Modeling and Analysis of TinyOS Sensor Node Firmware: A CSP Approach

ALLAN I. MCINNES, University of Canterbury

Wireless sensor networks are an increasingly popular application area for embedded systems. Individual sensor nodes within a network are typically resource-constrained, event-driven, and require a high degree of concurrency. This combination of requirements motivated the development of the widely-used TinyOS sensor node operating system. The TinyOS concurrency model is a lightweight non-preemptive system designed to suit the needs of typical sensor network applications. Although the TinyOS concurrency model is easier to reason about than preemptive threads, it can still give rise to undesirable behavior due to unexpected interleavings of related tasks, or unanticipated preemption by interrupt handlers. To aid TinyOS developers in understanding the behavior of their programs we have developed a technique for using the process algebra Communicating Sequential Processes (CSP) to model the interactions between TinyOS components, and between an application and the TinyOS scheduling and preemption mechanisms. Analysis of the resulting models can help TinyOS developers to discover and diagnose concurrency-related errors in their designs that might otherwise go undetected until after the application has been widely deployed. Such analysis is particularly valuable for the TinyOS components that are used as building-blocks for a large number of other applications, since a subtle or sporadic error in a widely-deployed building-block component could be extremely costly to repair.

Categories and Subject Descriptors: D.2.4 [**Software Engineering**]: Software/Program Verification—*Formal methods, Model checking*; D.4.7 [**Operating Systems**]: Organization and Design—*Real-time systems and embedded systems*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*mechanical verification*

General Terms: Design, Languages, Verification

Additional Key Words and Phrases: Firmware, wireless sensor network, TinyOS, nesC, CSP, process algebra

ACM Reference Format:
McInnes, A. I. 2011. Modeling and analysis of TinyOS sensor node firmware: a CSP approach. ACM Trans. Embedd. Comput. Syst. 0, 0, Article 00 (2011), 23 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

1. INTRODUCTION

Wireless sensor networks are becoming an increasingly widespread application of embedded systems. In the past decade they have moved from being used primarily in research applications to deployment in a wide range of industrial and commercial contexts. One of the most popular sensor network operating systems for both research and commercial use is TinyOS [Levis et al. 2005; Culler 2006]. TinyOS and the applications that run on it are written in nesC [Gay et al. 2003], a dialect of C that adds support for a component-based programming model. TinyOS applications are constructed as graphs of components that interact both with each other, and with the underlying

Author's address: A. I. McInnes, Department of Electrical and Computer Engineering, University of Canterbury, Private Bag 4800, Christchurch 8140, New Zealand.

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

TinyOS scheduler and platform hardware. Ensuring that all of these interactions are correct is an important part of developing a reliable TinyOS application.

Understanding how TinyOS application components interact with each other is complicated by the necessary inclusion of concurrency within the TinyOS model of computation. Concurrency is introduced by interrupts, which can cause one component to preempt the execution of another, and by tasks, which are a way for components to defer the execution of operations that are not time-critical [Levis et al. 2003]. Although the designers of TinyOS and nesC have taken care to provide a relatively easy-to-understand concurrency model, the possibility of unanticipated interleavings of different activities may still cause undesirable application behavior. For example, consider the following simplified extract from the `AlarmToTimerC` component [Sharp 2008] distributed as part of TinyOS 2.x, which creates a periodic timer from a hardware counter by using the task that signals a timer firing to also restart the hardware counter for the next timer interval:

```
1:   command void Timer.stop() { call Alarm.stop(); }
2:
3:   task void fired() {
4:       call Alarm.startAt(call Alarm.getAlarm(), m_period);
5:       signal Timer.fired();
6:   }
7:
8:   async event void Alarm.fired() { post fired(); }
```

The cycle of repeated timer firings will *usually* stop when `Timer.stop()` is called, because stopping the `Alarm` (line 1) disables the interrupt-driven `Alarm.fired()` event that triggers the next timer interval. However, should `Timer.stop()` be called after the `fired()` task has been posted to the task scheduler (line 8), but before that task has been executed the timer cycle will not actually stop. Later execution of the `fired()` task (lines 3–6), which does not check whether or not the timer is actually supposed to be active, will restart the timer. This erroneous behavior occurs because of an apparently unanticipated interleaving of the concurrent stop and restart activities that makes use of the alarm state to track the timer state fail in some instances. In practice, application developers are usually shielded from unexpected `AlarmToTimerC` behavior by the higher-level timer virtualization framework, which does correctly handle the interleaving of stop and restart activities. However, any developer that makes direct use of the `AlarmToTimerC` component may find their application susceptible to sporadic errors caused by periodic timers that fail to stop.

One method of helping developers understand the possible behaviors of concurrent programs is the application of model-checking to systematically explore the state-space of an abstract model of the program [Clarke et al. 1996]. TinyOS applications possess a number of features that make them well-suited to the application of model-checking. Unlike general C programs, TinyOS applications have a component-based structure that already provides a set of well-defined abstractions. In addition, the individual application components encapsulate state, and provide clearly-defined module interfaces that are easily translated into a modeling formalism. We have opted to use the process algebra CSP (Communicating Sequential Processes) [Roscoe 1998] as the formalism for building models to be checked, since its characteristics are well-matched to those of nesC and TinyOS:

— TinyOS applications are concurrent; CSP is designed to model concurrency.
— TinyOS applications are event-driven; CSP is an event-based formalism.

— TinyOS applications are hierarchical compositions of components; CSP process models are hierarchical compositions of processes.
— TinyOS applications are composed by wiring together interfaces; CSP processes are composed by wiring together channels.

In this paper, we build on the intuitive correspondence between nesC programs and CSP processes to develop a systematic mapping from nesC to CSP, and add models of the TinyOS task-scheduler and preemption rules to construct CSP processes that model TinyOS applications:

— We define a mapping from nesC programs to CSP process models that formalizes the component interactions within a nesC program (section 3.1). This mapping is more accurate and comprehensive than previous efforts at process algebraic modeling of TinyOS applications, providing a model that correctly represents the sequential nature of most nesC function execution, and that includes representations of mutable variables, argument passing, and multiple levels of interrupt priority. As part of the mapping, we define several processes that ease the translation from nesC to CSP, by providing a nesC-like embedded domain-specific language over the underlying process model (section 3.2).
— We describe a new CSP model of TinyOS task-scheduling and preemption that includes modeling of prioritized interrupts and nested atomic blocks (section 3.3). This model clarifies the execution semantics of TinyOS, and allows interactions between TinyOS applications and the TinyOS concurrency mechanisms to be analyzed.
— As part of our TinyOS preemption model, we introduce a novel technique for modeling the relationship between multiple prioritized interrupts, based on tracking the execution context and corresponding priority level of each event in the interrupt handling function (section 3.3.2). In addition to being useful for modeling TinyOS applications this technique is applicable to the more general problem of modeling firmware interrupt-handling, and is also potentially useful for modeling priority-based preemptive multi-threading systems,
— We present an example that demonstrates how a CSP model of a TinyOS component can be used to expose and correct subtle concurrency-related errors in sensor-node firmware (section 4).

We discuss other approaches to analyzing TinyOS applications in section 5.

2. BACKGROUND

In this section we briefly review the basic features of TinyOS and CSP.

2.1. TinyOS and nesC

TinyOS is an operating system designed to meet the needs of programming resource-constrained network embedded systems [Levis et al. 2005]. TinyOS is not a hard real-time operating system, but provides an event-driven reactive programming model within a very small footprint. The TinyOS distribution provides a lightweight task-scheduling system, and a library of components that implement services useful in networked embedded systems, such as communications, timing, and sensing.

The nesC language was created to support TinyOS [Gay et al. 2003]. Reflecting TinyOS' component-based philosophy, nesC programs are compositions of interacting components. There are two kinds of components: *modules*, which implement new functionality, and *configurations*, which define connections between components. Each nesC component has one or more *interfaces*, made up of *commands* and *events* (e.g. Fig. 1). Commands request that a component perform some service, while events signal to a component that an activity it requested has been completed, or that an ex-

```
interface Timer<precision_tag> {
    command void startPeriodic(uint32_t period);
    command void stop();
    event void fired();
    ...
}
```

Fig. 1. `Timer` interface

```
module BlinkC {
    uses interface Timer<TMilli> as Timer0;
    uses interface Leds;
    uses interface Boot;
} implementation {
    event void Boot.booted() {
        call Timer0.startPeriodic( 250 );
        ...
    }
    event void Timer0.fired() {
        call Leds.led0Toggle();
    }
}
```

Fig. 2. `BlinkC` module

```
configuration BlinkAppC { } implementation {
    components MainC, BlinkC, LedsC;
    components new TimerMilliC() as Timer0;

    BlinkC -> MainC.Boot;
    BlinkC.Timer0 -> Timer0;
    BlinkC.Leds -> LedsC;
}
```

Fig. 3. `BlinkAppC` configuration

ternal phenomenon such as an interrupt has occurred. A component that *provides* an interface must implement each command, and may signal any of the events. A component that *uses* an interface must implement handlers for each event, and may call any of the commands (e.g. Fig. 2). Configuration components connect ("wire") interface providers to interface users (e.g. Fig. 3).

