A Hierarchical Approach for Power Management in Mobile Embedded Systems

Arliones Stevert Hoeller Jr.

Lucas Francisco Wanner

Antônio Augusto Fröhlich

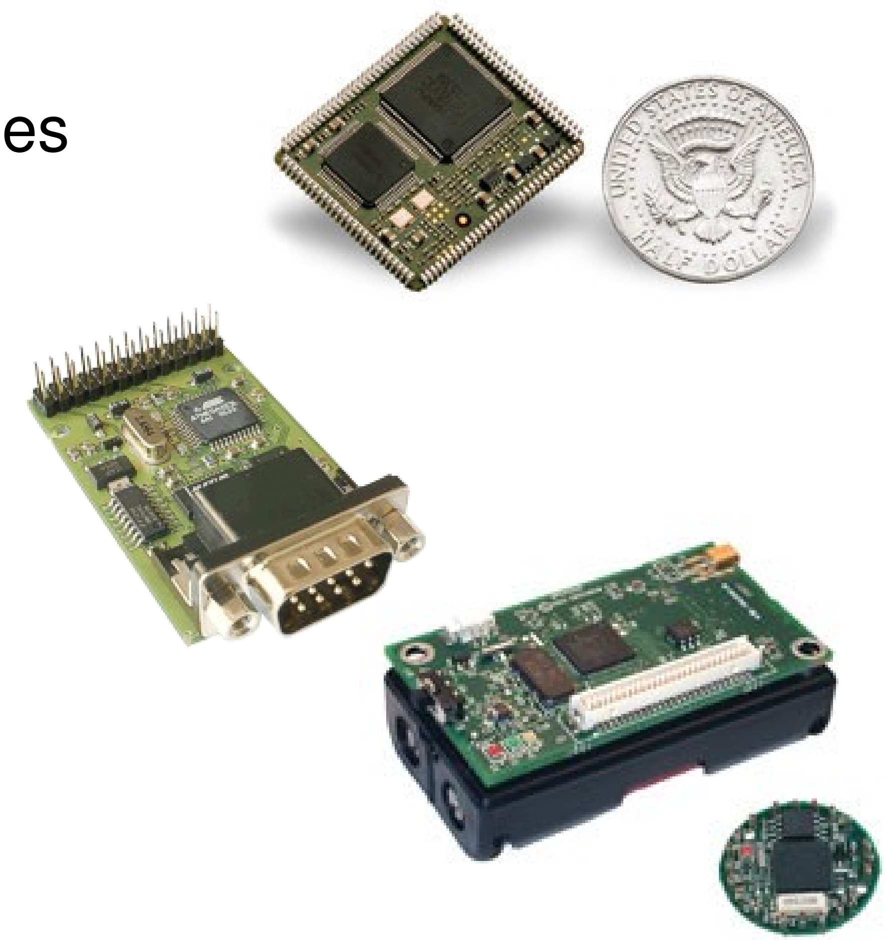
DIPES 2006

Outline

- Power management infrastructure
- Power management in deeply embedded systems
- Application-driven power management
- Implementation in EPOS
- First results
- Further work

Deeply Embedded Systems (DES)

- Application-specific, tiny devices with resource limitations
  - 8-bit, 3-4 Mhz microcontrollers
  - Small data and program memories
  - Several peripherals



Battery = System Lifetime

Power Management Infra-structure

- Hardware support for power management
  - Operating modes (CPU, UART, ADC, etc)
  - Frequency and voltage scaling
  - Event counters
- Power management standards (e.g. APM, ACPI)
  - At the Hardware/Software interface
  - Too complex for DES
- Deeply embedded systems solutions
  - Context-specific
  - Partial hardware coverage
  - Compromise application portability

Power-awareness in DES

- DES “Operating Systems”
  - Simplistic HALs
  - No or few high-level abstractions
  - Power management targeted at the CPU
  - Power state transitions implemented by APP
- DES Energy-Awareness
  - Contestable energy efficiency
    - Peripherals often consume more than peripherals
  - Compromised portability
    - Hardware details exposed to applications

