the sonar mapping, and enters a new mode that maps the arm button to the resume handler. The reset mode is always "active" independent of the transitions triggered by the other sensors.

Timing constraints are present on both device accesses and reaction times. The sonar, for example, is configured to trigger when it detects an obstacle within 30cm, and this is determined by checking the status bit after the round-trip time of 2ms. The pulsing rate for the sonar is once every two seconds, and this is specified by the designer in the trigger definition. The reaction times here are scoped to be from event detection to an observable event in the response, namely changing the motor direction or speed. These would be specified also by the designer as part of the response record.

To show that this style of description is easily retargetable, consider that the pause and reset buttons are replaced with an infrared receiver. It can interpret four commands, where two of the commands trigger the same events as the bumper and the sonar. The only modifications are limited to the mapping functions, where the pause and reset commands from the IR replace the buttons in the entries, and the turn and backward commands from the IR are added to the mapping, without replacing the bumper and the sonar. Scheduling ensures timing of the infrared unit is satisfied. The user-code requires no change. (See also slide 12)

7 Conclusions and Open Issues

We have presented a retargetable way of organizing the software of embedded systems. The use of modes isolates the architecture-specific features from high-level behavior while providing important information for the scheduler to meet timing constraints on complex control-dominated behavior. Event-based scheduling enables satisfaction of timing constraints without overconstraining the system.

Accurate timing estimation is an important enabling technique for software synthesis. The use of bytecode, while slower than native code, has the potential of yielding more predictable timing behavior with its extra level of indirection. It may be possible to parameterize the ISA timing so that timing analysis needs to be done only once on the byte code.

The open issues include extensions and refinements to the proposed model. First, events are assumed to be detected one at a time, but will they automatically be queued? If so, what is the precise semantics of the queued events on a mode transition? Does it make sense for an event to remain in the queue across multiple mode transitions? Or should they be flushed implicitly? Does the user code have the option of flushing events? Should the events be time-stamped, and how does this complicate scheduling? Will it be possible to do out-of-order dispatching? In addition, will it make sense to generalize triggers to not just physical events, but general boolean conditions? Are there other useful mode operators?

References

[Ada95] The Ada 9X Design Team, *Ada 95 Rationale: The Language/The Standard Libraries*, published by Intermetrics, January 1995.

[BG91] G. Berry, G. Gonthier, "The ESTEREL synchronous programming language: design, semantics, implementation," *Science of Computer Programming*, Vol. 19, no.2, pp. 87-152. November 1991.

[CB94] P. Chou, G. Borriello, "Software Scheduling in the Co-Synthesis of Reactive Real-Time Systems," in *Proc. DAC*, June 1994. pp. 1-4.

[COB92] P. Chou, R. Ortega, G. Borriello, "Synthesis of the Hardware/Software Interface in Microcontroller-Based Systems," in *Proc. ICCAD*, 1992, pp. 488-495.

[COB95] P. Chou, R. Ortega, G. Borriello, "Interface Co-Synthesis Techniques for Embedded Systems," in *Proc. ICCAD*, 1995. pp. 280-287.

[CTGC] M. Cornero, F. Thoen, G. Goossens, and F. Curatelli, "Software Synthesis for Real-Time Information Processing Systems," in P. Marwedel and G. Goossens (ed.), *Code Generation for Embedded Processors*, Kluwer Academic Publishers, 1995, pp. 260-296.

[CWB94] P. Chou, E.A. Walkup, G. Borriello, "Scheduling issues in the Co-Synthesis of Reactive Real-Time Systems," *IEEE Micro*, August 1994. pp. 37-47.

[GF96] D. Gaudreau, P. Freedman, "Temporal Analysis and Object-Oriented Real-Time Software Development: a Case Study with ROOM/ObjectTime," in *Proc. 1996 IEEE Real-Time Technology and Applications Symposium*, June 1996. pp. 110-118.

[GH95] R. Gerber, S. Hong, "Compiling Real-Time Program with Timing Constraint Refinement and Structural Code Motion," *IEEE Trans. on SW Engineering*, vol.21, no.5, May 1995.

[HLN+] D. Harel *et al*, "STATEMATE: a working environment for the development of complex reactive systems," *IEEE Trans. on SW Engineering*, vol.16, no.4, pp. 403-414, April 1990.

[Nils96] K. Nilsen, "Issues in the Design and Implementation of Real-Time Java," document, <http://www.newmonics.com/webroot/technologies/java/RTJI.ps>, Revised July 19, 1996.

[VLD96] S. Vercauteren, B.Lin, H. De Man, "A Strategy for Real-Time Kernel Support in Application-Specific HW/SW Embedded Architectures," in *Proc. DAC*, 1996.

[You96] J. Young, *JavaTime*, <http://www-cad.eecs.berkeley.edu/~jimy/java/index.html>, August 1996.