Flux Group Technical Note 2002-02, May 17 2002
http://www.cs.utah.edu/~regehr/papers/cee

Composable Execution Environments

John Regehr Alastair Reid Kirk Webb Jay Lepreau
School of Computing, University of Utah
{regehr, reid, kwebb, lepreau}@cs.utah.edu
http://www.cs.utah.edu/flux

Abstract

Many problems in developing and maintaining systems software stem from inappropriate choices of *execution environments*: sets of rules and conventions for structuring code. We have developed *composable execution environments* (CEE), a new way to compose systems software that is based on two main capabilities: the hierarchical composition of execution environments, and the late binding of design requirements to implementation constructs.

Our contributions include a number of techniques to support CEE: first, an environment description language in which compositions of environments are specified; second, the task / scheduler logic for checking safety properties of compositions, which exploits the low-level hierarchical scheduling relationships that are present in systems software; and third, integrated support for real-time schedulability analysis. Together, the CEE model and these techniques enable developers to create systems software out of multiple restricted, domain-specific execution environments and to more easily retarget code to new environments.

1 Introduction

Creating good systems software often depends on choosing appropriate execution environments. For example, a small embedded system is often structured as a *cyclic executive* — a glorified infinite loop. In this non-preemptive, statically scheduled environment it is difficult or impossible to write a program containing a race condition or a deadlock. At the other end of the spectrum, preemptive multithreading with blocking locks enables very flexible resource utilization. It also permits threads with short response time requirements to run in a timely way even if other threads run for long periods without voluntarily blocking. However, developers have a notoriously difficult time creating correct systems using concurrent threads [13, 20]. There is a tension between picking a powerful execution environment such as preemptive multithreading, and picking a restricted environment such as a cyclic executive or a non-preemptive event loop. Furthermore, different parts of a single system are often best structured using different execution environments.

This paper describes CEE, a model for systems software programming that supports *first-class, composable execution environments*. We define an *execution environment* to be a set of rules and conventions for structuring systems code along axes such as sequencing, concurrency, atomicity, resource management, and real-time constraints. In other words, execution environment considerations come from architectural issues rather than from functional issues. Restricted execution environments help make systems easier to design, understand, and analyze by enforcing rules that deprive developers of unwanted or unneeded design freedom while maintaining freedom along axes that help solve specific problems. For example, Click [14] is all about packet processing: its push/pull distinction and strict restrictions on concurrency and communication make it much easier to create and debug Click graphs than unstructured C++ code that performs the same functions.

CEE provides two complementary benefits. First, it permits systems to be constructed by hierarchically composing execution environments. The execution environments supported by CEE include traditional ones such as non-preemptive event loops and preemptive multithreading, as well as more restricted, domain-specific execution environments such as Port-Based Objects [26] and TinyOS [11]. We work with, rather than attempting to supplant, existing component languages such as those used to compose Port-Based Objects and TinyOS systems. This is necessary because some of the valuable features provided by these systems, such as the nested component interconnection model in TinyOS, have nothing to do with execution environments.

The second main benefit of CEE is that it permits the *late binding* of some types of design requirements to implementation constructs. As an example of late binding, consider rate-monotonic scheduling [18]. By delaying the binding of threads to priorities until an entire system is available for analysis, it avoids composing prioritized threads: they are known to compose poorly. Our work is analogous but goes much farther: we permit the late binding of components to threads, threads to schedulers, and critical sections to synchronization primitives.

In systems software developed with traditional languages and module systems, execution environment considerations are *implicit* and *hardwired*. Real-time constraints, atomicity requirements, and design rules are buried in the code where they are hard to find, and hard to modify in response to changes in hardware capabilities or software requirements. In our programming model, execution environment considerations are *explicit* and *flexible*: environments declare their requirements and assumptions, and our analysis tools determine a concrete implementation of the requirements that does not break any of the assumptions. Explicit requirements makes it easier to develop, maintain, and evolve systems software, since it helps cut down on subtle, non-local errors that can be caused by changes in execution environments. Furthermore, common strategies for generalizing the execution environment, e.g., using a preemptive kernel rather than a single-threaded event loop, can cause the benefits of the more restricted environment to be lost. By supporting multiple concurrent environments, we often permit part of the system to be generalized while preserving the benefits of restricted environments elsewhere.

The work described in this paper applies where there is significant diversity in execution environments. This includes most embedded systems, all general-purpose operating systems, most complicated server and middleware infrastructures, and some distributed systems. The focus of this paper is on embedded systems, for a number of reasons. First, it tends to be hard to fix bugs in embedded systems after deployment: for this reason a more formal underpinning for systems software is particularly valuable for embedded systems. Second, there is pressure to produce embedded software quickly, and products often form product families. Members of the family are similar enough that the similarity must be exploited, while there are too many differences to be ignored [29]. Third, because time and space efficiency are so important, embedded systems tend to be low-level and hand-rolled, and developers have significant design freedom with respect to execution environments. This can result in evolution and maintenance headaches as the capabilities of embedded systems track Moore's Law. Finally, although we plan later to apply CEE to other classes of systems software, embedded systems offer an especially promising starting point. They are small and they are typically more static than higher-level software.

Our contributions are as follows. We present CEE, a unified model capable of describing a large class of execution environments including those commonly found in embedded systems. We have developed novel techniques that help make CEE into a practical way to build systems software: the environment description language (EDL) for specifying hierarchies of environments (Section 4), and the task / scheduler logic, a way to reason about safety properties for concurrent processes using hierarchical schedulers (Section 5). We have also integrated a real-time task model with CEE (Section 6). In Section 7 we evaluate CEE by using it to combine TinyOS and AvrX [4] environments in ways that can help solve real-world problems.

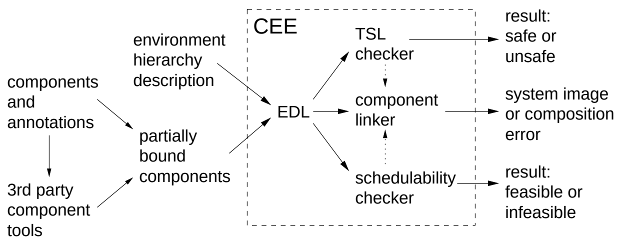
CEE
environment hierarchy description
components and annotations
3rd party component tools
partially bound components
EDL
TSL checker
component linker
schedulability checker
result: safe or unsafe
system image or composition error
result: feasible or infeasible

Figure 1: Overview of CEE

2 Overview of the CEE Architecture

Figure 1 depicts the architecture of CEE. On the left are the inputs (environment hierarchies and application code); in the center is the CEE toolchain; and on the right are the outputs: analysis results and the application binary. This section describes the parts of the model and how they fit together, while Sections 4–6 go into more detail about important elements.

Components: A major motivation for CEE is to make component programming feasible when writing programs that involve multiple execution environments (i.e., systems software). Components can be *partially bound* (they interact with their scheduling environment through a layer of indirection such as a function call or macro) or *fully bound* (they interact with their scheduling environment directly). CEE can work with both kinds of components, but the ways in which fully bound components can be composed is necessarily limited.

