A Specification Idiom for Reactive Systems

Nigam Sridhar1 and Jason O. Hallstrom2
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

Example: Timer

```java
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 — \( f_t \)

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

Specifying Timer: Attempt 1

```java
command void start(uint32_t delay);
    requires: !self.active
    ensures: self.active ∧ self.period = delay ∧
    (∃k: tc < i: ¬∃j < i: (f_t[j] = self) ∧ (f_t[j], m = stop) ∧ (f_t[j], m = fired))
    ⇒ (f_t[i] = self) ∧ (f_t[i], m = fired)
command void stop();
    requires: self.active
    ensures: !self.active ∧ self.period = 0 ∧
    (∃k: tc < i: ¬∃j < i: (f_t[j] = self) ∧ (f_t[j], m = fired) ∧ (f_t[i], m = start))
```

Problem: Method specs no longer independent!

Specifying Timer: Attempt 2 — Trace Invariant

```java
∀t ∈ N {  
    [3]: t < j < i: (f_t[j], t = self) ∧ (f_t[j], m = stop) ∧ 
    ¬∃j: j < i: (f_t[j], s = self) ∧ (f_t[j], m = fired) ∧ 
    ∃k: i < k: (f_t[k], s = self) ∧ (f_t[k], m = fired) 
    ⇒ (f_t[i], t = self) ∧ (f_t[i], m = start) 
∨ [3]: t < i: (f_t[i], t = start) \( \land \) (f_t[i], m = fired) \( \land \) 
    ¬∃j: j < i: (f_t[j], s = self) ∧ (f_t[j], m = fired) \( \land \) 
    ∃k: i < k: (f_t[k], s = self) ∧ (f_t[k], m = fired) 
    ⇒ (f_t[i], t = self) ∧ (f_t[i], m = start) 
}
```

Generalized Specification Idiom

```java
∀t ∈ N {  
    [3]: t < j < i: (f_t[j], t = self) ∧ (f_t[j], m = cancelSPOp) \( \land \) 
    ¬∃j: j < i: (f_t[j], s = self) \( \land \) (f_t[j], m = SPOpDone) \( \land \) 
    ∃k: i < k: (f_t[k], s = self) \( \land \) (f_t[k], m = SPOpDone) 
    ⇒ (f_t[i], t = self) \( \land \) (f_t[i], m = SPOpStart) 
∨ [3]: t < i: (f_t[i], t = self) \( \land \) (f_t[i], m = SPOpDone) \( \land \) 
    ¬∃j: j < i: (f_t[j], s = self) \( \land \) (f_t[j], m = SPOpDone) \( \land \) 
    ∃k: i < k: (f_t[k], s = self) \( \land \) (f_t[k], m = SPOpDone) 
    ⇒ (f_t[i], t = self) \( \land \) (f_t[i], m = SPOpStart) 
}
```

Sample Traces

Valid Trace

Invalid Trace