A Specification Idiom for Reactive Systems

Nigamanth Sridhar
Electrical and Computer Engineering
Cleveland State University
n.sridhar1@csuohio.edu

Jason O. Hallstrom
School of Computing
Clemson University
jasonoh@cs.clemson.edu

Abstract

Interrupt- and event-driven applications constitute an important system class, with connections to desktop computing, embedded systems, and sensor networks. We refer to this set of applications collectively as reactive systems. In this paper, we present a specification idiom for documenting reactive system behavior. Specifically, we discuss an approach to documenting split-phase operations — operations that involve a request, followed by a deferred out-of-context callback. We derive the idiom by example using interfaces from the TinyOS library, a popular component library for sensor network applications. We conclude with a broader discussion of specification idioms for reactive systems.

1 Introduction

In this paper, we focus on the specification of reactive systems. We define this class as including all systems in which invocation sequences originate from outside the main thread of control (e.g., main()). The discussion is presented in the context of nesC [2], using examples from the TinyOS [3] library. The basic principles of the approach, however, are applicable to a range of languages and systems, including standard event-based systems in Java, and interrupt-based systems in embedded C dialects.

Reactive systems involve significant interactions with the external environment. Software for such systems must account for (perhaps) unexpected forms in which the environment may “intrude” on the computation being performed. So much so, that reactive systems are commonly implemented using an event-driven programming style. In such a style, rather than representing a program as a sequence of actions to be performed, the programmer encodes the application’s behavior in a state machine that can include transitions that are initiated both internally in the application, as well as those initiated by the environment.

Event-driven programming, while well-suited to accommodating interrupt behavior, poses a significant burden in terms of program understanding. Right at the surface, several problems are made manifest: Program logic is scattered throughout the source base; consequently, state variables are shared throughout, requiring careful management by the programmer. This problem has been labeled “stack ripping”.

Contract-based specifications [5] are highly useful for designing component-oriented software. Pre- and post-conditions provide an easy way for programmers to ensure that the software components they use (from a library, perhaps) are being used correctly. While contract specifications are easy to write for sequential programs, they are not directly well-suited for concurrent systems. In particular, traditional pre/post-conditional specifications offer no way of encoding event or interrupt behavior. The latest attempt at defining interface contracts for TinyOS components suffers this fate [1]; the specifications are too shallow and do not allow for implementation freedom.

We take the position that the concept of a trace can be used to specify reactive behavior in a precise manner. Given the high degree of expressivity of trace variables, this may not be surprising (though we believe our approach is novel). Here is the surprising part: We also take the position that 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.

It is important to emphasize that we are taking a position on the concept of a trace as a specification mechanism for reactive systems. We are not taking a position on the particular specification syntax. Hence, we adopt elements of several notations to emphasize our neutrality on this point.

Paper Organization. We present a brief overview of TinyOS and nesC in Section 2. We outline our specification approach using an example in Section 3 and show how the structure of our specification approach is an idiom that can be used for components that exhibit reactive behavior. Finally, we discuss our current work extending the approach into the broader domain of reactive systems in Section 4.
2 TinyOS and nesC

TinyOS [3] is a software operating environment designed specifically to support sensor network development. Programs for TinyOS are written in nesC [2], a dialect of C that supports a component-oriented, event-driven programming style. TinyOS is a componentized operating system that supports these applications and provides abstractions for interacting with the underlying hardware. Rather than a general-purpose OS, TinyOS is a collection of nesC components, from which a designer can pick and choose when developing the single application that will run on a particular device.

A nesC program consists of (i) interfaces, (ii) modules, and (iii) configurations. A nesC interface is analogous to a Java interface and defines the command signatures that must be provided by implementations of that interface. Furthermore, nesC interfaces are bi-directional. An interface may define one or more events that will be signaled by an implementation. In effect, an event declaration defines the signature of the handler that will be invoked on clients of the interface when the event is signaled by an implementation.

Operations that involve input/output, and hence require the processor to wait, are implemented as split-phase operations. The basic idea here is to avoid putting the processor in a position where it has to block waiting for some I/O operation to complete. Instead, the operation is split into two phases. In the first phase, the component that is invoking the I/O operation (for example, sending a wireless message through the radio) calls a command to initiate the operation (send()). At this point, the component that receives this command immediately returns control to the caller after registering the request. This prevents the caller from blocking on the processor. At a later point, when the operation has completed, an event is signaled (sendDone()) on the calling component, notifying it of the completion of the split-phase operation.

3 The Specification Approach

Here we outline the key elements of our specification approach. Consider the Timer interface presented below:

```plaintext
interface Timer {
  modeled by: (active: boolean, period: nat number)
  initial state: (false, 0)
  command void start(uint32_t delay);
  command void stop();
  command bool is_active();
  event void fired();
}
```

Implementations of the interface are used to generate events at a desired periodicity. The interface provides commands to start and stop a timer and to check its current state (i.e., started or stopped). It also provides an event that serves as the timer’s 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. So what should the contract specification of the start() operation look like? Using simple state predicates, we can write a specification that looks like this (based on [1]):

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

While this specification indeed communicates what the state of the component is upon termination of the start() operation, this alone does not communicate the actual observable behavior of the component. In particular, the piece that is omitted is that at a future time (more precisely, delay time units later), the fired() event will be signaled.

