View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Cleveland-Marshall College of Law

Cleveland State University
EngagedScholarship@CSU

Electrical Engineering & Computer Science Faculty Publications

Electrical Engineering & Computer Science Department

2014

Supporting the Specification and Runtime Validation of Asynchronous Calling Patterns in Reactive Systems

Jiannan Zhai
Clemson University

Nigamanth Sridhar
Cleveland State University, n.sridhar1@csuohio.edu

Jason O. Hallstrom
Clemson University

Follow this and additional works at: https://engagedscholarship.csuohio.edu/enece_facpub

Part of the Electrical and Computer Engineering Commons
How does access to this work benefit you? Let us know!

Original Citation

J. Zhai, N. Sridhar and J. O. Hallstrom, "Supporting the specification and runtime validation of asynchronous calling patterns in reactive systems," in Runtime Verification: 5th International Conference, RV 2014, Toronto, ON, Canada, September 22-25, 2014. Proceedings, B. Bonakdarpour and S. A. Smolka, Eds. Cham: Springer International Publishing, 2014, pp. 108-123.

Repository Citation

Zhai, Jiannan; Sridhar, Nigamanth; and Hallstrom, Jason O., "Supporting the Specification and Runtime Validation of Asynchronous Calling Patterns in Reactive Systems" (2014). *Electrical Engineering & Computer Science Faculty Publications*. 372.
https://engagedscholarship.csuohio.edu/enece_facpub/372

This Conference Proceeding is brought to you for free and open access by the Electrical Engineering & Computer Science Department at EngagedScholarship@CSU. It has been accepted for inclusion in Electrical Engineering & Computer Science Faculty Publications by an authorized administrator of EngagedScholarship@CSU. For more information, please contact library.es@csuohio.edu.

Supporting the Specification and Runtime Validation of Asynchronous Calling Patterns in Reactive Systems

Jiannan Zhai[1], Nigamanth Sridhar[2], and Jason O. Hallstrom[1]

[1] School of Computing, Clemson University, Clemson, SC USA 29634
[2] Electrical and Computer Engineering, Cleveland State University, Cleveland, OH USA 44115

Abstract. Wireless sensor networks (*"sensornets"*) are highly distributed and concurrent, with program actions bound to external stimuli. They exemplify a system class known as *reactive systems*, which comprise execution units that have "hidden" layers of control flow. A key obstacle in enabling reactive system developers to rigorously validate their implementations has been the absence of precise software component specifications and tools to assist in leveraging those specifications at runtime. We address this obstacle in three ways: (*i*) We describe a specification approach tailored for reactive environments and demonstrate its application in the context of sensornets. (*ii*) We describe the design and implementation of extensions to the popular *nesC* tool-chain that enable the expression of these specifications and automate the generation of runtime monitors that signal violations, if any. (*iii*) Finally, we apply the specification approach to a significant collection of the most commonly used software components in the *TinyOS* distribution and analyze the overhead involved in monitoring their correctness.

1 Introduction

In software development, there is a behavioral spectrum that runs from purely *synchronous* to purely *asynchronous*. A purely synchronous system contains a single thread of control, typically originating from `main()`. Traditional component-based specification and validation strategies were designed with these systems in mind and have proven to be effective in ensuring application correctness. Toward the middle of this spectrum are the more common applications, comprising multiple threads that communicate through narrow interfaces, or through a small set of shared variables, essentially forming a collection of synchronous, semi-independent activities. In this context, component-based specification and validation mechanisms begin to break down; they were not designed to handle frame property violations originating from outside the main control thread. At the far end of the spectrum, in the presence of pure asynchrony, component-based specification and validation mechanisms break down entirely.

A *reactive system* is one in which an invocation sequence may originate from outside the main thread of control (e.g., `main()`). Such systems are increasingly important, particularly in the context of embedded applications, which tend to spend much of their time in a reduced power state to conserve energy, waking in response to internal and external interrupts. We focus on the rigorous characterization and validation of such systems. The discussion is presented in the context of *nesC* [12], a component-based dialect of the C programming language, using examples from the *TinyOS* [14] distribution, the most popular operating system (library) of its kind for building wireless sensor network systems. However, the basic principles of the runtime verification approach are applicable to a range of languages and systems, including standard event-based systems developed in Java, and interrupt-based systems developed in other embedded C dialects.

Reactive systems often depend on external stimuli, e.g., from an attached sensor or control system. These systems are commonly implemented using an *event-driven* programming style, encoding the application's behavior in the form of a state machine, with actions tied to each state. The transitions among these states are initiated internally by the application, as well as through external signals. In this style of expression, all concurrent behaviors are explicit. So while well-suited to accommodating interrupt behavior, it poses a significant burden in terms of program understanding. Program logic is partitioned into disjoint units that are often textually distant; the state shared among these units must be managed manually, including control flow state [2]. Not only are these programs more difficult to understand, the transition from synchrony to asynchrony precludes the application of contract-based specification and validation mechanisms — arguably the most powerful tools for ensuring program correctness.

Contract specifications [23] have proven valuable for developing and validating component-based software. Unfortunately, pre- and post-conditions do not support the encoding of event semantics, which dictate properties on the *call sequence* of an execution. Without encoding call sequence properties, the contracts are not as useful; the pre- and post-conditions need to be contextualized by *when* a particular method invocation must occur. The latest attempt at defining interface contracts for TinyOS components suffers this same limitation [3]. The contracts do not preserve the timing context of method calls, offer little abstraction, and leave virtually no implementation freedom.