The behavior of nesC programs is largely event-driven. Hardware interrupts trigger events, the effects of which propagate up the component graph as a cascade of further events. Where the response to an event is not time-critical the overall responsiveness of the application may be improved by deferring the response until the processor is idle, allowing other events to be processed in the interim. Deferred execution is accomplished by posting a *task* that executes the deferred activities.

TinyOS provides a non-preemptive task scheduler that queues pending tasks in first-in/first-out order, and executes those tasks whenever the processor is not occu-

pied responding to an interrupt. Each task is executed in a run-to-completion style, which makes tasks atomic with respect to each other and thereby prevents data races between tasks. Execution in a task context is thus referred to as *synchronous* execution. Tasks can be preempted by interrupt-triggered *asynchronous* events. Commands or events able to be executed in an interrupt context are required to be labeled as asynchronous using the `async` keyword [Gay et al. 2005]. This allows the nesC compiler to check for possible race conditions.

The reliability of TinyOS applications can be improved prior to deployment by subjecting the application to simulation via TOSSIM [Levis et al. 2003], unit testing [Woehrle et al. 2007], and perhaps lab testing. But uncovering concurrency-related errors via testing and simulation is a difficult task, due to the timing sensitivity of many concurrency errors. Furthermore, even if an error is found during testing, pinning down the actual source of the problem can be a time-consuming process. It is therefore worthwhile to consider other options for analyzing the concurrent behavior of TinyOS applications.

2.2. Communicating Sequential Processes

CSP [Roscoe 1998] is a mathematical theory of concurrency and interaction, in which concurrent systems are modeled as collections of event-transition systems (processes) that interact by synchronizing on shared events. CSP events are abstract symbols that represent interactions or communications. For example, a process model of an online purchasing system might include events that represent ordering an item, confirming the order, providing payment, and shipping the ordered item. Interfaces between processes are defined using channels that carry values of some specified type; each occurrence of a value being passed through a channel corresponds to a single event.

Sequential processes are defined by using the prefix operator, $\rightarrow$, to specify sequences of events. For example, an online purchase can be described by the process

$$\begin{aligned} &\mathit{OnlinePurchase} = \\ &\quad \mathit{order}?\mathit{item} \rightarrow \mathit{confirm}!\mathit{item} \rightarrow \mathit{request\_payment}!\mathit{cost}(\mathit{item}) \rightarrow \\ &\qquad \mathit{receive\_payment}?p \rightarrow \mathit{ship}!\mathit{item} \rightarrow \mathit{SKIP} \end{aligned}$$

where $?$ and $!$ indicate channel input and output respectively, and *SKIP* is a primitive process representing successful termination. Unfortunately, the purely sequential online purchase process defined above will ship an item regardless of the amount of payment received. However, the CSP process notation also provides a variety of other operators for defining behaviors, such as:

— Conditional behavior (if $p = \mathit{cost}(\mathit{item})$ then ... else ...)
— Alternative courses of action ($\mathit{DisplayCart} \,\Box\, \mathit{DisplayCatalog}$)
— Nondeterministic behavior ($\mathit{Transaction} \sqcap \mathit{Error}$)
— Process sequencing ($\mathit{Server}(x) = \mathit{Login};\ \mathit{Hello};\ \ldots$)
— Parallel execution ($\mathit{Servers} = \mathit{Server}(1) \,|||\, \mathit{Server}(2)$)
— Interaction through an interface ($\mathit{Customer} \,[\![\, \mathit{OrderEvents} \,]\!]\, \mathit{Servers}$)

CSP includes a rich theory of process refinement and equivalence based on analyzing the sequences of events that processes can be observed to perform. The FDR2 model-checker [Gardiner et al. 2005] is a mature analysis tool that can be used to automatically check CSP models for properties such as deadlock or livelock, and to evaluate process models for conformance to specifications.

3. CSP MODELS OF TINYOS APPLICATIONS

Our CSP models of TinyOS applications are broken into two major parts. One part is the mapping from nesC to CSP that allows application code to be translated into

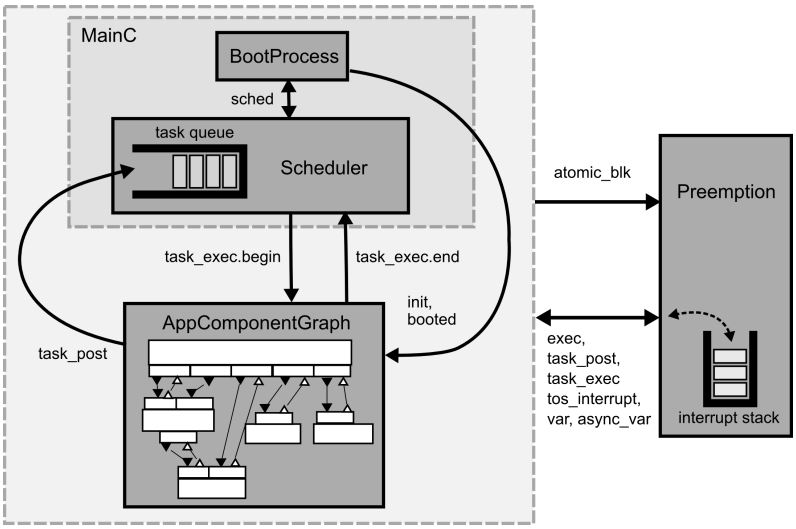

Fig. 4. Structure of the CSP TinyOS model

a model in which each component in the application is represented by a CSP process, and the overall application is defined by combining the component processes. The other part is a model of the TinyOS runtime environment that provides boot-up, scheduling, and preemption mechanisms, and acts as a context for the application model. Fig. 4 shows the structure of our CSP model. In the remainder of this section we explain the behavior of the processes that make up the model, and the significance of the channels through which the processes interact.

3.1. Mapping nesC to CSP

Since we are interested in examining the component interactions within a TinyOS application our mapping from nesC to CSP focuses on modeling the execution of nesC commands and events. We model nesC components as CSP processes, and model the nesC commands, events, interrupts, variable operations, and task control functions as CSP events.

3.1.1. CSP Event Structure. The CSP events that represent nesC commands and events are compound symbols that encode information about the command or event, the interface and component with which it is associated, and the execution context (synchronous or asynchronous). For example, initiation of the `read()` command from the `Read` interface by a component `SenseC` in a synchronous execution context is represented by the event *exec.begin.sync.SenseC.Read_read*.

Although it is tempting to map execution of a nesC command or event directly to a single CSP event, as previous work on process algebraic analysis of TinyOS applications has done [Rosa and Cunha 2007], such a mapping does not accurately represent command/event execution. Commands and events in nesC are compiled down to C functions [Levis 2006]. Execution of a command that calls another command will not complete until the called command has completed, and control has returned to the calling command. To capture the behavior of function calls in the CSP model we represent command/event execution as a pair of events that indicate the beginning and end of an execution.

```
interface Send {
  command error_t send(message_t* msg, uint8_t len);
  event void sendDone(...);
}
module SenderC {
  provides interface SplitControl;
  uses interface Send;
  uses interface Timer<TMilli> as MTimer;
} ...
```

datatype $Msg = hello \mid goodbye \mid value.\{0\,..\,10\}$
datatype $SendIF = Send\_send.Msg \mid Send\_sendDone$
datatype $SenderC\_IF = SenderC.\bigcup\{SendIF, SplitControlIF\}$
$\mid SenderC\_MTimer.TimerIF$

Fig. 5. Datatypes for interfaces

In general then, nesC commands and events are represented by CSP events associated with the *exec* channel:

channel $exec : Exec.SyncType.NesC\_Function$

where the types

datatype $Exec = begin \mid end$
datatype $SyncType = sync \mid async.IntPriority$
datatype $IntPriority = prio.\{0\,..\,MAX\_PRIORITY\}$

contain symbols that encode the execution step, the execution context, and, in asynchronous contexts, the interrupt preemption level. The use of *SyncType* will become clear when we introduce modeling of interrupt preemption in section 3.3.

