Initiating a Design Pattern Catalog for Embedded Network Systems

Sally K. Wahba, Jason O. Hallstrom, and Neelam Soundarajan

Dependable Systems Research Group, School of Computing

October 27, 2010
Design Patterns

Overview

- **Definition**
  - A general reusable solution to a commonly occurring problem in software design

- **Benefits**
  - Record expert knowledge
  - Create vocabulary
Design Patterns (cont’d)

Patterns in Sensor Networks

- **Problems**
  - Most design patterns are inapplicable
    - Resource requirements
    - Programming models

- **Process**
  - Identify common design challenges
  - Study expert-crafted systems
  - Codify design patterns
  - Formally specify design patterns
Structural Formalism

Motivation
- Helps refine our understanding of candidate pattern
- Improves pattern description
- Ensures that system examples exhibit the candidate structure

Basic Structure
- A pattern is represented by an assertion in first-order logic
- The terms of the formalism correspond to structural pattern elements
- Relations capture the relationships among pattern elements
## Structural Formalism Terms

<table>
<thead>
<tr>
<th>Term</th>
<th>Description of Elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C$</td>
<td>Constants</td>
</tr>
<tr>
<td>$V$</td>
<td>Variables</td>
</tr>
<tr>
<td>$O$</td>
<td>Operations</td>
</tr>
<tr>
<td>$F_h$</td>
<td>Header files</td>
</tr>
<tr>
<td>$F_i$</td>
<td>C or nesC implementation files</td>
</tr>
<tr>
<td>$F_n$</td>
<td>nesC configuration files</td>
</tr>
</tbody>
</table>
## Structural Formalism Relations

### Relations

<table>
<thead>
<tr>
<th>Relation</th>
<th>Domain</th>
<th>Intent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implemented_in(o, f)</td>
<td>O x F_{h,i}</td>
<td>Operation o is defined in file f</td>
</tr>
<tr>
<td>Defined_in(c, f)</td>
<td>C x F_{h,i}</td>
<td>Constant c is defined in file f</td>
</tr>
<tr>
<td>Defined_in(v, f)</td>
<td>V x F_{h,i}</td>
<td>Variable v is defined in file f</td>
</tr>
<tr>
<td>Accessed_in(c, o)</td>
<td>C x O</td>
<td>Constant c is accessed in operation o</td>
</tr>
<tr>
<td>Accessed_in(v, o)</td>
<td>V x O</td>
<td>Variable v is accessed in operation o</td>
</tr>
<tr>
<td>Global(v)</td>
<td>V</td>
<td>Variable v is global</td>
</tr>
<tr>
<td>Invocation(o_1, o_2)</td>
<td>O x O</td>
<td>Operation o_1 invokes operation o_2</td>
</tr>
<tr>
<td>Passed_to(c, o)</td>
<td>C x O</td>
<td>Constant c is passed as an argument to operation o</td>
</tr>
<tr>
<td>Passed_to(v, o)</td>
<td>V x O</td>
<td>The value of variable v is passed as an argument to operation o</td>
</tr>
<tr>
<td>Includes(f_1, f_2)</td>
<td>F_{h,i,n} x F_{h}</td>
<td>File f_1 includes file f_2</td>
</tr>
<tr>
<td>Computed_from(o, v, c)</td>
<td>O x V x C</td>
<td>The value of variable v in operation o is computed from a constant c</td>
</tr>
<tr>
<td>Computed_from(o, v, v_1)</td>
<td>O x V x V</td>
<td>The value of variable v in operation o is computed from the value of v_1</td>
</tr>
<tr>
<td>Reachable(o_1, o_2)</td>
<td>O x O</td>
<td>Operation o_1 is reachable from operation o_2</td>
</tr>
<tr>
<td>Externalizes(o)</td>
<td>O</td>
<td>Operation o communicates a message to an external device</td>
</tr>
</tbody>
</table>
Basestation Dictionary Pattern

Description

- **Intent:** provide a mapping from space-efficient keys to larger constants
- **Participants:** Basestation Dictionary and Embedded Key User
- **Known Uses:** LiteOS and Mantis
- **Consequences:**
  - Ensures unique keys
  - Guarantees consistency between keys on embedded devices and basestation
  - Maintains key map at the basestation
Basestation Dictionary Pattern (cont’d)