We use the concept of a *trace* to specify reactive behavior in a precise manner. Given the high degree of expressivity of trace variables, this may not be surprising (though our approach is novel). Here is the surprising part: The trace —traditionally viewed as a brute-force, heavy-weight mechanism— can be used to specify reactive behavior in a manner that is both *concise* and *accessible.* Using the trace construct, we define the notion of a *promise* that an operation makes about its future behavior. This promise, captured in a specialized `promises` clause, accompanies traditional pre- and post-conditions in the contract.

There has been extensive work in runtime validation using various temporal logics and associated tools. Despite their expressive power, there is little evidence of programmer adoption. The contributions of this paper are of an applied

nature, serving as a bridge from the theoretical programming languages community to a popular programming domain. The goal is to provide a practical toolset, both in terms of language extensions and supporting software tools, to enable practitioners to make use of temporal concepts. Our specification approach is to recast traditional temporal specifications as time-indexed state vectors, and to introduce suitable language notations to integrate the resulting conditions as part of state-based pre- and post-conditions. The supporting tools check these conditions to the extent possible.

To support the use of promises in sensornet development, we extend the *nesC* tool-chain to accommodate an optional `promises` clause as part of a method's signature. At compile-time, the `promises` are used to generate runtime monitors that are woven throughout the resulting application image. If a `promise` is violated, the monitors signal the violation, notifying the developer, and potentially triggering corrective measures. We describe the design and implementation of the tool-chain extensions and demonstrate their use across a significant set of commonly used components within the TinyOS distribution. Finally, we present a detailed analysis of the runtime overhead these extensions introduce and show that the overhead is modest in most cases.

2 TinyOS and nesC

TinyOS [14] is a software component library designed for constructing sensornets. The components and the programs which use them are written in *nesC* [12], a dialect of C that supports component-oriented, event-driven programming.

A nesC program consists of *interfaces* and *modules*. A nesC interface is analogous to a Java interface and defines the *command* signatures that must be provided by implementations of that interface. An interface may additionally define one or more *events* that will be signaled by an implementation. An event declaration defines the signature of its callback handler.

A nesC module defines a set of interfaces *provided* by the component, and a set of interfaces *used* by the component. The module is then responsible for implementing the commands that it *provides* and relies on the commands that it *uses* to satisfy those implementations. The module is also responsible for implementing the events (i.e., handlers) defined by the interfaces that it uses.

Long-running operations in TinyOS are implemented as *split-phase* operations. In the first phase, the component that initiates the operation (e.g., sending a message) calls a command to initiate the operation (`send()`). The component that receives the command immediately returns control to the caller after registering the request. This prevents the processor from blocking, allowing the caller to continue execution. At a later point, when the operation has completed, an event is signaled (e.g., `sendDone()`, originating from interrupt context) to the calling component notifying it of the completion of the split-phase operation.

3 The Specification Approach

```
1 interface Timer {
2   modeled by: (active: boolean, period: nat number)
3   initial state: (false, 0)
4   command void start(uint32_t delay);
5   command void stop();
6   event void fired();
7 }
```

Consider the `Timer` interface shown above. The interface provides commands to start and stop a timer, and an event that serves as the timer's periodic signal. A component using this interface can start a timer, with the expectation that when `delay` time units have elapsed, the `fired()` event will be signaled. Using simple state predicates, a first spec attempt might look as follows (based on [3]):

```
1 command void start(uint32_t delay);
2   requires: !self.active
3   ensures: self.active ∧ self.period = delay
```

While the spec captures the state change induced by the call to `start()`, it does not capture the most important impact of the call — at a future time (i.e., `delay` time units later), the `fired()` event will be signaled. Using a temporal specification to capture this liveness property, a second attempt might look like:

```
1     start() ⇝ fired()
```

But such temporal specifications do not coexist well with state contracts, compromising compositional reasoning [18]. The desired goal is to express the direct relationship between the call to `start()` and the signaling of `fired()`. To do so, we introduce our main specification mechanism — namely, $f\tau$, pronounced "future trace" of execution. The future trace of a component is the sequence of method footprints (both incoming and outgoing) that the component will ultimately participate in. Using $f\tau$, we can make an assertion that as a result of the call to `start()`, the `fired()` event will be signaled in the future. To simplify the expression of assertions defined over $f\tau$, we introduce two predicates, $CallAt()$ and $CallBet()$:

$$
\begin{aligned}
&CallAt(source, target, method, time) \equiv \\
&\quad (f\tau[time].s = source) \wedge (f\tau[time].t = target) \wedge (f\tau[time].m = method)
\end{aligned}
$$

$CallAt()$ is *true* if the *source* object places a call to the *method* body provided by the *target* object at the specified *time*, where *time* is defined as an index into $f\tau$.

$$
\begin{aligned}
&CallBet(source, target, method, lb, ub) \equiv \\
&\quad (\ \ \exists ft : lb < time < ub : \\
&\qquad (f\tau[time].s = source) \wedge (f\tau[time].t = target) \wedge (f\tau[time].m = method)\ \ )
\end{aligned}
$$

$CallBet()$ evaluates to *true* iff the call occurs within a specified window, given by lower-bound lb, and upper-bound ub, again defined as indices into $f\tau$. When applying these predicates, we often wish to disregard the *source* and/or *target* clauses. Rather than introducing additional predicates, we introduce the special object value $-$, indicating "don't care"; $object = -$ evaluates to *true* for all *object* values. With these definitions in place, consider a third attempt at specifying `Timer.start()`:

```
1 command void start(uint32_t delay);
2   requires: !self.active
3   ensures: self.active ∧ self.period = delay ∧ CallBet(self, −, fired, now, ∞)
```