The *NesC_Function* type that appears in the definition of *exec* is a set that is specific to the TinyOS application being modeled. It contains symbols that encode the component and interface of each command or event, and, for those functions that take arguments, the set of argument values the function can receive (see Fig. 5). We encode interfaces as types that combine the interface name with the name of each function in the interface, using the format <*interface-name*>_<*function-name*> (see Fig. 5). The interface of a component is the union of the individual interfaces the component provides and uses. We prepend the component interface set with the component name to model the component-local namespaces used in nesC [Levis 2006]. In the case of interfaces that are renamed within the component, we incorporate the new name into the prepended component name (see Fig. 5). Finally, the *NesC_Function* type is the union of all component interfaces. For example, in an application containing the components `MainC`, `SenderC`, and `TimerC`, the corresponding *NesC_Function* is:

nametype $NesC\_Function = \bigcup\{MainC\_IF, SenderC\_IF, TimerC\_IF\}$

Asynchronous execution is triggered by interrupts, which are processed by corresponding interrupt handlers. We model the interrupt handlers using events with a similar structure to those used to model regular nesC functions, although using separate channels to allow interrupt handling to be easily distinguished from other nesC operations. The interrupt channel is:

channel $tos\_interrupt : Exec.IntPriority.IntHandler$

where *Exec* and *IntPriority* are as defined above, and *IntHandler* is a type similar in concept to *NesC_Function* but specifically used to represent interrupt handlers.

Synchronous execution is managed through the TinyOS task scheduler. We model task execution much as commands, events, and interrupt handlers are modeled. Posting of tasks to the scheduler is modeled by the *task_post* channel, while initiation and completion of task execution are modeled as *task_exec* events:

channel *task_post* : *SyncType*.*Task*
channel *task_exec* : *Exec*.*Task*

Task is a datatype specific to the application being modeled. The *Task* datatype consists of symbols that represent each task in the application, using the form <*component-name*>_<*task-name*>. For example, the symbol for a task `sendDone()` in a module `SenderC` would be *SenderC_sendDone*.

Task execution events should only be generated by the TinyOS scheduler, and thus can only occur in synchronous execution contexts. In contrast, task post events may occur both in synchronous and asynchronous contexts. Indeed, posting a task is the standard way to transition from asynchronous processing to synchronous processing. Consequently, we include *SyncType* information in the *task_post* events, providing a way to identify the execution context of a given post operation.

Within the body of a task, command, event, or interrupt handler, a program may modify or read state variables that are shared with other functions. We model operations on variables in terms of "get" and "set" operations on those variables. Variables intended only for use in a synchronous context are accessed through the *var* channel. Variables able to be used in both synchronous and asynchronous contexts are accessed through the *async_var* channel, which includes information about the execution context of the operation.

datatype *VarOp* = *getv* | *setv*
channel *var* : *VarOp*.*NesC_Variable*
channel *async_var* : *SyncType*.*VarOp*.*NesC_Variable*

The type *NesC_Variable* that appears in both the *var* and *async_var* channel declarations is an application-specific type that defines both variable names and variable data types. For example, in an application with a single variable used to track the operational state of the program, we might define *NesC_Variable* as

datatype *State* = *STATE_IDLE* | *STATE_PROCESSING*
datatype *NesC_Variable* = *state*.*State*

In this example, the event *async_var*.*async*.*prio*.3.*setv*.*state*.*STATE_IDLE* represents setting the value of the *state* variable to an idle state, with the operation being executed from an interrupt of priority level 3.

3.1.2. Modules. We model modules as processes that provide a selection of behaviors corresponding to the functions implemented by the module. A module model may also contain processes representing the key state variables used by the module implementation. The function part is defined as a parallel composition of processes that represent individual command, event-handling, or task functions. The state-variables are similarly defined as a parallel composition of processes that represent each variable. The module model itself is then a parallel composition of the function and variable parts, interfacing with each other through a set of channels that represent the variable value assignment and retrieval. The general form of module process models is shown in Fig. 6. Several examples of this approach to modeling modules appear in section 4.

$$
\begin{array}{l}
Module\_\langle component\text{-}name\rangle = \\
\quad \textsf{let} \\
\quad\quad StateEvents = VarEvents(\{v_1, \cdots, v_N\}) \\
\quad\quad StateVars = (V_1 \;|||\; \cdots \;|||\; V_N) \\
\quad\quad Functions = \\
\quad\quad\quad \textsf{let} \\
\quad\quad\quad\quad C_N = \langle cmd\text{-}N\text{-}behavior\rangle;\ C_N \\
\quad\quad\quad\quad \cdots \\
\quad\quad\quad\quad E_N = \langle event\text{-}N\text{-}behavior\rangle;\ E_N \\
\quad\quad\quad\quad \cdots \\
\quad\quad\quad\quad T_N = \langle task\text{-}N\text{-}behavior\rangle;\ T_N \\
\quad\quad\quad \textsf{within} \\
\quad\quad\quad\quad (C_1 \;|||\; \cdots \;|||\; T_N) \\
\quad \textsf{within} \\
\quad\quad (Functions \;[\![\, StateEvents \,]\!]\; StateVars)
\end{array}
$$

Fig. 6. General structure of module process models

Although the module structure in Fig. 6 appears to place all functions in a parallel composition, the actual execution sequence of the synchronous functions is constrained to follow a path prescribed by the TinyOS scheduler model (section 3.3.1) and the call-graph of the application. In an earlier version of our nesC-to-CSP mapping [McInnes 2009] we suggested that the synchronous parts of modules should be modeled as a single process external choice ($\Box$) over the synchronous functions. This works well for simple modules and small state-spaces (as the examples in section 4 show), but our experiences with larger modules have shown that modeling the synchronous functions using parallel composition opens new interaction possibilities by allowing modules to safely make callbacks, and decomposes the model state-space in a way that permits faster compilation by FDR2.

3.1.3. Commands and Events. Within a module process, we model the behavior of an individual function f as the process

$$FnDef(f, Body) = exec.begin.sync.f \rightarrow Body\ ;\ exec.end.sync.f \rightarrow SKIP$$

where *Body* is a process defining the actions performed by the command or event. Note that *FnDef* provides a model for synchronous functions only, since its events include the symbol *sync*. Because asynchronous functions can be called from both synchronous and asynchronous contexts, we require a slightly more complex model to properly capture their behavior. The process

$$
\begin{array}{l}
AsyncFnDef(f, Body) = \\
\quad \Box\, s : SyncType \bullet exec.begin.s.f \rightarrow Body(s)\ ;\ exec.end.s.f \rightarrow SKIP
\end{array}
$$

is similar to *FnDef*, but keeps track of the *SyncType* execution context of the initiating event, and also propagates that execution context to the *Body* process.

The preceding definitions model simple functions that lack arguments. This is a reasonable approach for many nesC commands and events. Indeed, we prefer to use function definitions without arguments wherever possible, since doing so reduces the size of the model state-space. However, in some situations it is useful to be able to model argument passing. To achieve this, we incorporate the argument values as additional components of the *exec* events, and assume that the *Body* process is defined in a way that allows it to accept argument values.

$$FnDefArgs(f, Body) =$$
$$\quad exec.begin.sync.f?arg \rightarrow (Body(arg)\ ;\ exec.end.sync.f.arg \rightarrow SKIP)$$

$$AsyncFnDefArgs(f, Body) =$$
$$\quad \Box\, s : SyncType \bullet exec.begin.s.f?arg \rightarrow$$
$$\qquad (Body(s)(arg)\ ;\ exec.end.s.f.arg \rightarrow SKIP)$$

Thus, for example, a synchronous function *MessageP.Send_send* that takes an argument *msg* could be defined as

$$FnDefArgs(MessageP.Send\_send, \lambda\, msg \bullet <do\ something\ with\ msg>)$$

where the *Body* process is defined as a lambda function that accepts the argument *msg* and then performs various actions.

In general, the actions performed in the *Body* of a function will be calls to other functions, and variable reads or writes. We use CSP sequential composition (;), conditionals (if ... then), and recursion to capture control-flow, and model function calls with the process

$$FnExec(s)(f) = exec.begin.s.f \rightarrow exec.end.s.f \rightarrow SKIP$$

where s indicates the execution context of the call, and f may contain both a function name and any argument values passed to the function. Synchronizing *FnExec* with a corresponding *FnDef* process has the effect of triggering execution of the function f, and blocking the *FnExec* process until the body of the *FnDef* has completed. Blocking *FnExec* in this way provides the desired behavior for nested command and event executions. For example, in the composite process

$$FnDef(App.Say\_hello, FnExec(sync)(MessageP.Send\_send.hello))$$
$$[\![\{\!|exec.begin.sync.MessageP, exec.end.sync.MessageP|\!\}]\!]$$
$$FnDefArgs(MessageP.Send\_send, \lambda\, msg \bullet <do\ something\ with\ msg>)$$

the function *App.Say_hello* cannot complete until execution of *MessageP.Send_send* has completed. Similary, the synchronization on *MessageP* execution events ensures that *MessageP.Send_send* cannot begin execution until *App.Say_hello* permits it. The sequence of events produced by the process will therefore be:

$$exec.begin.sync.App.Say\_hello$$
$$\quad exec.begin.sync.MessageP.Send\_send.hello$$
$$\quad\quad \cdots <do\ something\ with\ 'hello'> \cdots$$
$$\quad exec.end.sync.MessageP.Send\_send.hello$$
$$exec.end.sync.App.Say\_hello$$

3.1.4. Tasks. The behavior of tasks is modeled similarly to the behavior of commands and events, as a task body surrounded by beginning and end events:

$$task(t, Body) = task\_exec.begin.t \rightarrow Body\ ;\ task\_exec.end.t \rightarrow SKIP$$

Unlike commands or events, tasks are triggered by the scheduler (section 3.3). Tasks are added to the scheduler by posting, an action modeled by the processes

$$post(t) = task\_post.sync.t \rightarrow SKIP$$
$$async\_post(s)(t) = task\_post.s.t \rightarrow SKIP$$

The *post* process can only be used in synchronous contexts, while *async_post* tracks the execution context and can thus be used in asynchronous contexts.