Sample Code

```c
// header.h
#define MESSAGE_1 1 /* "test 3 passed in state 12" */
#define MESSAGE_2 2 /* "check failed" */
...

// Client.nc
#include "header.h" ...
module Client { uses interface Logger; ... }
implementation { ...
  task void checkState() { ...
    if (state == 12 && testPassed) {
      call Logger.writeMessage(MESSAGE_1);
    } ...
  } ...
}```
Basestation Dictionary Pattern (cont’d)

Structural Specifications

\[ \exists \text{client, sender, communicator } \in F_i; \]
\[ \text{keys } \in F_h; \text{ key}_1 \ldots \text{ key}_n \in C; \]
\[ \text{communicate, send, operation}_1 \ldots \text{ operation}_n \in O \]

\[ \text{Defined in} (\text{key}_1, \text{keys}) \land \ldots \land \text{Defined in} (\text{key}_n, \text{keys}) \land \]
\[ \text{Implemented in} (\text{operation}_1, \text{client}) \land \ldots \land \]
\[ \text{Implemented in} (\text{operation}_n, \text{client}) \land \]
\[ \text{Implemented in} (\text{communicate, communicator}) \land \]
\[ \text{Implemented in} (\text{send, sender}) \land \]
\[ \text{Externalizes} (\text{send}) \land \text{Includes} (\text{client, keys}) \land \]
\[ (\text{Accessed in} (\text{key}_1, \text{operation}_1) \lor \ldots \lor \]
\[ \text{Accessed in} (\text{key}_n, \text{operation}_1)) \land \ldots \land \]
\[ (\text{Accessed in} (\text{key}_1, \text{operation}_n) \lor \ldots \lor \]
\[ \text{Accessed in} (\text{key}_n, \text{operation}_n)) \land \]
\[ \text{Invocation}(\text{operation}_1, \text{communicate}) \land \ldots \land \]
\[ \text{Invocation}(\text{operation}_n, \text{communicate}) \land \]
\[ \text{Passed to}(\text{key}_1, \text{communicate}) \land \ldots \land \]
\[ \text{Passed to}(\text{key}_n, \text{communicate}) \land \]
\[ \text{Reachable}(\text{send, communicate}) \]

<table>
<thead>
<tr>
<th>Term</th>
<th>Elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C$</td>
<td>Constants</td>
</tr>
<tr>
<td>$V$</td>
<td>Variables</td>
</tr>
<tr>
<td>$O$</td>
<td>Operations</td>
</tr>
<tr>
<td>$F_h$</td>
<td>Header files</td>
</tr>
<tr>
<td>$F_i$</td>
<td>Impl. files</td>
</tr>
<tr>
<td>$F_n$</td>
<td>Conf. files</td>
</tr>
</tbody>
</table>
Control Flow Mediator Pattern

Description

- **Intent**: simplify control flow and synchronization logic by linearizing interrupt-triggered events, while minimizing execution in interrupt context
- **Participants**: Handler and Mediator
- **Known Uses**: TinyThread and TinyOS
- **Consequences**:
  - Reduces the time spent executing in interrupt context
  - Handles interrupts outside interrupt context
  - Simplifies system maintenance
Control Flow Mediator Pattern (cont’d)

Mediator

if (state == INITING)
else if (state == INITED && (failures <= MAX))

Diagram:

- Node 1 (INITING)
- Node 2 (INITED)
- Node 3 (failures <= MAX)
Control Flow Mediator Pattern (cont’d)

Mediator

if (state == INITING)
else if (state == INITED)
        && (failures <= MAX) )

if (state == INITING)
else if (state == INITED)
        && (failures <= MAX) )
Control Flow Mediator Pattern (cont’d)

Mediator

if (state == INITING)
else if ((state == INITED) && (failures <= MAX))
Control Flow Mediator Pattern (cont’d)

Sample Code

```c
// states.h
enum State {OFF, INITING, INITED, STARTING, STARTED, ..., ERROR};

// client.nc
#include "states.h" ...
State state = OFF; int failures = 0; ...
command void Init.init() {
    state = INITING; failures = 0; post mediator();
} ...

event void Device.initDone(bool success) {
    if (success) {
        state = INITED; failures = 0; post mediator();
    } else { failures++; post mediator(); }
} ...