Proposal

- **Application-Driven** Power Management
  - Uniform PM API (software and hardware)
    - Including high-level abstractions
  - Hierarchical state transitions propagation
    - Formalized through Petri Nets
  - Static-metaprogrammed implementation
    - No unnecessary overhead
  - Example

```
file.power(OFF);
communicator.power(OFF);
cout.power(ON);
```

Power management API

- Interface is simple and uniform

```
power(Mode s)
Mode power()
```

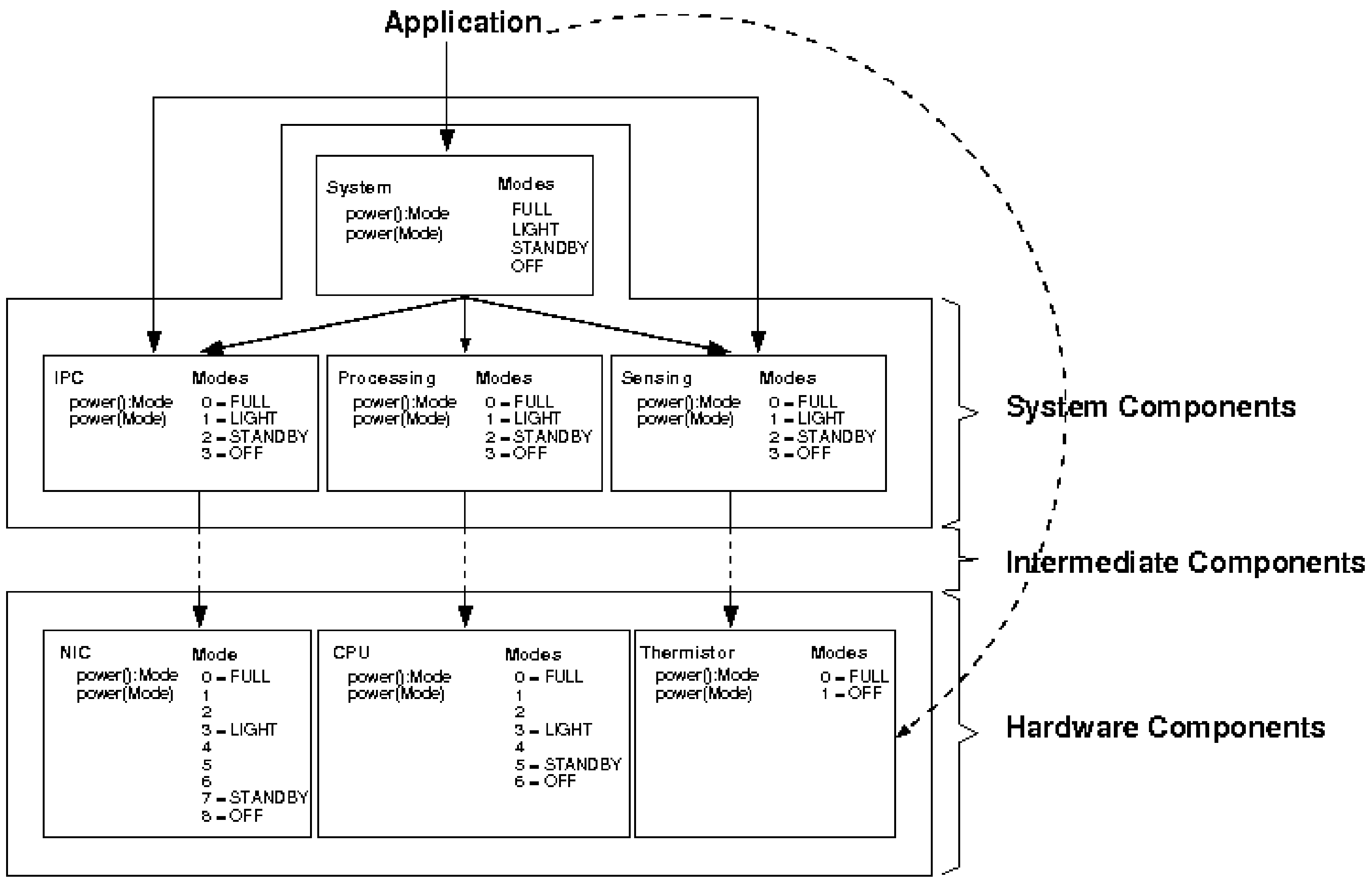
- Configurable operating modes
  - Components export all operating modes
  - Application programmer binds real operating modes to “label” modes

```
[FULL | LIGHT | STANDBY | OFF]
```