3.1.5. Variables. We model each state variable as a process that can report its current value (*getv*) or accept a new value (*setv*). However, we make a distinction between those variables intended for use only in a synchronous context, and those that can be used in asynchronous contexts. This allows us both to track the execution context of variable accesses, and to flag potential race conditions when variables are accessed from the wrong execution context. The process definition for synchronous-only variables is

$$
\begin{aligned}
&Var(name, val) = \\
&\quad (var.getv.name!val \rightarrow Var(name, val)) \\
&\quad \Box\ (var.setv.name?val' \rightarrow Var(name, val')) \\
&\quad \Box\ (async\_var?\_ : SyncType?\_ : VarOp!name?\_ \rightarrow tos\_datarace \rightarrow STOP)
\end{aligned}
$$

while that for asynchronous variables is

$$
\begin{aligned}
&AsyncVar(name, val) = \\
&\quad (async\_var?s : SyncType!getv!name!val \rightarrow AsyncVar(name, val)) \\
&\quad \Box\ (async\_var?s : SyncType!setv!name?val' \rightarrow AsyncVar(name, val')) \\
&\quad \Box\ (var?\_ : VarOp!name?\_ \rightarrow tos\_datarace \rightarrow STOP)
\end{aligned}
$$

3.1.6. Configurations. We model the inter-component connections defined by a configuration as a parallel composition of component processes synchronizing on events in their connected interfaces. However, because *exec* events are structured to give each module its own namespace (section 3.1.1), modules do not by themselves have any events in common. We therefore use the CSP renaming operator to map events associated with one module to corresponding events in the other module.

The wiring of components A and B to each other is modeled by the process

$$
\begin{aligned}
&wiring(A, B, Connections) = \\
&\quad \text{let} \\
&\qquad SharedIF = \{exec.e.s.IFa.f \mid (IFa, \_, Fns) \in Connections, \\
&\qquad\qquad\qquad\qquad e \in Exec, f \in Fns, s \in SyncType\} \\
&\quad \text{within} \\
&\qquad (A \ [\![SharedIF]\!] \\
&\qquad\quad (B [\![exec.e.s.IFb.f \leftarrow exec.e.s.IFa.f \mid (IFa, IFb, Fns) \in Connections, \\
&\qquad\qquad\qquad\qquad e \in Exec, f \in Fns, s \in SyncType]\!]))
\end{aligned}
$$

where *Connections* is a set of 3-tuples specifying the interfaces to be connected. The *wiring* process renames *exec* events for *IFb.f* to appear as the corresponding *IFa.f* events, and synchronizes A and B on the shared *IFa.f* events.

Multiple-wiring of interfaces results in fan-in and fan-out of calls to and from the multiply-connected component. Fan-in is well-modeled by using a multiple-renaming scheme. Modeling fan-out is slightly more complex, since it is necessary to enforce serial execution of the fan-out functions. Our approach to modeling fan-out is to insert an intermediate process between the caller and the functions called in the fan-out. This approach resembles the nesC compiler's use of intermediate functions to implement fan-out [Gay et al. 2005].

$$
\begin{aligned}
&FanOut(comp, Fns, OutComps) = \\
&\quad \text{let} \\
&\qquad AwaitOut = exec.begin?s : SyncType!comp?f : Fns \rightarrow ExecOut(s, f, OutComps) \\
&\qquad ExecOut(s, f, \{\}) = exec.end.s.comp.f \rightarrow AwaitOut \\
&\qquad ExecOut(s, f, Comps) = \\
&\qquad\quad \sqcap c : Comps \bullet exec.begin.s.c.f \rightarrow exec.end.s.c.f \rightarrow ExecOut(s, f, Comps \setminus \{c\}) \\
&\quad \text{within} \\
&\qquad AwaitOut
\end{aligned}
$$

```
module SenderC {
  ...
  event void MTimer.fired() { call Send.send(msg, ...); }
  async event void Radio.txDone() { signal Message.sendDone();}
```

$$event(SenderC\_MTimer.Timer\_fired, call(SenderC.Send\_send.msg))$$
$$async\_event(SenderC.Radio\_txDone, \lambda\, context \bullet async\_signal(context)(SenderC.Message\_sendDone))$$

Fig. 7. NesC statements and corresponding CSP expressions

3.2. Easing the Translation

Although the mapping described above allows nesC programs to be in modeled in CSP, the resulting model can be complex to construct, and may bear little resemblance to the original nesC program. However, we can overlay the basic process model with what amounts to an embedded domain-specific language that more closely matches nesC syntax, simplifying the translation to CSP.

For synchronous functions, we use the processes

$$command(f, Body) = FnDef(f, Body)$$
$$command\_args(f, Body) = FnDefArgs(f, Body)$$
$$event(f, Body) = FnDef(f, Body)$$
$$event\_args(f, Body) = FnDefArgs(f, Body)$$
$$call(f) = FnExec(sync)(f)$$
$$signal(f) = FnExec(sync)(f)$$

These allow us to translate nesC statements into similar looking CSP expressions, as the `MTimer.fired()` example in Fig. 7 illustrates.

Similarly, for asynchronous functions, and calls from asynchronous contexts, we define the processes

$$async\_command(f, Body) = AsyncFnDef(f, Body)$$
$$async\_command\_args(f, Body) = AsyncFnDefArgs(f, Body)$$
$$async\_event(f, Body) = AsyncFnDef(f, Body)$$
$$async\_event\_args(f, Body) = AsyncFnDefArgs(f, Body)$$
$$async\_call(s)(f) = FnExec(s)(f)$$
$$async\_signal(s)(f) = FnExec(s)(f)$$

The asynchronous *call* and *signal* must be invoked with an argument specifying the caller's execution context. The initial execution context is typically set in the interrupt handler that triggers a particular asynchronous activity, and then propagated through *FnExec* and *AsyncFnDef* calls. The propagation through *AsyncFnDef*s can be handled by defining the *Body* of the *AsyncFnDef* as an anonymous function that takes an execution context as an argument (see Fig. 7).

For variable access, we define the processes

$$assign(name, val) = var.setv.name!val \rightarrow SKIP$$
$$read(name, P) = var.getv.name?v \rightarrow P(v)$$
$$async\_assign(s)(name, val) = async\_var.s.setv.name!val \rightarrow SKIP$$
$$async\_read(s)(name, P) = async\_var.s.getv.name?v \rightarrow P(v)$$

As with the asynchronous *call* and *signal*, asynchronous operations on variables require the specification of an execution context. Furthermore, the *read* processes use a continuation-passing style in which the *read* is parameterized by a process that represents the continuation of the program behavior, and to which is passed the retrieved variable value. By using anonymous functions to handle this argument passing we can achieve a compact yet readable notation for expressing operations on variables. For example, without the *read* and *assign* processes the nesC

```
y = x + 1;
```

would be written as

$$var.getv.xvar?x \rightarrow var.setv.yvar!(x+1) \rightarrow SKIP,$$

while using *read* and *assign* allows us to write

$$read(xvar, \lambda x \bullet assign(yvar, x+1))$$

Finally, we define some auxiliary processes for creating interrupt handlers. The basic interrupt handler is parameterized by a function name $f \in IntHandler$, a priority level $priority \in IntPriority$, and a *Body* that specifies the behavior of the interrupt handler.

$$TOSH\_INTERRUPT(f, priority, Body) =$$
$$\quad tos\_interrupt.begin.priority.f \rightarrow$$
$$\quad\quad Body(async.priority)\ ;\ tos\_interrupt.end.priority.f \rightarrow SKIP$$

A variation on the basic interrupt handler permits the creation of handlers that accept a value of some kind, allowing compact expression of interrupt handlers that read a value from a hardware device at the start of their execution.

$$TOSH\_INTERRUPT\_input(f, priority, Body) =$$
$$\quad tos\_interrupt.begin.priority.f?x \rightarrow$$
$$\quad\quad (Body(async.priority)(x)\ ;\ tos\_interrupt.end.priority.f.x \rightarrow SKIP)$$

TinyOS also allows the definition of interrupts handlers that are not preemptible. These are referred to as "signals". We model a signal as an interrupt with a priority level higher than any other interrupt.

$$TOSH\_SIGNAL(f, Body) =$$
$$\quad TOSH\_INTERRUPT(f, prio.SIGNAL\_PRIORITY, Body)$$

$$TOSH\_SIGNAL\_input(f, Body) =$$
$$\quad TOSH\_INTERRUPT\_input(f, prio.SIGNAL\_PRIORITY, Body)$$

3.3. Scheduling and Preemption Models

Mapping from nesC to CSP is only one part of modeling a TinyOS application. TinyOS applications are composed of both the nesC program, and the infrastructure of TinyOS (see Fig. 4). Given a model of a nesC program, *AppComponentGraph*, we model the complete TinyOS application as a composite process that combines the *AppComponentGraph* with models of the TinyOS boot-up and scheduling systems, and of interrupt preemption.