How can we capture this relationship between the start() operation and the fired() event? Using a temporal specification, we can capture it as a liveness property:

```plaintext
interface Timer
  start() ⇒ fired()
```

But such temporal specifications do not coexist well with component contracts [4]. In effect, what we like to see is an expression of the direct relationship between the call to start() and the signaling of fired(). In order to establish this relationship, we introduce our main specification mechanism — \( f^\tau \), pronounced “future trace” of execution. The future trace of a component is the sequence of method footprints (both incoming and outgoing) that the component is involved in. Using \( f^\tau \), we can make an assertion that as a result of this call to start(), the fired() event will be signaled in the future. Consider the following revision:

```plaintext
command void start(uint32_t delay);
  requires: !self.active
  ensures: self.active ∧ self.period = delay ∧
    (∃i:tc<τ:(fr[i].s = self) ∧ (fr[i].m = fired))
```

The last conjunct of the post-condition for start() states that at some time \( i \) in the future, a fired() event will be signaled. Note that \( tc \) refers to the index of “this call” in the future trace. “.s” indicates the source of a call. “.t” indicates the target of the call. “.m” indicates the method.

Consider the rest of this 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.

```plaintext
command void stop();
  requires: self.active
  ensures: !self.active ∧ self.period = 0 ∧
    ¬(∃tc<τ:(fr[i].s = self) ∧ (fr[i].m = fired))
```

Upon observing the preceding two method contracts, we can see that neither is complete. The post-conditions for the two methods actually depend on each other in order to be complete. 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 spec for `start()` and `stop()` results in this next attempt:

```plaintext
1  command void start(uint32_t delay);
2  requires: !self.active
3  ensures: self.active ∧ self.period = delay ∧
4      (∃ t:i < j < i: (f\tau[i], t = self) ∧ (f\tau[j], m = stop)
5         ⇒ (f\tau[i], s = self) ∧ (f\tau[j], m = fired))
6
7  command void stop();
8  requires: self.active
9  ensures: !self.active ∧ self.period = 0 ∧
10       (∃ t:i < j < i: (f\tau[i], s = self) ∧ (f\tau[j], m = fired)
11          ⇒ (∃ j:i < k < j: (f\tau[i], t = self) ∧ (f\tau[j], m = fired))
12             ∧ (∃ k:i < j < k: (f\tau[j], s = self) ∧ (f\tau[k], m = fired))
13             ∧ (∃ k:i < l < k: (f\tau[l], t = self) ∧ (f\tau[k], m = start)))
```

This specification, while more accurate, introduces a new undesirable, namely that the contract for each method is no longer independent of other methods in the interface. Moreover, these contracts express more than what these methods actually are intended to do. The post-condition of a method is only supposed to capture what is true about the component upon successful termination. The last conjunctions of the post-conditions of the two methods are really predicates on the future behavior of the component.

One way of removing this interdependence across method contracts is to elevate the predicate on \( f\tau \) from the post-conditions of individual methods to an invariant on the component. By doing this, we can ensure that method contracts refer only to the particular operation in question, and not to other (public) operations in the interface. By expressing the behavioral specification as a component-wide invariant, we can capture all interleavings of method invocations in a succinct form, not disallowing any legal behavior.

The following is a predicate on all traces of `Timer` (we will refer to this invariant later as \( \mathcal{E}_{\text{Timer}} \):

\[
\forall t \in \mathbb{N} (\exists i: (f\tau[i], t = self) ∧ (f\tau[i], m = stop) ∧
\neg \exists j: t < j < i: (f\tau[j], t = self) ∧ (f\tau[j], m = stop)
\land
(\exists k: i < k < j: (f\tau[k], s = self) ∧ (f\tau[k], m = fired))
\land
(\exists k: i < j < k: (f\tau[j], s = self) ∧ (f\tau[k], m = fired))
\land
(\exists k: i < l < k: (f\tau[l], t = self) ∧ (f\tau[k], m = start)))
\]

The first disjunct in the invariant says that each call to `start()` results in a `fired()` event signal unless the timer is canceled by way of a call to `stop()`. The second disjunct states that every signal to `fired()` must have been preceded by a call to `start()`. That is, the `fired()` event will not be signaled by the timer spontaneously. Given this invariant on behavior traces that a timer component produces, the method contracts themselves can be simple state assertions on the abstract model.

While the invariant above captures the behavior of the timer interface completely, the relationship between `start()` and `fired()` is left implicit. In particular, the fact that the two are part of the same split-phase operation is not captured. To make it easier for developers to see the relationship, we would like for it to be explicit. To do so, we introduce a new clause in the specification called a promise.

The promises clause defines an obligation that the component must meet at some point after termination of the current method. As such, the promises clause is a dual to the expects clause (introduced in [4]), which describes the obligations that this component expects clients to meet after successful termination of the current method.