Power State Transition Propagation

- Components are hierarchically organized
- Application programmer sees high-level abstractions (e.g., `Thread`, `File`, `Alarm`, `Temperature_Sensor`)
- Operating mode transition messages are propagated throughout components following a dependency graph

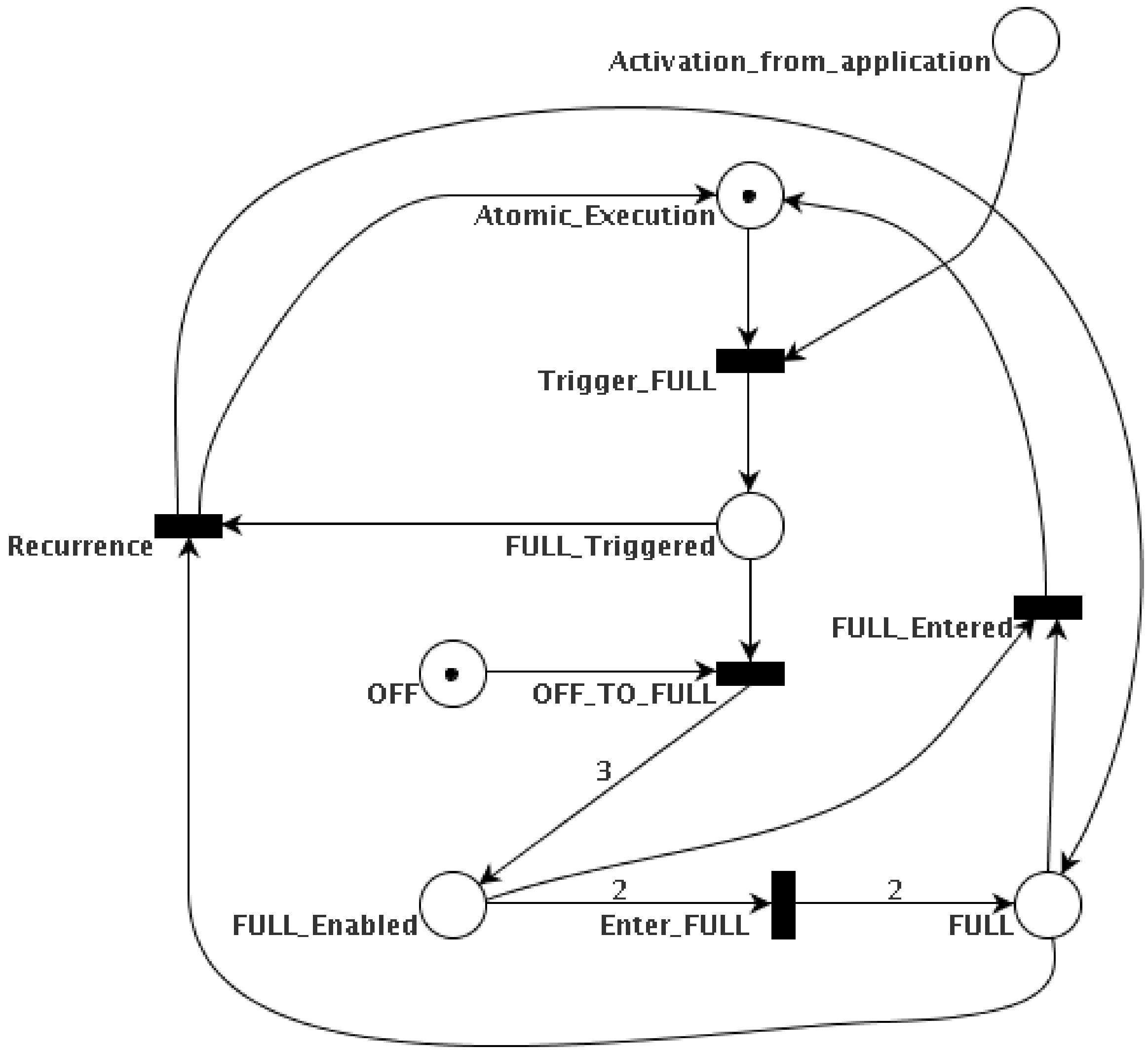