The process *Configuration_MainC* (section 3.3.1) is our abstract model of the TinyOS `MainC` component. It encapsulates the boot process and the task scheduler. The *Preemption* process (section 3.3.2) models the relationship between synchronous and asynchronous execution in TinyOS. We developed these models based on existing descriptions of TinyOS execution semantics [Levis 2006], and examination of the

TinyOS 2.x source code. Using these processes, the complete application model is defined as:

$$
\begin{array}{l}
Application(AppComponentGraph, BootWiring, InitWiring) = \\
\quad \textsf{let} \\
\quad\quad App = \\
\quad\quad\quad ((AppComponentGraph \\
\quad\quad\quad\quad [\![\{exec.e.s.b.f \mid e \in Exec, s \in SyncType, b \in BootWiring, f \in BootIF\}]\!] \\
\quad\quad\quad\quad\quad FanOut(MainC, BootIF, BootWiring)) \\
\quad\quad\quad\quad [\![\{exec.e.s.i.f \mid e \in Exec, s \in SyncType, i \in InitWiring, f \in InitIF\}]\!] \\
\quad\quad\quad\quad\quad FanOut(MainC\_SoftwareInit, InitIF, InitWiring)) \\
\quad\quad AppMainIF = \\
\quad\quad\quad \{exec.e.s.f \mid e \in Exec, s \in SyncType, f \in MainC\_IF\} \\
\quad\quad\quad\quad \cup \{\!| task\_post, task\_exec |\!\} \\
\quad\quad PreemptIF = \\
\quad\quad\quad \{\!| atomic\_blk, task\_exec, task\_post, exec, var, async\_var, tos\_interrupt |\!\} \\
\quad \textsf{within} \\
\quad\quad (App \; [\![AppMainIF]\!] \; Configuration\_MainC) \; [\![PreemptIF]\!] \; Preemption
\end{array}
$$

where *Configuration_MainC* and *AppComponentGraph* synchronize on all events prefixed by *exec.begin.s.MainC* or *exec.end.s.MainC* for $s \in SyncType$, and all events associated with the *task_post* and *task_exec* channels. Both processes synchronize with *Preemption* on all events associated with the channels *atomic_blk*, *task_exec*, *task_post*, *exec*, *var*, *async_var*, *tos_interrupt*. The *FanOut* processes ensure that all components with *ApplicationComponentGraph* that are associated with the *Boot* or *Init* interfaces are connected to *Configuration_MainC*.

3.3.1. MainC and Scheduler. The TinyOS `MainC` component provides platform and software initialization, and is also the component in which the TinyOS scheduler resides [Levis 2006]. The `MainC` component has two interfaces, modeled in CSP as

$$
\begin{array}{l}
\textsf{datatype } BootIF = Boot\_booted \\
\textsf{datatype } InitIF = Init\_init \\
\textsf{datatype } MainC\_IF = MainC.BootIF \mid MainC\_SoftwareInit.InitIF
\end{array}
$$

Within `MainC`, system startup activities are handled by `RealMainP`. Following the standard `RealMainP` module [Levis 2007], we define *BootProcess* as

$$
\begin{array}{rl}
BootProcess = atomic(call(MainC\_SoftwareInit.Init\_init) \; ; & \\
\qquad\qquad sched.runTasks \rightarrow sched.doneTasks \rightarrow SKIP) \; ; & \\
\qquad signal(MainC.Boot\_booted) \; ; \; sched.runTasks \rightarrow SKIP &
\end{array}
$$

Note that *BootProcess* does not model platform initialization, but does include software initialization. The meaning of the $atomic()$ wrapper around software initialization will be clarified in the discussion of *Preemption*.

The *MainC* process combines *BootProcess* with a model of the scheduler:

$$Configuration\_MainC = BootProcess \; [\![\; \{\!| sched |\!\} \;]\!] \; Scheduler$$

Synchronization between *BootProcess* and *Scheduler* prevents tasks from being executed during initialization, and also prevents the boot process from signaling *MainC.Boot_booted* until any tasks posted during initialization have been executed.

The *Scheduler* process models the standard TinyOS FIFO scheduler. In TinyOS 2.x, posting a task to the scheduler when it is already enqueued has no effect [Levis 2006]. Our model reflects this behavior.

Internally, the *Scheduler* model is split into several processes, each representing a different scheduler state:

$$
\begin{array}{l}
Scheduler = \\
\quad \textsf{let} \\
\quad\quad SchInit(Q, P) = \cdots \\
\quad\quad SchNext(\langle\rangle, \_) = \cdots \\
\quad\quad SchNext(\langle t\rangle \frown Q, P) = \ldots \\
\quad\quad SchExec(Q, P, t) = \cdots \\
\quad \textsf{within} \\
\quad\quad SchInit(\langle\rangle, Task)
\end{array}
$$

In the initial state, tasks may be posted to the scheduler, but are not executed. The scheduler transitions to a state in which tasks can be executed when the *sched.runTasks* event occurs. Tasks are only added to the task queue if they are in the set, P, of "postable" tasks not yet enqueued. The queue is modeled as a sequence. Notationally, $\langle a, b\rangle$ is a sequence containing values a and b, and $s \frown t$ is the concatenation of the sequences s and t. The process modeling the initial scheduler state is:

$$
\begin{array}{l}
SchInit(Q, P) = \\
\quad (sched.runTasks \rightarrow SchNext(Q, P)) \\
\quad \Box\ (task\_post?s : SyncType?t : P \rightarrow SchInit(Q \frown \langle t\rangle, P \setminus \{t\})) \\
\quad \Box\ (task\_post?s : SyncType?t' : (Task \setminus P) \rightarrow SchInit(Q, P))
\end{array}
$$

If the task queue is empty, then any task can be posted. In addition, the empty-queue state is the only one in which the *sched.doneTasks* event, which triggers the *MainC.Boot_booted* event, can occur. This ensures that tasks posted during software initialization are cleared before the boot signal is sent.

$$
\begin{array}{l}
SchNext(\langle\rangle, \_) = \\
\quad (task\_post?s : SyncType?t \rightarrow SchNext(\langle t\rangle, Task \setminus \{t\})) \\
\quad \Box\ (sched.doneTasks \rightarrow SchInit(\langle\rangle, Task))
\end{array}
$$

If the task queue is not empty, the task at the head of the queue is executed. If a task is in the postable set, posting that task causes it to be added to the end of the task queue and removed from the postable set. Otherwise, the task posting is ignored. Note that a task is added to the postable set as soon as it has started executing. This allows tasks to post themselves.

$$
\begin{array}{l}
SchNext(\langle t\rangle \frown Q, P) = \\
\quad (task\_exec.begin!t \rightarrow SchExec(Q, P \cup \{t\}, t) \\
\quad \Box\ (task\_post?s : SyncType?t' : P \rightarrow SchNext(\langle t\rangle \frown Q \frown \langle t'\rangle, P \setminus \{t'\})) \\
\quad \Box\ (task\_post?s : SyncType?t'' : (Task \setminus P) \rightarrow SchNext(\langle t\rangle \frown Q, P))
\end{array}
$$

Finally, when the scheduler is executing a task, it waits for the task to complete before returning to a state in which it is ready to execute another task.

$$
\begin{array}{l}
SchExec(Q, P, t) = \\
\quad (task\_exec.end.t \rightarrow SchNext(Q, P)) \\
\quad \Box\ (task\_post?s : SyncType?t' : P \rightarrow SchExec(Q \frown \langle t'\rangle, P \setminus \{t'\}, t)) \\
\quad \Box\ (task\_post?s : SyncType?t'' : (Task \setminus P) \rightarrow SchExec(Q, P, t))
\end{array}
$$

Although the model just described uses an explicit sequence-based representation of the scheduler queue to allow a clearer presentation, better model-checking performance with FDR2 can be achieved by constructing a functionally identical task scheduler model that models the queue as a chain of smaller buffer processes.