Third-party component languages: There are many component languages available (including "plain old C"). Rather than invent yet another component language, CEE allows third-party component languages to be embedded in scheduler hierarchies.

Environment Description Language (EDL): The EDL compiler processes descriptions of environment hierarchies; it resolves references between environments, integrates component annotations and the results of analyses, generates a checkable task / scheduler logic specification for a system, and generates a linking specification that is passed to the CEE linker.

Task / Scheduler Logic: Components and environments in EDL instantiate *prototypes* that are accompanied by specifications in task / scheduler logic (TSL).

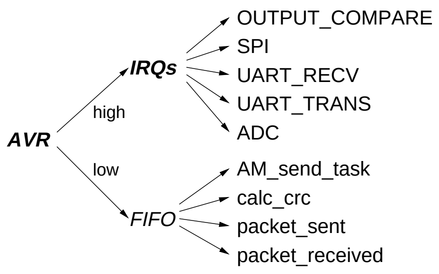

Figure 2: The TinyOS scheduling hierarchy with tasks from the `cnt_to_rfm` TinyOS kernel. In this and subsequent figures preemptive schedulers are shown in a bold, oblique font, non-preemptive schedulers are oblique, and non-scheduler entities in the hierarchy are in the standard font. Annotations are in a smaller font.

TSL provides the concurrency semantics for EDL; it is used to verify safety properties of the system as a whole, and also to enable optimizations.

Real-time scheduling: Real time tasks can be annotated with their real time properties (worst case execution time, period, etc.) enabling CEE to check schedulability, assign priorities, and merge logically separate tasks into a single physical thread when appropriate.

3 Background: TinyOS and AvrX

TinyOS [11] and AvrX [4] are operating systems for Atmel AVR microcontrollers; they will serve as the primary examples of systems throughout this paper. TinyOS was written to run on the *mote* hardware developed at UC Berkeley. It has a static memory model that can result in systems with a very small footprint (some motes have only 512 B of RAM). TinyOS embeds a component model that makes it easy to compose new systems, often without touching source code.

The execution environment structure of TinyOS, shown in Figure 2, is basically identical to that of the bottom half of a modern general-purpose operating system: it runs in a single address space and consists of interrupt handlers with partial preemption relations running at high priority; at lower priority there is a non-preemptive FIFO scheduler that runs *tasks*. Tasks provide a level of deferred processing, reserving interrupt context for short, time-critical sections of code.

In this paper we also use *task* in a more general way: a task can simply be a piece of code to be executed by the system. This sense of task does not connote execution in any particular environment.

Like TinyOS, AvrX targets very small systems. It provides features found in most real-time executives: preemptive multithreading, mutual exclusion, timers, message queues, and a debugging monitor that can be used to single-step the processor. AvrX is structured so that unused code is not included in applications. AvrX saves RAM by running all interrupt handlers on a kernel stack.

4 Specifying Hierarchies of Environments

This section describes the CEE environment description language and the embedding of other component languages within it. Subsequent sections describe TSL (the logic we developed to detect systems composition errors) and SPAK (the schedulability analysis tool that we use to check real-time properties).

4.1 The Environment Description Language

A hierarchy of execution environments is expressed as a scheduling hierarchy. For example, Figure 2 shows a typical TinyOS configuration. At the top level is the AVR microcontroller. The AVR supports 23 interrupt handlers, most of which are not present in any given TinyOS kernel. Interrupts are run in preference to user-mode code; in user mode, TinyOS runs a non-preemptive task scheduler.

Systems are described in EDL by a number of object instance definitions such as the following:

```
code = TinyOS{
  desc = "cnt_to_rfm.desc"
};
avr = AVR{
  irq = irq,
  user = fifo
};
irqs = IRQs{
  irq10 = IRQ{fn = code.desc.OUTPUT_COMPARE},
  irq16 = IRQ{fn = code.desc.SPI},
  irq18 = IRQ{fn = code.desc.UART_RECV},
  irq21 = IRQ{fn = code.desc.UART_TRANS},
  irq22 = IRQ{fn = code.desc.ADC},
  default_irq = IRQ{fn = code.desc._unexpected_}
};
fifo = FIFO{
  tasks =
    [ TASK{func = code.desc.AM_send_task},
      TASK{func = code.desc.calc_crc},
      TASK{func = code.desc.packet_sent),
      TASK{func = code.desc.packet_received}
    ],
  max_queue = 6
};
```

Every node in the scheduler hierarchy in Figure 2 has a corresponding instance definition in this description. Each object definition is of the form `Type{field1, ... fieldn}` where the `Type` determines which fields may be defined and their meaning, while fields define properties of the instance. The instance `code` is an embedded TinyOS system and contains a pointer to a TinyOS description file. EDL code can refer to objects inside the TinyOS system using references of the form `code.desc.<fieldname>`.

The types `AVR`, `IRQs`, `IRQ`, `FIFO`, and `TASK` used above are, in fact, "prototypes:" incomplete object instances to be completed later. We use prototypes whenever we want to reuse a piece of system description (of which types are just one example). We shall not show how to define prototypes in this paper.

As is common when comparing diagrams with text, the figure quickly conveys the gist of the design (and, for that reason, we shall use figures in the remainder of this paper) while the textual definition expresses details that would be hard to include in the figure. In this case, the textual definition specifies which code is used, the default interrupt handler to use if no interrupt handler is explicitly specified, and the maximum number (6) of tasks that can be in the scheduling queue at once.

Whereas execution environments form a tree, the code itself can form an arbitrary directed graph and a single piece of code is often connected either directly or indirectly to multiple parts of the execution environment hierarchy. In other words, the traditional modular decomposition of a system is often quite independent of the execution environment structure. For example, TinyOS code to control a radio transceiver might be accessed both by two separate interrupt handlers. It is this sharing that leads to race conditions: if every resource could only be attached to just one branch of the scheduler hierarchy, there would be no race conditions.

4.2 Interacting with Component Languages

As demonstrated in the previous section, EDL acts as a glue language for component languages such as TinyOS and Click. Since each component language may be internally quite complex and we wish to support a wide range of languages, our implementation is designed to work with the implementation of other languages rather than to fight against it. Thus, we make very few assumptions about any given component language while providing ways to take advantage of those that are able to provide more information.

The absolute minimum required of a component language is that it provides a standard interface for invoking the component language compiler, and that all use of schedulers, tasks, and locks is virtualized and adheres to a standard API. Interfacing with the runtime behavior is largely a matter of identifying the scheduler interactions and modifying the code to conform to our APIs. In many cases this can be done by modifying a few header files and a few lines of the scheduler. Interfacing with the build process is largely a matter of understanding a maze of Makefiles but is made feasible by the fact that we are primarily interested in eliminating the final linking step in order to permit what was once a complete system to be embedded in a larger system. Issues of name clashes between clients can be resolved either using the C preprocessor or using a tool to rename symbols in object files.