Accessing the API



Operating Mode Transition Nets

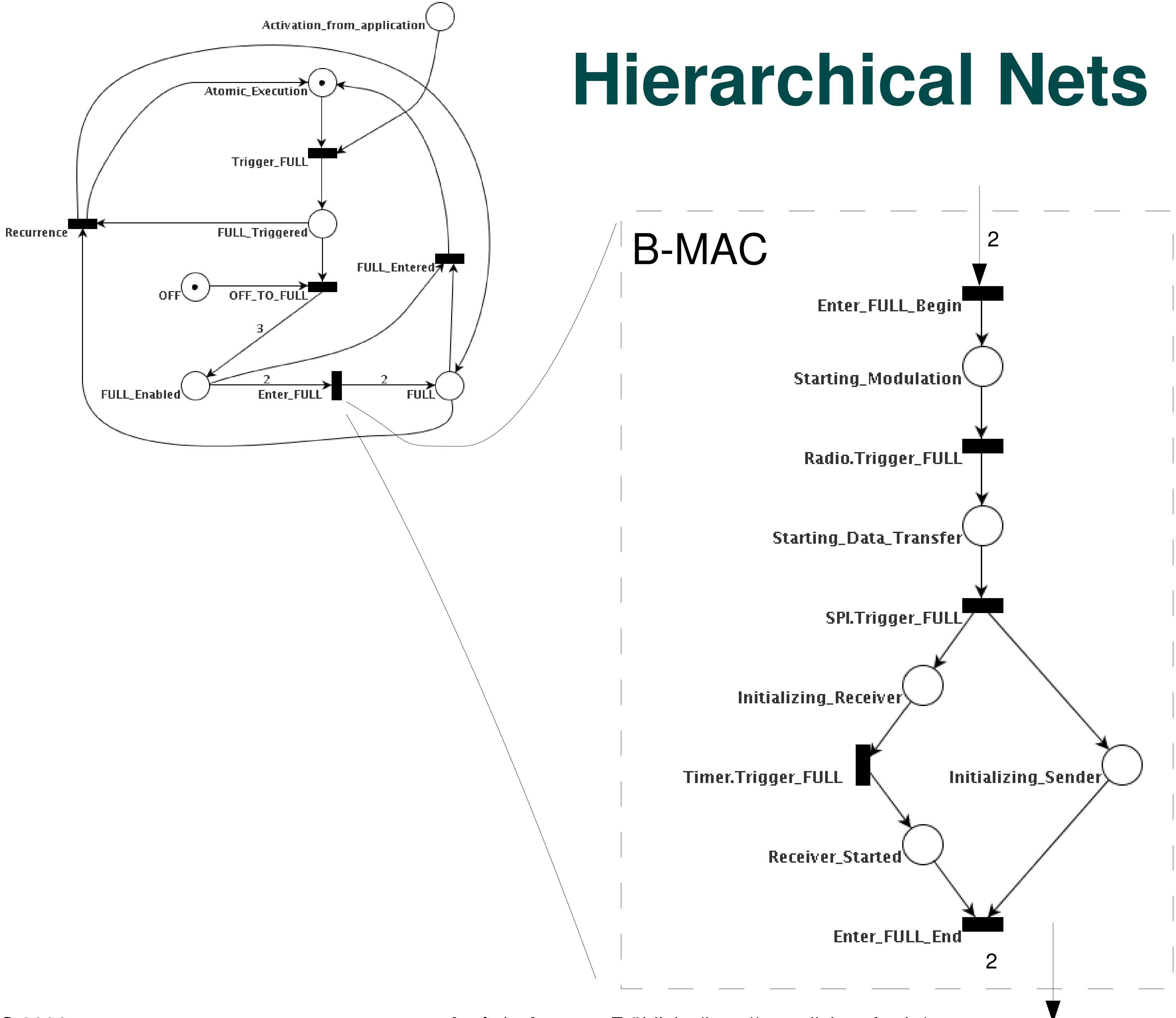
- Formalized through Petri Nets
- Define conditions to transitions
- Organized to represent the proposed power state transition mechanism
- Generalized transition structure
  - OFF, STANDBY, LIGHT, FULL
  - Finite reachability graph => state analysis