3.3.2. Preemption. The *Preemption* process models preemption of synchronous execution by asynchronous events, and also preemption of low-priority interrupts by higher-priority interrupts. It does this by blocking all events associated with the preempted execution context whenever a preemption event occurs, and only allowing their execution to resume when the preempting activity has completed. We split *Preemption* into two major execution modes, which represent synchronous and asynchronous execution:

$$
\begin{array}{l}
Preemption = \\
\quad \textsf{let} \\
\quad\quad SyncExec(atomicDepth) = \cdots \\
\quad\quad AsyncExec(\langle\rangle, atomicDepth) = \cdots \\
\quad\quad AsyncExec(Stack, atomicDepth) = \cdots \\
\quad \textsf{within} \\
\quad\quad SyncExec(0)
\end{array}
$$

Each execution mode places constraints on the functions that an application can execute. To apply these constraints, the *Preemption* process synchronizes with the scheduler and application models on all events associated with the *task_exec* and *exec* channels, as well as *atomic_blk*, *task_post*, *var*, *async_var*, and *tos_interrupt*. Within both execution modes, the application can prevent preemption from occurring by declaring an *atomic* block. The transition into and out of an atomic block is communicated to the *Preemption* model via the events *atomic_blk.begin* and *atomic_blk.end*. These events are used in the application model to define an atomic block via the process

$$atomic(Block) = atomic\_blk.begin \rightarrow Block \ ; \ atomic\_blk.end \rightarrow SKIP$$

which wraps the execution of the process *Block* in the required *atomic_blk* events.

The synchronous execution mode permits task execution, function execution, and operations on both synchronous and asynchronous variables. If the application is not in an atomic block then interrupt events may occur. When an interrupt occurs, the interrupt and its priority level are pushed onto the stack of active interrupts, which is modeled as a sequence, and execution transitions to the asynchronous mode. The process model is

$$
\begin{array}{l}
SyncExec(atomicDepth) = \\
\quad (\Box\, s : SyncActions \bullet s \rightarrow SyncExec(atomicDepth)) \\
\quad \Box\ (atomicDepth = 0\ \&\ tos\_interrupt.begin?p : IntPriority?i \rightarrow \\
\qquad\qquad\qquad AsyncExec(\langle (p,i) \rangle, 0)) \\
\quad \Box\ (atomic\_blk.begin \rightarrow \textsf{let}\ depth = atomicDepth + 1\ \textsf{within} \\
\qquad\qquad\qquad \textsf{if}\ depth > MAX\_ATOMIC\_DEPTH \\
\qquad\qquad\qquad \textsf{then}\ tos\_atomic\_err \rightarrow STOP \\
\qquad\qquad\qquad \textsf{else}\ SyncExec(depth)) \\
\quad \Box\ (atomicDepth > 0\ \&\ atomic\_blk.end \rightarrow SyncExec(atomicDepth - 1))
\end{array}
$$

where

$$
\begin{array}{l}
SyncActions = \{\!| task\_exec, task\_post.sync, exec.begin.sync, exec.end.sync, \\
\qquad\qquad var, async\_var.sync |\!\}
\end{array}
$$

Although nested atomic blocks are optimized away by the nesC compiler [Levis 2006], the preemption model is constructed on the assumption that program translation will be from source code. Therefore, the *SyncExec* process permits nesting of atomic blocks, although the nesting is necessarily restricted to a finite depth to allow model-checking. Following the approach used by Kleine [Kleine and Helke 2009], we

ensure the soundness of this abstraction by by emitting the *tos_atomic_err* event to flag situations where the maximum depth is exceeded.

In the asynchronous mode, the *Preemption* process will only perform events allowed in the current priority level. By synchronizing with the application model on the *exec*, *task_post*, *var*, and *async_var* channels, the *Preemption* process thereby causes all events associated with lower priority execution contexts to be blocked. Interrupts with a higher priority than the current context can occur, in which case the interrupt is pushed onto the stack of active interrupts, and the execution context shifts to the priority level of the new interrupt. This causes the events associated with the previous context to become blocked, while allowing events at the new priority level to proceed. Thus execution of the new interrupt preempts execution of the lower-priority interrupt. When the active interrupt completes, it is removed from the interrupt stack, and execution of the next interrupt on the stack resumes. Once the interrupt stack is empty execution returns to the synchronous mode.

$$
\begin{aligned}
&AsyncExec(\langle\rangle, atomicDepth) = SyncExec(atomicDepth)\\
&AsyncExec(Stack, atomicDepth) =\\
&\quad \mathsf{let}\\
&\qquad (prio.priority, intr) = head(Stack)\\
&\qquad Ps = \{prio.p \mid prio.p \in IntPriority, p > priority\}\\
&\quad \mathsf{within}\\
&\qquad (\Box\, a : AsyncActions(prio.priority) \bullet a \rightarrow AsyncExec(Stack, atomicDepth))\\
&\qquad \Box\ (tos\_interrupt.end.prio.priority.intr \rightarrow AsyncExec(tail(Stack), atomicDepth))\\
&\qquad \Box\ (atomicDepth = 0\ \&\ tos\_interrupt.begin?p : Ps?i \rightarrow\\
&\qquad\quad AsyncExec(\langle(p, i)\rangle \frown Stack, 0))\\
&\qquad \Box\ (atomic\_blk.begin \rightarrow \mathsf{let}\ depth = atomicDepth + 1\ \mathsf{within}\\
&\qquad\qquad\qquad\qquad \mathsf{if}\ depth > MAX\_ATOMIC\_DEPTH\\
&\qquad\qquad\qquad\qquad \mathsf{then}\ tos\_atomic\_err \rightarrow STOP\\
&\qquad\qquad\qquad\qquad \mathsf{else}\ AsyncExec(Stack, depth))\\
&\qquad \Box\ (atomicDepth > 0\ \&\ atomic\_blk.end \rightarrow AsyncExec(Stack, atomicDepth - 1))
\end{aligned}
$$

where

$$
\begin{aligned}
AsyncActions(p) = \{&exec.e.async.p.f \mid e \in Exec, f \in NesC\_Function\}\\
&\cup \{|task\_post.async.p, async\_var.async.p|\}
\end{aligned}
$$

4. EXAMPLE

Given both a mapping from nesC programs to CSP processes, and a model of TinyOS scheduling and preemption, we can now look at an example of using CSP models of TinyOS applications to analyze the component interactions within an application. We return to the `AlarmToTimerC` component discussed at the beginning of the paper, and show how a CSP model can be used to diagnose the unstoppable timer problem and to verify a solution.

4.1. Modeling

A simplified version of the `AlarmToTimerC` module appears in Fig. 8. To conserve space and focus on the functionality relevant to this example we omit functions used for running the timer in a one-shot mode, and functions associated with the extended interface for querying the timer state. The corresponding CSP model appears in Fig. 9. Note that the model completely omits the `m_period` variable, and abstracts from the actual alarm timing. However, the basic structure and behavior of the nesC code is clearly reflected in the model.

```
generic module AlarmToTimerC(typedef precision_tag) {
    provides interface Timer<precision_tag>;
    uses interface Alarm<precision_tag,uint32_t>;
} implementation {
  uint32_t m_period;

  command void Timer.startPeriodic(uint32_t period) {
      call Alarm.startAt(call Alarm.getNow(), period);
      m_period = period;
  }

  command void Timer.stop() { call Alarm.stop(); }

  task void fired() {
      call Alarm.startAt(call Alarm.getAlarm(), m_period);
      signal Timer.fired();
  }

  async event void Alarm.fired() { post fired(); }
}
```

Fig. 8. `AlarmToTimerC` module

$$
\begin{array}{l}
\textsf{datatype } AlarmToTimerC\_IF = AlarmToTimerC.\bigcup\{AlarmIF, TimerIF\} \\
Module\_AlarmToTimerC = \\
\quad \textsf{let} \\
\quad\quad StartStop = (command(AlarmToTimerC.Timer\_startPeriodic, \\
\quad\quad\quad\quad call(AlarmToTimerC.Alarm\_start)) \; ; \; StartStop) \\
\quad\quad\quad\quad \Box \; (command(AlarmToTimerC.Timer\_stop, \\
\quad\quad\quad\quad call(AlarmToTimerC.Alarm\_stop)) \; ; \; StartStop) \\
\quad\quad FiredTask = (task(AlarmToTimerC\_fired, \\
\quad\quad\quad\quad call(AlarmToTimerC.Alarm\_start) \; ; \\
\quad\quad\quad\quad signal(AlarmToTimerC.Timer\_fired)) \; ; \; FiredTask) \\
\quad\quad AlarmFired = async\_event(AlarmToTimerC.Alarm\_fired, \\
\quad\quad\quad\quad \lambda context \bullet async\_post(context)(AlarmToTimerC\_fired)); \\
\quad\quad\quad\quad AlarmFired \\
\quad \textsf{within} \\
\quad\quad (StartStop \;|||\; FiredTask \;|||\; AlarmFired)
\end{array}
$$