The last conjunct states that at some time in the future (*i.e.*, after the current time, *now*), a `fired` event will be signaled. Now let us consider the rest of the interface. The `stop()` command stops an active timer. In terms of $f\tau$, the command guarantees that there is no `fired()` signal in the future, between current time and the "end" of time.

```
1 command void stop();
2   requires: self.active
3   ensures: !self.active ∧ self.period = 0 ∧ ¬CallBet(self, −, fired, now, ∞)
```

While individually meaningful, the specifications miss a key relationship *between* the two commands. In the case of `start()`, the method can guarantee a `fired()` event in $f\tau$ only if there is no call to `stop()` in the intervening duration. Similarly, a call to `start()`, after a call to `stop()` will, in fact, introduce a `fired()` event in $f\tau$. Accounting for this in the specifications of `start()` and `stop()` results in this next attempt:

```
1 command void start(uint32_t delay);
2   requires: !self.active
3   ensures: self.active ∧ self.period = delay ∧
4         ∃i : now < i : [CallAt(−, self, stop, i) ∧ ¬CallBet(self, −, fired, i, ∞)] ∨
5                   [¬CallBet(−, self, stop, now, i) ∧ CallAt(self, −, fired, i)]
6 command void stop();
7   requires: self.active
8   ensures: !self.active ∧ self.period = 0 ∧
9         ∀i : now < i : CallAt(self, −, fired, i) ⟹ CallBet(−, self, start, now, i)
```

While improved, the specifications are no longer independent. A post-condition is intended to capture only what is true about the component upon successful termination. The last conjunct in each post-condition is a predicate on the *future* behavior of the component. One way of addressing this is to elevate predicates on $f\tau$ to an invariant on the component, succinctly capturing all correct interleavings of command invocations. Each command specification then refers only to the corresponding command, independent of other commands. The invariant for the `Timer` interface is as follows:

```
1 ∀i : [[CallAt(−, self, start, i)
2         ⟹ ∃j : i < j : CallAt(self, −, fired, j) ∨ CallBet(−, self, stop, i, j)] ∧
3    [CallAt(self, −, fired, i)
4         ⟹ ∃h : h < i : CallAt(−, self, start, h) ∧ ¬CallBet(−, self, stop, h, i)]]
```

The first conjunct states that each call to `start()` results in a future call to `fired()`, or there is an interleaving call to `stop()`. The second conjunct states that every call to `fired()` must have been preceded by a call to `start()`, and there must have been no interleaving call to `stop()`. Given this invariant, the command contracts can again be expressed as simple state assertions on the abstract model. However, the split-phase correspondence between `start()` and `stop()` is left implicit. This is a useful relationship for developers, one that can be captured with a new `promises` clause.

The `promises` clause defines an obligation that a component must meet at some point after termination of the current command. It is the dual of the

`expects` clause [18], which describes the obligations that a component expects *clients* to meet after successful termination of an operation. The key difference between `expects` and `promises` is in the "direction" of the deferred method call.

```
1 command void start(uint32_t delay);
2   requires: !self.active
3   ensures: self.active ∧ self.period = delay
4   promises: signal caller.fired()
```

Operationally, in addition to the control-flow context and variable values in each state of the program, each component maintains a *promise set* – a set of actions that it has promised to other components. For example, upon successful termination of the `start()` method, the `Timer` component promises to signal `fired()` on the caller. The complete specification of `Timer` is as follows:

```
1 interface Timer {
2   modeled by: (active: boolean, period: nat number)
3   initial state: (false, 0)
4   maintains:
5   ...invariant clause presented above...
6   command void start(uint32_t delay);
7     requires: !self.active
8     ensures: self.active ∧ self.period = delay
9     promises: signal caller.fired()
10  command void stop();
11    requires: self.active
12    ensures: !self.active ∧ self.period = 0
13  event void fired();
14    requires: self.active
15    ensures: !self.active ∧ self.period = delay
16 }
```

The `promises` clause on `start()` specifies both halves of the split-phase operation, adding significant reasoning value for client programmers. Consider a program that invokes `foo()`, followed, after a delay of 1000 time units, by `bar()`:

```
1 void op1() {  foo(); call Timer.start(1000);  }
2 ...
3 event void Timer.fired() {  bar();  }
```

After calling `foo()`, `op1()` starts a timer and terminates. The call to `bar()` appears within the event handler of `fired()`. Without the `promises` clause, there is no indication of where program control will continue once the timer expires.