Component systems that provide only a "`.o`" interface necessarily force us to use externally acquired knowledge in order to bound their behavior, or to simply make pessimistic assumptions about them. Additional information about components can come from a variety of sources. For example, if we need to know for how long a particular TinyOS task runs in order to perform a real-time analysis, we can make inferences from how it is used (e.g., the task is called once per incoming packet, and therefore its run-time cannot be longer than the total time taken by TinyOS to process an incoming packet) or we can simply measure its run time and make the provisional assumption that its worst-case run time is not much different. Another way to learn about components is to mechanically analyze their source or object code (we do this for TinyOS systems in Section 5) or make use of properties that are known to be true of all valid compositions. For example, resources other than packet buffers in TinyOS are directly accessed by at most one component.

4.3 Encapsulating Foreign Schedulers

The ideal software component for CEE has a well-defined entry point that, when called, returns control to the caller within bounded time. In practice reuse of systems software is seldom so convenient, and we have developed a number of techniques for dealing with components with different control-flow characteristics. In general there are two ways to deal with components that do not gracefully return control: to alter them to change this behavior and to create a scheduling hierarchy such that they can be preempted when the need arises. We use both methods.

The TinyOS scheduler, for example, is easy to work with in the sense that it returns control to its caller when it has no work to perform. However, in practice all TinyOS kernels execute this scheduler inside an infinite loop that executes the AVR `sleep` instruction when there are no tasks to execute, putting the processor to sleep until the next interrupt arrives. We replicate the sleep-loop when using CEE to create kernels where TinyOS is the only user-level scheduler, but it must be replaced, for example, when running TinyOS tasks in a thread as we do in Section 7.

When system response-time requirements make it necessary to preempt TinyOS components, we run either a single TinyOS component, or an entire TinyOS scheduler, in a preemptible thread. Threads act as virtual processors and so we need a virtual version of the sleep-loop that native TinyOS uses; this is accomplished by having the thread block on a semaphore each time it has no work to perform. Since the scheduler is no longer implicitly awakened by an interrupt, an explicit signal must be sent to the semaphore each time work is sent to the corresponding TinyOS scheduler. The CEE linker automatically generates the code to do this, when necessary, by inspecting the path between the task being posted and the root of the scheduling hierarchy.

The AvrX RTOS, which we currently use as our thread implementation, never returns control once called. We could arrange for it to return control, or to be preemptible, but we have not done so. Consequently, AvrX does not compose well: other environments can be run in AvrX threads, but not the other way around. AvrX has no idle thread, but rather invokes the processor `sleep` function when it has nothing to run.

4.4 Linking Components

The EDL compiler emits a lexically flat linking specification that is passed to the *CEE linker* whose job it is to generate a system image. The linking specification encodes the type of each scheduler in the hierarchy as well as extra information such as priorities, stack sizes, and the like. The linker is general purpose but contains non-generic modules to deal with specific schedulers. For example, when an AvrX thread is instantiated, the linker calls a routine that generates code to statically allocate a thread control block and to dynamically initialize the thread as part of the overall system initialization. The modules may also perform additional scheduler-specific checking. An extreme example is the modules for the AVR processor: since we cannot generate fresh hardware matching the user's specification, the module can only check that the scheduler connections match those provided by the physical processor.

5 Task / Scheduler Logic

CEE adds flexibility to systems software. Increased flexibility creates the potential for new kinds of programmer errors and therefore it is important that we have the ability to detect these errors. In particular, concurrency problems that are notoriously difficult to reproduce and fix.

This section defines the "Task / Scheduler Logic" (TSL) that we use to detect race conditions in systems. TSL consists of a language for describing concurrency-related properties of schedulers and code, and a logic for reasoning about compositions of these components (i.e., schedulers and ordinary code). The advantage that TSL has over traditional concurrency languages and logics is that it focuses on a specific domain: it makes direct use of low-level hierarchical scheduling relationships that are present in most kinds of systems software. At present TSL only checks for race conditions; we would like to extend TSL to check liveness properties in the future.

In designing TSL (and, indeed, all of CEE), we had two usability goals in mind: that TSL should provide benefit even at low levels of programmer effort/investment; and that TSL should be usable by "normal" programmers. To achieve our first goal, we took care that our system would work with no input annotations and give progressively better results as the precision of the annotations improved. To achieve our second goal, we restricted the form of TSL specification to what we hoped systems programmers could quickly understand and could use correctly. We also restricted the need to write TSL specifications to the less common parts of system design where considerable expertise and understanding is already required (like implementing a new scheduler or supporting a new processor). Where this could not be done, we wrote tools to automate the process.

TSL is primarily concerned with reasoning about when tasks can run and conflicts between running tasks. When tasks can run is determined by the scheduler hierarchy and the use of locks, while conflicts between running tasks are determined by the function callgraph and which resources each function accesses.

We use the following naming conventions in the remainder of this section. The variables s, $s1$, etc. range over schedulers; the variables e, $e1$, etc. range over entrypoints; the variables r, $r1$, etc. range over resources; and the variables l, $l1$, etc. range over locks.

5.1 TSL Specifications

There are two aspects to TSL: what the user sees and how we find races in TSL specifications. This section is about what the user sees.

We treat tasks as a degenerate form of scheduler. Schedulers are required to form a tree: a parent scheduler decides if and when its children run. We write $s1 \lhd s2$ when $s1$ is the parent of $s2$. Schedulers are the only mechanism that can cause two tasks to run concurrently. That is, multiprocessors, interrupt mechanisms, etc. are all modeled as schedulers. For each scheduler s, we define $\rightarrow_s$ to be a relation between child schedulers such that $s1 \rightarrow_s s2$ if $s2$ can start to run while $s1$ is running. A number of common relations between tasks can be expressed in terms of $\rightarrow$:

- $s1$ is strictly higher priority than $s2$ on a uniprocessor scheduler is modeled by $\neg(s1 \rightarrow s2) \wedge (s2 \rightarrow s1)$. Although priority does not have a first-class representation in TSL, it is an important concept since systems that are described in TSL will often map to implementations containing priority-based schedulers.

- $s1$ and $s2$ may run simultaneously (may preempt each other on a uniprocessor scheduler, or may be scheduled by different processors on an SMP) is modeled by $s1 \rightarrow s2 \wedge s2 \rightarrow s1$.

- $s1$ and $s2$ cannot run simultaneously (are mutually non-preemptible on a uniprocessor scheduler) is modeled by $\neg(s1 \rightarrow s2) \wedge \neg(s2 \rightarrow s1)$.

Although TSL is applicable to multiprocessor systems, we restrict our attention to uniprocessor systems for the remainder of this paper. Thus, $s1 \rightarrow s2$ iff $s2$ can preempt $s1$.

In this paper, we use the term 'lock' to mean any mechanism used to make a section of code atomic with respect to another piece of code including spinlocks, disabling interrupts, and traditional thread locks. That is, a *lock* is anything which can be taken and released and which overrides concurrency relations introduced by schedulers. For each lock l, we specify the set b_l of schedulers that cannot start to run if l is held. Each lock is provided by a particular scheduler and b_l may only contain children of that scheduler. A consequence of this is that TSL rejects the classic programming error of attempting to block on a thread lock in an interrupt handler as a malformed TSL specification.