task void mediator() {
    if (state == INITING) { ...
        /* access driver state and request init. */ ...
    } else if ((state == INITED) && (failures <= MAX)) {
        /* access driver state and request startup */ ...
    } else { ... } ...
} ...
```
Control Flow Mediator Pattern (cont’d)

Structural Specifications

<table>
<thead>
<tr>
<th>Term</th>
<th>Elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C$</td>
<td>Constants</td>
</tr>
<tr>
<td>$V$</td>
<td>Variables</td>
</tr>
<tr>
<td>$O$</td>
<td>Operations</td>
</tr>
<tr>
<td>$F_h$</td>
<td>Header files</td>
</tr>
<tr>
<td>$F_i$</td>
<td>Impl. files</td>
</tr>
<tr>
<td>$F_n$</td>
<td>Conf. files</td>
</tr>
</tbody>
</table>

$\exists$ client, handlers_1 ... handlers_n, controller $\in F_i$; 
states $\in F_h$; state_1 ... state_n $\in C$; program_state $\in V$;
mediator, manager, handle_1 ... handle_n $\in O$;

Defined_in(state_1, states) $\land$ ... $\land$
Defined_in(state_n, states) $\land$
Global(program_state) $\land$
Implemented_in(handle_1, handlers_1) $\land$ ... $\land$
Implemented_in(handle_n, handlers_n) $\land$
Implemented_in(mediator, controller) $\land$
Implemented_in(manager, client) $\land$
Includes(controller, states) $\land$
Includes(handlers_1, states) $\land$ ... $\land$
Includes(handlers_n, states) $\land$
Computed_from(handle_1, program_state, state_1) $\land$ ... $\land$
Computed_from(handle_n, program_state, state_n) $\land$
Invocation(manager, mediator) $\land$
Accessed_in(program_state, mediator)
Port Template Pattern

Description

- Intent: define the pin-level control interface for an embedded hardware module in such a way that the driver can be easily configured for use on different platforms where the pin assignments may vary.
- Participants: Chip-level Driver and Port Template
- Known Uses: LiteOS and TinyOS
- Consequences:
  - Decreases implementation errors
  - Increases portability of driver code
  - Guarantees safe driver instantiation
Port Template Pattern (cont’d)

Sample Code

```
// portspins.h
#define RADIO_PORT PORTC
#define RADIO_DDR DDRC
#define RADIO_BIT 0
...

// PortsPinC.nc
#include "portspins.h"
module PortsPinC {
  provides interface PortsPin;
}
implementation {
  void command PortsPin.deepSleep() {
    RADIO_DDR |= (1 << RADIO_BIT);
    RADIO_PORT &= ~(1 << RADIO_BIT);
  }
  ...
}
```
Port Template Pattern (cont’d)

Structural Specifications

<table>
<thead>
<tr>
<th>Term</th>
<th>Elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>Constants</td>
</tr>
<tr>
<td>V</td>
<td>Variables</td>
</tr>
<tr>
<td>O</td>
<td>Operations</td>
</tr>
<tr>
<td>( F_h )</td>
<td>Header files</td>
</tr>
<tr>
<td>( F_i )</td>
<td>Impl. files</td>
</tr>
<tr>
<td>( F_n )</td>
<td>Conf. files</td>
</tr>
</tbody>
</table>

\[ \exists \text{pin}_1 \ldots \text{pin}_n, \text{port}_1 \ldots \text{port}_n \in C; \]
\[ \text{operation}_1 \ldots \text{operation}_n \in O; \]
\[ \text{pins}, \text{ports} \in F_h; \text{driver} \in F_i \]

\[
\begin{align*}
\& \text{Defined}\_\text{in}(\text{pin}_1, \text{pins}) \land \ldots \land \\
\& \text{Defined}\_\text{in}(\text{pin}_n, \text{pins}) \land \\
\& \text{Defined}\_\text{in}(\text{port}_1, \text{ports}) \land \ldots \land \\
\& \text{Defined}\_\text{in}(\text{port}_n, \text{ports}) \land \\
\& \text{Implemented}\_\text{in}(\text{operation}_1, \text{driver}) \land \ldots \land \\
\& \text{Implemented}\_\text{in}(\text{operation}_n, \text{driver}) \land \\
\& \text{Includes}(\text{driver}, \text{pins}) \land \text{Includes}(\text{driver}, \text{ports}) \land \\
(\text{Accessed}\_\text{in}(\text{pin}_1, \text{operation}_1) \lor \ldots \lor \\
\text{Accessed}\_\text{in}(\text{pin}_n, \text{operation}_1) \lor \\
\text{Accessed}\_\text{in}(\text{port}_1, \text{operation}_1) \lor \ldots \lor \\
\text{Accessed}\_\text{in}(\text{port}_n, \text{operation}_1)) \land \ldots \land \\
(\text{Accessed}\_\text{in}(\text{pin}_1, \text{operation}_n) \lor \ldots \lor \\
\text{Accessed}\_\text{in}(\text{pin}_n, \text{operation}_n) \lor \\
\text{Accessed}\_\text{in}(\text{port}_1, \text{operation}_n) \lor \ldots \lor \\
\text{Accessed}\_\text{in}(\text{port}_n, \text{operation}_n))
\end{align*}
\]
Future Work

What’s next?

- Identify more patterns
- Automate identification process
- Identify more pattern applications
- Evaluate identified patterns
Questions and Comments