3.1 The Invariant as an Idiom

The invariant on the future trace has broad applicability in reactive programming. In nesC, the invariant serves as an idiom for specifying interfaces that contain a split-phase operation started by `SPOpStart()` and completed by `SPOpDone()`; and contain an operation `cancelSPOp()`, used to cancel an operation after it has been initiated. The invariant idiom for such a component is:

$$
\begin{aligned}
&1\quad \forall i : [[\mathit{CallAt}(-,\mathtt{self},\mathtt{SPOpStart}, i) \\
&2\qquad\quad \Longrightarrow \exists j : i < j : \mathit{CallAt}(\mathtt{self}, -,\mathtt{SPOpDone}, j) \vee \mathit{CallBet}(-,\mathtt{self},\mathtt{SPOpCancel}, i, j)] \ \wedge \\
&3\quad\ \ [\mathit{CallAt}(\mathtt{self}, -,\mathtt{SPOpDone}, i) \\
&4\qquad\quad \Longrightarrow \exists h : h < i : \mathit{CallAt}(-,\mathtt{self},\mathtt{SPOpStart}, h) \wedge \neg \mathit{CallBet}(-,\mathtt{self},\mathtt{SPOpCancel}, h, i)]]
\end{aligned}
$$

The structure mirrors the "instantiated" invariant for the `Timer` interface. As another example, consider applying the idiom to the `Send` interface in TinyOS, used to send wireless messages in a network. The idiom correspondence is as follows: `send()` corresponds to `SPOpStart()`, `sendDone()` corresponds to `SPOpDone()`, and `cancel()` corresponds to `cancelSPOp()`. Combining the instantiated specification idiom with the usual state predicates yields the following specification:

```
1  interface Send {
2    modeled by: (active: boolean, message: string)
3    initialization ensures: (false, <>)
4    maintains:
5      ...instantiated invariant...
6    command error_t send(message_t* msg, uint8_t len);
7      requires: !self.active
8      ensures: self.active ∧ self.message = #msg
9      promises: signal caller.sendDone()
10   command error_t cancel(message_t* msg);
11     ...standard state conditions...
12   event void sendDone(message_t* msg);
13     ...standard state conditions...
14 }
```

3.2 Refining Promises

Conditional Promise. Consider the `Send` interface. When `send()` is invoked, the message to be sent is placed in an outgoing buffer. If this step completes, `send()` returns `SUCCESS`; otherwise, it returns `FAIL`. The return value communicates to the client that `sendDone()` will be signaled only if the message is successfully scheduled for transmission. Accordingly, we modify the specification of `send()`:

```
1 command error_t send(message_t* msg, uint8_t len);
2   requires: !self.active
3   ensures: self.active ∧ self.message = #msg
4   promises: (retval == SUCCESS) ⟹ signal caller.sendDone()
```

Conditional promises, which allow for a promise to be made contingent on a state assertion, are a specialization of the basic idiom. The basic idiom assumes that commands always complete in a state that guarantees the promise. Conditional promises can be used in cases where such an assumption is unrealistic.

Timed Promise. It is often useful to specify *when* invocations must occur. Consider again the `Timer` interface. When a timer is started, it is not enough to promise that `fired()` will eventually be signaled. It is also necessary to state that the event will be signaled after `delay` time. We can strengthen the specification of `start()` as follows:

```
1 command void start(uint32_t delay);
2   requires: !self.active
3   ensures: self.active ∧ self.period = delay
4   promises: signal caller.fired() within delay
```

Repeat Promise. In some cases, a single split-phase `SPOpStart()` can lead to multiple event signals. Consider, for example, a periodic timer. In such cases, the `promises` clause includes the `repeat` keyword, signifying that the event will be signaled continuously until the `cancel` operation is called by the client. We can specify the start of a periodic timer using a repeat promise as follows:

```
1 command void startPeriodic(uint32_t delay);
2   requires: !self.active
3   ensures: self.active ∧ self.period = delay
4   promises: signal caller.fired() within delay repeat
```

Notice here that the promise includes both a time limit and a repeat condition. In practice, most promises have multiple refinement annotations.

4 nesC / TinyOS Tool-Chain Extensions

To assist developers use our approach, we have developed extensions to the nesC compiler . Specifically, we have extended the nesC parser to accommodate a variation on the specification syntax introduced in the previous sections. Further, we have modified the compiler to enable the generation of runtime monitoring logic used to detect promise violations. This logic is automatically woven throughout the source base, if requested. For our case studies, we target a significant subset of the components and applications included in the TinyOS 2.1.1 distribution.

4.1 Annotations

To support `promises`, we introduce command-level annotations within the nesC interface grammar. When specifying that a given command issues a promise, the developer introduces the following annotation on the event signature, where the `<event>` parameter specifies the signature of the event to be invoked in the future: `@promises <event>`

To support refined promises, three subordinate annotations (applied beneath the root `@promises` annotation) are introduced. The first is used to support a conditional promise; it imposes a condition on the return value of the initiating command. A `<condition>` clause specifies a value to compare against the initiating command's return value. Only if these values match is a promise made: `@condition <condition>`

The second subordinate annotation supports timed promises. The annotation specifies that the promised event will be invoked within `<p>` time units, where the unit of measure is (at present) specified at compile time: `@within <p>`

The final subordinate annotation supports repeat promises. This annotation accepts no parameters and specifies that the promised event will be invoked repeatedly: `@repeat`

Consider the application of these annotations in specifying the behavior of the `SplitControl` power management interface in TinyOS. The interface has two commands, `start()` and `stop()`, with two corresponding events, `startDone()` and `stopDone()`. The `start()`/ `startDone()` operation is used to initialize a peripheral, while the `stop()/stopDone()` operation is used to put a peripheral into a low-power state. The commands, return codes, and events have the usual meanings. The annotated signature of `start()` is:

```
1 // @promises startDone
2 // @condition SUCCESS
3 command error_t start();
```