Since CEE is concerned with schedulers, we only need a simple model of code. In fact, we provide two complementary ways of specifying code. (We motivate the need for two methods in Section 5.4.)

In the first (traditional) method, code consists of *entrypoints* and *resources*. Entrypoints are functions that may be invoked and resources are, by definition, things that must be accessed atomically. Entrypoints may acquire locks before accessing resources. We say that an entrypoint e accesses a resource r with a set of locks ls (written $e \downarrow_{ls} r$) if all the locks in ls are acquired before any access of r by any function directly or indirectly invoked by e. When a scheduler (typically a task) s invokes an entrypoint e, we write $s \lhd e$.

In the second method, we annotate entrypoints with the current set of preemptions that may occur and use TSL to report preemptions introduced by changes to the scheduling hierarchy. This amounts to trusting the original system designer to get the design right: a not unreasonable starting point. We write $e1 \leadsto_{ls} e2$ if it is considered safe for entrypoint $e2$ to be invoked while $e1$ is running and holding a set ls of locks.

The above are all local properties: they can be determined by independently inspecting schedulers, components, and the interconnections in the EDL specification. To determine the behavior of the complete system, we must combine these individual properties.

5.2 Reasoning about TSL Specifications

Race conditions occur when two tasks (or two instances of the same task) can execute simultaneously and both access the same resource. Thus we must determine when two tasks can execute simultaneously. We do this by extending the relation $\rightarrow_s$ and the property b_l.

Definition: *For a scheduler s, we define the relation $\Rightarrow_s$ to be the least relation containing $\rightarrow_s$ such that:*

$$s1 \Rightarrow_s s2 \wedge s1 \lhd s1' \implies s1' \Rightarrow_s s2$$

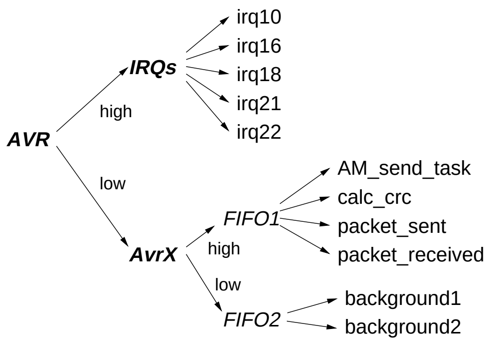

Figure 3: Composing TinyOS and AvrX

and

$$s1 \Rightarrow_s s2 \wedge s2 \lhd s2' \implies s1 \Rightarrow_s s2'$$

That is, if a scheduler s allows its children to run concurrently, then so can their descendents.

Definition: *For a lock l, we define the set B_l to be the least set containing b_l such that:*

$$s \in B_l \wedge s \lhd s' \implies s' \in B_l$$

That is, if a lock blocks a scheduler s, then it blocks any descendent s'.

With these definitions in hand, we can state precisely when two components can access a resource simultaneously:

Definition: *A* race condition *exists when two entrypoints access the same resource concurrently.*

$$\begin{array}{rcl} \mathit{race}(e1, e2, r) & \overset{def}{=} & e1 \Rightarrow_s e2 \\ & \wedge & e1 \downarrow_{ls1} r \\ & \wedge & e2 \downarrow_{ls2} r \\ & \wedge & \neg(\exists l \in ls1 : e2 \in B_l) \\ & \wedge & \neg(e1 \leadsto_{ls1} e2) \end{array}$$

In other words, a race exists if $e1$ and $e2$ both access a resource where: there exists a scheduler s that permits $e2$ to preempt $e1$, $e1$ does not hold any locks that overrides that preemption, and the user has not indicated that it is safe for $e2$ to preempt $e1$. A corollary of this definition is that if the nearest scheduler in the hierarchy common to $e1$ and $e2$ is non-preemptive, then a race condition cannot exist between $e1$ and $e2$. (Note: In fact, a race can only occur if at least one accessor modifies the resource — a detail we shall ignore in this paper.)

5.3 A Simple TSL Example

In general it is easier to meet deadlines in preemptively scheduled systems, while non-preemptive systems are easier to develop. Composing execution environments hierarchically affords us the potential to have the best of both worlds. Consider the example in Figure 3 where AvrX runs a non-preemptive TinyOS task scheduler in each of two threads.

We start by specifying the scheduler hierarchy:

$$\begin{array}{rcl c rcl}
\text{avr} & \lhd & \text{irqs} & & \text{avr} & \lhd & \text{avrx} \\
\text{irqs} & \lhd & \text{irq10} & \cdots & \text{irqs} & \lhd & \text{irq22} \\
\text{avrx} & \lhd & \text{fifo1} & & \text{avrx} & \lhd & \text{fifo2} \\
\text{fifo1} & \lhd & \text{AM\_send\_task} & \cdots & \text{fifo1} & \lhd & \text{packet\_received} \\
\text{fifo2} & \lhd & \text{background1} & & \text{fifo2} & \lhd & \text{background2}
\end{array}$$

and the preemption relations

$$\begin{array}{rcl c rcl}
\text{avrx} & \rightarrow_{AVR} & \text{irqs} & & & & \\
\text{irq10} & \rightarrow_{IRQs} & \text{irq10} & \cdots & \text{irq10} & \rightarrow_{IRQs} & \text{irq22} \\
 & & & \ddots & & & \\
\text{irq22} & \rightarrow_{IRQs} & \text{irq10} & \cdots & \text{irq22} & \rightarrow_{IRQs} & \text{irq22} \\
\text{fifo2} & \rightarrow_{AvrX} & \text{fifo1} & & & &
\end{array}$$

From this we can derive additional preemption relations and the impossibility of some preemptions. For example,

$$\begin{array}{rcl}
\text{AM\_send\_task} & \rightarrow & \text{irq10} \\
\text{background1} & \rightarrow & \text{AM\_send\_task} \\
\neg(\text{AM\_send\_task} & \rightarrow & \text{calc\_crc}) \\
\neg(\text{calc\_crc} & \rightarrow & \text{AM\_send\_task}) \\
\neg(\text{background1} & \rightarrow & \text{background2}) \\
\neg(\text{background2} & \rightarrow & \text{background1})
\end{array}$$

For any resources that are accessed only by tasks AM_send_task and calc_crc, or by tasks background1 and background2, a TSL derivation tells us that no locks are needed: the tasks are already mutually atomic. On the other hand, if tasks AM_send_task and background1 share a resource, or if AM_send_task shares a resource with irq10, then a lock is needed. TSL tells us that the lock to protect from an interrupt handler can be one-sided, because part of the definition of "interrupt handler" is that it executes atomically with respect to user mode. The lock enforcing atomic access to a resource shared by two tasks, on the other hand, must be two-sided. It can, however, be implemented by either of the two preemptive schedulers between the tasks and the root of the scheduling hierarchy, i.e., by taking a thread lock in the AvrX scheduler or by disabling interrupts in the AVR scheduler.