Generalized Transition Net

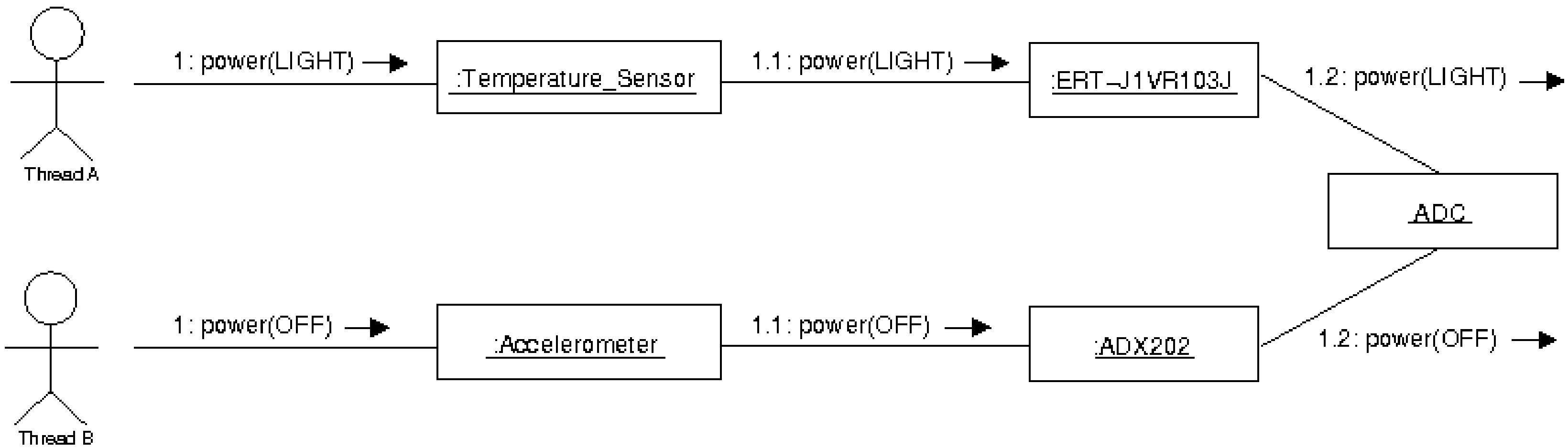


* the full version covers all operating modes

Hierarchical Nets



Shared Dependencies



- Solution
  - Reference counter for each operating mode
  - Component stays in the highest operating mode which has “users”