Fig. 9. *AlarmToTimerC* CSP model

In addition to the model of `AlarmToTimerC` it is useful to have a model of the `AlarmC` component to which it is connected. For this we use a simplified state-machine alarm component model (Fig. 10) that approximates the behavior of the platform-specific alarms provided with TinyOS. When the alarm is commanded to start it moves to the *Running* state, and can then signal an *AlarmHW_fire* interrupt. When the alarm is commanded to stop it moves to the *Idle* state. The process $skip() = \lambda \_ \bullet SKIP$ accepts an argument – necessary in the body of an *async_command* – but ignores the argument and simply passes control to the next process.

$$
\begin{array}{l}
\mathit{Module\_AlarmC} = \\
\quad \textsf{let} \\
\quad\quad \mathit{Idle} = (\mathit{async\_command}(\mathit{AlarmC.Alarm\_start}, \mathit{skip}());\ \mathit{Running}) \\
\quad\quad\quad\quad \Box\ (\mathit{async\_command}(\mathit{AlarmC.Alarm\_stop}, \mathit{skip}());\ \mathit{Idle}) \\
\quad\quad \mathit{Running} = (\mathit{TOSH\_SIGNAL}(\mathit{AlarmHW\_fire}, \\
\quad\quad\quad\quad\quad\quad \lambda\, \mathit{context} \bullet \mathit{async\_signal}(\mathit{context})(\mathit{AlarmC.Alarm\_fired}));\ \mathit{Idle}) \\
\quad\quad\quad\quad\quad\quad \Box\ (\mathit{async\_command}(\mathit{AlarmC.Alarm\_stop}, \mathit{skip}());\ \mathit{Idle}) \\
\quad \textsf{within} \\
\quad\quad \mathit{Idle}
\end{array}
$$

Fig. 10. Simple *AlarmC* CSP model

We combine the `AlarmToTimerC` and `AlarmC` components to form the `TimerC` configuration by wiring the two components together through the *Alarm* interface, and also causing the *Timer* interface of `AlarmToTimerC` to be accessible as a pass-through connection (a simple renaming of events).

$$
\begin{array}{l}
\mathit{Configuration\_TimerC} = \\
\quad \mathit{passthru}(\mathit{wiring}(\mathit{Module\_AlarmToTimerC}, \mathit{Module\_AlarmC}, \\
\quad\quad\quad\quad\quad\quad \{(\mathit{AlarmToTimerC}, \mathit{AlarmC}, \mathit{AlarmIF})\}), \\
\quad\quad\quad\quad \mathit{AlarmToTimerC}, \mathit{TimerC}, \{\mathit{TimerIF}\})
\end{array}
$$

4.2. Analysis

To analyze the `AlarmToTimerC` component, we embed it in a simple test application designed to run the timer, produce an observable *tick* event at each timer-firing, and command the timer to stop after 10 timer-firings have occurred (Fig. 11). We then use FDR2 to refinement-check this application model against a specification process that produces a finite number of ticks:

$$
\begin{array}{l}
\mathit{Ticks}(0) = \mathit{STOP} \\
\mathit{Ticks}(N) = \mathit{tick} \rightarrow \mathit{Ticks}(N-1)
\end{array}
$$

$$
\mathit{TimerTestApp} = \mathit{Application}(\mathit{Configuration\_TimerTestC}, \{\mathit{TimerTestC}\}, \{\})
$$

$$
\begin{array}{l}
\textsf{assert}\ \mathit{Ticks}(5) \sqsubseteq_T \mathit{TimerTestApp} \setminus (\Sigma \setminus \{\mathit{tick}\}) \\
\textsf{assert}\ \mathit{Ticks}(15) \sqsubseteq_T \mathit{TimerTestApp} \setminus (\Sigma \setminus \{\mathit{tick}\}) \\
\textsf{assert}\ \mathit{Ticks}(100) \sqsubseteq_T \mathit{TimerTestApp} \setminus (\Sigma \setminus \{\mathit{tick}\})
\end{array}
$$

The refinement assertions state that there should be no possible execution of *TimerTestApp* that produces an execution trace containing more than the specified number of *tick* events. As expected, a check of the first assertion fails because the test application is designed to run the timer for more than 5 ticks. However, checks of the second and third assertions also fail, indicating that the timer is not stopping after 10 timer-firings, or indeed after 100 timer-firings.

The following counterexample trace provided by FDR2 illustrates the problem:

$$
\begin{array}{l}
\ldots \\
\mathit{task\_exec.begin.TimerTestC\_stop} \\
\mathit{exec.begin.sync.TimerTestC.Timer\_stop} \\
\quad \mathit{tos\_interrupt.begin.prio}.255.\mathit{AlarmHW\_fire} \\
\quad \mathit{exec.begin.async.prio}.255.\mathit{AlarmToTimerC.Alarm\_fired} \\
\quad \mathbf{task\_post.async.prio.255.AlarmToTimerC\_fired} \\
\quad \mathit{exec.end.async.prio}.255.\mathit{AlarmToTimerC.Alarm\_fired}
\end{array}
$$

$$
\begin{array}{l}
Configuration\_TimerTestC = wiring(Module\_TimerTestC, Configuration\_TimerC, \\
\qquad\qquad\qquad\qquad\qquad\qquad \{(TimerTestC, TimerC, TimerIF)\}) \\
Module\_TimerTestC = \\
\quad \textsf{let} \\
\qquad StateEvents = VarEvents(\{tickCount\}) \\
\qquad StateVars = Var(tickCount, 0) \\
\qquad Boot = (event(TimerTestC.Boot\_booted, \\
\qquad\qquad call(TimerTestC.Timer\_startPeriodic)) \; ; \; Boot) \\
\qquad TimerFired = (event(TimerTestC.Timer\_fired, tick \rightarrow SKIP \; ; \\
\qquad\qquad\qquad read(tickCount, \lambda\, n \bullet \text{if } n < 10 \\
\qquad\qquad\qquad\qquad\qquad\qquad \text{then } assign(tickCount, n + 1) \\
\qquad\qquad\qquad\qquad\qquad\qquad \text{else } post(TimerTestC\_stop))) \; ; \\
\qquad\qquad\qquad\qquad\qquad\qquad TimerFired) \\
\qquad StopTask = (task(TimerTestC\_stop, call(TimerTestC.Timer\_stop)) \; ; \; StopTask) \\
\qquad Functions = (Boot \;|||\; TimerFired \;|||\; StopTask) \\
\quad \textsf{within} \\
\qquad (Functions \;[\![\, StateEvents \,]\!]\; StateVars)
\end{array}
$$

Fig. 11. TimerTestC CSP model

*tos_interrupt.end.prio.*255*.AlarmHW_fire*
exec.begin.sync.AlarmToTimerC.Alarm_stop
exec.end.sync.AlarmToTimerC.Alarm_stop
exec.end.sync.TimerTestC.Timer_stop
task_exec.end.TimerTestC_stop
...
task_exec.begin.AlarmToTimerC_fired
exec.begin.sync.AlarmToTimerC.Alarm_start
exec.end.sync.AlarmToTimerC.Alarm_start
...

As this trace shows, FDR2 is able to find a possible sequence of events in which the alarm interrupt preempts execution of the task that stops the timer, and posts the *AlarmToTimerC_fired* task. When that task is later executed it restarts the alarm, and the timer cycle begins anew despite the stop command. While this problematic sequence of events will not always occur, the fact that it *can* occur means that correct timer behavior depends on the unpredictable relative timing of interrupt and task executions. This leaves the application susceptible to seemingly random errors.

One way to remedy the problem is to add a state variable to the *AlarmToTimerC* component that is set to *true* when the timer is started, and to *false* when the timer is commanded to stop. The modified version of *AlarmToTimerC* should check the state variable before restarting the timer:

$$
\begin{array}{l}
StateVars = Var(running, false) \\
FiredTask = (task(AlarmToTimerC\_fired, \\
\qquad\qquad read(running, \lambda\, run \bullet \\
\qquad\qquad (\text{if } run \text{ then } call(AlarmToTimerC.Alarm\_start) \text{ else } SKIP) \; ; \\
\qquad\qquad\quad signal(AlarmToTimerC.Timer\_fired))) \; ; \; FiredTask)
\end{array}
$$

Rechecking the refinement assertions on the revised version of the application model indicates that the timer still produces more than 5 ticks, but there is no longer a possible sequence of events that can cause the timer to run indefinitely. The close resem-

blance between the model and the corresponding nesC code for `AlarmToTimerC` makes it easy to see how to translate the solution that was developed in the CSP model into application code:

```
bool running = FALSE;
task void fired() {
    if(running == TRUE)
        call Alarm.startAt(call Alarm.getAlarm(), m_period);
    signal Timer.fired();
}
```

5. RELATED WORK

The work most closely related to ours is probably Rosa and Cunha's [Rosa and Cunha 2007] effort to formalize nesC programs using LOTOS. Similar work by Völgyesi *et al.* [Völgyesi et al. 2005] proposed the use of hierarchical interface automata to analyze nesC component interactions. Our CSP modeling approach draws inspiration from both efforts. It extends that earlier work by using a dual-event model of function-calls that gives a more accurate representation of nesC execution, and by adding models of TinyOS scheduling and preemption to facilitate analysis of their impact on application behavior.