5.4 Implementing TSL

The TSL specifications for components are part of the EDL specification of that component: each component contains a string field called "tsl." For example the specification of fifo2 could be written like this:

```
fifo2 = FIFO{
  task1 = ...;
  task2 = ...;
  tsl = << EOF
    ${^} <| ${^.task1};
    ${^} <| ${^.task2};
EOF
};
```

The EDL compiler expands the references in the tsl string into:

```
    fifo1 <| fifo1.task1;
    fifo1 <| fifo1.task2;
```

A separate TSL checker parses all the tsl fields, translates them into Prolog statements, and invokes a Prolog program to check for races.

Putting the TSL specifications in the instance definitions like this would work but it would lead to poor reuse and require all users to write their own TSL specifications of each component. Instead, we put the TSL specification in the prototype `FIFO`, the prototype `AVR`, etc. so that specifications of schedulers and of TinyOS applications may be shared between all systems.

TSL specifications for schedulers are generally written by hand. This is feasible because there are few schedulers and because a user choosing a new scheduler is usually interested in precisely those properties of the scheduler that TSL describes. To aid us in generating TSL specifications for the remaining TinyOS components, we wrote a simple code analyzer that treats each byte of memory and each I/O port as an individual resource and treats each of the 24 independent interrupt disabling mechanisms as locks. This model is rather lax (it is often necessary to treat a set of variables as a single logical resource) but even so, TSL reported many race conditions in working TinyOS systems. On inspection, the problem is that TinyOS uses lock free synchronization in many cases and our analysis tool cannot identify these uses. This motivated the complementary approach of using $\rightsquigarrow$ to specify which entrypoints may run concurrently. This approach has proven to be simple and effective in actual use.

5.5 Synchronization Inference

Thus far, we have described the use of TSL to *check* that the user's choices of locks and schedulers are correct. In fact, TSL has enough information to make the choices of locks itself in almost all cases. If the user declares a lock as being "virtual," the TSL checker will search for a choice of physical lock that eliminates any races. A useful special case of synchronization inference is when a particular composition of environments does not permit concurrent access to a protected resource: in this case the locks can be dropped from the implementation.

We currently use a simple heuristic to pick a lock implementation. In the future we want to integrate lock choice with real-time analysis: this will be useful because the choice of locking has global side effects. For example, disabling interrupts is often the most efficient lock implementation, but it disables all preemptive scheduling in the system for the duration of the critical section. Thread locks, on the other hand, are less efficient but have fewer side effects.

6 Real-Time Scheduling

The scheduling model that underlies CEE is based on extensions to the existing response-time analysis for fixed-priority scheduling [28]. First, we make use of a model developed by Saksena and Wang [23] that shows how to map a *design model*, consisting of a set of preemp-

	Tasks 1–4		Tasks 5–6		CPU	
scenario	WCET	deadline	WCET	deadline	utilization	result of analysis
1	1	10	1	10	60%	schedulable w/o preemption
2	1	10	10	100	60%	schedulable with preemption
3	1	10	Task 5: 1 Task 6: 10	Task 5: 10 Task 6: 100	60%	*not schedulable*

Figure 5: Three different scenarios for the environment hierarchy in Figure 4

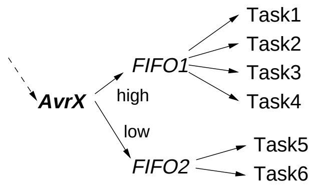

Figure 4: Scheduling hierarchy fragment for the real-time example

tive tasks with real-time requirements, to an *implementation model* that usually contains significantly fewer threads than the design model contained tasks. Saksena and Wang's contribution was to show how to perform this transformation without causing deadlines to be missed. We believe that techniques like this are important for embedded systems, where the memory overhead of threads is significant, and in server systems, where the context switches associated with threads can add significant CPU overhead. Furthermore, when synchronizing tasks are mapped to the same thread, TSL gets a chance to eliminate locks that have become superfluous.

We have extended Saksena and Wang's scheduling model to incorporate *task clusters*, or collections of tasks with real-time requirements that are designed to be mutually non-preemptible. In other words, while Saksena and Wang showed how introduce non-preemption into a preemptive system as a performance optimization, we showed how to make structured mixes of preemption and non-preemption into a first-class part of the programming model, while still meeting real-time deadlines [21]. We have implemented SPAK, a tool that can check the schedulability of systems containing task clusters; it serves as the real-time analysis tool for CEE. By using task clusters it is easy to embed, for example, a collection of TinyOS components in a real-time system without (1) breaking the invariant that TinyOS tasks be mutually non-preemptible, (2) making the system globally non-preemptible, or (3) sacrificing global schedulability analysis.

We introduced task clusters in a previous paper [21]. In this paper we integrate task clusters into CEE's hierarchical scheduling model: Section 7.3 provides an example showing how this works.

6.1 A Schedulability Example

Figure 4 is a subset of Figure 3, relabeled for easier discussion. Figure 5 shows three different scenarios for the real-time properties of tasks 1–6. In all three scenarios, tasks 1–4 have a worst-case execution time (WCET) of 1 ms and a period of 10 ms. In other words, each of tasks 1–4 may be invoked at most once every 10 ms, runs for at most 1 ms, and must finish within 10 ms after starting to run.

In scenario 1, tasks 5 and 6 each have WCET 1 ms and period 10 ms. In this case the overall task set can be shown to be schedulable with a non-preemptive scheduler. Informally, notice that even if all tasks become ready at the same time (this is the worst case for many scheduling scenarios) all components finish executing 6 ms later, meeting the 10 ms deadline of each component. So scenario 1 does not require a preemptive scheduler at all — AvrX could be eliminated from the picture in order to save memory and reduce processor overhead that comes from preemptions.

In Scenario 2 of Figure 5, tasks 5 and 6 have a worst-case execution time of 10 ms and run at most every 100 ms. This task set is not schedulable by a non-preemptive scheduler. To see this, assume that task 5 is signaled and begins to run, and that task 1 begins to run a short time later. By the time task 5 finishes executing, 10 ms later, task 1 will have missed its deadline. This task set, however, is schedulable with a preemptive scheduler: since its utilization is below the rate monotonic bound of 69%, it can be scheduled by a fixed priority scheduler [18] as long as scheduler FIFO1 is assigned a higher priority than FIFO2.

Scenario 3 is not schedulable under any scheduling discipline given the constraint that task 6 cannot be preempted by task 5. The developers of a system containing this task set have several options. All of the options require code to be restructured: there is simply no way around this. First, the run-time of task 6 can be reduced by optimizing it, or it might be possible to increase the deadline of task 5 by adding buffering somewhere in the system. Second, task 6 could be broken up into several segments, each with a shorter worst-case execution time. And finally, the mutual non-preemption relations between tasks 5 and 6 could be broken — this would require that developers add locks to protect any resources that tasks 5 and 6 share.