Table 1. Annotated TinyOS 2.1.1 Interfaces

Interface	Command	Promised Event	Periodicity	Timed	Condition
Send	send	sendDone	singleton	NO	SUCCESS
AMSend	send	sendDone	singleton	NO	SUCCESS
CC2420Config	sync	syncDone	singleton	NO	SUCCESS
Tcp	connect	connectDone	singleton	NO	SUCCESS
Mount	mount	mountDone	singleton	NO	SUCCESS
Read	read	readDone	singleton	NO	SUCCESS
ReadStream	postBuffer	bufferDone	singleton	NO	SUCCESS
	read	readDone	singleton	NO	SUCCESS
SplitControl	start	startDone	singleton	NO	SUCCESS
	stop	stopDone	singleton	NO	SUCCESS
Timer	startOneShot	fired	singleton	YES	(none)
	startPeriodic	fired	repeat	YES	(none)
ConfigStorage	read	readDone	singleton	NO	SUCCESS
	write	writeDone	singleton	NO	SUCCESS
	commit	commitDone	singleton	NO	SUCCESS
LogWrite	append	appendDone	singleton	NO	SUCCESS
	erase	eraseDone	singleton	NO	SUCCESS
	sync	syncDone	singleton	NO	SUCCESS
LogRead	read	readDone	singleton	NO	SUCCESS
	seek	seekDone	singleton	NO	SUCCESS

Next recall the `Timer` interface. This interface includes a command `startPeriodic()`, which makes a promise that the event `fired()` will be invoked repeatedly, with a period specified as argument. The command does not return a value, so the promise is unconditional. Here is the annotated signature of `startPeriodic()`:

```
1 // @promises fired
2 // @within dt
3 // @repeat
4 command void startPeriodic(uint32_t dt);
```

These are demonstrative examples. We have annotated *all* of the core interfaces in TinyOS 2.1.1 to specify the appropriate promises (Table 1).

4.2 Overhead Evaluation

To use the PromiseTracker tool with TinyOS applications, we recompiled all of the constituent applications to use the annotated interfaces and corresponding runtime monitors. The number and types of promises introduced in each application are summarized in Table 2a. Each application is intended to illustrate only one or two TinyOS concepts. As such, each application uses a small number of split-phase operations. Table 2b shows the overhead introduced by PromiseTracker. In absolute terms, the overhead is nearly the same in each application.

To evaluate PromiseTracker in a realistic scenario, we instrumented a common spanning tree data collection protocol. Upon deployment, the nodes in the network organize themselves into a spanning tree, with the base-station at the root of the tree. All nodes collect data from their sensors and transmit the data up the tree toward the root. When instrumented with PromiseTracker, the spanning tree protocol uses a total of 30 promises and nearly all of the core interfaces in TinyOS. In terms of overhead, RAM usage increased by 33% (from 1,612b to 2,138b), and ROM usage increased by 13% (from 35,404b to 40,130b).

Table 2. TinyOS Evaluation Results

(a) Number of Clauses Introduced

Application	Number of Promises		
	single basic	single timed	repeat timed
Blink	0	0	3
BaseStation	4	0	0
MultihopOscilloscope	5	0	1
MultihopOscilloscopeLqi	5	0	1
MViz	5	0	2
Oscilloscope	3	0	1
PowerUp	0	0	0
RadioSenseToLeds	3	0	1
RadioCountToLeds	3	0	1
Sense	1	0	1

(b) Application Sizes After Injection

Application	Memory Overhead	
	RAM (bytes)/ overhead (%)	ROM (bytes)/ overhead (%)
Blink	672 / 92%	10260 / 74%
BaseStation	2111 / 16%	18696 / 16%
MultihopOscilloscope	3947 / 9%	34716 / 10%
MultihopOscilloscopeLqi	3030 / 12%	30604 / 12%
MViz	2176 / 18%	38814 / 10%
Oscilloscope	1020 / 56%	24948 / 30%
PowerUp	560 / 99%	7032 / 79%
RadioSenseToLeds	990 / 58%	24890 / 30%
RadioCountToLeds	902 / 64%	19736 / 39%
Sense	696 / 83%	15480 / 48%

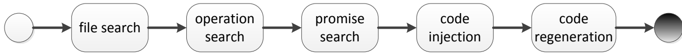

Fig. 1. Monitor Generation Process

4.3 Monitoring Promises at Runtime

The runtime monitoring logic generated by PromiseTracker is automatically woven into a target system image to detect and report violations at runtime. This is useful either as a debugging aid or as the foundation for fault recovery.

A summary of the monitor generation process is shown in Figure 1. The first step is the *file search*, which mirrors the behavior of the nesC *make* system. The project makefile is parsed to identify the top-level component, which is then parsed using the *nesC Analysis and Instrumentation Toolkit* [10] to identify all implementation modules linked (transitively) from the top-level component.

The next step, the *operation search*, is the most compute-intensive. All of the implementation modules identified in the previous step are parsed and analyzed. This yields three hash-tables containing information about all of the interfaces used in the target application, all of the commands invoked, and all of the events signaled, respectively.

At this point, the *promise search*, a second-level parse is performed on each of the interfaces identified in the previous step. For each command invoked in the application, the corresponding declaration in the interface is examined to determine whether there are associated promise annotations. If so, the annotations associated with the command are added to the information contained within the command hash-table.

Next, the *code injection* step is performed, which introduces the runtime monitoring logic. The most basic component of this step is the introduction of support components and data structures to record pending and failed promises. In addition, for each annotated command invoked in the application, instrumentation is introduced at the call site to capture the (perhaps conditional) promise being

made. Similarly, the corresponding `<event>` specified in the `promises` annotation is instrumented to capture the attempt to satisfy the promise.

Finally, the *code regeneration* step is performed to generate augmented nesC source materials ready for compilation and installation on the target device(s).