More recently, Bucur and Kwiatkowska [Bucur and Kwiatkowska 2009] built on the SATABS verification tool for ANSI-C, extending it to incorporate models of the TosThreads API for multithreading on TinyOS and the TinyOS kernel. Their work focuses on detecting violations of low-level safety-properties, such as misuse of pointers and out-of-bounds array access.

Xie *et al.* [Xie et al. 2006] proposed using the COSPAN model-checker for co-verification of TinyOS applications and hardware. Their approach to modeling TinyOS applications in the S/R language includes a model of scheduling and preemption. However, their scheduler model differs from ours in that it directly controls the execution of each individual function-call. Our model more closely resembles the TinyOS task scheduler, which is responsible for coordinating task execution but does not directly control individual function-calls. In addition, their scheme for modeling preemption is only able to represent a single priority level.

Kothari *et al.* [Kothari et al. 2008] developed a technique for extracting state-machines from TinyOS programs using symbolic execution. The resulting state-machines provide an abstracted view of the TinyOS program. In principle, symbolic execution could also be used to check the TinyOS program for desirable properties. However, doing so would require extending Kothari's method with some kind of specification language for defining the properties, and with a mechanism for checking those specifications during symbolic execution.

Kleine and Helke [Kleine and Helke 2009] proposed using CSP to verify implementations of real-time multithreaded applications. Their approach appears to be similar to ours, in that it splits the CSP model into separate parts representing the application, the operating system, and the runtime environment. However, their models are derived from a compiled intermediate representation of a program rather than from source code, and model a preemptive multithreading system rather than TinyOS-style mixed synchronous and interrupt-driven execution. Also, their preemption model relies on explicitly tracking the program counter state of each thread, whereas in our model that information is implicit in the execution state of each CSP process.

An alternative to static analysis of interaction models is runtime checking of the component interactions. To support such checks, Archer *et al.* [Archer et al. 2007] recently proposed the addition of interface contract specifications to TinyOS applications.

Runtime contract checking can provide useful diagnostic information when an error is encountered. But, as with other testing techniques, interface contracts cannot provide any information about bugs that are not triggered by the testing regime.

6. CONCLUSIONS

Although the TinyOS concurrency model is easier to reason about than shared-memory threads, unplanned interactions between background tasks and preemptive interrupts can lead to unexpected application behavior. To help TinyOS application developers gain a better understanding of the concurrent behavior of their applications we have developed an approach for modeling TinyOS applications using the process algebra CSP. Our approach provides a mapping from nesC programs to CSP processes, and a model of the TinyOS scheduling and preemption mechanisms.

Modeling and model-checking of TinyOS applications is complementary to verification methods such as testing and runtime contract checks. Model-checking permits analyses that can uncover low-probability bugs that might otherwise go undetected until the application has been widely deployed. Such analyses seem particularly important for TinyOS components that are intended to be building-blocks for creating large numbers of applications. In addition, model-checkers generate counterexample traces when an error is detected, providing a clear indication of the sequence of commands, events, tasks, and interrupts that led to the error. These counterexamples are useful for diagnosing and fixing errors in an application design, and are also potentially helpful for application developers seeking to gain insight into how the TinyOS scheduler, tasks, and preemption mechanisms interact with each other.

In addition to allowing application-level models to be analyzed, one advantage of using CSP as the modeling language and FDR2 as the model-checker is that refinement-checking can also be used to validate an abstract model of the sensor node behavior against the TinyOS application model. This is a capability not available with temporal-logic model checkers. The transitive and monotone nature of the refinement relation [Roscoe 1998] guarantees that properties checked on networks of valid abstract nodes also hold for identical networks of application-level models, without having to directly check the larger state-space of the application-level network model.

Our modeling approach does have some limitations. It does not presently support multiple simultaneous calls to the same function. Nor does it support return values from commands or events. Also, as with any application of model-checking, state-explosion is a concern. The example presented in this paper can be checked within a few seconds on a modern computer. The analysis of larger applications requires substantially more time and memory. In practice, careful use of abstraction is likely to be the key to successfully checking real applications.

REFERENCES

ARCHER, W., LEVIS, P., AND REGEHR, J. 2007. Interface contracts for TinyOS. In *Proc. of the 6th International Conference on Information Processing in Sensor Networks (IPSN '07)*. ACM, 158–165.

BUCUR, D. AND KWIATKOWSKA, M. 2009. Towards software verification for TinyOS applications. In *Proc. of the Workshop on Formal Approaches to Ubiquitous Systems (FAUSt'09)*.

CLARKE, E. M. ET AL. 1996. Formal methods: state of the art and future directions. *ACM Computing Surveys 28,* 4, 626–643.

CULLER, D. E. 2006. TinyOS: Operating system design for wireless sensor networks. *Sensors 23,* 5.

GARDINER, P. ET AL. 2005. *Failures-Divergences Refinement: FDR2 User Manual*. Formal Systems (Europe) Ltd.

GAY, D., LEVIS, P., CULLER, D., AND BREWER, E. 2005. nesC 1.2 language reference manual. `http://nescc.cvs.sourceforge.net/viewvc/*checkout*/nescc/nesc/doc/ref.pdf?revision=1.18`.

GAY, D., LEVIS, P., VON BEHREN, R., WELSH, M., BREWER, E., AND CULLER, D. 2003. The nesC language: A holistic approach to networked embedded systems. *ACM SIGPLAN Notices 38,* 5, 1–11.

KLEINE, M. AND HELKE, S. 2009. Low-level code verification based on CSP models. In *Proc. of the Brazilian Symposium on Formal Methods (SBMF'09)*. Springer-Verlag, Berlin, 266–281.

KOTHARI, N., MILLSTEIN, T., AND GOVINDAN, R. 2008. Deriving state machines from TinyOS programs using symbolic execution. In *Proc. of the 7th International Conference on Information Processing in Sensor Networks (IPSN '08)*. IEEE Computer Society, 271–282.

LEVIS, P. 2006. TinyOS programming. `http://csl.stanford.edu/~pal/pubs/tinyos-programming.pdf`.

LEVIS, P. 2007. TinyOS 2.x boot sequence. TinyOS Extension Proposal TEP-107, TinyOS Core Working Group.

LEVIS, P. ET AL. 2005. TinyOS: An operating system for sensor networks. In *Ambient Intelligence*, W. Weber, J. Rabaey, and E. Aarts, Eds. Springer, Berlin, 115–148.

LEVIS, P., LEE, N., WELSH, M., AND CULLER, D. 2003. TOSSIM: accurate and scalable simulation of entire TinyOS applications. In *Proc. of the 1st International Conference on Embedded Networked Sensor Systems (SenSys '03)*. ACM, 126–137.

MCINNES, A. I. 2009. Using CSP to model and analyze TinyOS applications. In *Proc. of the 16th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS'09)*. IEEE Computer Society, 79–88.

ROSA, N. S. AND CUNHA, P. R. F. 2007. Using LOTOS for formalising wireless sensor network applications. *Sensors 7,* 8, 1447–1461.

ROSCOE, A. W. 1998. *The Theory and Practice of Concurrency*. Prentice Hall, Englewood Cliffs, NJ.

SHARP, C. 2008. AlarmToTimerC.nc. TinyOS 2.x Source. `http://code.google.com/p/tinyos-main/source/browse/trunk/tos/lib/timer/AlarmToTimerC.nc?r=5108`.

VÖLGYESI, P., MARÓTI, M., DÓRA, S., OSSES, E., AND LÉDECZI, Á. 2005. Software composition and verification for sensor networks. *Science of Computer Programming 56,* 1-2, 191–210.

WOEHRLE, M., PLESSL, C., BEUTEL, J., AND THIELE, L. 2007. Increasing the reliability of wireless sensor networks with a distributed testing framework. In *Proc. of the 4th Workshop on Embedded Networked Sensors (EmNets '07)*. ACM, 93–97.

XIE, F., YANG, G., AND SONG, X. 2006. Compositional reasoning for hardware/software co-verification. In *Proc. of the 4th International Symposium on Automated Technology for Verification and Analysis (ATVA)*. Springer, 154–169.

Received March 2010; revised June 2011; accepted TBD 2011