	Size (bytes)		Latency (μs)		
TinyOS Version	Code	Data+BSS	Pin – Int	Int – Task	Task – Task
Original v0.6	5022	232	11.3	22.5	16.0
CEE baseline	5068	234	11.3	22.5	16.0
2-Level	6094	448	11.3	46.7	16.0 / 45.2
Ints in Thrds	6210	896	124	96.7	16.0

Figure 6: Static and dynamic overhead for TinyOS/AvrX configurations of the cnt_to_rfm kernel

6.2 Generating Schedules

When system developers specify the priorities of real-time tasks instead of specifying requirements such as period, deadline, and worst-case execution time, the real-time correctness of tasks cannot be verified. If priorities are specified in addition to real-time requirements, then the correctness of the priorities can be checked using a response time analysis such as the one in SPAK. However, in this case there is no need to specify the priorities: they can be generated by a suitable scheduling algorithm. Similarly, when collections of tasks that are designed to be mutually non-preemptible are given to SPAK, it can check their schedulability and it can also attempt to further reduce the number of preemptive threads used to run the task set.

Performing real-time analysis of an entire system is burdensome and, indeed, often unnecessary. Without any special support from SPAK, it is valid to simply ignore the non-real-time parts of a system as long as the ignored subsystems meet the *non-interference* property: they must not prevent the real-time part from meeting its deadlines. This is the insight behind RTLinux [31], which achieves non-interference by running at higher priority than Linux, and by virtualizing Linux's interrupt handling subsystem. In practice, non-real-time parts of the system usually contribute *blocking terms* to the real-time analysis because they spend time running with interrupts disabled or otherwise being non-preemptible. SPAK uses previously developed techniques [24] to support analysis of systems with blocking terms.

7 Case Study: Applying CEE to TinyOS

TinyOS is easy to use and does a good job of meeting its design goals. However, as sensor node hardware evolves and as people attempt to use it in more diverse situations (for example, the COTS-BOTS project at Berkeley [6] is using TinyOS motes to control robots) it is likely that changes will need to be made to its software architecture. The goal of this section is to anticipate some of these changes and to show that they can be accomplished within the CEE framework with minimal disruption of the TinyOS programming model.

We use a variety of metrics to evaluate the results. Some, such as code and data size, time to respond to an externally triggered interrupt, time to post a task from an interrupt, and time to post a task from a task, are shown in Figure 6. We discuss these numbers in subsequent sections. Although the latency numbers in Figure 6 are averages over at least 100 measurements, we do not report confidence intervals because the results were extremely predictable: the variation between measurements was below 100 ns in all cases.

All experiments for this section were run on "mica" motes, based on the Atmel ATmega103 CPU [3], running at 4 MHz and with 4 KB of SRAM and 128 KB of flash memory. These motes have a radio capable of sending up to 50 Kbps. We used TinyOS version 0.6 and AvrX version 2.6. Timing measurements were taken by setting output pins on the AVR and observing the results using a Tektronics TDS784D digital oscilloscope and an HP 16500A logic analyzer with an HP 16510B module. While taking measurements we set sampling rates to 25 MHz.

7.1 Reconstructing TinyOS

This experiment is the control, and it evaluates our ability to reconstruct a version of TinyOS that is essentially identical to the one from Berkeley. We used CEE to generate the `cnt_to_rfm` kernel, which periodically sends packets over the radio containing three-bit counter values. Its scheduling hierarchy is the one shown in Figure 2.

The results of this experiment can be seen by comparing the "original v0.6" and "CEE baseline" lines in Figure 6. The kernel generated by CEE is slightly larger because the CEE linker generates a few extra functions that are not inlined away. There is no significant difference in the time taken by these two kernels to respond to an externally triggered interrupt (Pin-Int), the time to post a task from an interrupt (Int-Task), and the time to post a task from a task (Task-Task). In fact, although the code is arranged slightly differently, the instructions generated for the schedulers of both kernels are exactly the same.

7.2 Adding a Second Level of Deferred Processing

Since tasks — the deferred processing mechanism within TinyOS — are not preemptible, care must be taken to avoid letting them run for too long. Specifically, any tasks with response time requirements shorter than the execution time of the longest-running task will not

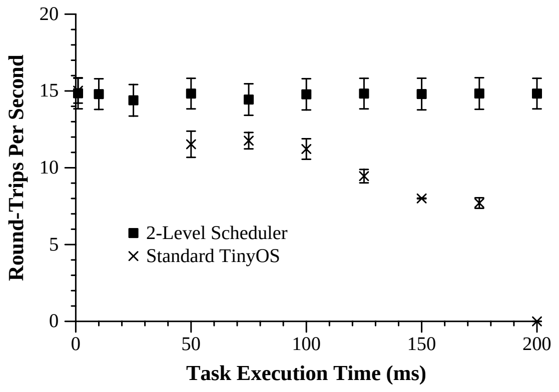

Figure 7: A CPU-intensive task run every 250 ms degrades network throughput of a standard TinyOS kernel; a kernel with a second level of deferred processing is not affected

always meet their deadlines. This kind of problem will become more common as mote hardware becomes capable, for example, of running sophisticated algorithms such as Fourier transforms and public key cryptography. Our solution is to run the TinyOS task scheduler in an AvrX thread, and to run a second instance of the TinyOS task scheduler in a second, lower priority thread. This is the environment hierarchy depicted in Figure 3. The low-priority scheduler runs *background tasks* that cannot interfere with the response times of foreground tasks.

To evaluate the utility of this modified TinyOS environment structure, we used existing TinyOS components to create a "ping responder" kernel that replies to packets that it receives over the radio. To measure the performance of the ping responder we had it ping-pong packets with another mote. If the ping responder took more than 500 ms to reply, the sending node considered a packet loss to have occurred and resent a packet. The average baseline performance of this setup, with TinyOS kernels created by the Berkeley toolchain, is 15.4 round-trips per second.

We modeled a CPU-intensive task on the ping responder by using a timer interrupt to post a task four times per second. The experimental procedure was to vary the run-time of the CPU-intensive task while measuring ping throughput on a TinyOS kernel compiled by the Berkeley toolchain, and a TinyOS/AvrX kernel with a two-level scheduler created by CEE. Figure 7 shows the results of this experiment: the CPU-intensive task interferes with throughput when run by the foreground scheduler, but not when run by a background scheduler. Confidence intervals are computed at 95%. We do not have data points for standard TinyOS for 10 and 25 ms because the network subsystem crashed inexplicably (we hope to have traced the cause of this problem by the time of the final paper), or at 200 ms because no packets were reliably returned at this point.

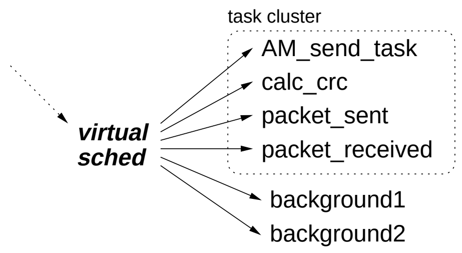

Figure 8: Scheduling hierarchy before schedulability analysis

The "2-Level" line of Figure 6 shows that including AvrX adds about a kilobyte of code and 224 bytes of data to TinyOS; this represents less than 1% of the flash memory and less than 3% of the SRAM on the ATmega103. Posting a task from an interrupt takes about 24 μs longer than in the standard TinyOS kernel since it involves signaling a semaphore and switching to a thread context. The two task-task numbers respectively indicate the cost to post a task within the same scheduler, and to post a task to the other scheduler.