Implementation Details. The `PromiseTracker` interface lies at the core of the system. The interface provides commands to register new promises, flag that particular promises have been satisfied, and check for pending promises. An implementation of this interface is linked into every monitored application. This single instance is shared across all module implementations that invoke methods involving a promise.

During code analysis, each call site involving a command that establishes a promise is identified. To differentiate these promises and monitor their correctness over time, the analysis stage assigns a unique identifier to each promise, a `promiseID`. The identifier serves as an index into an array that stores information about each promise. The data structure used to store information about an unbounded promise is as follows:

```
1 struct UnboundedPromise { uint8_t state; }
```

`UnboundedPromise` defines a single field, `state`, used to record the current state of the promise. There are only two possibilities, `PENDING` and `SUCCESS`. The first indicates that a promise of future behavior has been made. The latter indicates that there is no pending promise. It is interesting to note that these are the only two states required since an unbounded promise can never be violated in a finite prefix of a computation. However, recording unbounded promises at the time they are made and keeping a tally of unfulfilled promises is a valuable tool for system developers. This class of problems (unfulfilled promises) represents a large class of errors in embedded networked systems; the identification of where these errors originate is useful. The data structure used to store information about a timed promise is as follows:

```
1 struct TimeBoundedPromise {
2     bool repeat;    uint8_t state;
3     uint32_t timeConstraint, startTime;  }
```

`TimeBoundedPromise` defines four fields. The first, `repeat`, is a boolean that records whether the promise is a repeat promise. The second, `timeConstraint`, stores the time constraint, `<p>`, specified as part of the `@within` annotation. The third, `startTime`, stores the time at which the promise obligation was registered. (Comparing the current system time to `startTime` and `timeConstraint` is performed to detect timing failures.) Finally, the `state` field records the current state of the promise. As before, a promise may be in the `PENDING` or `SUCCESS` state. In addition, a timed promise may be in the `MARKED` or `FAIL` states. When a promise is `MARKED`, it indicates that the specified future event has been signaled, but the timing has not yet been checked. The `FAIL` state indicates that a promise of future behavior was not satisfied within the specified time limit.

The essential elements of the `PromiseTracker` interface are: `makePromise()`, `markPromise()`, and `checkPromise()`. Calls to these methods are inserted

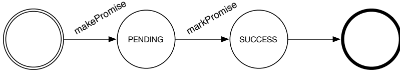

Fig. 2. Singleton, Unbnd. Tracking

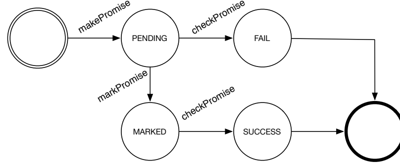

Fig. 3. Singleton, Timed Tracking

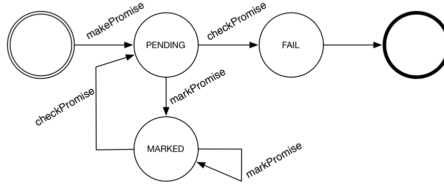

Fig. 4. Repeat, Timed Tracking

automatically during the instrumentation process. When a command that includes a `promises` clause is invoked, `makePromise()` is called to register the promise of future behavior. Note that if the promise is a conditional promise, the return value of the command is compared to the <`condition`> specified in the `@condition` annotation; `makePromise()` is not called if there is a mismatch. The call results in the corresponding promise being marked as `PENDING`. Similarly, a call to `markPromise()` is introduced in the corresponding event. In the case of an unbounded promise, the call results in the promise state being set to `SUCCESS`. In the case of a timed promise, the state is set to `MARKED`. The complete lifecycle of an unbounded promise is illustrated in Figure 2.

The lifecycle of a singleton, timed promise is more complicated, as shown in Figure 3. The call to `markPromise()` is not the end of the lifecycle; an additional step remains. Specifically, the monitoring logic must check whether the promise was satisfied within its deadline. This is done using the `checkPromise()` method. At the time the promise was made, `makePromise()` initiates a timer with a period equal to the specified promise deadline. When the timer fires, `checkPromise()` is invoked. If `checkPromise()` finds the promise in the `PENDING` state, it means the promise has not been kept, and therefore, the deadline has not been met. If the state is `MARKED`, it means the promised event has already been signaled within the deadline. For singleton, timed promises, if the deadline is met properly, the promise is marked `SUCCESS`, otherwise it is marked `FAIL`.

The lifecycle of a repeat, timed promise is similar, as shown in Figure 4. This type of promise is also examined by `checkPromise()` when the deadline timer expires. If the promised event has been signaled by the deadline (`MARKED`), the promise is returned to the `PENDING` state to wait for the next promised event. If the promised event has not yet been signaled (`PENDING`), the promise has been violated and is marked `FAIL`.

4.4 Using PromiseTracker during Development

Once interfaces have been annotated using `promises` clauses to establish links between commands and events, the PromiseTracker tool can be used as a debugging aid during development. When a developer chooses to use a particular interface, the `promises` provide a better understanding of command and event behaviors. During the development cycle, the developer can use PromiseTracker to identify the promises that have been made, and to inject code to monitor these promises. At any point during execution, the developer can query the state of all promises in the system. Errors involving promise violations are notoriously difficult to identify using traditional debugging methods. The capability that PromiseTracker affords in tracking the status of each promise provides value to developers, making the development process more predictable.