$$P_j = \sum_{i=j-1}^{0} opmode_i$$

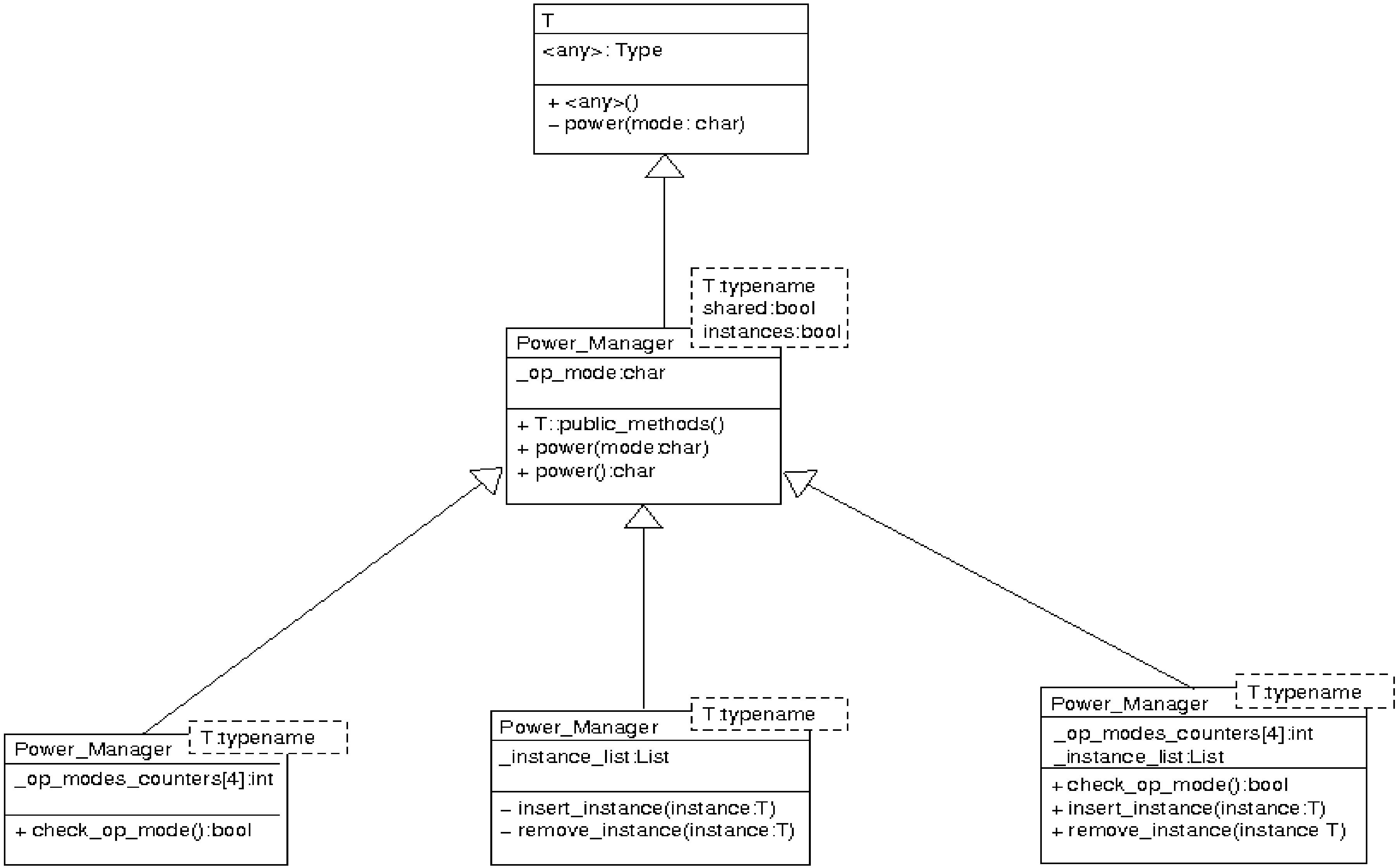
- j is the wanted mode
- opmode is the set of counters
- If Pj = 0 then j is entered

Implementation in EPOS

- *Application-Oriented System Design*
  - Domain Engineering methodology
  - Components + Aspects + Frameworks

- Power Management API
  - Power management is a non-functional feature of computing systems
  - This interface was modeled as an Aspect Program implemented using C++ meta-programming techniques