7.3 Automatic Schedule Generation

The approach in the previous section uses manual priority assignment: developers must specify which scheduler each task runs on. This complicates the programming model and is probably a poor design in the long run because developers will often want to reuse existing TinyOS components, and may not know how long tasks specified in these components run at each priority.

An alternative solution to creating a background scheduler is to use SPAK to generate a mapping of tasks to schedulers. We accomplish this by using EDL to specify that TinyOS tasks are scheduled by a "virtual scheduler" as depicted in Figure 8. Tasks scheduled by a virtual scheduler can belong to task clusters and have real-time properties such as periods and worst-case execution times. The linking specification containing virtual schedulers is run through SPAK, which converts the virtual scheduler into a number of non-preemptive schedulers, each of which runs in a preemptive thread. SPAK's goal is to instantiate as few threads as possible. Ideally, only a single thread is required in which case the preemptive scheduler can be omitted.

Assigning real-time parameters to TinyOS tasks is not always straightforward: the predominant scheduling models, including the one we use, are based on *periodic* tasks that recur after a fixed time interval. However, the periodic task model can be used to analyze non-periodic tasks as long as a minimum time between arrivals of the tasks can be found. Since the ping responder kernel from the previous section attains at most about 15 packets per second, and because all of the tasks in that kernel are related to packet processing, we assume that the tasks have a 7 ms period. In this kernel task execution

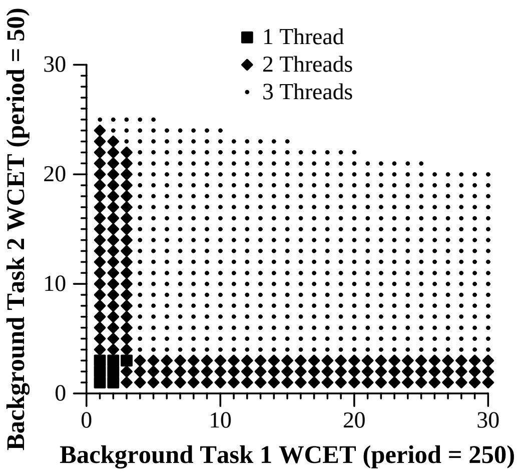

Figure 9: Effect of varying task requirements on the number of threads required to permit all deadlines to be met

times are very deterministic, with AM_send_task running for about 150 μs, calc_crc running for 3160 μs, and packet_sent running for 17 μs. Since these tasks share packet buffers, we put them into a task cluster to ensure that they will not preempt each other in the generated system.

To evaluate the automatic schedule generation, we assume that the radio tasks will be running alongside two CPU-intensive tasks with periods of 250 ms and 50 ms. We then vary the execution time requirement of the CPU-intensive tasks. Figure 9 shows, for a variety of task parameters, the smallest number of threads that SPAK could instantiate while still guaranteeing that all deadlines will be met. Blank space in the figure indicates the region where no feasible schedule exists.

No lines exist in Figure 6 to describe the performance of TinyOS kernels with schedules generated by SPAK. This is because their performance is the same as the performance of the two-level scheduler. The difference between the two approaches is in the programming model, not in the generated code.

7.4 Making TinyOS Interrupts Preemptible

The motivation for our final TinyOS execution environment modification is a high-frequency pulse-width modulation (PWM) application. PWM is used to perform fine-grained software control of power sent to devices such as motors and speakers. The ATmega103 chips are capable of running a hand-written interrupt handler up to about 100 KHz, or every 10 μs.

We found that interrupt handlers in common TinyOS kernels sometimes disable interrupts for up to about 80 μs, preventing reliable PWM over 12.5 KHz ($1/80\,\mu$s). Since AvrX disables interrupts for at most about 15 μs, we reasoned that running the TinyOS interrupt handling code in high-priority AvrX threads would

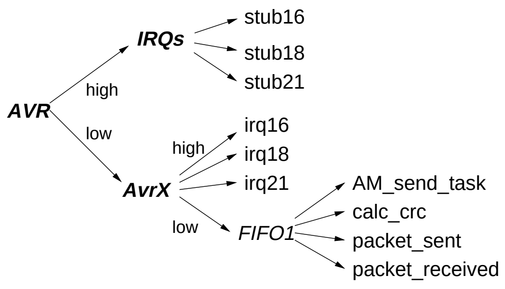

Figure 10: Running TinyOS interrupt handlers in threads

be a good way to keep hardware interrupt latency low, permitting PWM interrupts up to 66 KHz. In the modified kernels, an AvrX semaphore is used to provide the atomicity properties between interrupt handlers that were previously achieved by running with interrupts disabled.

This arrangement of execution environments, shown in Figure 10, is analogous to architectures such as MERT [5], RTLinux [31], and TimeSys Linux/GPL [27]. We expected it to just work, and in fact it does work in TinyOS kernels that do not use the radio. There were two problems running radio interrupt handlers in threads. First, the radio on the motes is software-driven at the bit level, and an interrupt handler polls the radio every 50 μs. However, the context switch to and from an AvrX thread takes longer than 50 μs. Polling the radio less often would solve this problem but prevents the motes from receiving radio packets (they can still send). The second obstacle is that the serial peripheral interface (SPI) interrupt for the radio must be serviced within a handful of microseconds after being asserted. Bits and, hence, packets are lost if this deadline is missed. The time that it takes to dispatch an AvrX thread is considerably longer than the deadline for servicing this interrupt. This too can be worked around by running the time-critical part of the interrupt handler, which is only a handful of instructions, in the "real" interrupt handler, and the rest in a thread.

The lesson to be learned from this example is that systems software can sometimes be too sensitive for functionally transparent changes to execution environments to produce a working system. Had we known about the extremely short deadline of the SPI interrupt, and if SPAK contained comprehensive support for system overheads like interrupt dispatch times and preemptive vs. non-preemptive context switch times, it could have told us that a naive application of the architecture in Figure 10 would not work.

The "ints in thrds" line of Figure 6 shows that it takes 124 μs to start running the TinyOS interrupt handling code once an interrupt is asserted, and that it takes 96.7 μs to start running a task from an interrupt. These numbers are long because of a known problem in AvrX:

in some circumstances it initiates a wasteful context switch in the code to signal a semaphore.

8 Related Work

Ousterhout [20] claimed that threads are fundamentally the wrong programming abstraction for most systems. Burns and Wellings [7] advocate allowing developers to pick from a variety of restricted scheduling models for real-time ADA that have different levels of generality and analyzability. Adya et al. [1] describe the conflict between thread- and event-based programming within their programming team and have developed a programming style that allows the two styles to interact safely. These works, and many others, provide arguments for choosing one execution environment over another as well as some techniques to let multiple environments coexist. However, they provide little help in building systems out of generic environments or in changing the set of environments and their relationships to each other.

Many research systems have component-based software and explicit software architectures, such as Click [14], TinyOS [11], Koala [29], Ptolemy [17], VEST [25], SEDA [30], and our own Knit [22]. Most of these systems can also analyze properties of component compositions and a few of them allow heterogeneous component languages to be embedded inside them. Our work differs in that we focus on reasoning about the hierarchical scheduling relationships created by compositions of environments, and on providing tools and techniques that make it as easy as possible to create new compositions of environments.