Operationally, in addition to the program context and variable values in each state of the program, each component also 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.

Combining all the elements of the specification described so far, we have the following specification:

```plaintext
1  interface Timer {
2    modeled by: (active: boolean, period: nat number)
3    initial state: (false, 0)
4    maintains: \mathcal{E}_{\text{Timer}}
5
6  command void start(uint32_t delay);
7  requires: !self.active
8  ensures: self.active ∧ self.period = delay
9  promises: signal caller.fired()
10
11  command void stop();
12  requires: self.active
13  ensures: !self.active ∧ self.period = 0
14
15  command bool is_active();
16  ensures: is_active = self.active
17
18  event void fired();
19  requires: self.active
20  ensures: !self.active ∧ self.period = delay
21  }
```

The promises clause in the `start()` operation’s post-condition indicates the two pieces of the split-phase operation. To a client programmer, this adds a lot of value from a reasoning perspective. Let us look at split-phase operations a little closer. In a programming language that support blocking I/O operations, a timer interface typically includes a `wait()` or `sleep()` operation. Consider this block of code in a client program that uses a timer:

```plaintext
1  interface Timer {
2    modeled by: (active: boolean, period: nat number)
3    initial state: (false, 0)
4    maintains: \mathcal{E}_{\text{Timer}}
5
6  command void start(uint32_t delay);
7  requires: !self.active
8  ensures: self.active ∧ self.period = delay
9  promises: signal caller.fired()
10
11  command void stop();
12  requires: self.active
13  ensures: !self.active ∧ self.period = 0
14
15  command bool is_active();
16  ensures: is_active = self.active
17
18  event void fired();
19  requires: self.active
20  ensures: !self.active ∧ self.period = delay
21  }
```
In this case, the call to `foo()` executes, and then the thread on which this program is executing `sleeps` for 1000 time units, after which the call to `bar()` executes. The key thing to note is that program control remains in `op1` the entire time. Contrast this with code using split-phase methods:

```c
void op1() {
  foo();
  Timer.sleep(1000);
  bar();
}
```

Following the call to `foo()`, `op1()` starts a timer for 1000 time units. Notice that immediately after making this call, `op1()` terminates. The call to `bar()` which previously appeared on the very next line in program text after the timer call, now appears in the event handler for the `fired()` event. So when reading the client program, in the absence of the promises clause in the specification of `Timer.start()`, there is no indication of where program control will continue once the timer period expires.

### 3.1 The Invariant as an Idiom

Surprisingly, the invariant on the future trace, \( f \tau \), presented in the `Timer` component turns out to have a much wider range of application. The structure of the invariant serves as an idiom to specify the behavior of any c component that:

- contains at least one split-phase operation, initiated by some operation `SPOpStart()` and completed by some operation `SPOpDone()`, and
- contains an operation `cancelSPOp()` that can cancel the split-phase operation after it has been initiated.

The specification idiom for the invariant for such a component is as follows.

\[
\forall t \in \mathbb{N} \ ( \exists i : t < i : (fr[i].t = self) \land (fr[i].m = cancelSPOp) \land \\
\neg \exists j : t < j < i : (fr[j].s = self) \land (fr[j].m = SPOpDone) \land \\
(\exists k : i < k : (fr[k].s = self) \land (fr[k].m = SPOpDone)) \implies \\
(\exists l : i < l < k : (fr[l].t = self) \land (fr[l].m = SPOpStart))
\]

As with the `Timer` interface, the first disjunct of the invariant states that each call to `SPOpStart()` to initiate the split-phase operation leads to a `SPOpDone()` event signal that completes the operation, unless the operation is canceled by a `cancelSPOp()` call. The second disjunct states that the `SPOpDone()` event cannot fire spontaneously and must be triggered by a prior call to the initiating `SPOpStart()` command.

### 4 Discussion

The preliminary work presented in this paper is a first step in documenting **specification idioms** — language-specific specification patterns similar in spirit to those codified by the Gang of Four, and that we hope will be successful for similar reasons: Despite decades of research in program specification and verification, a complete specification and verification solution seems out of reach in the near future. In the meantime, it would be useful to begin collecting reusable elements of best practice in software specification to share that knowledge throughout the community. We think that the idiom presented here for specifying reactive systems is a step in that direction. We envision a much broader catalog of idioms for reactive systems in general, and embedded network systems in particular.

But there is much work to be done. Beyond surveying the literature to identify and document other specification idioms and developing a suitable documentation mechanism, there are still issues to be solved with our first pattern — and the issues are non-trivial. Some of the most pressing issues that we are currently pursuing include:

- Embedded systems typically run on resource-limited hardware. Most of the time, these nodes are powered by consumer-grade batteries, and hence sport highly power-efficient and small processors, memories, etc. A lot of attention, therefore, needs to be placed on specifying the time and space performance of the associated software. We are working on adding such support.

- With reactive systems, the behavioral response of the environment surrounding a software component is frequently as important as the behavior of the component itself. To reason correctly about the behavior of a reactive software system, it is important to bring the environment and its effects, particularly time-sensitive spontaneous interrupts, into the picture.

### References