Power Manager Aspect

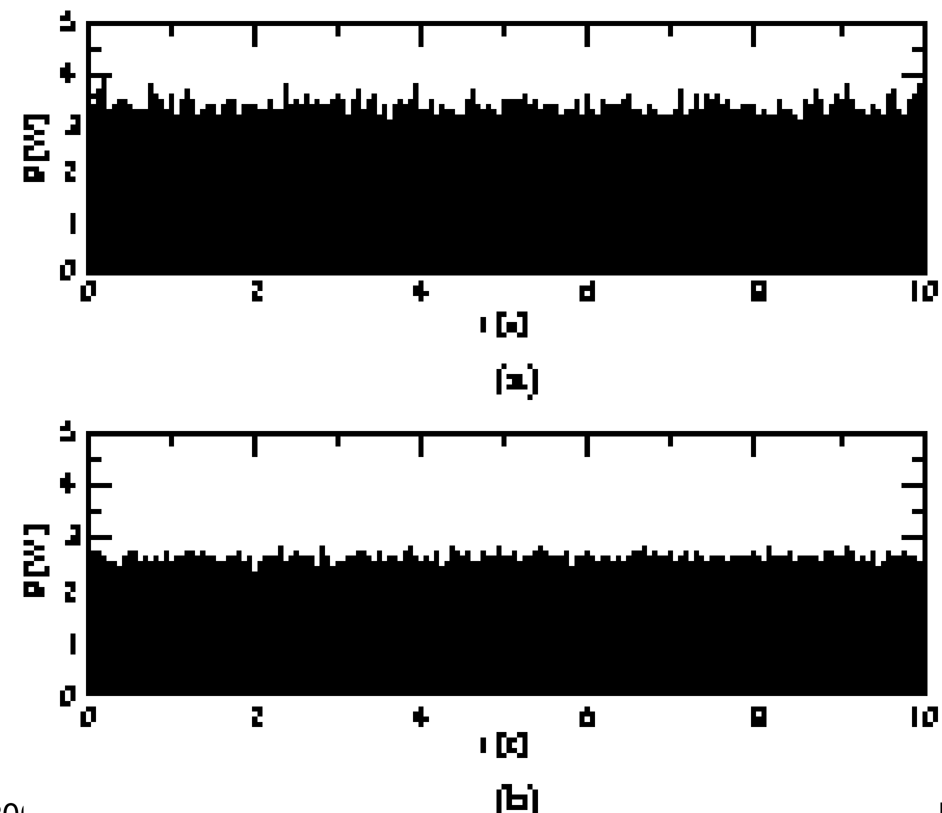


Sample Application

```
#include <system.h>
#include <sensor.h>
#include <uart.h>
#include <alarm.h>

using namespace System;

System sys;
Temperature_Sensor sensor;
UART uart;

void alarm_handler() { uart.put(sensor.sample()); }

int main() {
    Handler_Function handler(& alarm_handler);
    Alarm alarm(1000000,& handler);
    while(1) {
        sys.power(STANDBY);
    }
}
```

Power Consumption in Example

- Hardware
  - STK-500, ATMega16, 10 KΩ Thermistor
- Software
  - EPOS (`System, Alarm, UART, Temp_Sensor`)



Consumption: 3.96 J

38.1 % less energy

Consumption: 2.45 J

Summary

- Application-Driven power management in deeply embedded systems
  - Little overhead
  - Platform-independent (application portability)
- Uniform API
  - Configurable operating modes
  - On high-level abstractions
  - Easier application programming

On-going Work

- Dynamic power manager
  - A thread shuts down everything that is not in use at the moment
    - Usage/reference counters
    - Enabled test on most operations
- Static power consumption model
  - Enables design space exploration
  - Profiling of use cases
    - Classify high-level components in regard to power consumption
    - Sometimes even estimating battery time

Complex example

```
MP3_Player player;
Display display;

//...

void main() {
  //...
  char * file_name = select_file();
  display.power(LIGHT);
  play_file(file_name);
  //...
}

//...

void play_file(char * file_name) {
  File file(file_name);
  player.load(file);
  file.power(OFF);
  player.play();
}
```