A Specification Idiom for Reactive Systems

Nigamanth Sridhar[1] and Jason O. Hallstrom[2]
1: Electrical and Computer Engineering, Cleveland State University
2: School of Computing, Clemson University

31st International Conference on Software Engineering, New Ideas and Early Results Track

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.*

Problem Context

- Reactive systems are hard to specify without temporal properties
  - Systems are driven by external stimuli from the environment
  - Some systems are inherently reactive: e.g., wireless sensor networks
- Temporal properties are hard to capture in a call/return programming model
  - e.g., **Split-phase operations**: Call & return realized as separate functions

TinyOS

- Operating system designed to support sensor network development
- Component library provides access to low-level hardware entities
- Execution driven by interrupts and a lightweight **task** scheduler

nesC Interfaces

- Bi-directional; **commands** flow into the component, and **events** flow out of the component
- Non-blocking operations implemented in a split-phase manner

This work is supported by NSF CAREER grants CNS-0746632 and CNS-0745846.

Example: Timer

```
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();
}
```

Key Points

- 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
- Specification needs to capture relation between `start()` and `fired()`

The Specification Approach

Requirement

We need to establish the relation between the initialization and completion of split-phase operations.

Key Specification Mechanism — $f\tau$

The **future trace** of a system is the sequence of method invocations that *must* occur after a given execution point.

Specifying Timer: Attempt 1

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

$$
\begin{aligned}
&(\exists i : \texttt{tc} < i :\\
&\quad \neg\exists j : \texttt{tc} < j < i : (f\tau[j].t = \texttt{self}) \wedge (f\tau[j].m = \texttt{stop})\\
&\quad\quad \Longrightarrow \ (f\tau[i].s = \texttt{self}) \wedge (f\tau[i].m = \texttt{fired}))
\end{aligned}
$$

```
command void stop();
  requires: self.active
  ensures: !self.active ∧ self.period = 0 ∧
```

$$
\begin{aligned}
&(\exists i : \texttt{tc} < i : (f\tau[i].s = \texttt{self}) \wedge (f\tau[i].m = \texttt{fired})\\
&\quad \Longrightarrow \ \exists j : \texttt{tc} < j < i : (f\tau[j].t = \texttt{self}) \wedge (f\tau[j].m = \texttt{start}))
\end{aligned}
$$

Problem: Method specs no longer independent!

Specifying Timer: Attempt 2 — Trace Invariant

$$
\begin{aligned}
&\forall t \in \mathbb{N} \ (\\
&\quad [\exists i : t < i : (f\tau[i].t = \texttt{self}) \wedge (f\tau[i].m = \texttt{stop}) \wedge\\
&\quad\quad \neg\exists j : t < j < i : (f\tau[j].s = \texttt{self}) \wedge (f\tau[j].m = \texttt{fired}) \wedge\\
&\quad\quad\quad \exists k : i < k : (f\tau[k].s = \texttt{self}) \wedge (f\tau[k].m = \texttt{fired})]\\
&\quad\quad\quad \Longrightarrow\\
&\quad\quad\quad \exists l : i < l < k : (f\tau[l].t = \texttt{self}) \wedge (f\tau[l].m = \texttt{start})\\
&\quad \vee\\
&\quad [\exists i : t < i : (f\tau[i].s = \texttt{self}) \wedge (f\tau[i].m = \texttt{fired}) \wedge\\
&\quad\quad \neg\exists j : j < i : (f\tau[j].s = \texttt{self}) \wedge [(f\tau[j].m = \texttt{fired}) \wedge\\
&\quad\quad\quad \exists k : i < k : (f\tau[k].s = \texttt{self}) \wedge (f\tau[k].m = \texttt{fired})]\\
&\quad\quad\quad \Longrightarrow\\
&\quad\quad\quad \exists l : i < l < k : (f\tau[l].t = \texttt{self}) \wedge (f\tau[l].m = \texttt{start}))
\end{aligned}
$$

Generalized Specification Idiom

$$
\begin{aligned}
&\forall t \in \mathbb{N} \ (\\
&\quad [\exists i : t < i : (f\tau[i].t = \texttt{self}) \wedge (f\tau[i].m = \texttt{cancelSPOp}) \wedge\\
&\quad\quad \neg\exists j : t < j < i : (f\tau[j].s = \texttt{self}) \wedge (f\tau[j].m = \texttt{SPOpDone}) \wedge\\
&\quad\quad \exists k : i < k : (f\tau[k].s = \texttt{self}) \wedge (f\tau[k].m = \texttt{SPOpDone})]\\
&\quad\quad \Longrightarrow\\
&\quad\quad \exists l : i < l < k : (f\tau[l].t = \texttt{self}) \wedge (f\tau[l].m = \texttt{SPOpStart})\\
&\quad \vee\\
&\quad [\exists i : t < i : (f\tau[i].s = \texttt{self}) \wedge (f\tau[i].m = \texttt{SPOpDone}) \wedge\\
&\quad\quad \neg\exists j : j < i : (f\tau[j].s = \texttt{self}) \wedge [(f\tau[j].m = \texttt{SPOpDone}) \wedge\\
&\quad\quad \exists k : i < k : (f\tau[k].s = \texttt{self}) \wedge (f\tau[k].m = \texttt{SPOpDone})]\\
&\quad\quad \Longrightarrow\\
&\quad\quad \exists l : i < l < k : (f\tau[l].t = \texttt{self}) \wedge (f\tau[l].m = \texttt{SPOpStart}))
\end{aligned}
$$

Sample Traces