Concurrency languages/logics such as temporal logic [15], CSP [12], and the pi-calculus [19] address concurrency issues in a more general way than does TSL. By limiting the domain of applicability and the expressive power of TSL, we hope to make it simpler for systems programmers to use. It is difficult to imagine typical systems programmers routinely writing temporal logic specifications of their systems.

Needham and Lauer's result about the duality of process- and message-based operating systems [16] suggests that TSL could have been based on resources rather than on schedulers. In our experience, however, a scheduler-centric view of systems software gets to the heart of the concurrency and atomicity issues that we are interested in.

Meta-compilation [9] is an extensible framework for developing state-based checkers, and ESC [8] looks for common programming errors such as null pointer dereferences and out-of-bounds accesses. Flanagan and Abadi [10] use types to detect race conditions and Aldrich et al. [2] show how to eliminate synchronization from Java programs when doing so does not affect correctness. These are all examples of lightweight formal methods that use annotation and analysis to detect incorrect programs. Our TSL checker is similar in intent but, because it focuses on analyzing scheduler hierarchies instead of code, is complementary to these tools.

9 Conclusion

An important difference between systems software and application code is that systems software executes in a wider variety of execution environments. These environments are both a source of difficulty when developing software and a source of opportunities for systems software to meet its requirements. We have shown that the composable execution environment model can help tame the complexity of interacting environments. First, we have described EDL: a language for specifying hierarchical compositions of execution environments. We have shown that EDL can include existing execution environments such as TinyOS. Second, we have described TSL, the task / scheduler logic, which makes it possible to reason about safety properties of environment hierarchies specified in EDL. Third, we have shown how to analyze the schedulability of environment hierarchies that use a mix of preemptive and non-preemptive fixed priority scheduling. And finally, we have shown that CEE can be used to change the execution environment structure of TinyOS kernels in useful ways.

Acknowledgments: The authors would like to thank Mike Jones and Fred Schneider for helpful advice during early stages of this work, and David Culler for getting us some motes on short notice. Thanks also to Mike Parker who helped enormously with the logic analyzers.

References

[1] Atul Adya, Jon Howell, Marvin Theimer, William J. Bolosky, and John R. Douceur. Cooperative task management without manual stack management. In *Proc. of the 2002 USENIX Annual Technical Conf.*, Monterey, CA, June 2002.

[2] Jonathan Aldrich, Craig Chambers, Emin Gün Sirer, and Susan Eggers. Eliminating unnecessary synchronization from Java programs. In *Proc. of the Static Analysis Symp.*, Venezia, Italy, September 1999.

[3] Atmel, Inc. ATmega103 datasheet, 2001. `http://www.atmel.com/atmel/acrobat/doc0945.pdf`.

[4] Larry Barello. The AvrX real time kernel. `http://www.barello.net/avrx/`.

[5] Doug Bayer and Heinz Lycklama. MERT – A multi-environment real-time operating system. In *Proc. of the 9th ACM Symp. on Operating Systems Principles*, pages 33–42, Austin, TX, November 1975.

[6] Sarah Bergbreiter and Kristofer Pister. COTS-BOTS specifications, March 2002.

`http://www.eecs.berkeley.edu/~sbergbre/CotsBots/specs.pdf`.

[7] Alan Burns and Andy J. Wellings. Restricted tasking models. In *Proc. of the 8th Intl. Real-Time Ada Workshop*, pages 27–32, Ravenscar, UK, December 1997.

[8] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. Research Report 159, Compaq Systems Research Center, Palo Alto, CA, December 1998.

[9] Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proc. of the Fourth Symposium on Operating Systems Design and Implementation*, pages 1–16, San Diego, CA, October 2000. USENIX Association.

[10] Cormac Flanagan and Martin Abadi. Types for safe locking. In S.D. Swierstra, editor, *ESOP'99 Programming Languages and Systems*, volume 1576 of *Lecture Notes in Computer Science*, pages 91–108. Springer-Verlag, March 1999.

[11] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, and Kristofer Pister. System architecture directions for networked sensors. In *Proc. of the 9th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 93–104, Cambridge, MA, November 2000.

[12] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Englewood Cliffs, NJ, 1985.

[13] Michael B. Jones. What really happened on Mars?, December 1997. `http://research.microsoft.com/~mbj/Mars_Pathfinder/`.

[14] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The Click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, August 2000.

[15] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[16] Hugh C. Lauer and Roger M. Needham. On the duality of operating systems structures. *ACM Operating Systems Review*, 13(2):3–19, April 1979.

[17] Edward A. Lee. Overview of the Ptolemy project. Technical Memorandum UCB/ERL M01/11, University of California at Berkeley, March 2001.

[18] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, January 1973.

[19] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes (parts I and II). *Information and Computation*, 100:1–77, 1992.

[20] John K. Ousterhout. Why threads are a bad idea (for most purposes), 1996. Invited talk at the 1996 USENIX Technical Conference.

[21] John Regehr. Scheduling tasks with mixed preemption relations for robustness to timing faults. In submission, May 2002. `http://www.cs.utah.edu/~regehr/papers/mixed/`.

[22] Alastair Reid, Matthew Flatt, Leigh Stoller, Jay Lepreau, and Eric Eide. Knit: Component composition for systems software. In *Proc. of the 4th Symp. on Operating Systems Design and Implementation*, pages 347–360, San Diego, CA, October 2000.

[23] Manas Saksena and Yun Wang. Scalable real-time system design using preemption thresholds. In *Proc. of the 21st IEEE Real-Time Systems Symp.*, Orlando, FL, November 2000.

[24] Lui Sha, Ragunathan Rajkumar, and John Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, 1990.

[25] John A. Stankovic, Hexin Wang, Marty Humphrey, Ruiqing Zhu, Ramasubramaniam Poornalingam, and Chenyang Lu. VEST: Virginia embedded systems toolkit. In *Proc. of the IEEE/IEE Real-Time Embedded Systems Workshop*, London, UK, December 2001.

[26] David B. Stewart, Richard A. Volpe, and Pradeep K. Khosla. Design of dynamically reconfigurable real-time software using port-based objects. *IEEE Transactions on Software Engineering*, 23:759–776, December 1997.

[27] TimeSys Linux/GPL. `http://www.timesys.com/prodserv`.

[28] Ken Tindell, Alan Burns, and Andy J. Wellings. An extendible approach for analysing fixed priority hard real-time tasks. *Real-Time Systems Journal*, 6(2):133–151, March 1994.

[29] Rob van Ommering. Building product populations with software components. In *Proc. of the 24th Intl. Conf. on Software Engineering*, Orlando, FL, May 2002.

[30] Matt Welsh, David Culler, and Eric Brewer. SEDA: An architecture for well-conditioned scalable internet services. In *Proc. of the 18th ACM Symp. on Operating Systems Principles*, Banff, Canada, November 2001.

[31] Victor Yodaiken. The RTLinux manifesto. In *Proc. of The 5th Linux Expo*, Raleigh, NC, March 1999.