5 Related Work

Specification techniques for reactive systems usually include explicit statements of safety and progress properties. Popular specification languages such as UNITY [6] and TLA [19] model concurrency using nondeterministic interleaving of actions. Other major approaches to capturing concurrent behavior include rely-guarantee [1,15,29], hypothesis-conclusion [6], and assumption-commitment [8]. All these techniques suffer from a similar problem; they do not map well to procedural languages.

Contract specifications [23] map well to procedural code, and [18] presents techniques to capture concurrent behavior in contracts. The `promises` clause we have presented is a dual to the `expects` clause presented in [18]. Contract specifications have been written for TinyOS before [3]; however, these contracts do not capture the reactive nature of the components. In particular, these contracts do not capture the relationship between the halves of a split-phase operation.

Others have worked on capturing the behavior of TinyOS applications. [17] presents a technique to automatically derive state machines from TinyOS programs. They use symbolic execution to infer the execution trace of an application, and based on this trace, to construct a finite state machine that represents the behavior of the program. There has also been work in runtime monitoring of TinyOS applications [13]. TOSTracer is a lightweight monitor that runs concurrently with the application program and generates a sequence diagram representation of the application's execution. [4] describes work on verifying TinyOS programs using the CBMC bounded model checker [7].

Li and Regehr [22] present T-Check, a model checking approach for finding interaction bugs in sensor networks. T-Check is implemented on top of Safe TinyOS [9] and allows developers to specify both safety and liveness properties. T-Check incorporates multiple models of non-determinism in order to explore the complete state space of a sensornet. Some of the liveness bugs that T-Check can capture (node-level bugs) can be expressed as promises. Kleenet [26] is a tool based on symbolic execution for discovering interaction bugs in sensor networks. Kleenet has been integrated into Contiki [11].

Several authors have considered monitoring runtime errors using pre-defined specifications. The Monitoring and Checking framework (MaC) [20] is an approach to conducting runtime analysis of a system's execution. MaC uses a formal language to specify execution requirements, which assert events and conditions in a high-level manner. A monitoring script is used to link the high-level events and conditions with low-level information at runtime. Monitored information is converted to events, which are verified based on the requirements. Based on MaC, [28] presents an approach that uses verification results and user specifications to detect errors and adjust the system back to normal execution. [21] presents an approach that not only monitors execution and logs errors, but also takes programmers' system recovery specification as input to perform a desired repair. These efforts focus on monitoring program execution using user-defined specs, whereas our work is focused on tracking split-phase operations at runtime by extending the nesC tool-chain to support command-level annotations.

Dustminer [16] is a diagnostic tool that automatically detects root causes of interaction bugs and performance anomalies in sensor networks. For example, after analyzing collected logs from good nodes and crashed nodes in a sensor network running LiteOS [5], the *packet received* event was identified as highly correlated with the *get current radio handle* event in the good nodes, whereas it was highly correlated with the *get serial send function* event in the crashed nodes. By capturing unexpected event sequences that cause errors, Dustminer focuses on non-localized errors when nodes run distributed protocols. As such, Dustminer helps with diagnosing errors that occur in distributed scenarios, which are usually hard to reproduce. However, Dustminer is not designed to help localize the events in the code that cause these errors.

[30] presents a technique for TinyOS applications that reconstructs control-flow paths between procedures based on captured concurrent events and control-flow paths inside each event. The target program is statically analyzed, and tracing statements are inserted in each event function body. At runtime, the recorded trace is stored in RAM, and then compressed and transferred to flash. When an error is detected, the stored trace is sent to the base-station. By replaying the trace and reproducing the execution sequence in a simulator or debugger, the programmer is better able to locate the fault and the call sequence that led to the fault. This tool requires manual operations and depends highly on the capability of the programmer to identify the error and problematic trace.

There is a vast literature base exploring runtime monitoring for error detection. [27] presents an approach to monitoring the execution of reactive systems and recovering from runtime failures. This approach uses a module that learns the behavior of a reactive system, and when an error is detected, applies a repair action. The choice of which repair action to use is based on an analysis of the execution history. [24] presents a discussion of how to design runtime monitors for real-time systems. The focus is on how to enforce real-time guarantees. Copilot [25] focuses on hard real-time guarantees. The monitoring system samples observable state variables; the monitor and the system share a global clock.

6 Conclusion

Asynchronous behavior in reactive systems is difficult to capture using traditional contract-based specification mechanisms. Such behavior is usually captured using temporal specifications, but the mapping between such specifications and corresponding implementations in procedural languages is cumbersome. In this paper, we have presented a specification idiom that can be used to capture asynchronous behavior in reactive systems using the concept of a *future trace*. When a split-phase operation is initiated, the start command makes a *promise* that an event will be signaled in the future. The promise is encoded as part of the method's contract along with its pre- and post-condition.

The `promises` clause offers a way to capture asynchronous behavior in contract specifications that can be easily integrated with software written in procedural languages such as C. Split-phase operations are particularly common in embedded systems, where blocking operations are not viable. At this point, the promises we are able to specify and capture are only local to a single sensor node. While these represent a large class of potential interaction bugs, interactions between commands and events across nodes represent an even larger class of such bugs. These are even harder to find. We are currently working on extending the semantics of the `promises` clause to be able to express such cross-node promises. Once the semantics are extended, tool support can be readily added. In fact, we already have tools that can capture execution snapshots across nodes in a sensor network and check predicates; promises can be added to such a set of predicates.

As a case study, we have written specifications for TinyOS, which is designed for sensornets. As a way of enforcing promises at runtime, we have implemented a runtime monitoring infrastructure that runs in parallel with the application running on an embedded microcontroller. The runtime monitor, PromiseTracker, injects bookkeeping calls to track each promise made, and to check if the promise is satisfied. This runtime monitor, implemented for TinyOS 2.1.1, serves as a powerful debugging aid in the presence of asynchronous behavior.

Acknowledgments. This work was supported in part by NSF grants CNS-0746632, CNS-0745846, and CNS-1126344.

References

1. Abadi, M., Lamport, L.: Composing specifications. TOPLAS 15(1), 73–132 (1993)
2. Adya, A., et al.: Cooperative task management without manual stack management. In: USENIX 2002, pp. 289–302 (2002)
3. Archer, W., et al.: Interface contracts for tinyos. In: IPSN 2007, pp. 158–165. ACM Press, New York (2007)
4. Bucur, D., Kwiatkowska, M.: On software verification for sensor nodes. J. Syst. Softw. 84, 1693–1707 (2011)
5. Cao, Q., et al.: The liteos operating system: Towards unix-like abstractions for wireless sensor networks. In: IPSN 2008, Washington, DC, USA, pp. 233–244 (2008)
6. Chandy, K.M., Misra, J.: Parallel Program Design: A Foundation. Addison-Wesley, Reading (1988)

7. Clarke, E., Kroning, D., Lerda, F.: A tool for checking ansi-c programs. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 168–176. Springer, Heidelberg (2004)
8. Collette, P.: Composition of assumption-commitment specifications in a UNITY style. SCP 23, 107–125 (1994)
9. Cooprider, N., et al.: Efficient memory safety for tinyos. In: SenSys 2007, pp. 205–218. ACM, New York (2007)
10. Dalton, A.R., Hallstrom, J.O.: nait: A source analysis and instrumentation framework for nesc. J. Syst. Softw. 82, 1057–1072 (2009)
11. Dunkels, A., et al.: Contiki - a lightweight and flexible operating system for tiny networked sensors. In: LCN 2004, Washington, DC, USA, pp. 455–462 (2004)
12. Gay, D., et al.: The nesC language: A holistic approach to networked embedded systems. In: PLDI 2003, pp. 1–11. ACM Press (June 2003)
13. Hammad, M., Cook, J.: Lightweight monitoring of sensor software. In: SAC 2009, pp. 2180–2185. ACM, New York (2009)
14. Hill, J., et al.: System architecture directions for networked sensors. In: ASPLOS, pp. 93–104. ACM Press (November 2000)
15. Jones, C.B.: Tentative steps toward a development method for interfering programs. TOPLAS 5(4), 596–619 (1983)
16. Khan, M.M.H., et al.: Dustminer: troubleshooting interactive complexity bugs in sensor networks. In: SenSys 2008, pp. 99–112. ACM, New York (2008)
17. Kothari, N., et al.: Deriving state machines from tinyos programs using symbolic execution. In: IPSN 2008, pp. 271–282. IEEE, Washington, DC (2008)
18. Kumar, S., et al.: Encapsulating concurrency as an approach to unification. In: SAVCBS 2004, Newport Beach, CA (October 2004)
19. Lamport, L.: The temporal logic of actions. TOPLAS 16(3), 872–923 (1994)
20. Lee, I., et al.: A monitoring and checking framework for run-time correctness assurance. In: Proc. Korea-U.S. Tech Conf. Strat. Tech., Vienna, VA (October 1998)
21. Lewis, C., Whitehead, J.: Runtime repair of software faults using event-driven monitoring. In: ICSE 2010, pp. 275–280. ACM, New York (2010)
22. Li, P., Regehr, J.: T-check: bug finding for sensor networks. In: IPSN 2010, pp. 174–185. ACM Press, New York (2010)
23. Meyer, B.: Applying "design by contract". Computer 25(10), 40–51 (1992)
24. Peters, D.K., Parnas, D.L.: Requirements-based monitors for real-time systems. SIGSOFT Softw. Eng. Notes 25, 77–85 (2000)
25. Pike, L., Goodloe, A., Morisset, R., Niller, S.: Copilot: a hard real-time runtime monitor. In: Barringer, H., et al. (eds.) RV 2010. LNCS, vol. 6418, pp. 345–359. Springer, Heidelberg (2010)
26. Sasnauskas, R., et al.: Kleenet: discovering insidious interaction bugs in wireless sensor networks before deployment. In: IPSN, New York, NY, pp. 186–196 (2010)
27. Seshia, S.A.: Autonomic reactive systems via online learning. In: Proc. IEEE ICAC. IEEE Press (June 2007)
28. Sokolsky, O., et al.: Steering of real-time systems based on monitoring and checking. In: WORDS 1999, p. 11. IEEE Computer Society, Washington, DC (1999)
29. Stark, E.W.: A proof technique for rely guarantee properties. In: Maheshwari, S.N. (ed.) FSTTCS 1985. LNCS, vol. 206, pp. 369–391. Springer, Heidelberg (1985)
30. Sundaram, V., et al.: Efficient diagnostic tracing for wireless sensor networks. In: SenSys 2010, pp. 169–182. ACM, New York (2010)
31. Yang, J.: Clairvoyant: a comprehensive source-level debugger for wireless sensor networks. In: SenSys 2007, pp. 189–203. ACM, New York (2007)

Post-print standardized by MSL Academic Endeavors, the imprint of the Michael Schwartz Library at Cleveland State University, 2016